3., aktualisierte und überarbeitete Auflage

Michael Inden

Der Weg zum Java-Profi

Konzepte und Techniken für die professionelle Java-Entwicklung

dpunkt.verlag

Dipl.-Inform. Michael Inden ist Oracle-zertifizierter Java-Entwickler für JDK 6. Nach seinem Studium in Oldenburg war er lange Zeit als Softwareentwickler und -architekt bei verschiedenen internationalen Firmen tätig.

Dabei hat er über 15 Jahre Erfahrung beim Entwurf objektorientierter Softwaresysteme gesammelt, an diversen Fortbildungen und an mehreren Java-One-Konferenzen in San Francisco teilgenommen. Sein besonderes Interesse gilt dem Design qualitativ hochwertiger Applikationen mit ergonomischen, grafischen Oberflächen sowie dem Coaching von Kollegen.

Papier plus PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:
www.dpunkt.de/plus

Der Weg zum Java-Profi

Konzepte und Techniken für die professionelle Java-Entwicklung

3., aktualisierte und überarbeitete Auflage

Michael Inden

dpunkt.verlag

Michael Inden
michael_inden@hotmail.com

Lektorat: Dr. Michael Barabas
Technischer Review: Torsten Horn, Aachen
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Michael Inden
Herstellung: Frank Heidt, Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN
Buch 978-3-86490-203-1
PDF 978-3-86491-637-3
ePub 978-3-86491-638-0

3., aktualisierte und überarbeitete Auflage 2015
Copyright © 2015 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.
5 4 3 2 1 0

Inhaltsübersicht

1 Einleitung

Java-Grundlagen, Analyse und Design

2 Professionelle Arbeitsumgebung

3 Objektorientiertes Design

4 Java-Grundlagen

Bausteine stabiler Java-Applikationen

5 Das Collections-Framework

6 Applikationsbausteine

7 Multithreading

8 Fortgeschrittene Java-Themen

9 Programmierung grafischer Benutzeroberflächen mit Swing

10 Basiswissen Internationalisierung

Die Neuerungen in Java 8

11 Lambda-Ausdrücke

12 Bulk Operations on Collections

13 JSR-310: Date And Time API

14 Einstieg JavaFX 8

15 Weitere Änderungen in JDK 8

Fallstricke und Lösungen im Praxisalltag

16 Bad Smells

17 Refactorings

18 Entwurfsmuster

Qualitätssicherungsmaßnahmen

19 Programmierstil und Coding Conventions

20 Unit Tests

21 Codereviews

22 Optimierungen

Anhang

Literaturverzeichnis
Index

Inhaltsverzeichnis

1 Einleitung

1.1 Über dieses Buch

1.1.1 Motivation

1.1.2 Was leistet dieses Buch und was nicht?

1.1.3 Wie und was soll mithilfe des Buchs gelernt werden?

1.1.4 Wer sollte dieses Buch lesen?

1.2 Aufbau des Buchs

1.3 Konventionen und ausführbare Programme

I Java-Grundlagen, Analyse und Design

2 Professionelle Arbeitsumgebung

2.1 Vorteile von IDEs am Beispiel von Eclipse

2.2 Projektorganisation

2.2.1 Projektstruktur in Eclipse und Ant

2.2.2 Projektstruktur für Maven und Gradle

2.3 Einsatz von Versionsverwaltungen

2.3.1 Arbeiten mit zentralen Versionsverwaltungen

2.3.2 Dezentrale Versionsverwaltungen

2.3.3 VCS und DVCS im Vergleich

2.4 Einsatz eines Unit-Test-Frameworks

2.4.1 Das JUnit-Framework

2.4.2 Vorteile von Unit Tests

2.5 Debugging

2.5.1 Fehlersuche mit einem Debugger

2.5.2 Remote Debugging

2.6 Deployment von Java-Applikationen

2.6.1 Das JAR-Tool im Kurzüberblick

2.6.2 JAR inspizieren und ändern, Inhalt extrahieren

2.6.3 Metainformationen und das Manifest

2.6.4 Inspizieren einer JAR-Datei

2.7 Einsatz eines IDE-unabhängigen Build-Prozesses
2.7.1 Ant im Überblick
2.7.2 Maven im Überblick
2.7.3 Builds mit Gradle
2.7.4 Vergleich von Ant, Maven und Gradle
2.8 Weiterführende Literatur
3 Objektorientiertes Design
3.1 OO-Grundlagen
3.1.1 Grundbegriffe
3.1.2 Beispielentwurf: Ein Zähler
3.1.3 Vom imperativen zum objektorientierten Entwurf
3.1.4 Diskussion der OO-Grundgedanken
3.1.5 Wissenswertes zum Objektzustand
3.2 Grundlegende OO-Techniken
3.2.1 Schnittstellen (Interfaces)
3.2.2 Basisklassen und abstrakte Basisklassen
3.2.3 Interfaces und abstrakte Basisklassen
3.3 Wissenswertes zu Vererbung
3.3.1 Probleme durch Vererbung
3.3.2 Delegation statt Vererbung
3.4 Fortgeschrittenere OO-Techniken
3.4.1 Read-only-Interface
3.4.2 Immutable-Klasse
3.4.3 Marker-Interface
3.4.4 Konstantensammlungen und Aufzählungen
3.4.5 Value Object (Data Transfer Object)
3.5 Prinzipien guten OO-Designs
3.5.1 Geheimnisprinzip nach Parnas
3.5.2 Law of Demeter
3.5.3 SOLID-Prinzipien
3.6 Formen der Varianz

3.6.1 Grundlagen der Varianz
3.6.2 Kovariante Rückgabewerte
3.7 Generische Typen (Generics)
3.7.1 Einführung
3.7.2 Generics und Auswirkungen der Type Erasure
3.8 Weiterführende Literatur
4 Java-Grundlagen
4.1 Die Klasse `Object`
4.1.1 Die Methode `toString()`
4.1.2 Die Methode `equals()`
4.2 Primitive Typen und Wrapper-Klassen
4.2.1 Grundlagen
4.2.2 Konvertierung von Werten
4.2.3 Wissenswertes zu Auto-Boxing und Auto-Unboxing
4.2.4 Ausgabe und Verarbeitung von Zahlen
4.3 Stringverarbeitung
4.3.1 Die Klasse `String`
4.3.2 Die Klassen `StringBuffer` und `StringBuilder`
4.3.3 Ausgaben mit `format()` und `printf()`
4.3.4 Die Klasse `StringTokenizer`
4.3.5 Die Methode `split()` und das 1x1 der regulären Ausdrücke
4.4 Datumsverarbeitung
4.4.1 Fallstricke der Datums-APIs
4.4.2 Das `Date`-API
4.4.3 Das `Calendar`-API
4.5 Innere Interfaces und innere Klassen
4.5.1 Varianten innerer Klassen
4.5.2 Innere Interfaces
4.5.3 Designbeispiel mit inneren Klassen und Interfaces
4.5.4 Lokal definierte Klassen und Interfaces
4.6 Ein- und Ausgabe (I/O)

4.6.1 Dateibehandlung und die Klasse `File`
4.6.2 Ein- und Ausgabestreams im Überblick
4.6.3 Zeichencodierungen bei der Ein- und Ausgabe
4.6.4 Speichern und Laden von Daten und Objekten
4.6.5 Dateiverarbeitung in JDK 7
4.7 Fehlerbehandlung
4.7.1 Einstieg in die Fehlerbehandlung
4.7.2 Checked Exceptions und Unchecked Exceptions
4.7.3 Exception Handling und Ressourcenfreigabe
4.7.4 Besonderheiten beim Exception Handling mit JDK 7
4.7.5 Assertions
4.8 Weiterführende Literatur
II Bausteine stabiler Java-Applikationen
5 Das Collections-Framework
5.1 Datenstrukturen und Containerklasse
5.1.1 Wahl einer geeigneten Datenstruktur
5.1.2 Arrays
5.1.3 Das Interface `Collection`
5.1.4 Das Interface `Iterator`
5.1.5 Listen und das Interface `List`
5.1.6 Mengen und das Interface `Set`
5.1.7 Grundlagen von hashbasierten Containern
5.1.8 Grundlagen automatisch sortierender Container
5.1.9 Die Methoden `equals()`, `hashCode()` und `compareTo()` im Zusammenspiel
5.1.10 Schlüssel-Wert-Abbildungen und das Interface `Map`
5.1.11 Erweiterungen am Beispiel der Klasse `HashMap`
5.1.12 Entscheidungshilfe zur Wahl von Datenstrukturen
5.2 Suchen, Sortieren und Filtern
5.2.1 Suchen
5.2.2 Sortieren von Arrays und Listen

5.2.3 Sortieren mit Komparatoren
5.2.4 Filtern von Collections
5.3 Utility-Klassen und Hilfsmethoden
5.3.1 Nützliche Hilfsmethoden
5.3.2 Dekorierer `synchronized`, `unmodifiable` und `checked`
5.3.3 Vordefinierte Algorithmen
5.4 Containerklassen: Generics und Varianz
5.5 Fallstricke im Collections-Framework
5.5.1 Wissenswertes zu Arrays
5.5.2 Wissenswertes zu `Stack`, `Queue` und `Deque`
5.6 Weiterführende Literatur
6 Applikationsbausteine
6.1 Einsatz von Bibliotheken
6.2 Google Guava im Kurzüberblick
6.2.1 String-Aktionen
6.2.2 String-Konkatenation und -Extraktion
6.2.3 Erweiterungen für Collections
6.2.4 Weitere Utility-Funktionalitäten
6.3 Wertebereichs- und Parameterprüfungen
6.3.1 Prüfung einfacher Wertebereiche und Wertemengen
6.3.2 Prüfung komplexerer Wertebereiche
6.4 Logging-Frameworks
6.4.1 Apache log4j
6.4.2 Tipps und Tricks zum Einsatz von Logging mit log4j
6.5 Konfigurationsparameter und -dateien
6.5.1 Einlesen von Kommandozeilenparametern
6.5.2 Verarbeitung von Properties
6.5.3 Die Klasse `Preferences`
6.5.4 Weitere Möglichkeiten zur Konfigurationsverwaltung
7 Multithreading
7.1 Threads und Runnables

7.1.1 Definition der auszuführenden Aufgabe
7.1.2 Start, Ausführung und Ende von Threads
7.1.3 Lebenszyklus von Threads und Thread-Zustände
7.1.4 Unterbrechungswünsche durch Aufruf von `interrupt()`
7.2 Zusammenarbeit von Threads
7.2.1 Konkurrierende Datenzugriffe
7.2.2 Locks, Monitore und kritische Bereiche
7.2.3 Deadlocks und Starvation
7.2.4 Kritische Bereiche und das Interface `Lock`
7.3 Kommunikation von Threads
7.3.1 Kommunikation mit Synchronisation
7.3.2 Kommunikation über die Methoden `wait()`, `notify()` und `notifyAll()`
7.3.3 Abstimmung von Threads
7.3.4 Unerwartete `IllegalMonitorStateExceptions`
7.4 Das Java-Memory-Modell
7.4.1 Sichtbarkeit
7.4.2 Atomarität
7.4.3 Reorderings
7.5 Besonderheiten bei Threads
7.5.1 Verschiedene Arten von Threads
7.5.2 Exceptions in Threads
7.5.3 Sicheres Beenden von Threads
7.5.4 Zeitgesteuerte Ausführung
7.6 Die Concurrency Utilities
7.6.1 Concurrent Collections
7.6.2 Das Executor-Framework
7.6.3 Das Fork-Join-Framework
7.7 Weiterführende Literatur
8 Fortgeschrittene Java-Themen
8.1 Crashkurs Reflection
8.1.1 Grundlagen

8.1.2 Zugriff auf Methoden und Attribute
8.1.3 Spezialfälle
8.1.4 Type Erasure und Typinformationen bei Generics
8.2 Annotations
8.2.1 Einführung in Annotations
8.2.2 Standard-Annotations des JDKs
8.2.3 Definition eigener Annotations
8.2.4 Annotation zur Laufzeit auslesen
8.3 Serialisierung
8.3.1 Grundlagen der Serialisierung
8.3.2 Die Serialisierung anpassen
8.3.3 Versionsverwaltung der Serialisierung
8.3.4 Optimierung der Serialisierung
8.4 Objektkopien und das Interface `Cloneable`
8.4.1 Das Interface `Cloneable`
8.4.2 Alternativen zur Methode `clone()`
8.5 Garbage Collection
8.5.1 Grundlagen zur Garbage Collection
8.5.2 Herkömmliche Algorithmen zur Garbage Collection
8.5.3 Einflussfaktoren auf die Garbage Collection
8.5.4 Der Garbage Collector »G1«
8.5.5 Memory Leaks: Gibt es die auch in Java?!
8.5.6 Objektzerstörung und `finalize()`
8.6 Weiterführende Literatur
9 Programmierung grafischer Benutzeroberflächen mit Swing
9.1 Grundlagen zu grafischen Oberflächen
9.1.1 Überblick: Bedienelemente und Container
9.1.2 Einführung in das Layoutmanagement
9.1.3 Komplexere Layouts (Kombination von Layoutmanagern)
9.1.4 Grundlagen zur Ereignisbehandlung
9.1.5 Weitere gebräuchliche Event Listener

9.1.6 Varianten der Ereignisverarbeitung
9.2 Multithreading und Swing
9.2.1 Crashkurs Event Handling in Swing
9.2.2 Ausführen von Aktionen
9.2.3 Die Klasse `SwingWorker`
9.3 Zeichnen in GUI-Komponenten
9.3.1 Generelles zum Zeichnen in GUI-Komponenten
9.3.2 Kurzeinführung in Java 2D
9.3.3 Bedienelemente mit Java 2D selbst erstellen
9.4 Komplexe Bedienelemente
9.4.1 Grundlagen
9.4.2 Die Klasse `JList`
9.4.3 Die Klasse `JTable`
9.4.4 Die Klasse `JTree`
9.5 Weiterführende Literatur
10 Basiswissen Internationalisierung
10.1 Internationalisierung im Überblick
10.1.1 Grundlagen und Normen
10.1.2 Die Klasse `Locale`
10.1.3 Die Klasse `PropertyResourceBundle`
10.1.4 Formatierte Ein- und Ausgabe
10.1.5 Zahlen und die Klasse `NumberFormat`
10.1.6 Datumswerte und die Klasse `DateFormat`
10.1.7 Textmeldungen und die Klasse `MessageFormat`
10.1.8 Stringvergleiche mit der Klasse `Collator`
10.2 Programmbausteine zur Internationalisierung
10.2.1 Unterstützung mehrerer Datumsformate
10.2.2 Nutzung mehrerer Sprachdateien
III Die Neuerungen in Java 8
11 Lambda-Ausdrücke
11.1 Einstieg in Lambdas

11.1.1 Lambdas am Beispiel
11.1.2 Functional Interfaces und SAM-Typen
11.1.3 Type Inference und Kurzformen der Syntax
11.1.4 Lambdas als Parameter und als Rückgabewerte
11.1.5 Unterschiede: Lambdas vs. anonyme innere Klassen
11.2 Defaultmethoden
11.2.1 Interface-Erweiterungen
11.2.2 Vorgabe von Standardverhalten
11.2.3 Erweiterte Möglichkeiten durch Defaultmethoden
11.2.4 Spezialfall: Was passiert bei Konflikten?
11.2.5 Vorteile und Gefahren von Defaultmethoden
11.2.6 Statische Methoden in Interfaces
11.3 Methodenreferenzen
11.4 Fazit
12 Bulk Operations on Collections
12.1 Externe vs. interne Iteration
12.1.1 Externe Iteration
12.1.2 Interne Iteration
12.1.3 Externe vs. interne Iteration an einem Beispiel
12.2 Collections-Erweiterungen
12.2.1 Das Interface `Predicate<T>`
12.2.2 Die Methode `Collection.removeIf()`
12.2.3 Das Interface `UnaryOperator<T>`
12.2.4 Die Methode `List.replaceAll()`
12.3 Streams
12.3.1 Streams erzeugen — Create Operations
12.3.2 Intermediate und Terminal Operations im Überblick
12.3.3 Zustandslose Intermediate Operations
12.3.4 Zustandsbehaftete Intermediate Operations
12.3.5 Terminal Operations
12.3.6 Wissenswertes zur Parallelverarbeitung

12.4 Filter-Map-Reduce
12.4.1 Herkömmliche Realisierung
12.4.2 Filter-Map-Reduce mit JDK 8
12.5 Fallstricke bei Lambdas und funktionaler Programmierung
12.5.1 Java-Fehlermeldungen werden zu komplex
12.5.2 Fallstrick: Imperative Lösung 1:1 funktional umsetzen
12.6 Fazit
13 JSR-310: Date And Time API
13.1 Datumsverarbeitung vor JSR-310
13.2 Überblick über die neu eingeführten Klassen
13.2.1 Die Klasse `Instant`
13.2.2 Die Aufzählung `ChronoUnit`
13.2.3 Die Klasse `Duration`
13.2.4 Die Klassen `LocalDate`, `LocalTime` und `LocalDateTime`
13.2.5 Die Aufzählungen `DayOfWeek` und `Month`
13.2.6 Die Klassen `YearMonth`, `MonthDay` und `Year`
13.2.7 Die Klasse `Period`
13.2.8 Die Klasse `Clock`
13.2.9 Die Klasse `ZonedDateTime`
13.2.10 Beispiel: Berechnung einer Zeitdifferenz
13.2.11 Interoperabilität mit Legacy-Code
13.3 Fazit
14 Einstieg JavaFX 8
14.1 Einführung – JavaFX im Überblick
14.1.1 Motivation für JavaFX und Historisches
14.1.2 Grundsätzliche Konzepte
14.1.3 Layoutmanagement
14.2 Deklarativer Aufbau des GUIs
14.2.1 Deklarative Beschreibung von GUIs
14.2.2 Hello-World-Beispiel mit FXML
14.2.3 Diskussion: Design und Funktionalität strikt trennen

14.3 Rich-Client Experience
14.3.1 Gestaltung mit CSS
14.3.2 Effekte
14.3.3 Animationen
14.4 Neuerungen in JavaFX 8
14.4.1 Unterstützung von Lambdas als `EventHandler`
14.4.2 Texteffekte
14.4.3 Neue Controls
14.4.4 JavaFX 3D
14.5 Fazit
15 Weitere Änderungen in JDK 8
15.1 Erweiterungen im Interface `Comparator<T>`
15.2 Die Klasse `Optional<T>`
15.2.1 Grundlagen zur Klasse `Optional<T>`
15.2.2 Weiterführendes Beispiel und Diskussion
15.3 Parallele Operationen auf Arrays
15.4 Erweiterungen im Interface `Map<K, V>`
15.5 Erweiterungen im NIO und der Klasse `Files`
15.6 Erweiterungen im Bereich Concurrency
15.7 »Nashorn« – die neue JavaScript-Engine
15.8 Keine Permanent Generation mehr
15.9 Erweiterungen im Bereich Reflection
15.10 Base64-Codierungen
15.11 Änderungen bei Annotations
IV Fallstricke und Lösungen im Praxisalltag
16 Bad Smells
16.1 Programmdesign
16.1.1 Bad Smell: Verwenden von Magic Numbers
16.1.2 Bad Smell: Konstanten in Interfaces definieren
16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert
16.1.4 Bad Smell: Programmcode im Logging-Code

16.1.5 Bad Smell: Dominanter Logging-Code
16.1.6 Bad Smell: Unvollständige Betrachtung aller Alternativen
16.1.7 Bad Smell: Unvollständige Änderungen nach Copy-Paste
16.1.8 Bad Smell: Casts auf unbekannte Subtypen
16.1.9 Bad Smell: Pre-/Post-Increment in komplexeren Statements
16.1.10 Bad Smell: Keine Klammern um Blöcke
16.1.11 Bad Smell: Mehrere aufeinanderfolgende Parameter gleichen Typs
16.1.12 Bad Smell: Grundloser Einsatz von Reflection
16.1.13 Bad Smell: `System.exit()` mitten im Programm
16.1.14 Bad Smell: Variablendeklaration nicht im kleinstmöglichen Sichtbarkeitsbereich
16.2 Klassendesign
16.2.1 Bad Smell: Unnötigerweise veränderliche Attribute
16.2.2 Bad Smell: Herausgabe von `this` im Konstruktor
16.2.3 Bad Smell: Aufruf abstrakter Methoden im Konstruktor
16.2.4 Bad Smell: Referenzierung von Subklassen in Basisklassen
16.2.5 Bad Smell: Mix abstrakter und konkreter Basisklassen
16.2.6 Bad Smell: Öffentlicher Defaultkonstruktor lediglich zum Zugriff auf Hilfsmethoden
16.3 Fehlerbehandlung und Exception Handling
16.3.1 Bad Smell: Unbehandelte Exception
16.3.2 Bad Smell: Unpassender Exception-Typ
16.3.3 Bad Smell: Exceptions zur Steuerung des Kontrollflusses
16.3.4 Bad Smell: Fangen der allgemeinsten Exception
16.3.5 Bad Smell: Rückgabe von `null` statt Exception im Fehlerfall
16.3.6 Bad Smell: Unbedachte Rückgabe von `null`
16.3.7 Bad Smell: Sonderbehandlung von Randfällen
16.3.8 Bad Smell: Keine Gültigkeitsprüfung von Eingabeparametern
16.3.9 Bad Smell: Fehlerhafte Fehlerbehandlung
16.3.10 Bad Smell: I/O ohne `finally`
16.3.11 Bad Smell: Resource Leaks durch Exceptions im Konstruktor
16.4 Häufige Fallstricke

16.5 Weiterführende Literatur

17 Refactorings

17.1 Refactorings am Beispiel

17.2 Das Standardvorgehen

17.3 Kombination von Basis-Refactorings

17.3.1 Refactoring-Beispiel: Ausgangslage und Ziel

17.3.2 Auflösen der Abhängigkeiten

17.3.3 Vereinfachungen

17.3.4 Verlagern von Funktionalität

17.4 Der Refactoring-Katalog

17.4.1 Reduziere die Sichtbarkeit von Attributen

17.4.2 Minimiere veränderliche Attribute

17.4.3 Reduziere die Sichtbarkeit von Methoden

17.4.4 Ersetze Mutator- durch Business-Methode

17.4.5 Minimiere Zustandsänderungen

17.4.6 Führe ein Interface ein

17.4.7 Spalte ein Interface auf

17.4.8 Führe ein Read-only-Interface ein

17.4.9 Führe ein Read-Write-Interface ein

17.4.10 Lagere Funktionalität in Hilfsmethoden aus

17.4.11 Trenne Informationsbeschaffung und -verarbeitung

17.4.12 Wandle Konstantensammlung in `enum` um

17.4.13 Entferne Exceptions zur Steuerung des Kontrollflusses

17.4.14 Wandle in Utility-Klasse mit statischen Hilfsmethoden um

17.5 Defensives Programmieren

17.5.1 Führe eine Zustandsprüfung ein

17.5.2 Überprüfe Eingabeparameter

17.6 Weiterführende Literatur

18 Entwurfsmuster

18.1 Erzeugungsmuster

18.1.1 Erzeugungsmethode

18.1.2 Fabrikmethode (Factory method)
18.1.3 Erbauer (Builder)
18.1.4 Singleton
18.1.5 Prototyp (Prototype)
18.2 Strukturmuster
18.2.1 Fassade (Façade)
18.2.2 Adapter
18.2.3 Dekorierer (Decorator)
18.2.4 Kompositum (Composite)
18.3 Verhaltensmuster
18.3.1 Iterator
18.3.2 Null-Objekt (Null Object)
18.3.3 Schablonenmethode (Template method)
18.3.4 Strategie (Strategy)
18.3.5 Befehl (Command)
18.3.6 Proxy
18.3.7 Beobachter (Observer)
18.3.8 MVC-Architektur
18.4 Weiterführende Literatur
V Qualitätssicherungsmaßnahmen
19 Programmierstil und Coding Conventions
19.1 Grundregeln eines guten Programmierstils
19.1.1 Keep It Human-Readable
19.1.2 Keep It Simple And Short (KISS)
19.1.3 Keep It Natural
19.1.4 Keep It Clean
19.2 Die Psychologie beim Sourcecode-Layout
19.2.1 Gesetz der Ähnlichkeit
19.2.2 Gesetz der Nähe
19.3 Coding Conventions
19.3.1 Grundlegende Namens- und Formatierungsregeln

19.3.2 Namensgebung
19.3.3 Dokumentation
19.3.4 Programmdesign
19.3.5 Klassendesign
19.3.6 Parameterlisten
19.3.7 Logik und Kontrollfluss
19.4 Sourcecode-Prüfung mit Tools
19.4.1 Metriken
19.4.2 Sourcecode-Prüfung im Build-Prozess
20 Unit Tests
20.1 Testen im Überblick
20.1.1 Was versteht man unter Testen?
20.1.2 Testarten im Überblick
20.1.3 Zuständigkeiten beim Testen
20.1.4 Testen und Qualität
20.2 Wissenswertes zu Testfällen
20.2.1 Test-Driven Development (TDD) im Überblick
20.2.2 Testfälle mit JUnit 4 definieren
20.2.3 Problem der Komplexität
20.3 Motivation für Unit Tests aus der Praxis
20.3.1 Unit Tests für Weiterentwicklungen
20.3.2 Unit Tests und Legacy-Code
20.4 Fortgeschrittene Unit-Test-Techniken
20.4.1 Test-Doubles
20.4.2 Testen mit Stubs
20.4.3 Testen mit Mocks
20.4.4 Unit Tests von privaten Methoden
20.5 Unit Tests mit Threads und Timing
20.6 Test Smells
20.7 JUnit Rules und parametrierte Tests
20.7.1 JUnit Rules im Überblick

20.7.2 Parametrierte Tests
20.8 Nützliche Tools für Unit Tests
20.8.1 Hamcrest
20.8.2 MoreUnit
20.8.3 Infinitest
20.8.4 Cobertura
20.8.5 EclEmma
20.9 Schlussgedanken
20.10 Weiterführende Literatur
21 Codereviews
21.1 Definition
21.2 Probleme und Tipps zur Durchführung
21.3 Vorteile von Codereviews
21.4 Codereview-Tools
21.5 Codereview-Checkliste
22 Optimierungen
22.1 Grundlagen
22.1.1 Optimierungsebenen und Einflussfaktoren
22.1.2 Optimierungstechniken
22.1.3 CPU-bound-Optimierungsebenen am Beispiel
22.1.4 Messungen – Erkennen kritischer Bereiche
22.1.5 Abschätzungen mit der O-Notation
22.2 Einsatz geeigneter Datenstrukturen
22.2.1 Einfluss von Arrays und Listen
22.2.2 Optimierungen für `Set` und `Map`
22.2.3 Design eines Zugriffsinterface
22.3 Lazy Initialization
22.3.1 Lazy Initialization am Beispiel
22.3.2 Konsequenzen des Einsatzes der Lazy Initialization
22.3.3 Lazy Initialization mithilfe des PROXY-Musters
22.4 Optimierungen am Beispiel

22.5 I/O-bound-Optimierungen
22.5.1 Technik – Wahl passender Strategien
22.5.2 Technik – Caching und Pooling
22.5.3 Technik – Vermeidung unnötiger Aktionen
22.6 Memory-bound-Optimierungen
22.6.1 Technik – Wahl passender Strategien
22.6.2 Technik – Caching und Pooling
22.6.3 Optimierungen der Stringverarbeitung
22.6.4 Technik – Vermeidung unnötiger Aktionen
22.7 CPU-bound-Optimierungen
22.7.1 Technik – Wahl passender Strategien
22.7.2 Technik – Caching und Pooling
22.7.3 Technik – Vermeidung unnötiger Aktionen
22.8 Weiterführende Literatur
VI Anhang
A Einführung in die UML
A.1 Die UML im Überblick
A.2 Strukturdiagramme – statische Modelle
A.2.1 Klassendiagramme
A.2.2 Objektdiagramme
A.2.3 Komponentendiagramme
A.2.4 Paketdiagramme
A.3 Verhaltensdiagramme – dynamische Modelle
A.3.1 Anwendungsfalldiagramme
A.3.2 Sequenzdiagramme
A.3.3 Kommunikationsdiagramme
A.3.4 Zustandsdiagramme
A.3.5 Aktivitätsdiagramme
A.4 Weiterführende Literatur
B Überblick über den Softwareentwicklungsprozess
B.1 Vorgehensmodelle

B.1.1 Aufgaben und Phasen beim Softwareentwurf
B.1.2 Wasserfallmodell und V-Modell
B.1.3 Extreme Programming (XP)
B.1.4 Test-Driven Development (TDD)
C Grundlagen zur Java Virtual Machine
C.1 Wissenswertes rund um die Java Virtual Machine
C.1.1 Ausführung eines Java-Programms
C.1.2 Sicherheit und Speicherverwaltung
C.1.3 Sicherheit und Classloading
Literaturverzeichnis
Index

Vorwort zur 3. Auflage

Zunächst einmal bedanke ich mich bei allen Lesern der beiden vorherigen Auflagen. Nur durch das große Interesse und den dadurch begründeten Zuspruch und Erfolg des Buchs wurde diese vollständig überarbeitete und erweiterte 3. Auflage möglich. Diese halten Sie jetzt in den Händen und ich freue mich, dass Sie sich für dieses Buch interessieren. Es soll Ihnen einen fundierten Einstieg in die professionelle Java-Programmierung ermöglichen und damit Ihren Weg zum Java-Profi erleichtern. Neben Hinweisen zur Sprache selbst gebe ich immer wieder auch Tipps aus dem Praxisalltag, weise auf Fallstricke hin und zeige Lösungswege auf. Aber was wäre dieses Buch ohne die vielfältigen Neuerungen aus Java 8? Diesem Thema ist ein ganzer Teil des Buchs gewidmet. Schauen wir kurz darauf, wie sich diese Auflage ansonsten von den vorherigen unterscheidet.

Änderungen in dieser 3. Auflage

Als Vorbereitung für diese 3. Auflage habe ich das Buch mehrfach von vorne bis hinten gelesen und viele Textpassagen kritisch beleuchtet. Dadurch konnten kleinere Unstimmigkeiten, missverständliche Passagen oder Tippfehler erkannt und korrigiert werden. Zum Teil haben mich bei deren Entdeckung einige Leser durch Hinweise per Mail unterstützt. Vielen Dank dafür! Explizit möchte ich hier Christian Rudolph erwähnen, der diverse Hinweise gegeben hat.

Die Impulse durch verschiedene Anregungen und Wünsche von Lesern sowie von Kollegen und Freunden habe ich mit meinen Ideen kombiniert. Daraus sind diverse Ergänzungen in den bereits vorhandenen Kapiteln entstanden, die ich nun aufliste:

- **Professionelle Arbeitsumgebung** – Dieses Kapitel wurde deutlich erweitert. Dabei wurde der Tatsache Rechnung getragen, dass dezentrale Versionsverwaltungen immer wichtiger werden. Auch Beschreibungen zum Deployment von Java-Anwendungen und insbesondere zum JAR-Format wurden ergänzt. Eine weitere größere Änderung besteht darin, dass nun anstelle von Ant das modernere Gradle als Build- Tool zum Einsatz kommt. Zudem gibt es einen Kurzeinstieg in Maven.
- **Objektorientiertes Design** – Hier wurden die Texte leicht neu gegliedert und an diversen Stellen überarbeitet, insbesondere die Abschnitte zu Basisklassen wurden umgestaltet. Hinzugekommen ist auch die Darstellung von Designaspekten, im Speziellen den SOLID-Prinzipien und dem Law of Demeter.
- **Java-Grundlagen** – Bei den Java-Grundlagen wurden drei wesentliche Anpassungen vorgenommen: Erstens wurden die Beschreibungen zur Verarbeitung von Zahlen mit primitiven Typen und Wrapper-Klassen erweitert.

Zweitens wird nun ein viel ausführlicher Blick auf die Behandlung von Fehlern geworfen. Und drittens wurden Abschnitte zu verschiedenen Neuerungen aus Java 7 integriert.

- **Das Collections-Framework** – Die Struktur des Textes wurde leicht modifiziert und die jeweiligen Unterkapitel beginnen nun häufig mit einem einleitenden Beispiel, wodurch die Verständlichkeit verbessert wurde. Auch wurden viele Beispiele überarbeitet, um für mehr Klarheit zu sorgen. Im Speziellen sind die Hinweise zur Implementierung der Methode `hashCode()` korrigiert und nachvollziehbarer gestaltet worden.
- **Applikationsbausteine** – Das Kapitel Applikationsbausteine wurde radikal geändert. Statt der Beschreibung eigener kleinerer Applikationsbausteine wird nun ein Blick auf Google Guava (und ein wenig Apache Commons) als hilfreiche Bibliotheken geworfen. Dadurch konnten einige Abschnitte vollständig entfallen und Beispiele eleganter formuliert werden.
- **Multithreading** – Der Text wurde insbesondere im Bereich der Producer-Consumer-Kommunikation zur Darstellung der Zusammenarbeit von Threads umgestaltet. Zudem wurde ein Abschnitt zum Fork-Join-Framework ergänzt.
- **Fortgeschrittene Java-Themen** – Es wurden diverse kleinere Korrekturen vorgenommen. Zudem wurden die Beschreibung zum Zugriff auf generische Typinformationen und Erläuterungen zum G1 Garbage Collector in das Kapitel integriert. Außerdem wurden die thematisch in sich abgeschlossenen Abschnitte zur Internationalisierung in ein eigenes Kapitel ausgelagert.
- **Programmierung grafischer Benutzeroberflächen mit Swing** – Das Kapitel zu grafischen Benutzeroberflächen wurde ausgedünnt und auf das reduziert, was man zum Verständnis von Swing-Bedienoberflächen an Wissen benötigt, um diese weiterhin unterstützen zu können. Neuentwicklungen werden wohl auf JavaFX setzen. Dieser neuen Technologie ist im Teil über Java 8 ein eigenes Kapitel gewidmet.
- **Refactorings** – Zum besseren Verständnis der Abläufe bei Refactorings wurden ein ausführliches Beispiel zur Überarbeitung und zur Kombination von Refactorings ergänzt sowie ein paar Verbesserungen in den gesamten Text integriert.
- **Programmierstil und Coding Conventions** – Die Beispiele zur Wahrnehmung von Sourcecode wurden überarbeitet, zudem fanden ein paar textuelle Korrekturen statt. Die größten Änderungen finden sich im Bereich Tooling. Dort wird u. a. die Einbindung in einen Gradle-Build gezeigt sowie die Tools JDepend und SonarQube kurz beschrieben.
- **Unit Tests** – Dieses Kapitel wurde deutlich überarbeitet und umgestaltet. Dabei sind meine Erfahrungen als Trainer der Zühlke Academy sowie die Anregungen durch Diskussionen mit Kollegen eingeflossen. In diesem Kapitel findet sich jetzt noch mehr Information rund ums Unit-Testen, etwa Hinweise zu Fallstricken

beim Testen und wie man Tests lesbar und verständlich gestaltet.

- **Optimierungen** – Neben ein paar leichten textuellen Verbesserungen wurden alle Performance-Messungen mit JDK 7 (und teilweise JDK 8) neu ausgeführt. Zudem wurde der Teil über CPU-bound-Optimierungen vollständig neu gestaltet.

Erweiterungen

Hinzugekommen ist ein ganzer Teil mit fünf Kapiteln zu Java 8:

- **Lambda-Ausdrücke** – Kapitel 11 startet mit Lambda-Ausdrücken, einer der bedeutsamsten Änderungen der Sprache seit der Einführung von Generics in Java 5. Lambdas können zu einer vollkommen neuen Denkweise führen und bilden die Grundlage für die funktionale Programmierung mit Java.
- **Bulk Operations on Collections** – Kapitel 12 zeigt, wie sich Lambdas gewinnbringend mit den diversen Erweiterungen im Collections-Framework, insbesondere dem Stream-API, kombinieren lassen. Dort wird unter anderem eine mächtige Filter-Map-Reduce-Funktionalität bereitgestellt – ähnlich wie man dies von NoSQL-Datenbanken zur Verarbeitung großer Datenmengen (Big Data) kennt.
- **JSR-310: Date And Time API** – Lange Zeit war die Verarbeitung von Datumsund Zeitangaben mit Java-Bordmitteln eher mühsam und zudem fehlerträchtig. Mit Java 8 ändert sich dies grundlegend. Das neue Date And Time API bereitet in seiner Nutzung viel Freude. Kapitel 13 gibt dazu einen Überblick.
- **Einstieg JavaFX 8** – Nicht nur intern, sondern auch im sichtbaren Bereich der Benutzeroberfläche wurde in Java 8 einiges verbessert. JavaFX ist nun Bestandteil des JDKs und der JRE. Neben Detailverbesserungen sind die Bedienelemente `DatePicker` und `TreeTableView` sowie die Unterstützung von 3D-Darstellungen bedeutende Erweiterungen. Kapitel 14 beginnt mit einem allgemeinen Einstieg in JavaFX und geht danach auf die Besonderheiten von JavaFX 8 ein.
- **Weitere Änderungen in JDK 8** – Neben den in den zuvor genannten Kapiteln behandelten recht fundamentalen Änderungen enthält Java 8 noch eine Vielzahl weiterer, zum Teil kleinerer Verbesserungen, die aber allesamt das Programmiererleben deutlich erleichtern. Einige wesentliche werden in Kapitel 15 vorgestellt.

Entfallene Themen

Aus drucktechnischen Gründen mussten die Kapitel des Anhangs (Einführung in die UML, Überblick über den Softwareentwicklungsprozess und Grundlagen zur Java

Virtual Machine) aus der Druckversion des Buchs herausgenommen werden. Diese stehen aber als PDF auf der Seite des Verlags zum Download bereit. Auch das Kapitel zu Datenbanken ist aus diesem Buch entfernt worden, um den Rahmen nicht zu sprengen. Es ist ein Ergänzungsband geplant, der auch die Thematik der immer aktueller werdenden NoSQL-Datenbanken aufgreifen wird. Der Erscheinungstermin war bei Drucklegung dieses Buchs noch nicht bekannt.

Danksagung

Ein Fachbuch zu schreiben ist eine schöne, aber arbeitsreiche und langwierige Aufgabe. Alleine kann man dies kaum bewältigen, daher möchte ich mich an dieser Stelle bei allen bedanken, die direkt oder indirekt dazu beigetragen haben.

Bei der Erstellung des Manuskripts konnte ich auf ein starkes Team an Korrekturlesern zurückgreifen, insbesondere diesmal auch auf Benjamin Muschko und Hans Dockter als Experten zu Gradle sowie Hendrik Schreiber, selbst Autor eines Java-Fachbuchs zu Optimierungen. Vielen Dank an euch!

Einige Tipps erhielt ich von Tim Bötzmeyer und Reinhard Pupkes. Auch haben mich folgende Personen hervorragend unterstützt: Merten Driemeyer, Dr. Clemens Gugenberger, Dr. Carsten Kern, Florian Messerschmidt und Andreas Schöneck. Darüber hinaus kamen gute Anmerkungen von verschiedenen Zühlke-Kollegen: Michael Haspra, Jörg Keller, Rick Janda, Franziska Meyer, Sagi Nedunkanal, Joachim Prinzbach und Dr. Christoph Schmitz. Der Java-8-Teil entstammt weitestgehend meinem Buch »Java 8 – Die Neuerungen«. Allen dort Beteiligten danke ich ebenfalls.

Neben den Korrekturlesern möchte ich einen ganz herzlichen Dank an meinem Arbeitgeber Zühlke Engineering AG und insbesondere meinen Chef Wolfgang Giersche für die gewährte freie Zeit zum Finalisieren des Buchs aussprechen. Das war eine große Hilfe in der letzten heißen Phase vor der Abgabe des Manuskripts.

Wie immer geht natürlich auch ein Dankeschön an das Team des dpunkt.verlags (vor allem Dr. Michael Barabas, Martin Wohlrab, Vanessa Wittmer, Miriam Metsch und Birgit Bäuerlein) für die gute Zusammenarbeit. Außerdem möchte ich mich bei Torsten Horn für die fundierte fachliche Durchsicht sowie bei Ursula Zimpfer für ihre Adleraugen beim Copy-Editing bedanken.

Abschließend geht natürlich ein lieber Dank an meine Frau Lilija für ihr Verständnis und ihre Unterstützung. Bei der Erstellung dieser dritten Auflage war ich glücklicherweise weit weniger im Stress als bei Wurf eins und zwei.

Anregungen und Kritik

Ich wünsche allen Lesern viel Freude und einige neue Erkenntnisse durch die Lektüre dieser 3. Auflage. Möge Ihnen der »Weg zum Java-Profi« mit meinem Buch ein wenig leichter fallen.

Trotz großer Sorgfalt lassen sich leider bei einem so umfangreichen Buch Fehler nicht vollständig vermeiden. Falls Ihnen ein solcher auffällt oder eine Formulierung missverständlich sein sollte, so zögern Sie nicht, mir dies mitzuteilen. Haben Sie Anregungen, Verbesserungsvorschläge oder fehlt Ihnen noch eine Information? Sie erreichen mich per Mail unter: `michael_inden@hotmail.com`

Zürich und Aachen, im Dezember 2014

Michael Inden

Danksagung zur 2. Auflage

Bei der Erstellung der 2. Auflage konnte ich wieder auf ein starkes Team an Korrekturlesern zurückgreifen. Einige Tipps erhielt ich von Dr. Alexander Kort und Reinhard Pupkes. Auch haben mich folgende Personen hervorragend unterstützt: Stefan Bartels, Tim Bötzmeyer, Rudolf Braun, Andreas Bubolz, Merten Driemeyer, Bernd Eckstein, Dr. Clemens Gugenberger, Peter Kehren, Dr. Carsten Kern, Dr. Iris Rottländer, Roland Schmitt-Hartmann und Andreas Schöneck.

Dabei möchte ich folgende vier Personen herausheben: Stefan Bartels für seine sprachliche Gründlichkeit, Andreas Bubolz für seine Genauigkeit und Dr. Clemens Gugenberger sowie Andreas Schöneck für die ganzen hilfreichen Anregungen.

Danksagung zur 1. Auflage

Zu meiner Zeit bei der Heidelberger Druckmaschinen AG in Kiel ist bei den Vorbereitungen zu Codereviews und der Ausarbeitung von Vorträgen zum ersten Mal der Gedanke an ein solches Buch entstanden. Danke an meine damaligen Kollegen, die an diesen Meetings teilgenommen haben. Als Veranstalter und Vortragender lernt man immer wieder neue Details. Dietrich Mucha und Reinhard Pupkes danke ich für ihre Korrekturen und Anmerkungen, die gemeinsamen Erfahrungen beim Ausarbeiten von Coding Conventions und Codereviews sowie die nette Zeit beim Pair Programming. Die Zusammenarbeit mit Tim Bötzmeyer hat mir viel Freude bereitet. Unsere langen, interessanten Diskussionen über Java und die Fallstricke beim OO-Design haben mir diverse neue Einblicke verschafft.

Auch einige Kollegen bei der IVU Traffic Technologies AG in Aachen haben mich mit Korrekturen und Anregungen unterstützt. Unter anderem danke ich Rudolf Braun, Christian Gehrmann, Peter Kehren, Felix Korb und Roland Schmitt-Hartmann für den einen oder anderen Hinweis und Tipp, um den Text weiter zu verbessern. Mein spezieller Dank gilt Merten Driemeyer, der sich sehr gründlich mit frühen Entwürfen des Manuskripts beschäftigt und mir an diversen Stellen fachliche und sprachliche Tipps gegeben hat. Gleiches gilt für Dr. Iris Rottländer, die sowohl formal als auch inhaltlich an vielen Stellen durch ihre Anmerkungen für eine Verbesserung des Textes gesorgt hat. Auch Dr. Carsten Kern und Andreas Schöneck

haben gute Hinweise gegeben und einige verbliebene kleinere Fehler aufgedeckt. Last, but not least haben die Anmerkungen von Dr. Clemens Gugenberger und unsere nachfolgenden Diskussionen einigen Kapiteln den letzten Feinschliff gegeben. Gleiches gilt für Stefan Bartels, der mich immer wieder durch gute Anmerkungen unterstützt und damit zur Verständlichkeit des Textes beigetragen hat. Alle sechs haben mir entscheidend geholfen, inhaltlich für mehr Stringenz und Klarheit zu sorgen. Mein größter Dank geht an Andreas Bubolz, der mich immer wieder unterstützt und enorm viel Zeit und Mühe investiert hat. Als Korrekturleser und Sparringspartner in vielen Diskussionen hat er diverse Unstimmigkeiten im entstehenden Text aufgedeckt.

1 Einleitung

Bevor es mit den Programmierthemen losgeht, möchte ich Ihnen dieses Buch vorstellen. Ich beginne damit, warum dieses Buch entstanden ist und wie es Ihnen hoffentlich helfen kann, ein noch besserer Java-Entwickler zu werden. Danach folgt eine Gliederung des Inhalts, damit Sie sich gut im Buch zurechtfinden.

1.1 Über dieses Buch

1.1.1 Motivation

Mein Ziel war es, ein Buch zu schreiben, wie ich es mir selbst immer als Hilfe gewünscht habe. Die hier vorgestellten Hinweise und Techniken sollen Sie auf Ihrem Weg vom engagierten Hobbyprogrammierer oder Berufseinsteiger zum erfahrenen Softwareentwickler begleiten. Dieser Weg ist ohne Anleitung gewöhnlich steinig und mit einige Mühen, Irrwegen und Problemen verbunden. Einige dieser leidvollen Erfahrungen möchte ich Ihnen ersparen. Aber auch erfahreneren Softwareentwicklern soll dieses Buch eine Möglichkeit geben, über die im täglichen Einsatz lieb gewonnenen Gewohnheiten nachzudenken und die eine oder andere davon zu ändern, um die Produktivität weiter zu steigern. Mein Wunsch ist, dass sich nach Lektüre des Buchs für Sie die Ingenieurdisziplin der Softwareentwicklung mit der Kunst des Programmierens verbindet und Dinge auf einmal einfach so auf Anhieb funktionieren. Das ist etwas ganz anderes, als vor jedem Gang in die Testabteilung Magenschmerzen zu bekommen.

Sowohl der Berufseinstieg als auch die tägliche Arbeit können manchmal frustrierend sein. Meiner Meinung nach soll Softwareentwicklung aber Spaß und Freude bereiten, denn nur so können wir exzellente Resultate erzielen. In der Praxis besteht jedoch die Gefahr, in die Fettnäpfchen zu treten, die im Sourcecode hinterlassen wurden. Dies geschieht meistens dadurch, dass die existierende Lösung softwaretechnisch umständlich oder schlecht implementiert ist und/oder nicht bis zu Ende durchdacht wurde. Der Sourcecode ist dann häufig schwierig wart- und erweiterbar. Manchmal bereitet bereits das Auffinden der Stelle, an der man Modifikationen durchführen sollte, Probleme.

Dieses Buch soll aufzeigen, wie man die zuvor beschriebene »Altlasten«-Falle vermeidet oder aus ihr herauskommt und endlich Sourcecode schreiben kann und darf, der leicht zu lesen ist und in dem es Spaß macht, Erweiterungen zu realisieren. Grundlage dafür ist, dass wir uns einen Grundstock an Verhaltensweisen und an Wissen aneignen.

1.1.2 Was leistet dieses Buch und was nicht?

Wieso noch ein Buch über Java-Programmierung? Tatsächlich kann man sich diese Frage stellen, wo es doch unzählige Bücher zu diesem Thema gibt. Bei vielen davon handelt es sich um einführende Bücher, die häufig lediglich kurz die APIs anhand simpler Beispiele vorstellen. Die andere große Masse der Java-Literatur beschäftigt sich mit speziellen Themen, die für den »erfahrenen Einsteiger« bereits zu komplex geschrieben und in denen zu wenig erklärt ist. Genau hier setzt dieses Buch an und wagt den Spagat, den Leser nach der Lektüre einführender Bücher abzuholen und so weit zu begleiten, dass er mit einem guten Verständnis die Spezialliteratur lesen und gewinnbringend einsetzen kann.

Ziel dieses Buchs ist es, dem Leser fundierte Kenntnisse in Java und einigen praxisrelevanten Themenbereichen, unter anderem dem Collections-Framework und im Bereich Multithreading, zu vermitteln. Es werden vertiefende Blicke auf die zugrunde liegenden Details geworfen, um nach Lektüre des Buchs professionelle Programme schreiben zu können. Wie bereits angedeutet, bietet dieses Buch keinen Einstieg in die Sprache selbst, sondern es wird bereits einiges an Wissen oder Praxiserfahrung vorausgesetzt. In zwei einleitenden Grundlagenkapiteln über objektorientiertes Design und Java wird allerdings die Basis für das Verständnis der Folgekapitel geschaffen.

In diesem Buch versuche ich, einen lockeren Schreibstil zu verwenden und nur an den Stellen formal zu werden, wo dies wirklich wichtig ist, etwa bei der Einhaltung von Methodenkontrakten. Da der Fokus dieses Buchs auf dem praktischen Nutzen und dem guten Verständnis von Konzepten liegt, werden neben APIs auch häufig vereinfachte Beispiele aus der realen Welt vorgestellt. Die meisten der abgebildeten Listings stehen als kompilierbare und lauffähige Programme auf der Webseite zum Buch zum Download bereit. Im Buch selbst werden aus Platzgründen und zugunsten einer besseren Übersichtlichkeit in der Regel nur die wichtigen Passagen abgedruckt.

1.1.3 Wie und was soll mithilfe des Buchs gelernt werden?

Dieses Buch zeigt und erklärt einige in der Praxis bewährte Ansätze, Vorgehens- und Verhaltensweisen, ohne dabei alle Themengebiete bis in kleinste Detail auszuleuchten. Wichtiges Hintergrundwissen wird jedoch bei Bedarf vermittelt. Es wird der pragmatische Weg gegangen und bevorzugt die in der täglichen Praxis relevanten Themen vorgestellt. Sollte ein Thema bei Ihnen besonderes Interesse wecken und Sie weitere Informationen wünschen, so finden sich in den meisten Kapiteln Hinweise auf weiterführende Literatur. Dies ist im Prinzip auch schon der erste Tipp: ***Lies viele Bücher und schaffe damit eine breite Wissensbasis.*** Ich zitiere hier aus Jon Bentleys Buch »Perlen der Programmierkunst« [6]: ***»Im Stadium des Entwurfsprozesses ist es unschätzbar, die einschlägige Literatur zu kennen.«***

Diesem Hinweis kann ich mich nur anschließen und möchte Ihnen hier speziell einige – meiner Meinung nach – ganz besondere Bücher ans Herz legen und empfehle ausdrücklich, diese Bücher begleitend oder ergänzend zu diesem Buch zu lesen:

- »**SCJP – Sun Certified Programmer & Developer for Java 2**« [82] – Die Vorbereitung zur Zertifizierung als Java-Programmierer mit all seinen Fallstricken und kniffligen Details wird auf unterhaltsame Weise von Kathy Sierra und Bert Bates aufbereitet.
- »**The Java Programming Language**« [2] – Ein unglaublich gutes Buch von Ken Arnold, James Gosling und David Holmes über die Sprache Java, das detailreich, präzise und dabei angenehm verständlich zu lesen ist.
- »**Effective Java**« [8] und [9] – Dieses Buch von Joshua Bloch habe ich auf der Java One 2001 in San Francisco gekauft und es hat mein Denken und Programmieren in Java stark beeinflusst. Mittlerweile existiert eine zweite Auflage, die auf JDK 6 aktualisiert wurde.
- »**Entwurfsmuster**« [30] – Das Standardwerk der sogenannten »Gang of Four« (Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides) habe ich 1998 kennengelernt und mit großem Interesse gelesen und gewinnbringend eingesetzt. Die vorgestellten Ideen sind beim Entwurf guter Software enorm hilfreich.
- »**Head First Design Patterns**« [28] – Dieses Buch einer anderen »Gang of Four« (Eric Freeman, Elizabeth Freeman, Kathy Sierra und Bert Bates) lässt Entwurfsmuster als ein unterhaltsames Thema erscheinen und erleichtert den Einstieg.
- »**Refactoring**« [27] – Einige Tricks und Kniffe zur Verbesserung von Sourcecode lernt man von Kollegen oder durch Erfahrung. Martin Fowler fasst dieses Wissen in dem genannten Buch zusammen und stellt ein systematisches Vorgehen zur Sourcecode-Transformation vor.
- »**Refactoring to Patterns**« [54] – Dieses Buch von Joshua Kerievsky verknüpft das Standardwerk zu Refactorings von Martin Fowler mit den Ideen der Entwurfsmuster.
- »**Code Craft: The Practice of Writing Excellent Code**« [33] – Ein sehr lesenswertes Buch von Pete Goodlife, das diverse gute Hinweise gibt, wie man exzellenten Sourcecode schreiben kann, der zudem (nahezu) fehlerfrei, gut testbar sowie einfach zu warten ist.

Lesen hilft uns bereits, aber nur durch Übung und Einsatz in der Praxis können wir unsere Fähigkeiten verbessern. Weil ein Buch jedoch nicht interaktiv ist, werde ich bevorzugt eine schrittweise Vorstellung der jeweiligen Themen vornehmen, wobei zum Teil auch bewusst zunächst ein Irrweg gezeigt wird. Anhand der vorgestellten Korrekturen erkennt man dann die Vorteile viel deutlicher, als wenn nur eine reine

Präsentation der Lösung erfolgen würde. Mit dieser Darstellungsweise hoffe ich, dass Sie sich ein paar gute Gewohnheiten antrainieren. Das fängt mit scheinbar einfachen Dingen wie der Vergabe von sinnvollen Namen für Variablen, Methoden und Klassen an und endet in der Verwendung von problemangepassten Entwurfsmustern. Anfangs erfordert dies erfahrungsgemäß ein wenig Fleiß, Einarbeitung, Disziplin und eventuell sogar etwas Überwindung. Daher werde ich bei der Vorstellung einer Technik jeweils sowohl auf die Vorteile als auch die Nachteile (wenn vorhanden) eingehen.

1.1.4 Wer sollte dieses Buch lesen?

Dieses Buch konzentriert sich auf Java als Programmiersprache – allerdings benötigen Sie bereits einige Erfahrung mit Java, um die Beispiele sowie die beschriebenen Tücken nachvollziehen zu können und möglichst viel von den Tipps und Tricks in diesem Buch zu profitieren. Wenn Sie dieses Buch in den Händen halten, gehe ich davon aus, dass Sie sich bereits (etwas) mit Java auseinandergesetzt haben.

Das Buch richtet sich im Speziellen an zwei Zielgruppen: Zum einen sind dies engagierte Hobbyprogrammierer, Informatikstudenten oder Berufseinsteiger, die von Anfang an lernen wollen, wie man professionell Software schreibt. Zum anderen sind dies erfahrenere Softwareentwickler, die ihr Wissen in einigen fortgeschritteneren Themen komplettieren wollen und vermehrt Priorität auf sauberes Design legen oder Coding Conventions, Codereviews und Unit-Testen bei der Arbeit etablieren wollen.

Abhängig vom Kenntnisstand zu Beginn der Lektüre starten Entwickler mit Erfahrung bei Teil II oder Teil III des Buchs und können die dort vorgestellten Techniken sofort gewinnbringend in der Praxis einsetzen. Lesern mit noch relativ wenig Erfahrung empfehle ich, den ersten Teil konzentriert und vollständig durchzuarbeiten, um sich eine gute Basis zu verschaffen. Dadurch wird das Verständnis der später vorgestellten Themen erleichtert, denn die nachfolgenden Teile des Buchs setzen die Kenntnis dieser Basis voraus und das Niveau nimmt ständig zu. Ziel ist es, nach Lektüre des Buchs den Einstieg in die professionelle Softwareentwicklung mit Java erreicht zu haben und viele dazu erforderliche Techniken sicher zu beherrschen. Das Buch ist daher mit diversen Praxistipps gespickt, mit denen Sie auf interessante Hintergrundinformationen oder auf mögliche Probleme hingewiesen werden und die wie folgt in den Text integriert sind:

Tipp: Praxistipp

In derart formatierten Kästen finden sich im späteren Verlauf des Buchs immer wieder einige wissenswerte Tipps und ergänzende Hinweise zum eigentlichen Text.

1.2 Aufbau des Buchs

Der Aufbau des Buchs gliedert sich in mehrere Teile. Folgende Aufzählung konkretisiert die dort vorgestellten Themen:

- **Teil I** **»Java-Grundlagen, Analyse und Design«** – Dieser Teil legt die Grundlagen für einen guten Softwareentwurf, in dem sowohl auf objektorientiertes Design als auch auf eine produktive Arbeitsumgebung mit den richtigen Hilfsmitteln eingegangen wird. Teil I beginnt in Kapitel 2 mit der Vorstellung einer sinnvoll ausgestatteten Arbeitsumgebung, die beim Entwickeln von professionellen Programmen hilft und es ermöglicht, gute Resultate produzieren zu können. Anschließend wird in Kapitel 3 das Thema objektorientiertes Design beschrieben. Als Notation wird dabei die UML verwendet. Damit sind die Grundlagen für einen professionellen, objektorientierten Softwareentwurf gelegt und die Vorbereitungen zum Implementieren getroffen. Kapitel 4 stellt dann noch einige wichtige Java-Sprachelemente vor, die zum Verständnis der Beispiele in den folgenden Kapiteln notwendig sind.
- **Teil II** **»Bausteine stabiler Java-Applikationen«** – Der zweite Teil beschäftigt sich mit Komponenten und Bausteinen stabiler Java-Applikationen. Es werden wichtige Kenntnisse fundamentaler Java-APIs vermittelt, aber auch fortgeschrittene Java-Techniken behandelt. Zunächst stellt Kapitel 5 das Thema Collections vor, um eine effiziente Wahl von Datenstrukturen zur Speicherung und Verwaltung von Daten zu ermöglichen. In Kapitel 6 beschäftigen wir uns mit der Erstellung wiederverwendbarer Softwarebausteine. Es ist wichtig, das Rad nicht immer neu zu erfinden, sondern auf einer stabilen Basis aufzubauen. Ein weiterer wichtiger Baustein beim professionellen Programmieren ist Multithreading. Kapitel 7 gibt eine fundierte Einführung und stellt auch fortgeschrittenere Themen, wie das Java-Memory-Modell und die Parallelverarbeitung mit Thread-Pools, vor. Weitere fortgeschrittenere Themen, etwa Garbage Collection, werden in Kapitel 8 behandelt. Damit besitzen Sie dann schon eine gute Wissensbasis, aber was wären die meisten Programme ohne eine grafische Benutzeroberfläche? Weil das so wichtig ist, geht Kapitel 9 auf dieses Thema detailliert ein. Das Thema Internationalisierung und damit die Besonderheiten, die bei der Unterstützung verschiedener Länder und Sprachen zu beachten sind, werden in Kapitel 10 thematisiert.
- **Teil III** **»Die Neuerungen in Java 8«** – In diesem Teil gebe ich in verschiedenen Kapiteln einen Überblick zu Java 8 und seine wegweisenden Neuerungen, etwa Lambda-Ausdrücke und das Stream-API. Kapitel 11 startet mit einer Vorstellung von Lambdas, die zu einer vollkommen neuen Denkweise führen und die Grundlage für die funktionale Programmierung mit Java bilden. Damit stellen Lambdas eine der bedeutsamsten Änderungen der Sprache seit der Einführung von Generics in Java 5 dar. Kapitel 12 zeigt dann, wie sich Lambdas gewinnbringend mit den diversen Erweiterungen im Collections-Framework, insbesondere dem Stream-API, kombinieren lassen. Auch die lange Zeit, in der die Datumsarithmetik eher mühsam und zudem fehlerträchtig war, ist mit Java 8

vorbei. Das neue Date And Time API erleichtert die Arbeit deutlich. Kapitel 13 gibt dazu einen Überblick. Nicht nur intern, sondern auch aufseiten der Benutzeroberflächen wurde in Java 8 einiges verbessert. JavaFX ist nun fester Bestandteil des JDKs und schickt sich an, Swing bald als GUI-Framework abzulösen. Kapitel 14 gibt einen Einstieg und beschreibt neben neuen Bedienelementen auch die Unterstützung für Darstellungen in 3D. Neben einigen recht fundamentalen Änderungen enthält Java 8 auch noch eine Vielzahl weiterer, zum Teil kleinerer Verbesserungen, die allesamt das Programmiererleben erleichtern. Einige wesentliche werden in Kapitel 15 vorgestellt.

Nach der Lektüre dieser ersten drei Teile sind Sie programmiertechnisch fit und bereit für das Schreiben eigener Anwendungen mit komplexeren Aufgabenstellungen. Auf dem Weg zu guter Software werden wir über das eine oder andere Problem stolpern. Wie Sie mögliche Probleme erkennen und beheben, ist Thema der folgenden Teile.

- **Teil IV »Fallstricke und Lösungen im Praxisalltag«** – Der vierte Teil beschreibt anhand von Beispielen mögliche Probleme aus dem Praxisalltag und Lösungen. Auf diese Weise wird ein tieferes Verständnis für einen guten Softwareentwurf erlangt. Teil IV betrachtet in Kapitel 16 zunächst ausführlich mögliche Programmierprobleme, sogenannte »Bad Smells«. Diese werden analysiert und Lösungsmöglichkeiten dazu aufgezeigt. Diverse Umbaumaßnahmen werden in Kapitel 17 als Refactorings vorgestellt. Kapitel 18 rundet diesen Teil mit der Präsentation einiger für den Softwareentwurf wichtiger Lösungsideen, sogenannter Entwurfsmuster, ab. Diese sorgen zum einen dafür, ein Problem auf eine dokumentierte Art zu lösen, und zum anderen, Missverständnisse zu vermeiden, da die Entwickler eine eigene, gemeinsame Designsprache sprechen.
- **Teil V »Qualitätssicherungsmaßnahmen«** – Qualitätssicherung ist für gute Software elementar wichtig und wird in Teil V vorgestellt. In den Bad Smells gewonnene Erkenntnisse werden in Kapitel 19 zu einem Regelwerk beim Programmieren, sogenannten Coding Conventions, zusammengefasst. Um neue Funktionalität und Programmänderungen abzusichern, schreiben wir Unit Tests. Nur durch eine breite Basis an Testfällen haben wir die Sicherheit, Änderungen ohne Nebenwirkungen auszuführen. Kapitel 20 geht detailliert darauf ein. Eine Qualitätskontrolle über Codereviews wird in Kapitel 21 thematisiert. Zu einer guten Qualität gehören aber auch nicht funktionale Anforderungen. Diese betreffen unter anderem die Performance eines Programms. In Kapitel 22 stelle ich daher einige Techniken zur Performance-Steigerung vor.

1.3 Konventionen und ausführbare Programme

Verwendete Zeichensätze

Im gesamten Text gelten folgende Konventionen bezüglich der Schriftart: Der normale Text erscheint in der vorliegenden Schriftart. Dabei werden wichtige Textpassagen *kursiv* oder ***kursiv und fett*** markiert. Englische Fachbegriffe werden eingedeutscht groß geschrieben. Zusammensetzungen aus englischen und deutschen (oder eingedeutschten) Begriffen werden mit Bindestrich verbunden, z. B. Plugin-Manager. Namen von Refactorings, Bad Smells, Idiomen sowie Entwurfsmustern u. Ä. werden bei ihrer Verwendung in KAPITÄLCHEN dargestellt. Sourcecode-Listings sind in der Schrift `courier` gesetzt, um zu verdeutlichen, dass dieser Text einen Ausschnitt aus einem realen Java-Programm wiedergibt. Auch im normalen Text werden Klassen, Methoden, Konstanten und Übergabeparameter in dieser Schriftart dargestellt.

Verwendete Abkürzungen

Im Buch verwende ich die in Tabelle 1-1 aufgelisteten Abkürzungen. Weitere Abkürzungen werden im laufenden Text in Klammern nach ihrer ersten Definition aufgeführt und anschließend bei Bedarf genutzt.

Tabelle 1-1 *Verwendete Abkürzungen*

Abkürzung	Bedeutung
JDK	Java Development Kit
JLS	Java Language Specification
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java Virtual Machine
OO	Objektorientierung
TDD	Test-Driven Development
UML	Unified Modeling Language
XP	Extreme Programming
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
GUI/UI	(Graphical) User Interface
IDE	Integrated Development Environment
XML	Extensible Markup Language

Verwendete Java-Umgebungen und -Versionen

Wir werden uns in diesem Buch vorwiegend mit Beispielen aus dem Bereich der Java Standard Edition (Java SE) auseinandersetzen. Wo dies sinnvoll ist, werde ich auf einige Ideen und Themen aus dem Bereich Java Enterprise Edition (Java EE oder JEE) eingehen. Diese ist eine Erweiterung der Standard Edition um die Möglichkeit, Client-Server-Systeme und größere, verteilte Applikationen zu entwickeln.

Diese Editionen existieren in verschiedenen Versionen. Aktuell ist derzeit die Version 8 der Java Standard Edition, die im März 2014 veröffentlicht wurde. Weil ein Großteil der kommerziellen Java-Projekte noch den Vorgänger JDK 7 (oder zum

Teil noch JDK 6) nutzt, bildet JDK 7 die Grundlage für die Beispiele im Buch – natürlich abgesehen von den speziellen Kapiteln zu Java 8 und seinen Neuerungen. Sofern ansonsten schon JDK-8-Funktionalitäten verwendet wird, ist das jeweils entsprechend im Sourcecode kenntlich gemacht.

Verwendete Klassen aus dem JDK

Werden Klassen des JDKs zum ersten Mal im Text erwähnt, so wird deren voll qualifizierter Name, d. h. inklusive der Package-Struktur, angegeben: Für die Klasse `String` würde dann etwa `java.lang.String` notiert. Dies erleichtert eine Orientierung und ein Auffinden im JDK. Dies gilt insbesondere, da in den Listings nur selten `import`-Anweisungen abgebildet werden. Im nachfolgenden Text wird zur besseren Lesbarkeit auf diese Angabe verzichtet und nur der Klassenname genannt.

Im Text beschriebene Methodenaufrufe enthalten in der Regel die Typen der Übergabeparameter, etwa `substring(int, int)`. Sind die Parameter in einem Kontext nicht entscheidend, wird auf deren Angabe aus Gründen der besseren Lesbarkeit verzichtet oder aber durch die Zeichenfolge ... abgekürzt.

Download, Sourcecode und ausführbare Programme

Um den Rahmen des Buchs nicht zu sprengen, stellen die abgebildeten Programmlistings häufig nur Ausschnitte aus lauffähigen Programmen dar. Deren Formatierung weicht leicht von den Coding Conventions von Oracle[1] ab. Ich orientiere mich an denjenigen von Scott Ambler[2], der insbesondere (öffnende) Klammern in jeweils eigenen Zeilen vorschlägt. Weitere Informationen finden Sie in Kapitel 19. Ein Beispiel ist nachfolgend gezeigt:

```
public final class FormatingExample
{
    private static final Logger log = Logger.getLogger("FormatingExample");

    public static String asHex(final byte[] tele)
    {
        log.info("asHex(" + Arrays.toString(tele) + ")");

        final StringBuffer sb = new StringBuffer("0x");

        for (int i = 0; i < tele.length; i++)
        {
            final String hex = Integer.toHexString(tele[i]);
            sb.append(hex);
        }

        return sb.toString();
    }
    // …
}
```

Der Sourcecode der Beispiele kann auf der Webseite `www.dpunkt.de/java-`

`profi` heruntergeladen werden. Dort findet sich auch ein Eclipse-Projekt, zum Import. Weil dies ein Buch zum Mitmachen ist, sind viele der Programme mithilfe von Gradle-Tasks (die wir in Kapitel 2 kennenlernen) ausführbar. Deren Name wird in Kapitälchenschrift, etwa LOCALEEXAMPLE, angegeben.

Teil I

Java-Grundlagen, Analyse und Design

2 Professionelle Arbeitsumgebung

Das Bearbeiten von Sourcecode ist eine wichtige und elementare Aufgabe bei der Softwareentwicklung. Genau wie ein Handwerker braucht auch ein Softwareentwickler eine gut strukturierte Werkbank (Arbeits-/Entwicklungsumgebung) mit guten Werkzeugen (Tools). In diesem Kapitel werden wir damit beginnen, eine Arbeitsumgebung einzurichten, die sowohl die Arbeit erleichtert als auch Unterstützung beim Testen bietet und dadurch hilft, Fehler zu vermeiden.

Den zentralen Anlaufpunkt der Entwicklung, unsere Steuerzentrale, bildet eine sogenannte integrierte Entwicklungsumgebung, kurz IDE. Ich motiviere in Abschnitt 2.1 den Einsatz einer solchen und nenne drei mögliche Kandidaten, wobei in diesem Buch die Wahl auf die frei verfügbare IDE Eclipse fällt. In Abschnitt 2.2 betrachten wir zwei Vorschläge zur Organisation eines Softwareprojekts, also zur Strukturierung von Sourcecode und anderen Dateien (Bilder, Texte, Testcode usw.). Abschnitt 2.3 beschäftigt sich dann mit dem Thema Versionsverwaltungen. Diese helfen dabei, die Dateien eines Projekts sicher zu speichern und verschiedene Versionen davon abrufen zu können. Dazu werde ich kurz auf zentrale und dezentrale Versionsverwaltungen am Beispiel der Open-Source-Tools CVS, Subversion (SVN) sowie Git und Mercurial eingehen. Die professionelle Softwareentwicklung umfasst neben der Entwicklung und der späteren Auslieferung natürlich zuvor auch das Testen unserer Programme. In Abschnitt 2.4 werfen wir daher einen einführenden Blick auf die Erstellung von Unit Tests mithilfe von JUnit. Selbst wenn wir viele Unit Tests erstellen und unsere Programme sehr gewissenhaft testen, so wird es doch immer mal wieder zu unerklärlichem Programmverhalten oder gar Fehlern kommen. Für solche Fälle ist es zur Fehlersuche sehr wünschenswert und hilfreich, das Programm schrittweise ausführen zu können, es bei Bedarf an einer bestimmten Stelle zu unterbrechen und dann die Wertebelegungen von Attributen überprüfen zu können. Das Ganze ist mithilfe eines sogenannten Debuggers möglich. Eine Einführung in die Thematik bietet Abschnitt 2.5. Bestimmte lauffähige Stände wollen wir auch an Kunden veröffentlichen. Abschnitt 2.6 gibt einen Überblick über die Auslieferung (Packaging und Deployment) von Java-Programmen. Dazu lernen wir JAR-Dateien (Java Archive) kennen. Abschließend stelle ich die Vorteile eines von der IDE unabhängigen Build-Prozesses (Programme kompilieren, testen, starten, Auslieferungen erzeugen usw.) heraus. In Abschnitt 2.7 lernen wir dazu die Build-Tools Ant, Maven und insbesondere Gradle kennen. Letzteres wird zur Ausführung aller im Buch vorgestellten Beispielapplikationen genutzt.

2.1 Vorteile von IDEs am Beispiel von Eclipse

Zum Bearbeiten von Sourcecode empfehle ich den Einsatz einer IDE anstelle von Texteditoren. Zwar kann man für kleinere Änderungen auch mal einen Texteditor

nutzen, aber dieser bietet nicht die Annehmlichkeiten einer IDE: Dort finden Kompiliervorgänge automatisch und im Hintergrund statt, wodurch gewisse Softwaredefekte direkt noch während des Editierens erkannt und in einer To-do-/Task-Liste angezeigt werden können. IDEs analysieren zudem den Sourcecode und bereiten vielfältige Informationen auf. Das erlaubt unter anderem die Anzeige von Ableitungshierarchien und das Auffinden von Klassen über deren Namen. Auch das Verknüpfen der JDK-Klassen mit deren Sourcecode und das Anzeigen zugehöriger Dokumentation sind Vorteile von IDEs. Weiterhin werden automatische Transformationen und Änderungen von Sourcecode, sogenannte ***Refactorings***, unterstützt.

Für Java bieten sich verschiedene IDEs an. Sowohl Eclipse als auch NetBeans sind kostenlos. IntelliJ IDEA gibt es als kostenlose Community Edition sowie als kostenpflichtige Ultimate Edition. Alle IDEs haben ihre speziellen Vorzüge. Entscheiden Sie selbst und besuchen Sie dazu folgende Internetadressen:

- `http://www.eclipse.org/`
- `http://www.jetbrains.com/`
- `http://www.netbeans.org/`

Die genannten IDEs lassen sich durch die Integration von Tools, etwa Sourcecode-Checkern, Versionsverwaltungen, XML-Editoren, Datenbank-Tools, Profiling-Tools usw., erweitern. Im folgenden Text werde ich speziell auf Eclipse und entsprechende Tools eingehen. Nach der Lektüre dieses Kapitels haben Sie dann bereits eine Arbeitsumgebung, die professionelles Arbeiten erlaubt. Im Verlauf des Buchs werden thematisch passende, arbeitserleichternde Erweiterungen vorgestellt.

Basiskonfiguration für Eclipse

Zum Vermeiden von Fehlern und zur Sicherstellung einer guten Qualität empfiehlt es sich, den Sourcecode regelmäßig zu analysieren und dabei die Einhaltung gewisser Regeln und Standards zu beachten. In einem ersten Schritt kann man dazu auf die in Eclipse integrierte Sourcecode-Prüfung zurückgreifen. Umfangreichere Tests bieten Tools wie Checkstyle, FindBugs und PMD (vgl. Abschnitt 19.4).

Die in Eclipse integrierten Sourcecode-Prüfungen können wir im Einstellungsdialog WINDOW –> PREFERENCES konfigurieren. Dort wählen wir im Baum den Eintrag JAVA –> COMPILER –> ERRORS/WARNINGS. In dem zugehörigen Dialog nehmen wir Anpassungen in den Bereichen CODE STYLE, POTENTIAL PROGRAMMING PROBLEMS, UNNECESSARY CODE sowie NULL ANALYSIS vor. Abbildung 2-1 zeigt eine mögliche, sinnvolle Einstellung der Werte im Abschnitt CODE STYLE. Auch in den anderen Sektionen können Sie bei Interesse strengere Auswertungen wählen. Experimentieren Sie ruhig ein wenig mit den Einstellungen.

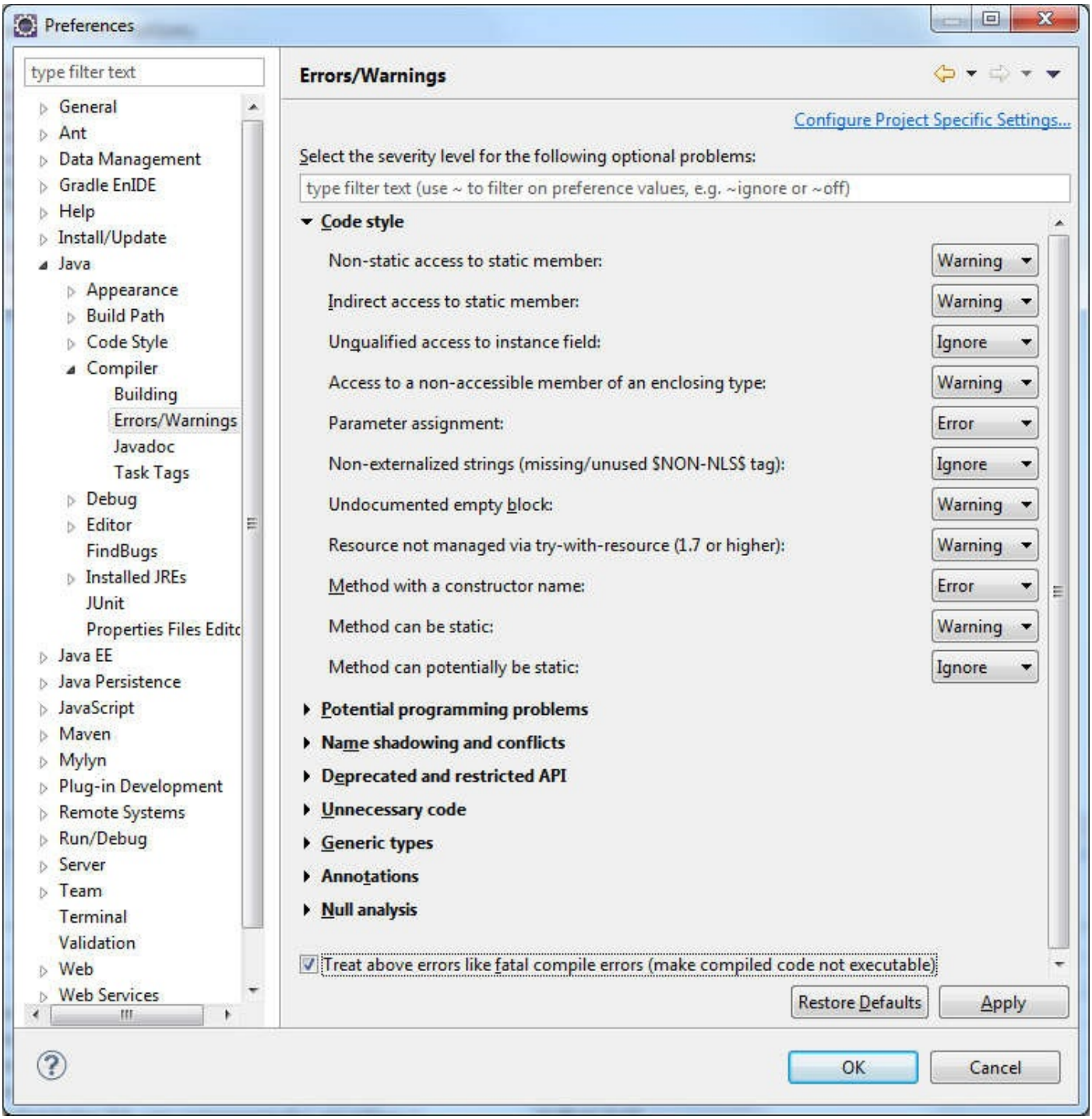


Abbildung 2-1 *Konfiguration von Java –> Compiler –> Errors/Warnings –> Code style*

Code style

Methoden, die genauso heißen wie die Klasse selbst, also dadurch sehr leicht mit einem Konstruktor verwechselt werden können, sowie Wertzuweisungen an Parameter werten wir als `Error`. Ein unqualifizierter Zugriff auf Attribute wird ignoriert. Alle anderen Werte setzen wir auf `Warning`.

Potential programming problems

Konvertierungen mithilfe von Auto-Boxing und -Unboxing stellen wir auf `Warning`. Eine versehentliche Zuweisung in einer booleschen Bedingung betrachten wir als

`Error`. Alle anderen Tests stellen wir auf `Warning`.

Unnecessary code

Unnötige `else`-Anweisungen sollen ignoriert werden, da es lediglich eine Stilfrage ist, ob man bei Auswertung einer Bedingung durch das `else` die Behandlung des anderen Zweiges darstellen möchte oder nicht. Die restlichen Tests stellen wir auf `Warning`.

Null analysis

Um uns vor Problemen mit unerwarteten `null`-Werten und Zugriffen darauf zu bewahren, stellen wir in dieser Sektion alles mindestens auf `Warning`.

2.2 Projektorganisation

Im Sourcecode erreicht man durch eine einheitliche Formatierung und Namensgebung eine gute Lesbarkeit. Wenn man das Ganze auf die Dateien eines Projekts überträgt, erleichtert eine einheitliche Projektstruktur die Orientierung in fremden (und auch in eigenen) Projekten. Daher stelle ich nun zwei bewährte Varianten vor, die Sie als Vorschlag und Ausgangsbasis für die Verzeichnisstruktur eigener Projekte nutzen können.

Als Erstes gehe ich auf die Projektstruktur ein, die entsteht, wenn man mit Eclipse ein Java-Projekt anlegt. Als Zweites beschreibe ich eine Projektstruktur, die sich durch die Verbreitung von Maven als Build-Tool als De-facto-Standard etabliert hat. Die Beispiele dieses Buchs folgen dieser zweiten Konvention.

2.2.1 Projektstruktur in Eclipse und Ant

Wenn man Projekte mit Eclipse anlegt und deren Inhalt verwaltet, so verwendet man ein Hauptverzeichnis mit dem Namen oder Synonym des Projekts. Nahezu alle anderen Daten werden in Unterordnern abgelegt. Lediglich einige (in Eclipse zunächst ausgeblendete) Metadateien (`.project`, `.classpath` usw.) findet man im Hauptverzeichnis. Die Projektstruktur wird zum besseren Verständnis im Folgenden zunächst grafisch dargestellt und danach kurz beschrieben:

```
+  <project-root>
|
+-src                          - Sourcecode unserer Applikation
|   +-ui                       - Grafische Oberfläche
|       +-FileDlg.java         - Die Klasse FileDlg
|
+-test                         - Sourcecode der Unit Tests
|   +-ui                       - Tests für die grafische Oberfläche
|       +-FileDlgTest.java     - Tests für die Klasse FileDlg
|
```

```
+-lib                          - Externe Klassenbibliotheken (JARs)
|   +-junit_4.11               - Ordner zur Strukturierung
|       +-junit.jar            - Wichtig zum Übersetzen der Unit Tests
|       +-hamcrest-core.jar    - Von JUnit benötigt
|
+-config                       - Konfigurationsdateien
|   +-images                   - Bilder
|   +-texts                    - Sprachressourcen
|
+-docs                         - Dokumentation
|
+-generated-reports            - Erzeugte Berichte (JUnit, Checkstyle usw.)
+-bin                          - Kompilierte Klassen und JAR der Applikation
```

Sourcecode und Tests

Den Sourcecode legen wir im Verzeichnis `src` ab. Darunter erfolgt die Aufteilung in Form von Packages. Parallel dazu werden Tests in einem Ordner `test` abgelegt. Dabei verwenden wir hier eine Spiegelung der Package-Struktur des `src`-Ordners. Das bietet zweierlei Vorteile: Erstens trennt man so Tests und Applikationscode im Dateisystem, wodurch sich ungewünschte Abhängigkeiten leicht erkennen lassen und später bei Auslieferungen die Testklassen nicht Bestandteil des Programms sein müssen. Zweitens liegen die Tests dadurch logisch in gleichen Packages wie der korrespondierende Sourcecode,[1] was den Zugriff auf alle Elemente des Packages (außer den privaten) möglich macht.

Weitere Dateien

In der Regel nutzt man zur Realisierung von Projekten verschiedene Klassenbibliotheken. Selbst in diesem einfachen Beispiel besitzen wir durch die Testklasse eine Abhängigkeit zur JUnit-Bibliothek. JUnit wird durch Eclipse bereits mitgeliefert und automatisch verwaltet. Andere Bibliotheken (oder eine aktuellere Version von JUnit) werden in einem Ordner `lib` gesammelt. Dabei erleichtert eine gut gewählte Verzeichnishierarchie die Übersicht über die verwendeten Bibliotheken und deren Versionen. Häufig sind für ein Projekt auch verschiedenste Konfigurationsdateien zu verwalten. Dazu bietet sich ein Ordner `config` an. Dort können etwa Textressourcen für Sprachvarianten und Konfigurationen für Logging usw. abgelegt werden. Verschiedene Arten von Dokumentation speichert man im Verzeichnis `docs`.

Tipp: Bibliotheken (JARs) in Eclipse einbinden

Wenn Sie Klassen aus Bibliotheken nutzen wollen, so liegen diese in Form von JARs (Java Archive) vor. Um diese in einem Eclipse-Projekt zu nutzen, gehen Sie wie folgt vor: Klicken Sie auf Ihr Projekt und wählen Sie im Kontextmenü PROPERTIES –> CONFIGURE BUILD PATH und in dem erscheinenden Dialog wählen Sie links den Eintrag JAVA BUILD PATH und dort den Tab LIBRARIES. Mithilfe der Buttons ADD JAR… bzw. ADD EXTERNAL JAR… können Sie die gewünschten Bibliotheken

einbinden. Ersteres wählen Sie, wenn sich die JAR-Dateien innerhalb Ihres Projekts (z. B. im Verzeichnis `lib`) befinden. Mithilfe von ADD EXTERNAL JAR… kann man JARs auch aus externen Verzeichnissen einbinden, etwa einem dedizierten Installationsverzeichnis einer Datenbank o. Ä.

Generierte Dateien

Gemäß der Faustregel »***Trenne generierte Dateien von regulärem Sourcecode***« werden generierte Dateien in separaten Verzeichnissen abgelegt. Vom Compiler generierte `.class`-Dateien liegen im Ordner `bin`. Durch Tests und andere Sourcecode-Prüfungen entstehende Berichte werden in einem Verzeichnis `generated-reports` gesammelt. Diese Aufteilung erleichtert die Übersicht und Trennung, was wiederum die später beschriebene Versionsverwaltung sowie die Automatisierung von Build-Läufen vereinfacht.

Auslieferungen und Releases

Normalerweise wird eine Applikation nicht nur innerhalb der IDE laufen, sondern vor allem als eigenständige Applikation. Häufig sollen davon auch verschiedene Versionen, sogenannte ***Releases***, erzeugt werden. Einige davon stellen stabile Softwarestände dar und können als offizielle Version oder ***Auslieferung*** an Kunden übergeben werden.

Für die aktuelle Programmversion bietet sich die Speicherung in einem Ordner `release` an. Damit die Applikation tatsächlich unabhängig von der IDE ausgeführt werden kann, müssen im Ordner `release` alle benötigten externen Bibliotheken, Konfigurationsdateien usw. bereitgestellt werden (z. B. durch Kopie) oder zugreifbar sein. Deren Pfade müssen in den sogenannten `CLASSPATH` aufgenommen werden. Dies ist die Menge von Verzeichnissen und Dateien, in denen die JVM nach Klassen und anderen Ressourcen, etwa Bildern, sucht. Der `CLASSPATH` kann als Startparameter der JVM gesetzt werden.

```
+ <project-root>
|
+-release              - Release-Ordner
    +-lib              - Kopie des lib-Verzeichnisses
    +-config           - Kopie des config-Verzeichnisses
    +-app.jar          - Applikation als JAR
```

Schwachpunkte der gezeigten Projektstruktur

Die dargestellte Verzeichnisstruktur eignet sich für viele Projekte recht gut, besitzt aber Schwachstellen. Zunächst einmal muss die Verzeichnisstruktur (bzw. Teile davon) für jedes Projekt erneut von Hand angelegt und auch gepflegt werden. Dabei besteht die Gefahr, dass sich Inkonsistenzen einschleichen: Heißt der Ordner mit den entstehenden Klassen *bin* oder *build*? Und wie derjenige mit den Fremdbibliotheken? In einem Projekt etwa *lib*, im anderen *libs*. Das setzt sich bei der Vergabe der Namen

von Unterordnern fort: Wird ein solcher *junit4* genannt oder *junit4.10* oder aber *junit_4.10*?

2.2.2 Projektstruktur für Maven und Gradle

In diesem Abschnitt schauen wir uns die von Maven und Gradle genutzte Verzeichnisstruktur als Alternative zu der von Eclipse standardmäßig verwendeten an. Diese Alternative hat sich durch die hohe Verbreitung von Maven als Build-Tool etabliert, wohl auch weil sie automatisch von Maven so angelegt werden kann.

Einheitliches Projektlayout

Das Problem möglicher Inkonsistenzen bezüglich der genutzten Verzeichnisse für kompilierte Klassen, Fremdbibliotheken oder Reports usw. wird von den beiden Build-Tools Maven und Gradle adressiert, indem diese eine einheitliche Verzeichnisstruktur für alle Projekte fordern. Dadurch ist auch für Projektneulinge sofort klar, wo sie nach Dateien gewünschten Inhalts suchen müssen.

In der standardisierten Verzeichnisstruktur befinden sich Sourcen und Tests in unterschiedlichen Verzeichnissen. Darüber hinaus benötigte Ressourcendateien werden wiederum in getrennten Verzeichnissen hinterlegt, nämlich wie folgt:

```
+ <project-root>
|
+-src
    +-main
    ¦   +-java               - Java-Klassen
    ¦   +-resources          - Konfigurationsdateien für Java-Klassen
    ¦
    +-test
        +-java               - Testklassen
        +-resources          - Konfigurationsdateien für Testklassen
```

Projektlayout am Beispiel

Nutzen wir die obige Projektstruktur für eine einfache HelloWorld-Applikation, so ergibt sich folgendes Verzeichnislayout, unter der Annahme, dass wir die Applikation mit einer Klasse `App.java` und die dazugehörige Testklasse `AppTest.java` im Package `de.javaprofi.helloworld` realisieren:

```
+-src
    +-main
    ¦   +-java
    ¦       +-de
    ¦           +-javaprofi
    ¦               +-helloworld
    ¦                       App.java
    ¦
    +-test
        +-java
            +-de
```

```
                +-javaprofi
                    +-helloworld
                            AppTest.java
```

Wenn man das Projekt mit Maven kompiliert und paketiert (`mvn package`), so entsteht ein Verzeichnis `target` als Ziel für kompilierte Klassen und weitere erzeugte Dateien wie z. B. Testresultate. Diese Struktur ist nachfolgend zum besseren Verständnis auszugsweise und gekürzt abgebildet:

```
+-target
    ¦   helloworld-1.0-SNAPSHOT.jar
    ¦
    +-classes
    ¦   +-de
    ¦       +-javaprofi
    ¦           +-helloworld
    ¦                   App.class
    ¦
    +-surefire-reports
    ¦       de.javaprofi.helloworld.AppTest.txt
    ¦       TEST-de.javaprofi.helloworld.AppTest.xml
    ¦
    +-test-classes
        +-de
            +-javaprofi
                +-helloworld
                        AppTest.class
```

Abweichungen im Projektlayout mit Gradle

Gradle verwendet für Sourcen, Tests und Ressourcen die Maven-Konventionen zur Strukturierung der Verzeichnisse. Allerdings besitzen die erzeugten Verzeichnisse einen leicht abweichenden Standard. Es entsteht ein Verzeichnis `build` mit verschiedenen Unterverzeichnissen wie folgt:

```
+  <project-root>
¦
+-build
¦   +-classes
¦   ¦   +-main
¦   ¦   ¦   +-de
¦   ¦   ¦       +-javaprofi
¦   ¦   ¦           +-helloworld
¦   ¦   ¦                   App.class
¦   ¦   ¦
¦   ¦   +-test
¦   ¦       +-de
¦   ¦           +-javaprofi
¦   ¦               +-helloworld
¦   ¦                       AppTest.class
¦   ¦
¦   +-dependency-cache
¦   +-libs
¦   ¦       helloworld.jar
¦   ¦
¦   +-reports
¦   ¦   +-tests
¦   ¦       ¦   index.html
¦   ¦       ¦
```

...

Verwaltung externer Abhängigkeiten

Eine Sache könnte uns noch wundern: Wo und wie werden denn die Abhängigkeiten zu externen Bibliotheken beschrieben? Wir finden in der Verzeichnisstruktur keinen Ordner mit Bibliotheken. Exakt! Während wir für das Eclipse-Projekt bzw. bei Builds mit Ant diese Verwaltung noch selbst erledigen mussten, helfen uns hier die Build-Tools Maven und Gradle, indem sie sich um die Auflösung und Bereitstellung externer Bibliotheken kümmern. Details dazu lernen wir später in Abschnitt 2.7 kennen.

Es bleibt für das Projektverzeichnis festzuhalten, dass die Automatiken recht gut verhindern, dass es zu Inkonsistenzen kommt, indem sie dafür sorgen, dass die Verzeichnisse konsistent benannt sind: Damit finden sich nach einem Update z. B. von Version 3 auf Version 4 von JUnit nicht plötzlich die JARs der aktuelleren Version in dem Verzeichnis *junit3* mit der alten Versionsnummer wieder.

2.3 Einsatz von Versionsverwaltungen

Nachdem wir zwei Alternativen zur Strukturierung von Projekten kennengelernt haben, widmen wir uns nun dem Thema Versionsverwaltungen. Diese ermöglichen die Speicherung und Verwaltung der Historie aller relevanten Dateien eines Softwareprojekts (vgl. dazu den folgenden Praxistipp). Dazu werden diese Dateien an einem zentralen Ort, dem sogenannten ***Repository***, gespeichert. Der Vorteil davon ist, dass sich hier nicht nur die aktuelle Version einer Datei befindet, sondern deren gespeicherte Historie zur Verfügung steht. Im Repository sind also alle früheren Stände enthalten, einsehbar und bei Bedarf wiederherstellbar. Das ist insofern von Bedeutung, als dass man auch einmal Experimente vornehmen und nötigenfalls zu einem älteren, funktionierenden Stand zurückkehren kann.

Darüber hinaus können verschiedene, voneinander unabhängige Entwicklungslinien, sogenannte ***Branches***, verwaltet werden. Diese können z. B. für Umbaumaßnahmen oder Experimente genutzt werden. Die Hauptentwicklungslinie wird ***Stamm***, ***Hauptast*** oder auch ***master*** bzw. ***default*** genannt. Den aktuellsten Stand einer Datei auf einer Entwicklungslinie nennt man ***HEAD***. Erfolgte Änderungen und Erweiterungen, die auf Branches entwickelt wurden, können über einen ***Merge***-Vorgang in den aktuellen Stand integriert werden. Diese Abläufe deutet Abbildung 2-2 an.

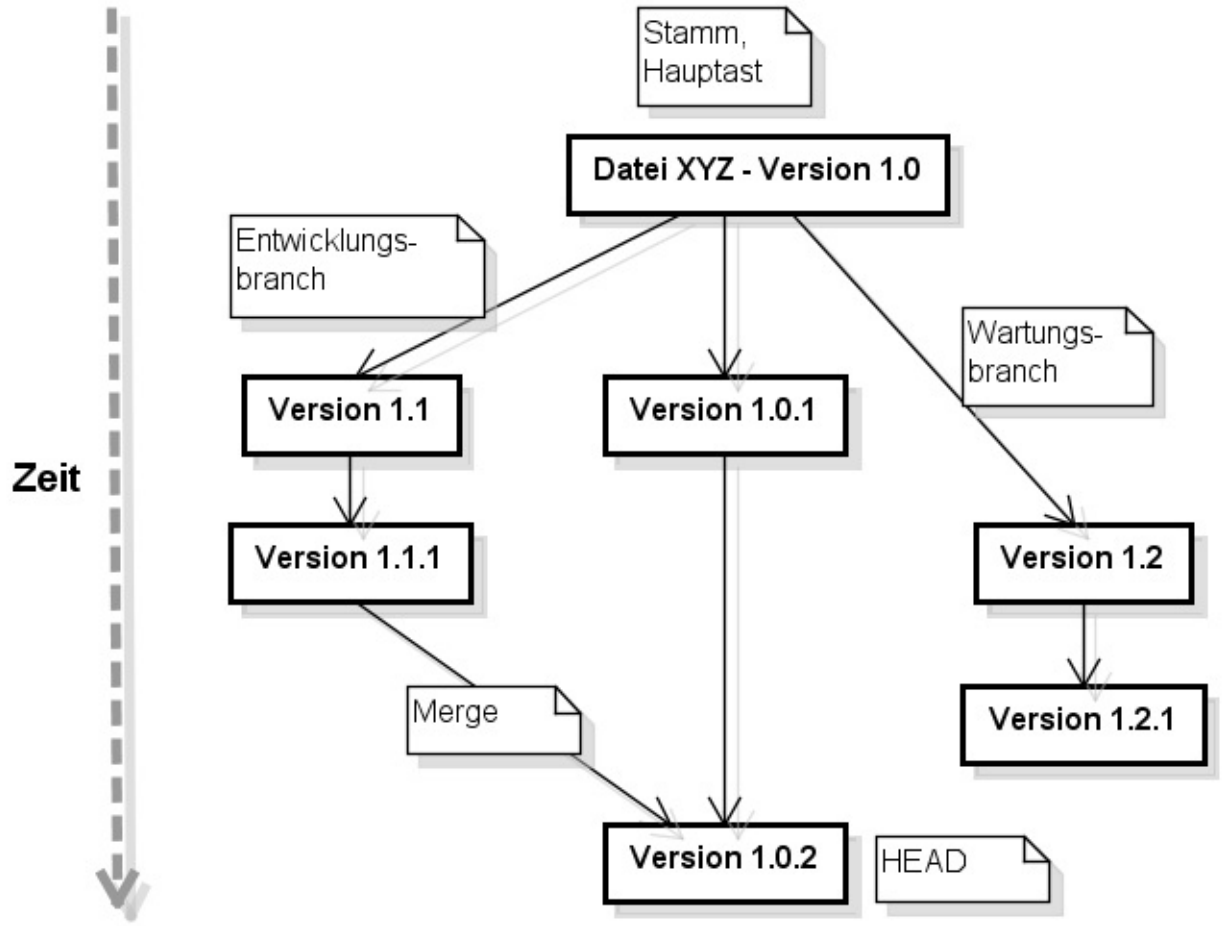


Abbildung 2-2 *Dateiversionsbaum*

Hilfreich ist, dass bei Änderungen an Dateien die Unterschiede zu beliebigen anderen Versionen ermittelt werden können. Außerdem können Versionen bei der Speicherung im Repository mit Kommentaren versehen werden, was den Überblick über Änderungen in den jeweiligen Versionen erleichtert.

Tipp: Inhalt des Repository

Anhand der in Abschnitt 2.2 beschriebenen Projektstruktur könnte man auf die Idee kommen, einfach alle Dateien eines Projekts in das Repository aufzunehmen. Dies ist aber nur begrenzt sinnvoll. Abgesehen von wenigen Ausnahmen sollten all diejenigen Dateien, die während des Build-Prozesses entstehen, *nicht* gesichert werden, eben weil sie sich immer wieder problemlos erneut erzeugen lassen. Dazu gehören vor allem die generierten `.class`-Dateien oder Berichte bzw. Dokumentationen. Ausnahmen können spezielle Versionen davon sein, die man historisieren möchte.

Was sollte also gespeichert werden? Die Frage lässt sich einfach beantworten: Alle Dateien, die sich nicht automatisch generieren lassen. Dazu gehören der Sourcecode, die Unit Tests, verschiedene Dokumentationen (UML-Diagramme, Textdokumente usw.), Konfigurationsdateien und benötigte Build-Skripte.

Motivation für Versionsverwaltungen

Jede Bank verwendet einen Tresor, um wertvolle Dinge zuverlässig zu schützen. Softwareentwicklung ohne Sicherung ist ähnlich unsicher wie eine Geldanlage in Omas Sparstrumpf. Also sichern Sie Ihren größten Schatz – Ihren Sourcecode – in einem virtuellen Safe! Falls Sie bis jetzt ohne Versionsverwaltung leben konnten, so sollten Sie sich durch die folgenden zwei Situationsbeschreibungen davon überzeugen lassen, schnellstmöglich ein solches System zu nutzen. Ansonsten könnte jeder kleine Fehler fatale Folgen haben, was ich mit zwei Anekdoten verdeutlichen möchte:

- Ich erinnere mich noch gut an die Tage mit Sicherungsdisketten. Es war eine mühselige und fehlerträchtige Arbeit, die aktuellen Daten auf die entsprechende Diskette zu überspielen. Als ich einmal die Arbeit eines Tages sichern wollte, habe ich beim Kopieren Quelle und Ziel vertauscht und damit hatte ich den gesamten Tag umsonst gearbeitet. Außerdem hatte ich danach zwei ältere Softwarestände, von denen ich nicht wusste, welche Änderungen diese enthielten. Um solche Situationen zu vermeiden, sollte häufig ins Repository gesichert werden. Somit bleibt selbst bei Problemen der maximale Datenverlust, d. h. die nicht gesicherten Änderungen, immer überschaubar.

  Diese Anekdote adressiert einen anderen wichtigen Aspekt: ***Datensicherung***. Ein Repository ist zwar ein guter Platz, um Daten zentral abzulegen und gezielt wiederherstellen zu können. Man ist so aber lediglich gegen die »eigene Dummheit« beim Ändern des Sourcecodes geschützt. Für eine weiter gehende Sicherheit der Daten ist es jedoch elementar wichtig, regelmäßig ein Backup des Repository zu erstellen, um das Repository an sich zu sichern, etwa gegen Festplattenfehler o. Ä.

- Stellen Sie sich vor, es ist Freitagmorgen und Ihr Chef betritt aufgeregt den Raum und verlangt für den späten Nachmittag eine Auslieferung, basierend auf dem letzten Auslieferungsstand des Programms von vor zwei Wochen mit genau zwei dedizierten Bugfixes. Ohne Versionsverwaltungssystem haben Sie jetzt ganz schlechte Karten, denn in der Zwischenzeit hat die gesamte Entwicklertruppe kräftig weitergearbeitet. Dies alles ohne Fehler wieder rückgängig zu machen ist nahezu unmöglich. Eine Sicherung des Sourcecodes gibt es zwar irgendwo, aber in der Hektik ist diese momentan nicht auffindbar oder sie ist auf der mobilen Festplatte des Kollegen, der natürlich gerade gestern krank geworden ist.

Die Moral von der Geschichte ist: Lassen Sie den Computer die Arbeit erledigen, die er viel besser kann als Sie: das Verwalten vieler Versionen.

Wie schon eingangs erwähnt, spricht neben den genannten Vorteilen noch ein weiteres wesentliches Argument für den Einsatz einer Versionsverwaltung: Man kann mithilfe von Branches sehr leicht experimentelle Versionen der Software entwickeln, ohne dabei Angst haben zu müssen, Auslieferungen oder die gesamte Weiterentwicklung zu stören. Verläuft ein solches Experiment erfolgreich, so kann man die Änderungen übernehmen. Ansonsten verwirft man sie einfach.

Varianten der Versionsverwaltung

Vor dem Aufkommen von dedizierten Programmen zur Versionsverwaltung wurden oftmals die Dateien eines Projekts lediglich in ein anderes Verzeichnis kopiert – zweckmäßigerweise in ein Verzeichnis mit einem Zeitstempel im Namen. Diese händische Versionsverwaltung scheint zunächst ganz natürlich und ist auch recht einfach, aber doch recht fehleranfällig, mitunter passieren Fehler beim Kopieren, wie ich es zuvor schon andeutete. Es gibt jedoch einen noch entscheidenderen Nachteil: Wenn man herausfinden möchte, was sich zwischen Versionen geändert hat, so ist dies einigermaßen mühselig festzustellen, z. B. durch den Einsatz von Programmen wie WinMerge o. Ä. Erschwerend kommt hinzu, dass mitunter versehentlich die zur Versionierung eigentlich notwendigen Kopien ausbleiben und somit gewisse Zwischenstände nicht gesichert sind. Dann besitzt man nur eine unvollständige Historie.

Um diesen Nachteilen zu begegnen und Änderungen an den Dateien eines Projekts feingranular speichern und später sehr gut nachvollziehen zu können, wurden Versionsverwaltungssysteme erfunden. Man unterscheidet zwischen ***zentralen*** und ***dezentralen Versionsverwaltungen***. Schon vor einigen Jahrzehnten entstanden zentrale Versionsverwaltungen (Version Control System, VCS). Dort erfolgt die Versionsverwaltung auf einem zentralen Server und erfordert von den Nutzern (Clients) somit (Netzwerk-)Zugriff darauf. Weil das Ganze eine gewisse Einstiegshürde und etwas initialen Aufwand benötigt, findet man auch weiterhin noch die zuvor beschriebene händische Versionsverwaltung. Dezentrale Versionsverwaltungen (Distributed Version Control System, DVCS) kombinieren die Vorteile beider Varianten: Sie sind nahezu so einfach in der Handhabung wie des Erstellen lokaler Kopien und bieten darüber hinaus leistungsstarke Versionsverwaltungsfunktionalität.

In den nachfolgenden Abschnitten wollen wir kurz auf die Arbeit mit zentralen und dezentralen Versionsverwaltungen eingehen.

2.3.1 Arbeiten mit zentralen Versionsverwaltungen

Während die lokale Versionsverwaltung durch Kopieren per Hand selbst schon für eine Person recht schnell an Grenzen stösst, so gilt dies umso mehr bei der Zusammenarbeit und Verwaltung von Änderungen im Team, weil hier deutlich mehr (eventuell auch konkurrierende) Änderungen und damit auch Abstimmungsbedarf entstehen. Als Abhilfe wurden zentrale Versionsverwaltungen entwickelt. Bekannte zentrale Versionsverwaltungen sind das Concurrent Version System (CVS) und Subversion (SVN). CVS ist älter und besitzt einige Einschränkungen, etwa bei Namensänderungen. SVN wurde als Nachfolger neu entwickelt und unterstützt diverse Dinge besser als CVS.

Repository

Wie bereits erwähnt, werden die Dateien eines Projekts mit ihrer gesamten Historie an einer zentralen Stelle, dem ***Repository***, gespeichert. Gewöhnlich liegt dieses Repository auf einem dedizierten Server und dort wird auch der Serveranteil zur Versionsverwaltung ausgeführt. Zum Zugriff darauf benötigt man als Nutzer eine spezielle Clientsoftware, mit der alle Aktionen zur Versionsverwaltung erfolgen, z. B. das Hinzufügen, Ändern, Aufbereiten der Versionshistorie usw. Diese erfordern jeweils Zugriffe auf das Repository. Der dezentrale Client-Server-Ansatz erlaubt es, an verschiedenen Orten verteilt an einem Projekt zu arbeiten, setzt aber Netzwerkzugriff voraus.

Arbeitsablauf

Zum Bearbeiten eines Versionsstands überträgt ein Entwickler den gewünschten Stand aller Dateien eines Projekts aus dem Repository in ein Arbeitsverzeichnis auf seinem Rechner. Diesen Vorgang bezeichnet man als »***Auschecken***« (Checkout). Anschließend arbeitet man auf lokalen Kopien der Dateien, der sogenannten ***Working Copy***, und nimmt dort Änderungen vor. Sind diese abgeschlossen oder haben diese einen stabilen Zwischenstand erreicht, so sollte eine Integration in das Repository erfolgen. Dadurch werden die Änderungen allen anderen Kollegen zugänglich gemacht. Dieser Vorgang wird »***Einchecken***« (Checkin) oder auch ***Commit*** genannt. Nach einem Commit müssen die Kollegen allerdings einen Abgleich mit dem Repository durchführen, damit die neu eingespielten Änderungen tatsächlich in ihrem Arbeitsbereich sichtbar werden. Diesen Abgleich nennt man ***Update*** oder ***Synchronize***. Die beschriebenen Arbeitsabläufe visualisiert Abbildung 2-3.

Besonderheiten beim Abgleich

Da die eigenen Änderungen jeweils nur auf lokalen Kopien der Originale erfolgen, kann es zu Problemen beim Abgleich kommen, wenn mehrere Entwickler die gleiche Datei verändert haben. In CVS werden bei einem ***Update*** die lokalen Änderungen mit der aktuellen Version aus dem Repository überschrieben. Ein ***Synchronize*** versucht die Änderungen der lokalen Version mit denen aus dem Repository abzugleichen. Dies funktioniert gut, wenn unterschiedliche Teile einer Datei von Änderungen betroffen sind. SVN integriert Änderungen bei einem Update automatisch, wenn aufgrund der Differenzanalyse keine Probleme festgestellt wurden. Widersprechen sich Änderungen an der lokalen Datei und deren Version im Repository, so muss man steuernd eingreifen. Eine solche Situation wird ***Konflikt*** genannt. Eine betroffene Datei lässt sich erst ins Repository integrieren, wenn alle dort gefundenen Konflikte behoben sind. Die Zusammenführung verschiedener Änderungen wird als ***Merge*** bezeichnet.

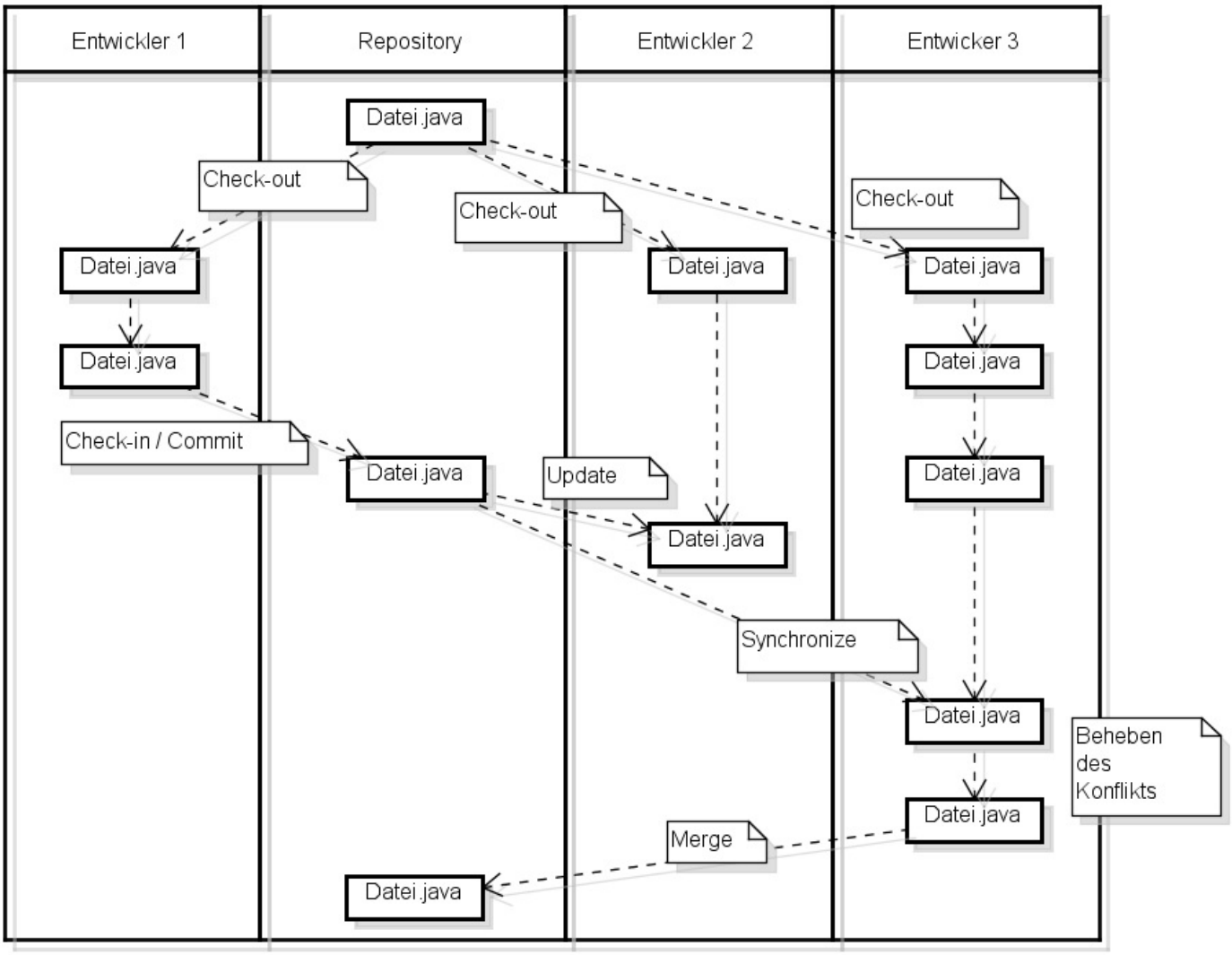


Abbildung 2-3 *Arbeitsablauf mit Versionsverwaltungen*

Tagging und Branching

Gehen wir noch einmal zu dem Beispiel zurück, in dem eine Auslieferung des Systems, basierend auf dem letzten Auslieferungsstand sowie ein paar dringend benötigten Fehlerbehebungen, gewünscht wird. Sie brauchen nun Zugriff auf den Stand des Projekts genau so, wie es ausgeliefert wurde. Eine Möglichkeit dazu besteht darin, Dateien anhand ihrer Versionsnummer zurückzuholen. Dieser Vorgang ist für CVS aufwendig, da die Versionsnummer von Datei zu Datei variiert.[2] In SVN ist dieser Vorgang leichter, da die Versionsnummer für alle Dateien des Repository einheitlich behandelt wird (siehe folgenden Praxistipp).

Alternativ können Dateien anhand eines speziellen Datums zurückgeholt werden. Dies ist nützlich, wenn wenige Dateien betroffen sind oder nur der zeitliche Verlauf von Interesse ist. Normalerweise soll ein Projekt jedoch wieder in einen Zustand gebracht werden, in dem es zu einem bestimmten Zeitpunkt oder Ereignis war, beispielsweise einer Auslieferung oder des Abschlusses größerer Änderungen. Das führt uns zum sogenannten Tagging, was wir uns nun anschauen.

Tipp: Versionsnummern in SVN

Die Versionsnummer einer Datei und von Verzeichnissen entspricht immer der Versionsnummer des Projekts zum Zeitpunkt der Änderung. Für Verzeichnisse ist dies die höchste Versionsnummer der enthaltenen Dateien und Verzeichnisse. Dadurch kann die Folge der Versionsnummern auch Lücken haben. Beim Auschecken wird jeweils die größte Versionsnummer aus dem Repository geholt, die kleiner oder gleich der angeforderten Versionsnummer beim Auschecken ist.

Tagging

Wenn man für Änderungen einen ganz speziellen Stand eines Projekts bearbeiten möchte, so wäre es umständlich und fehleranfällig, eine Menge von Dateien über deren Versionsnummer oder ein Datum zusammensuchen zu müssen. Als Abhilfe kann man zu einem beliebigen Zeitpunkt eine sogenannte ***Markierung*** (engl. ***Tag***) setzen. Vor größeren Änderungen und Auslieferungen sollte man dies in jedem Fall tun. Mithilfe von Markierungen kann später ein Zwischenstand exakt wiederhergestellt werden, um basierend darauf Fehlerkorrekturen durchführen zu können.

Man kann sich Markierungen wie eine Verbindungsschnur bzw. einen Zusammenschluss zwischen Versionen verschiedener Dateien vorstellen (vgl. Abbildung 2-4).

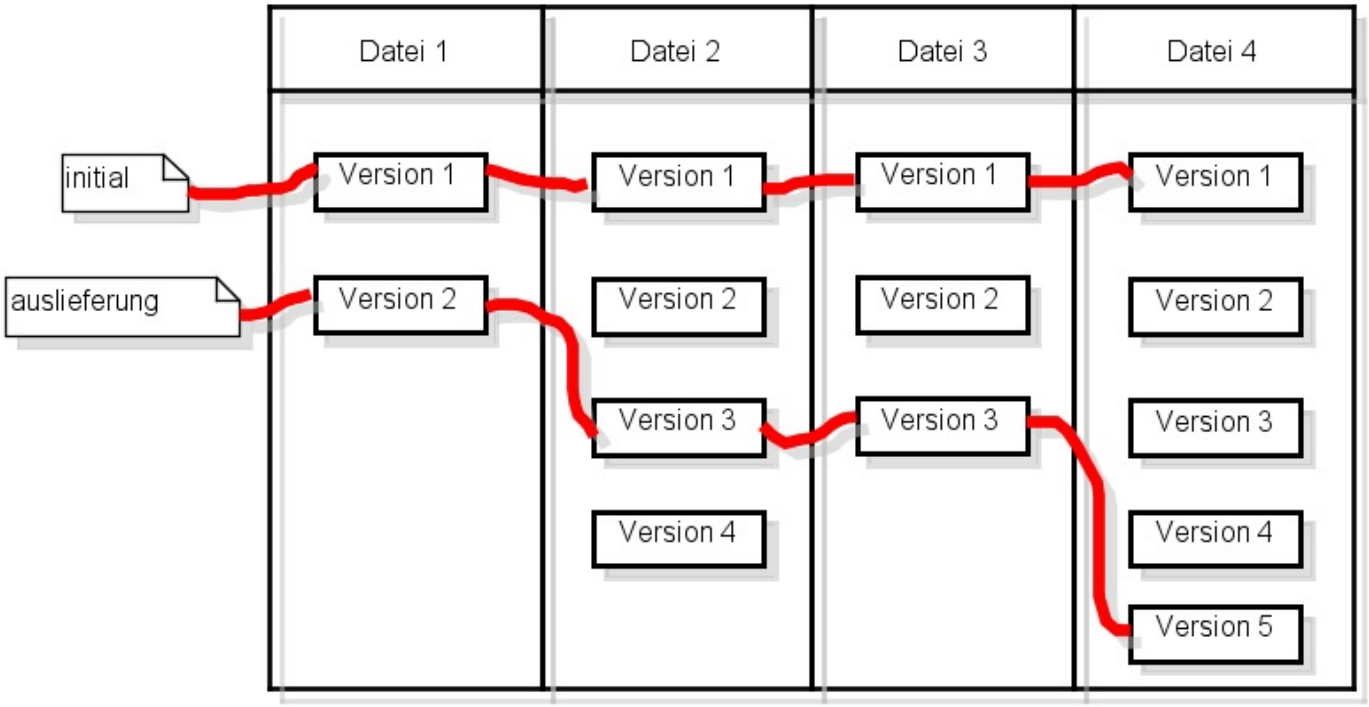


Abbildung 2-4 *Zwei Tags (miteinander verbundene Versionen von Dateien)*

Branching

Unter einer Verzweigung oder einem ***Branch*** versteht man eine eigene Entwicklungslinie. Einen solchen Abzweig kann man von jeder beliebigen anderen Entwicklungslinie abspalten, meistens geschieht dies aber vom Stamm, auch ***Main-***

Branch oder Trunk genannt. In einer solchen Entwicklungslinie entsteht eine eigene Historie, sodass parallel zu anderen Zweigen weitergearbeitet werden kann: Änderungen in einem Zweig haben keinen Einfluss auf andere. Dadurch sind Branches sehr hilfreich, um potenziell gefährliche oder umfangreiche Änderungen zu isolieren, bis sich diese stabilisiert haben.

Zur Motivation blicken wir auch hier wieder auf das vorherige Beispiel zurück. Der ältere Sourcecode-Stand konnte aufgrund einer gesetzten Markierung leicht wiederhergestellt werden. Auf dieser lokalen Kopie können Fehlerbehebungen erfolgen. Natürlich wäre es wünschenswert, die durchgeführten Änderungen im Repository zu sichern, damit man später auf diesem speziellen Stand weiterarbeiten kann. Dies ist aber nicht sinnvoll möglich, wenn kein Branch erzeugt wurde, weil man ansonsten neuere Versionen des Hauptastes mit einem älteren Stand überspielen würde. Das ist jedoch in der Regel nicht gewünscht. Als Abhilfe dienen Branches, die eine eigene, separate Historie bieten. Dadurch lassen sich Änderungen wieder sinnvoll ins Repository reintegrieren.

Tipps für die Arbeit mit Branches

Beim Erstellen von Branches sollte man etwas Vorsicht walten lassen. Unbedacht und selbst für kleinere Änderungen eingesetzt können zu viele Branches ein Projekt ins Chaos stürzen, da man schnell den Überblick verliert, was in welchem Branch erweitert oder korrigiert wurde. Als Folge divergiert die Sourcecode-Basis, sodass es mit der Zeit immer schwieriger wird, die Änderungen auf den Branches wieder korrekt zusammenzuführen (zu mergen).

Mit CVS bzw. SVN lassen sich Verbesserungen und Fehlerbehebungen eines Branches oftmals nur mit großer Mühe auf andere Branches übertragen. Allerdings ist nicht unbedingt die absolute Anzahl an Branches klein zu halten, sondern lediglich die Anzahl »aktiver« Branches, d. h., an denen gleichzeitig gearbeitet wird. Darüber hinaus sollte die Verzweigungstiefe und Komplexität der Branches überschaubar sein. Nur in wenigen Fällen ist es sinnvoll, von einer Verzweigung wiederum Verzweigungen zu erzeugen. Ein möglicher Grund dafür sind Wartungsaufgaben auf älteren Ästen.

Hält man die Anzahl parallel laufender Entwicklungen gering, so sinkt die Wahrscheinlichkeit für Konflikte beim Integrieren. Das erreicht man, ***indem man Branches möglichst häufig (z. B. täglich oder sobald die Arbeiten einen stabilen Zwischenstand erreicht haben) mit deren Ursprungsbranch synchronisiert*** und auch im abgezweigten Branch wichtige Änderungen des Ursprungsbranches wieder integriert.

Tipp: Namensgebung für Markierungen und Verzweigungen

Die Namen von Branches und Markierungen sollten einem einheitlichen Schema folgen und deren Bedeutung sollte offensichtlich sein. Der Name sollte alle

wichtigen Informationen über den Softwarestand enthalten und einer Konvention, etwa `ZWECK_KUNDE_VERSION_DATUM`, folgen. Versionsnummern für Releases sollten die Versionsangaben `Major.Minor.Patchlevel` enthalten. Damit ergibt sich als Beispiel folgender Name: `auslieferung_Meyer_V2.7.13_20080612`. Für den Zweck sind zumindest folgende Kürzel sinnvoll:

- `entwicklung` für Branches für Entwicklungen
- `auslieferung` für Auslieferungen an Kunden
- `release` für funktionsfähige Zwischenstände der Software

Die kritischen Aussagen zum Branching gelten insbesondere für VCS mit zentralem Repository und in wesentlich geringerem Maße für DVCS, da diese speziell dafür ausgelegt sind, mit vielen Branches und Versionsständen umgehen zu können.

2.3.2 Dezentrale Versionsverwaltungen

Während die zuvor vorgestellten Systeme CVS und SVN beide eine zentrale Versionsverwaltung mit einem zentralen Repository repräsentieren, werden neuerdings dezentrale Versionsverwaltungen, etwa Git[3] und Mercurial[4], immer beliebter. Git ist derzeit wohl die populärste dezentrale Versionsverwaltung und wurde von Linus Torvalds kurzerhand selbst geschrieben, da keine der herkömmlichen Versionsverwaltungen ausreichend Flexibilität und Funktionalität für die verteilte Entwicklung des Linux-Kernels bot. Insbesondere fehlte es an gutem Support für viele unabhängige Branches und das offline Arbeiten sowie für das Mergen oder Rücknehmen von verschiedenen Änderungen.

Mercurial ist eine valide und von mir bevorzugte Alternative zu Git, weil Mercurial teilweise einfacher in der Handhabung ist, wie es nachfolgend im Meinungskasten dargestellt wird.

Meinung: Git vs. Mercurial

Im Prinzip sind Git und Mercurial gleich mächtig. Meiner Meinung nach hinkt Git aber in der Benutzbarkeit etwas hinterher, weil hier zu viele Implementierungsdetails durchscheinen. Auch ist die grafische Integration von Mercurial (TortoiseHG[a]) übersichtlicher als das von Git mitgelieferte Git GUI. Zudem produziert das GUI von Git teilweise unverständliche Fehlermeldungen. TortoiseHG macht da einen reiferen Eindruck, ist fehlertolerant und optisch klarer strukturiert. Dies gilt meiner Ansicht nach auch noch, wenn man TortoiseGit[b] als GUI nutzt. Das zeigt sich in Kleinigkeiten: Commit-Kommentare werden standardmäßig bei Git über den vi eingegeben. Mercurial öffnet dagegen den vom System voreingestellten Texteditor, wodurch auch Einsteiger mit wenig Unix-Know-how keine Probleme haben sollten, einen Kommentar einzugeben. Entscheidend ist für mich aber die Art und Weise der Verwaltung und Angabe von

Revisionsnummern. In Mercurial sind für den Benutzer Revisionsnummern natürliche Zahlen und es werden nicht die internen hexadezimalen Keys wie bei Git nach außen exponiert. Ich referenziere einfach lieber Revision 4711 als 00eda2a680893a20ea11d48ddd884812dc97a718. Basierend auf den Ausführungen wird klar, dass ich persönlich Mercurial bevorzuge, da es sich für mich natürlicher in der Handhabung anfühlt. Ich empfehle Ihnen, einfach mal beide Versionsverwaltungen zu installieren und damit ein wenig herumzuspielen, um die eigene Präferenz zu finden.

[a] http://tortoisehg.bitbucket.org/

[b] https://code.google.com/p/tortoisegit/

Repositories

Für die Arbeitsweise mit CVS und SVN haben wir nach Lektüre der vorangegangenen Abschnitte ein erstes Verständnis aufgebaut. Dezentrale Versionsverwaltungen arbeiten gar nicht so verschieden dazu. Insbesondere fügen sie jedoch optional eine weitere Hierarchieebene ein. Schauen wir uns das im Folgenden genauer an.

Während bei zentralen Versionsverwaltungen jeweils Zwischenstände (Arbeitskopien) auf die Rechner der jeweiligen Entwickler übertragen werden, wird bei dezentralen Versionsverwaltungen lokal auf dem Rechner des Entwicklers im Arbeitsverzeichnis das gesamte Repository gespeichert.[5] Innerhalb des ***Arbeitsverzeichnisses*** existiert also ein vollständiger Stand des Projekts mitsamt seiner Historie. Mit diesem ***lokalen Repository*** kann man nahezu wie mit CVS oder SVN gewohnt arbeiten, allerdings mit dem großen Vorteil, dass alle Aktionen nicht über das Netzwerk, sondern lokal im Dateisystem und damit extrem performant ausgeführt werden können. ***Git und Mercurial ermöglichen also das Arbeiten mit lokalen Repositories unabhängig von einem zentralen Repository und Netzwerkzugang***.[6] Zunächst einmal erfolgt die Versionskontrolle dezentral und jedes Repository existiert für sich eigenständig. Es besteht aber die Möglichkeit, ein Repository im Netz zu veröffentlichen und so auch mit anderen teilen zu können.

Dadurch gibt es bei den Vertretern von DVCS im Unterschied zu den zentralen Versionsverwaltungen konzeptionell noch die Ebene über dem lokalen Repository, nämlich alle möglichen Repositories im Netz (***Remote Repository***). Für die Zusammenarbeit an einem größeren Projekt empfiehlt es sich, neben vielen verteilten Repositories auch ein ausgezeichnetes Projekt-Repository auf einem zentralen Server zu nutzen, ähnlich zu dem zentralen Server bei CVS/SVN. Dadurch gestaltet sich der Umstieg von CVS/SVN leichter, da man sich die lokalen Repositories als praktische lokale Zwischenspeicher vorstellen kann: Ein Feature, das man sich häufig für CVS/SVN gewünscht hat – insbesondere wenn die Zugriffe auf das zentrale Repository langsam oder aufgrund von Netzwerkproblemen nicht bzw. nur eingeschränkt möglich waren.

Tipp: Sicherungskopien

Weil jeder Benutzer ein vollständiges Repository auf seinem Rechner besitzt, ist es ganz einfach durch Kopie des Arbeitsverzeichnisses möglich, ein Repository neu zu erzeugen. Somit können auf einfache Art und Weise Sicherungskopien erstellt werden. Selbst bei Datenträgerdefekten auf einem Rechner kann man somit einen (nahezu) vollständigen Stand einfach durch Kopie eines anderen Repository wiederherstellen.

Begriffe und Arbeitsablauf im Überblick

Die Ideen und Abläufe sind bei der Arbeit mit einem lokalen Repository grundsätzlich ähnlich zu der von zentralen VCS gewohnten Arbeitsweise. In der Abbildung sehen wir einige Arbeitsschritte, die einen ersten Eindruck von der Arbeit mit einem DVCS vermitteln.

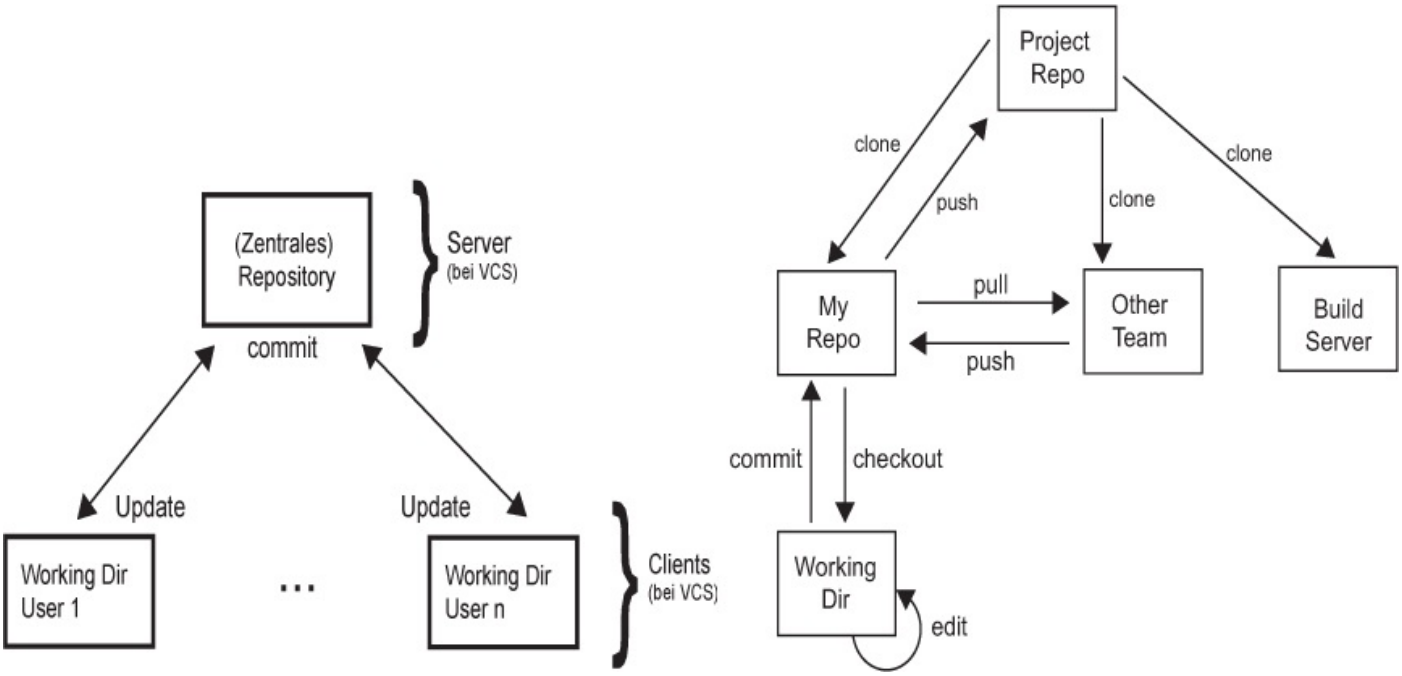


Abbildung 2-5 *Repository-Workflow für SVN (links) und DVCS (rechts)*

Für CVS und SVN haben wir bereits den typischen Arbeitsablauf und die entsprechenden Begriffe wie Checkout und Commit kennengelernt. Nachfolgend betrachten wir ein paar typische und grundlegende Arbeitsschritte beim Einsatz von DVCS, die aufgrund ihrer anderen Arbeitsweise auch eine leicht andere Begriffswelt besitzen, die folgende Auflistung näher bringt.

Die folgenden Ausführungen zeigen exemplarisch immer die Befehle sowohl für Mercurial als auch für Git. Das Kommando für Mercurial ist `hg`[7] und für Git `git`.

Repository erstellen

Sollen die Dateien in einem Verzeichnis versioniert werden, so kann man sehr einfach dort ein neues Repository erstellen: Man wechselt in das gewünschte Verzeichnis und

gibt dafür Folgendes ein – als Abkürzung verwende ich nachfolgend mitunter `git/hg`, was ausdrückt, dass die Syntax identisch ist:

```
git init       // für Git
// --------------------
hg  init       // für Mercurial
```

Das ist alles, um ein neues Repository anzulegen. Nun können Sie die zu versionierenden Dateien dem Repository hinzufügen. Bevor ich darauf eingehe, beschreibe ich den Fall, dass das Repository als Kopie eines anderen Repository erstellt wird.

Repository clonen

Zum Erstellen einer Kopie eines Repository nutzt man den `clone`-Befehl, der folgende Syntax besitzt:

```
git/hg clone /pfad/zum/repository ziel                  // lokales Repository
git/hg clone benutzername@host:/pfad/zum/repository     // Remote Repository
```

Nun zeige ich noch, wie man auf lokale Repositories zugreifen kann und deute für Mercurial den Zugriff auf ein Remote Repository unter Nutzung von HTTP, hier vereinfachend des eigenen Rechners (`localhost`), an:

```
git clone file:///C:/Users/Micha/Desktop/DVCS DVCS-GIT
// --------------------
hg clone file:///C:/Users/Micha/Desktop/DVCS DVCS-CLONE
hg clone http:/localhost:8000/DVCS DVCS-CLONE-BY-HTTP
```

Das erstmalige Initialisieren eines lokalen Repository kann durch eine Kopie eines anderen lokalen oder Remote Repository erfolgen. Dabei wird eine vollständige Kopie erzeugt, d. h. inklusive sämtlicher Branches und der gesamten Versionshistorie. Zudem erfolgt ein Checkout des Hauptastes, den man für Git ***master*** bzw. für Mercurial ***default*** nennt, statt ***trunk*** bei SVN.

Checkout

Nach dem Klonen eines Repository befindet man sich zunächst auf dem *master* bzw. *default*. Teilweise sollen aber auch andere Versionsstände bearbeitet werden, dazu dient ein `checkout`, der den gewünschten Versionsstand einer speziellen Revision oder eines Branches in das Arbeitsverzeichnis überträgt. Dabei hat man Zugriff auf jede beliebige Version. Wollten wir die ältere Revision 123 auschecken, so schreibt man Folgendes, wobei für `git` ein frei erfundener Hex-Key dargestellt ist:

```
git checkout 22a16bb5cdd8a8d72bda4276b720e0a86a2bad72
// --------------------
hg checkout -r 123
```

Um wieder auf die aktuellste Revision HEAD zurückzuwechseln, schreibt man einfach:

```
git checkout master
// --------------------
hg checkout default
```

Änderungen und Staging Area

Wenn Sie Änderungen an den lokalen Dateien vornehmen, werden diese nicht direkt ins lokale Repository übertragen. Git und Mercurial bieten eine weitere Zwischenebene: die sogenannte ***Staging Area***. Alle veränderten Dateien, die (später) ins Repository committed werden sollen, müssen zunächst der Staging Area hinzugefügt werden. Dazu dient der `add`-Befehl:

```
git/hg add <dateiname>
git/hg add *
```

Dies kann man beliebig für verschiedene Dateien spezifisch wiederholen oder die Wildcard `*` nutzen. Falls man eine oder mehrere Dateien einmal versehentlich der Staging Area hinzugefügt hat, so kann man diese bei Bedarf wieder daraus entfernen. Meistens wird man sich zunächst einen Überblick verschaffen wollen. Dazu dient der `status`-Befehl:[8]

```
git/hg status
```

Bei der Ausgabe symbolisiert ein `?`, dass die Datei nicht unter Versionskontrolle steht. Das `A` bedeutet Added, dass die Datei in der Staging Area hinzugefügt ist:

```
A ToBeRemovedFromStaging.txt
? ToBeAdded.txt
```

Anhand der Konsolenausgabe kann man diejenigen Dateien ermitteln, die man wieder entfernen möchte, etwa die Datei `ToBeRemovedFromStaging.txt`. Der dazu benötigte `remove`-Befehl besitzt folgende Syntax:

```
git/hg remove <dateiname>
git/hg remove *
```

Änderungen ins lokale Repository übertragen

Durch Aufruf des `commit`-Befehls werden Änderungen ins lokale Repository übertragen:

```
git/hg commit -m "Commit-Nachricht"
```

Der Parameter `-m` erlaubt es, beim Commit eine (möglichst) aussagekräftige Nachricht zu übergeben. Die Änderungen sind dann im lokalen Repository historisiert. Bei der Arbeit im Team wird man diese jedoch immer mal wieder mit dem Projekt-Repository abgleichen wollen. Dazu dient der später beschriebene `push`-Befehl.

Branching

Bekanntermaßen lassen sich durch Einsatz von Branches unabhängige Entwicklungen separat und isoliert voneinander mit unterschiedlichen Historien entwickeln. Wenn ein Repository neu erstellt wird, so existiert dort standardmäßig immer der Branch *master* (Git) bzw. *default* (Mercurial). Um Merge-Problemen möglichst zu vermeiden, war es bei CVS/SVN nicht ungewöhnlich, auf dem dazu korrespondierenden *trunk* zu

entwickeln. Ein analoges Vorgehen ist für Git und Mercurial zwar auch möglich, es bietet sich aufgrund ihrer deutlich besseren Unterstützung von Branching und Merging aber eher ein anderes Vorgehen an: Entwicklungen sollten auf eigenständigen Branches durchgeführt werden, wobei hier – wie schon für CVS/SVN erwähnt – eine Beschränkung auf eine überschaubare Anzahl an Branches aus purem Selbstschutz und zur besseren Übersicht eingehalten werden sollte.

Möchte man eine Funktionalität auf einem Branch entwickeln, so führt man Folgendes zu dessen Erstellung aus:

```
git branch mybranch        // Branch erstellen
git checkout mybranch      // Stand mybranch in Working Copy holen
// ---------------------
hg branch mybranch
```

Nun kann man Änderungen an den Dateien der Arbeitskopie vornehmen, die den Branch repräsentiert.

Merge

Irgendwann sind die Arbeiten an dem besonderen Feature auf dem Branch *mybranch* erfolgreich fertiggestellt und sollen wieder mit dem *master* bzw. *default* abgeglichen werden. Gerade im Zusammenführen von Änderungen und Branches liegt eine Stärke von DVCS.

Um einen Merge-Vorgang zu starten, muss man sich in dem Branch befinden, in den die Änderungen eingespielt werden sollen. Da wir bislang auf dem Branch `mybranch` gearbeitet haben, müssen wir mit dem Arbeitsverzeichnis zunächst wieder auf den *master* bzw. *default* wechseln. Dazu existieren folgende Kommandos:

```
git checkout master
// ---------------------
hg checkout default
```

Um dann einen Abgleich zwischen Branch und *master* bzw. *default* vorzunehmen, gibt man folgendes Kommando ein:

```
git merge mybranch    // Merge vom Branch mybranch
// ---------------------
hg merge mybranch     // Merge vom Branch mybranch
hg commit -m "Merged  Branch mybranch"
```

Im Vergleich zu zentralen Versionsverwaltungen benötigen obige Aktionen wenig Zeit, da keine aufwendigen Zugriffe über Netzwerk erfolgen müssen. Darüber hinaus kann in der Regel deutlich mehr an konkurrierenden Änderungen sinnvoll miteinander kombiniert werden, ohne dass dies Eingriffe des Entwicklers erfordert. Beim Mergen können natürlich dennoch ab und zu Konflikte auftreten, die manuell gelöst werden müssen.

Push und Pull

Nachdem nun die Arbeiten abgeschlossen sind oder einen sinnvollen Zwischenstand

erreicht haben, können und sollten diese mit dem Projekt-Repository abgeglichen werden.[9] Dazu dient der `push`-Befehl. Aber nicht nur man selbst, sondern auch andere Teammitglieder haben oftmals bereits Änderungen durchgeführt und diese möglicherweise schon an das Projekt-Repository übertragen. Um das eigene, lokale Repository mit demjenigen auf dem Projektserver abzugleichen, dient der `pull`-Befehl.

Ähnlich wie bei einem Update in SVN können bei einem `pull` mehrere Änderungen innerhalb der gleichen Datei existieren und zu behandeln sein. In der Regel können diese automatisch gemergt werden, weil DVCS diverse Informationen beim Branching zwischenspeichern. Bei Widersprüchen müssen diese allerdings von Hand aufgelöst werden.

2.3.3 VCS und DVCS im Vergleich

In diesem Abschnitt stelle ich nochmals zusammenfassend die Eigenschaften zentraler Versionsverwaltungen, insbesondere deren Nachteile, und die Vorteile dezentraler Versionsverwaltungen gegenüber und schließe mit einem Fazit.

Nachteile zentraler Versionsverwaltungen

Zentrale Versionsverwaltungen erleichtern zwar unbestritten die Verwaltung der Historie von Dateien enorm, jedoch besitzen sie auch ganz entscheidende Nachteile:

- **Single Point of Failure** – Durch ihre Ausrichtung auf das zentrale Repository besitzen sie einen ***Single Point of Failure***: Sobald das Repository einmal nicht mehr per Netz erreichbar ist, können nahezu keine Interaktionen mehr ausgeführt werden, die im Zusammenhang mit der Versionsverwaltung stehen. Schnell werden dann ganze Teams einige Stunden ausgebremst.
- **Zeitpunkt des Commits** – Es besteht ein Problem, den richtigen Zeitpunkt für einen Commit zu finden: Es ist teilweise recht schwierig zu entscheiden, wann und vor allem auch welche Änderungen eingecheckt und welche Teile besser noch nicht im Repository veröffentlicht werden sollten.

Wahl des richtigen Zeitpunkts zum Commit bei VCS

Intuitiv scheint klar, dass man mit einem Commit so lange warten sollte, bis die gemachten Änderungen einen sinnvollen Zwischenstand erreicht haben, und auch dass nur lauffähiger, möglichst gut getesteter Sourcecode ins Repository integriert werden sollte. Das Ganze hat jedoch folgenden Haken: Wenn man an Erweiterungen arbeitet, so erfüllt der neu erstellte Sourcecode manchmal die genannten Anforderungen (noch) nicht. Dafür gibt es unterschiedliche Gründe, etwa weil nicht alles zu 100% durchdacht ist oder einige Anforderungen unbekannt, unvollständig oder einfach falsch verstanden wurden. Für bestehenden Sourcecode gilt Ähnliches.

Möglicherweise sind Erweiterungen auch nur halbherzig umgesetzt und zudem kaum testbar. Wird derartiger Sourcecode eingecheckt, so ist er danach auch für alle Teammitglieder sichtbar. Das wiederum hat folgende Konsequenzen:

- Wenn man diesen Stand des Sourcecodes eincheckt, bevor er wirklich fertig ist, verärgert man damit eventuell andere oder sorgt dafür, dass keine korrekte Programmversion erstellt werden kann.
- Zögert man stattdessen das Einchecken eine Weile hinaus, so besitzt man für einige Dateien keine Versionshistorie, da lediglich lokal gearbeitet wird. Beim Commit entstehen dann oftmals große und schwer nachvollziehbare Änderungen.
- Wenn man als Ausweg auf einem Branch arbeitet, bis sich die Änderungen stabilisiert haben, so führt dies zu weiteren Problemen. Vor allem empfiehlt es sich, den eigenen Branch mit dem aktuellen Stand abzugleichen, damit man die Änderungen von Kollegen schon im eigenen Branch hat, um so die Reintegration des eigenen Branches in den Hauptstand zu erleichtern.

Manchmal gibt es Situationen, in denen einige Entwickler ihren Sourcecode einige Tage oder gar Wochen nicht ins Repository integrieren oder aber längere Zeit auf einem Branch arbeiten als Folge davon, dass es nicht leicht ist, den Zeitpunkt zu bestimmen, wann Sourcecode »reif« genug für das Repository ist. Je länger der Integrationsvorgang allerdings verzögert wird, desto höher ist die Wahrscheinlichkeit, dass auch Teamkollegen in der Zwischenzeit an denselben Dateien arbeiten. Dadurch gestaltet sich das Zusammenführen der erfolgten Änderungen immer schwieriger.

Während also – wie auch schon erwähnt – viele Branches in zentralen VCS eher Probleme bereiten, als diese zu lösen, adressieren die dezentralen Versionsverwaltungen dieses und andere Probleme.

Vorteile dezentraler Versionsverwaltungen

Mit DVCS wird dieses Szenario extrem entschärft: Jeder Entwickler besitzt lokal sein eigenes Repository. Dort kann er Änderungen ausführen, diese commiten, sie zurückrollen usw., alles ohne Einfluss auf die Teamkollegen. Wenn dann eine gute Qualität vorliegt, können die Modifikationen in ein (zentrales) Remote Repository überspielt werden. Indikatoren für den richtigen Moment sind, dass ein Entwickler mit seinen Änderungen zufrieden ist, die Tests erfolgreich laufen und bestenfalls sogar ein Codereview (Begutachtung des Sourcecodes durch verschiedene Entwickler, vgl. Kapitel 21) stattgefunden hat.

Die beschriebene Arbeitsweise für DVCS entspricht eigentlich genau derjenigen, die man sich als SVN-Benutzer schon immer gewünscht hat, nämlich immer dann, wenn a) keine Netzwerkverbindung bestand und b) man umfangreichere Änderungen durchführen musste und deren Zwischenstände im Repository sichern, aber nicht

allen direkt zugänglich machen wollte. Als Lösung hat man dann in SVN mit sogenannten Feature-Branches gearbeitet. Das läuft so lange gut, wie man nur auf den Branches entwickelt. Allerdings wird das schnell kompliziert, wenn man die Branches wieder zusammenführen muss. Weil DVCS mehr Informationen zu den Dateien speichern, fällt dort das Arbeiten mit Branches und vor allem das Vereinigen von Branches um einiges leichter als mit SVN.

Fazit

Verteilte Versionsverwaltungen erleichtern einem die Arbeit der Versionierung von Dateien sehr. Das gilt selbst dann, wenn man lediglich als Einzelperson lokal arbeiten möchte. Einfacher als mit Git oder Mercurial kann man dies kaum tun. In den letzten Jahren ist auch die Unterstützung durch grafische Tools immer besser geworden. Zur Integration der Versionsverwaltung in den Windows-Explorer sind die Tools TortoiseGit und TortoiseHg praktisch. Beide bieten auch eine grafische Oberfläche.

Teilweise mag in einigen Unternehmen der hohe Verbreitungsgrad von SVN möglicherweise (noch) den Einsatz von DVCS verhindern. Für DVCS spricht jedenfalls, dass man mit einem zentralen Repository, angelehnt an die bisherige Arbeitsweise, arbeiten kann, darüber hinaus aber von den vielfältigen Vorzügen von DVCS profitieren kann. Diese liste ich nachfolgend nochmals auf, um Sie für deren Einsatz zu begeistern und darüber hinaus eine Argumentationshilfe zu bieten, wenn Sie oder Ihre Firma einen Wechsel zu einem DVCS in Betracht ziehen.

- **Geschwindigkeit** -– Das lokale Repository ist vollständig im Dateisystem gespeichert und die meisten Aktionen und Arbeitsschritte erfordern keine Netzwerkzugriffe wie bei CVS oder SVN. Einzig der Abgleich mit Remote Repositories kann ohne Netz nicht erfolgen.
- **Einfache Installation und Nutzung** — Git und Mercurial sind im Internet frei verfügbar. Nach ihrer Installation (und minimaler Konfiguration) sind die beiden DVCS sofort einsatzbereit. Man kann dann mit der Arbeit beginnen, obwohl eventuell noch kein zentrales Repository eingerichtet ist.
- **Offline Arbeiten** -– Wünschenswert ist es häufig, auch offline arbeiten zu können. Insbesondere gilt dies etwa unterwegs, wenn nur eine eingeschränkte Verbindung zum Netz besteht. Zentrale Versionsverwaltungen setzen in der Regel eine funktionierende/ständige Verbindung zum Versionsverwaltungsserver voraus, um Arbeitsschritte durchführen zu können. DVCS erlauben das lokale Arbeiten und auch ein nachträglicher Abgleich mit einem anderen Repository ist (sehr häufig) problemlos möglich.
- **Einfaches Taggen und Branching** — Tagging und Branching sind zwei wichtige Hilfsmittel beim Verwalten von Versionen. Mit CVS oder SVN ist die Arbeit mit Tags und Branches teilweise recht aufwendig oder umständlich. Für DVCS ist das Ganze fast ein Kinderspiel.

- **Einfaches Commiten** -– Commits erfolgen ins lokale Repository und stehen damit nicht im Konflikt mit den Änderungen anderer. Zudem kann man Commits mithilfe der Staging Area vorbereiten. Das kann bei umfangreichen Änderungen hilfreich sein, um diese dann in einem Rutsch zu commiten.
- **Datensicherheit** — Jeder Clone eines Repository ist eine vollständige Kopie. Falls tatsächlich mal ein Datenträgerdefekt o. Ä. auftreten sollte, kann man einen Stand aus einem anderen Repository wiederherstellen (sofern es einen Clone gibt).

2.4 Einsatz eines Unit-Test-Frameworks

Um eine ausreichende Softwarequalität sicherzustellen, müssen Programme nach verschiedenen Kriterien geprüft werden. Eine mögliche Qualitätssicherungsmaßnahme auf Sourcecode-Ebene sind sogenannte Unit Tests, die hier kurz vorgestellt werden. Ausführlich wird das Thema Unit-Testen dann in Kapitel 20 besprochen. Ziel beim Unit-Testen ist es, das korrekte Arbeiten einer kleinen Einheit (Unit) – normalerweise einer Java-Klasse – für sich zu untersuchen: Das Zusammenspiel mit anderen Softwarekomponenten wird bei dieser Art von Test gewöhnlich nicht betrachtet. Das erleichtert den Fokus auf einen klar abgegrenzten Bereich der Funktionalität, wodurch sich gelieferte Ergebnisse besser mit gewünschten Resultaten vergleichen lassen.

Die erwarteten Ergebnisse und die korrespondierenden Testfälle werden in Form von kleinen Java-Klassen realisiert. Tests werden also nicht nur durchgeführt, sondern zunächst programmiert. So versucht man, auch die Entwickler zum Testen zu bewegen, die sich sonst gerne vor Tests drücken wollen. Gar keine Tests durchzuführen, stellt allerdings eine unangebrachte Art von Bequemlichkeit dar: Als Entwickler sollen Sie zumindest dafür sorgen, dass für Tester eine umfassende funktionale Prüfung der Applikation möglich ist, ohne dass die Tester dabei ständig über einfache Programmierfehler stolpern.

2.4.1 Das JUnit-Framework

JUnit ist ein in Java geschriebenes Framework, das beim Schreiben und bei der Automatisierung von Testfällen auf Klassenebene unterstützt.[10] Es ist durch seinen einfachen Aufbau leicht erlernbar und nimmt viel Arbeit beim Schreiben und Verwalten von Testfällen ab: Praktischerweise kümmert sich das Framework bereits um Dinge wie das Zählen und Berichten von Fehlern, sodass nur die Logik für die Testfälle selbst zu implementieren ist. Dabei unterstützt das Framework durch Methoden, mit denen Testbehauptungen aufgestellt und ausgewertet werden können.

Beispiel: Ein erster Unit Tests mit JUnit 4

Zum Test einer Applikationsklasse wird normalerweise eine korrespondierende Testklasse geschrieben. Häufig beginnt man zur Absicherung wichtiger Funktionalität eigener Klassen damit, zunächst einige zentrale Methoden durch Tests zu überprüfen. Dies kann anschließend schrittweise ausgeweitet werden. Dazu werden Testfälle in Form spezieller Testmethoden erstellt, die mit der Annotation[11] `@Test` markiert und `public` sein müssen. Zudem dürfen diese Methoden keine Parameter und keinen Rückgabewert besitzen. Ansonsten werden sie von JUnit nicht als Testfall betrachtet und bei der Testausführung ignoriert.

Schauen wir auf ein sehr einfaches erstes Beispiel, das lediglich das Gesagte verdeutlicht, aber noch keine Funktionalität testet:

```
import static org.junit.Assert.*;

import org.junit.Test;

public class FirstTestWithJUnit4
{
    @Test
    public void test()
    {
        fail("Not yet implemented");
    }
}
```

Im Listing sieht man den Import verschiedener Klassen und der Annotation `@Test` aus dem Package `org.junit`. Verschiedene Methoden der Klasse `Assert` werden statisch importiert. Damit erreicht man eine bessere Lesbarkeit beim Aufruf der Testmethoden.

Unit Tests mit Eclipse erstellen

Praktischerweise kann man sich ein derartiges Grundgerüst für einen Unit Test von Eclipse erzeugen lassen. Dazu wählt man im Kontextmenü des jeweiligen Projekts NEW –> JUNIT TEST CASE. Daraufhin erscheint ein Dialog, in dem man verschiedene Einstellungen vornehmen kann. Das ist exemplarisch in Abbildung 2-6 gezeigt.

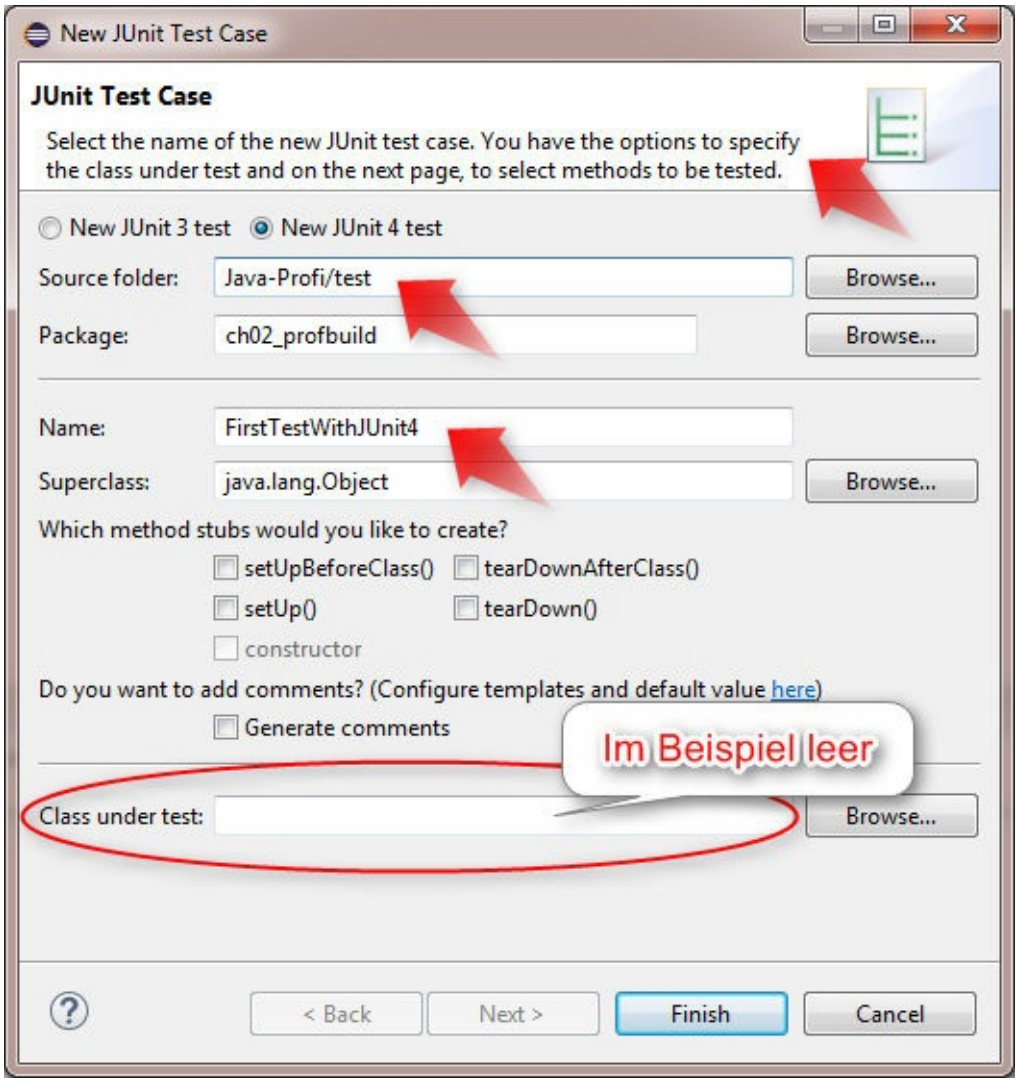


Abbildung 2-6 *Erstellen eines JUnit-Tests in Eclipse*

Man kann den Namen des Tests festlegen und angeben, ob der Test auf JUnit 3 oder 4 basieren soll. Wir nutzen das aktuellere JUnit 4 und verwenden als Namen `FirstTestWithJUnit4`. Auch sollten wir darauf achten, die Tests in einem speziellen Verzeichnis für Tests abzulegen, also je nach Verzeichnislayout etwa im Verzeichnis `test` oder `src/test/java`. Bestätigen wir den Dialog, so wird die im obigen Listing gezeigte Testklasse erzeugt, in der wir nun Testmethoden ergänzen könnten. Des Weiteren wird man im Normalfall im Textfeld `Class under test` die zu testende Klasse anwählen. Weil wir hier zur Einführung keinen Test einer Applikationsklasse, sondern nur einen minimalen Test als Ausgangspunkt erstellen, bleibt dieses Eingabefeld leer.

Tipp: Benennung von Testklassen und von Testmethoden

Die Klassen von Unit Tests sollten den Namen der zu testenden Source-Datei enthalten, um diese leicht miteinander zu assoziieren. ***Sinnvoll ist es***, **`Test`** ***als Postfix für die Testklasse zu verwenden***, so würde man etwa dem Test für die Klasse `MyClass` den Namen `MyClass`**`Test`** geben. Basierend auf dem Klassennamen findet man sehr schnell den entsprechenden Test. Würde man dagegen das Präfix `Test` für die Testklasse nutzen, so lassen sich Tests nur mit viel Aufwand finden: Alle Namen von Testklassen beginnen dann mit dem Präfix `Test`

und unterscheiden sich erst nach dem gemeinsamen Präfix.

Benennung von Testmethoden Mit JUnit 3 mussten alle Testmethoden mit dem Präfix `test` beginnen, um vom Framework erkannt und ausgeführt zu werden. Seit JUnit 4 ist das nicht mehr erforderlich, da dessen Architektur auf Annotations umgestellt wurde. Ich verwende zusätzlich zur Annotation auch weiterhin das Präfix `test`. Das ist nicht notwendig, erleichtert mir aber die Arbeit. So erreiche ich eine bessere Trennung zwischen Hilfsmethoden in der Testklasse und solchen, die Testfälle darstellen.

Schreiben und Ausführen von Tests

Nachdem wir nun wissen, wie man einfache Unit Tests erstellt, möchte ich das Ganze ausbauen. Wir lernen dazu u. a. folgende Funktionalitäten aus dem JUnit-Framework zum Ausführen von Tests bzw. zur vorherigen Definition von Testfällen inklusive der Auswertung von Bedingungen in Testfallmethoden kennen:

- **Auswertung von Bedingungen** – Die Klasse `Assert` stellt eine Menge von Prüfmethoden bereit, mit denen Bedingungen formuliert und dadurch Zusicherungen über den zu testenden Sourcecode geprüft werden können:
  - Durch Aufruf der überladenen Methoden `assertTrue()` und `assertFalse()` lassen sich boolesche Bedingungen prüfen. Erstere Methode geht davon aus, dass eine Bedingung erfüllt ist, und meldet ansonsten einen Fehler. Für `assertFalse()` gilt das Gegenteil.
- Mit den überladenen Methoden `assertNull()` bzw. `assertNotNull()` können Objektreferenzen auf `null` bzw. ungleich `null` geprüft werden.
- Die überladene Methode `assertEquals()` ermöglicht es, sowohl zwei Objekte auf inhaltliche Gleichheit (Aufruf von `equals(Object)`) als auch zwei Variablen primitiven Typs auf Gleichheit zu prüfen. Aufgrund möglicher Rundungsungenauigkeiten bei Berechnungen für die Typen `float` und `double` kann eine maximale Abweichung vom erwarteten Wert angegeben werden.
- Mithilfe der überladenen Methoden `assertSame()` bzw. `assertNotSame()` können Objektreferenzen auf Gleichheit bzw. Ungleichheit gemäß == geprüft werden.
- Über `fail()` kann man einen Testfall bewusst fehlschlagen lassen. Dies ist nützlich, um auf eine unerwartete Situation zu reagieren. Etwa wenn beim Parsing von Zahlen für eine ungültige Eingabe wider Erwarten keine Exception ausgelöst wird. Sollten bei der Abarbeitung von Testfällen Exceptions auftreten, erlaubt es JUnit 4, derartige Exceptions in der Annotation `@Test` mit dem Attribut `expected` zu spezifizieren.

Das initial erzeugte Unit-Test-Grundgerüst erweitern wir nun: Folgendes Listing zeigt einige der vorgestellten Methoden im Einsatz, wobei verschiedene

Testmethoden in diesem Beispiel bewusst Fehler provozieren:

```
import static org.junit.Assert.*;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

import org.junit.Test;

public class TestExample
{
    @Test
    public void testAssertTrue()
    {
        final List<String> names = new ArrayList<>();
        names.add("Max");
        names.add("Moritz");
        names.clear();
        assertTrue(names.isEmpty());
    }

    @Test
    public void testAssertFalse()
    {
        final List<Integer> primes = Arrays.asList(2, 3, 5, 7);
        //  Hier wird bewusst ein Fehler provoziert
        assertFalse(primes.contains(7));
    }

    @Test
    public void testAssertNull()
    {
        assertNull(null);
    }

    @Test
    public void testAssertNotNull()
    {
        // Hier wird bewusst ein Fehler provoziert
        assertNotNull("Unexpected null value", null);
    }

    @Test
    public void testAssertEquals()
    {
        assertEquals("EXPECTED", "expected".toUpperCase());
    }

    @Test
    public void testAssertEqualsWithPrecision()
    {
        assertEquals(2.75, 2.74999, 0.1);
    }

    @Test(expected = java.lang.NumberFormatException.class)
    public void testFailWithExceptionJUnit4()
    {
        // Hier wird bewusst ein Fehler provoziert
        final int value = Integer.parseInt("Fehler simulieren!");
        fail("calculation should throw an exception!");
```

```
    }
}
```

- **Testausführung** – JUnit ist in Eclipse integriert und erlaubt die Ausführung von Tests direkt aus der IDE. Tests kann man entweder über ein Kontextmenü oder über Buttons im GUI ausführen. Dabei kommt es zu einer Ausgabe ähnlich zu Abbildung 2-7. Ein roter Balken zeigt aufgetretene Fehler an. Im Idealfall sieht man ein »beruhigendes« Grün, das den erfolgreichen Abschluss aller Testfälle meldet.

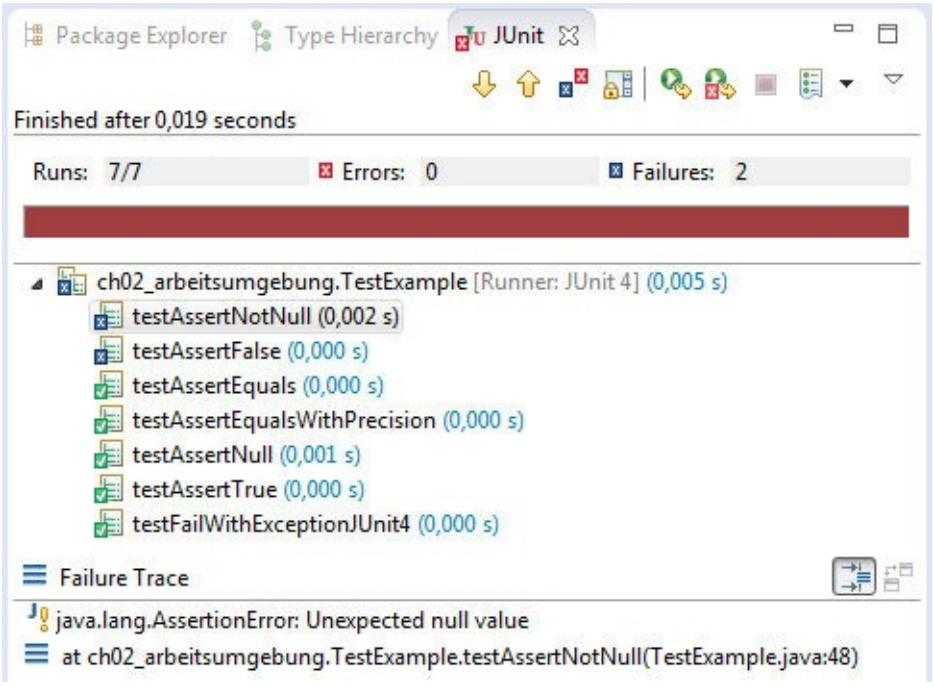


Abbildung 2-7 *Testausführung aus dem GUI der IDE*

Gleitkommawerte mit Abweichung testen

Bei Berechnungen mit Gleitkommazahlen kann es zu Rundungsfehlern kommen. Gerade im Bereich von Unit Tests ist es daher wünschenswert, eine kleine Abweichung von dem erwarteten Ergebnis noch als valide akzeptieren zu können. Im obigen Beispiel haben wir dies schon für fixe Zahlenwerte gesehen. Für Berechnungen ist das mit JUnit wie folgt möglich:

```
@Test
public void testDoubleWithDeviation()
{
    final double EPSILON = 0.0001;

    // Hier nur exemplarisch für eine komplexe Berechnung
    final double result = 10.0 / 3.0* 100;

    assertEquals(333.3333, result, EPSILON);
}
```

2.4.2 Vorteile von Unit Tests

Nachdem wir einen ersten Einblick in das Unit-Testen gewonnen haben, möchte ich

kurz mögliche positive Auswirkungen auf die Entwicklung sowie die Qualität nennen, um Sie für das Erstellen von Unit Tests zu ermuntern.

Regressionstest

Die Integration von Unit-Test-Frameworks in IDEs ermöglicht es, Unit Tests parallel zur eigentlichen Anwendung zu entwickeln und durchzuführen, ohne dafür die Arbeitsumgebung verlassen zu müssen.

Einmal programmierte Unit Tests sollten beliebig oft wiederholt werden können. Diese Form des Testens wird ***Regressionstest*** genannt. Nach Änderungen an der Software kann man durch erneute Ausführung der Unit Tests sicherstellen, dass auch weiterhin die erwarteten Resultate geliefert werden. Allerdings wäre es aufwendig, wenn man nach jeder kleineren Änderung alle Tests von Hand ausführen müsste. Hierfür bietet sich eine Automatisierung an. Die Unit Tests können dazu sowohl als ein Schritt im später vorgestellten Build-Prozess ausgeführt werden als auch über Tools parallel während des Entwickelns in der IDE (vgl. Abschnitt 20.8.3).

Positive Effekte von Unit Tests

Je komplexer Programme werden, desto wahrscheinlicher enthalten diese auch Fehler. Durch die entwicklungsbegleitende Erstellung von Unit Tests kann man bereits viele davon noch während der Implementierung – also frühzeitig – finden und beseitigen. Warum ist das ratsam? Fehler, die erst sehr spät im Entwicklungszyklus, eventuell sogar erst beim Kunden, gefunden werden, verursachen wesentlich mehr Ärger und Kosten, als wenn dies frühzeitig noch während der Entwicklung geschieht, etwa durch Unit Tests. Der Grund ist recht offensichtlich: Wenn Fehler erst kurz vor Auslieferungen oder gar erst vom Kunden gefunden werden, behindert es möglicherweise dessen tägliche Arbeit und verärgert ihn. Zudem lassen sich die Ursachen für den Fehler häufig nur schwierig nachvollziehen und es erfordert aufwendige Analysen und Nachtests, bis die Ursachen aufgedeckt werden können. Findet und korrigiert man Fehler dagegen noch während der Entwicklung, so spart dies einiges an Aufwand: Analysen sind in der Regel viel einfacher, weil die Funktionalität erst vor Kurzem entwickelt wurde und meistens mehr Informationen verfügbar sind. Außerdem kommt es auch zu weniger Ärger, da der fehlerbehaftete Softwarestand den Kunden gar nicht erst erreicht, wodurch dieses Vorgehen also auch unzufriedene Kunden vermeidet.

Fazit

In diesem Abschnitt wurde das Unit-Testen mithilfe des JUnit-Frameworks kurz vorgestellt. Damit haben Sie die Grundlagen für das Schreiben eigener Unit Tests kennengelernt und sollten die nachfolgenden Beispiele verstehen können. In Kapitel 20 gehe ich dann intensiver auf das Thema Testen mit JUnit ein. Weiter gehende

Informationen, Artikel sowie Tipps und Tricks finden Sie z. B. unter `www.junit.org`.

Hinweis: Das TestNG-Framework als Alternative zu JUnit

Wenn Sie die Arbeitsweise von JUnit verstanden haben, ist es nicht schwierig, diese auf TestNG zu übertragen. Weitere Informationen zu TestNG finden Sie unter `http://testng.org/doc/documentation-main.html`.

2.5 Debugging

Selbst wenn während der Entwicklung gründlich getestet wird und dazu diverse Tests existieren, die im besten Fall sehr viel Funktionalität des Programms überprüfen, verbleiben normalerweise immer noch einige Fehler im Programm. Um solche hartnäckigen Fehler aufzuspüren, ist es wünschenswert, die Programmabläufe im Detail, also am besten Schritt-für-Schritt, während der Ausführung beobachten zu können. Dazu ist der Einsatz eines sogenannten Debuggers (zu deutsch »Entwanzer«[12]) sinnvoll. Im JDK ist ein einfacher, kommandozeilenbasierter Debugger, der `jdb`, integriert. Allerdings ist dessen Einsatz kryptisch und mühselig und daher nur in Ausnahmefällen angebracht, zumal ein Qualitätsmerkmal aktueller IDEs genau darin besteht, leistungsfähige Debugger mit komfortabler, grafischer Benutzeroberfläche mitzuliefern. Dies gilt gleichermaßen für NetBeans, IntelliJ IDEA und Eclipse. Bevor ich konkret auf das Thema Debugging eingehe, möchte ich noch ein paar Details zur Fehlersuche darstellen.

Notwendige Informationen für eine sinnvolle Fehlersuche

Trotz aller Anstrengungen zum Erreichen einer guter Qualität werden irgendwann bis dahin unentdeckte Programmfehler auftreten. Für deren Behebung und die dazu notwendige vorangehende Analyse ist es meistens sehr hilfreich, wenn man möglichst umfangreiche Informationen zum Fehlverhalten besitzt.

Solange Programme einwandfrei funktionieren, wird man sich seltener auf Fehlersuche begeben als bei einem akut vorliegenden Problem. Nichtsdestotrotz sollte man immer auch einen Teil seiner Arbeitszeit in eine kontinuierliche Verbesserung bzw. Prüfung der bestehenden Implementierung stecken, um präventiv möglichen Fehlern vorzubeugen. Dazu kann man beispielsweise Abfragen und Prüfungen in das Programm integrieren, um die Validität von Eingabeparametern öffentlicher Methoden zu überprüfen und sicherzustellen.

Eine möglicherweise notwendige Fehleranalyse wird deutlich erleichtert, wenn die Historie der Methodenaufrufe (Stacktrace) bekannt ist, die zur Fehlersituation geführt hat. Diese Information besitzt man immer dann, wenn eine Exception ausgelöst wurde. Daher empfehle ich, in eigenen Programmen (schwerwiegende) Fehlersituationen durch Exceptions zu kommunizieren. Diese dienen sowohl dazu,

den Programmablauf in einer geeigneten Behandlung wieder auf Kurs zu bekommen als auch aussagekräftige Fehlermeldungen für den Kunden oder für Entwickler zu generieren.

Hilfreich für eine Problemanalyse ist zudem, die Exception mit möglichst viel Kontextinformationen anzureichern. Dort sollten etwa Angaben zu Wertebelegungen von Attributen, lokalen Variablen und Übergabeparametern usw. gemacht werden – aber natürlich beschränkt auf den relevanten Zustand! Mithilfe des zur Verfügung stehenden Stacktrace kann man die verursachende Stelle vielfach recht gut eingrenzen, und die bereitgestellten Kontextinformationen helfen bei der weiteren Analyse. Auf diese Weise lassen sich dann in der Regel die Ursachen für Fehler leichter finden.

Leider ist die Fehlersuche in der Praxis dann doch häufig noch etwas komplizierter. Nach der Korrektur offensichtlicher oder per ausgelöster Exception auffindbarer Programmfehler wird es zunehmend schwieriger, verbliebene Fehler zu entdecken. Beispielsweise äußern sich diese hartnäckigen Fehler dadurch, dass Berechnungen falsche Ergebnisse liefern – jedoch nur unter ganz speziellen Rahmenbedingungen. Alle diese Situationen und Randfälle im Voraus zu bedenken, wird kein Mensch schaffen. Was kann man also tun? Für solche Fälle ist der Einsatz von Debugging-Tools sinnvoll. Daher wenden wir uns nun diesem Thema zu.

Meinung: Fehlerbehandlung und Exceptions

Manchmal hört man, dass Exceptions etwas Schlechtes und zu Vermeidendes sind. Aber oftmals ist es zur Fehlerbehebung hilfreich, wenn bei schwerwiegenden Fehlern oder Problemen aussagekräftige Exceptions ausgelöst werden. Allerdings gilt diese derart positiv formulierte Aussage nur während der Entwicklungsphase und des Tests. Nach Auslieferung an Kunden sind Exceptions immer ärgerlich – auch wenn die Fehlersituation mit Stacktrace deutlich besser analysierbar ist als ohne.

2.5.1 Fehlersuche mit einem Debugger

Ein in die IDE integrierter Debugger ist ein komfortables Hilfsmittel, das die Fehlersuche zum Teil enorm vereinfachen kann. Dafür besonders praktisch ist es, dass die Ausführung eines Programms an nahezu beliebiger Stelle angehalten, die Belegung von Variablen angeschaut, überprüft (und sogar verändert) und das Programm anschließend fortgesetzt werden kann. Eine Programmstelle, an der die Ausführung unterbrochen werden soll, nennt man ***Breakpoint***. Einen solchen kann man im Editor der IDE für die gewünschte Programmzeile setzen (in Eclipse erscheint durch einen Doppelklick ein kleiner blauer Punkt), einen bestehenden Breakpoint kann man auf die gleiche Weise wieder entfernen.[13] Abbildung 2-8 zeigt einen Programmausschnitt einer Methode `doActionForAll()` einer Klasse `GameModel` mit vier gesetzten Breakpoints.

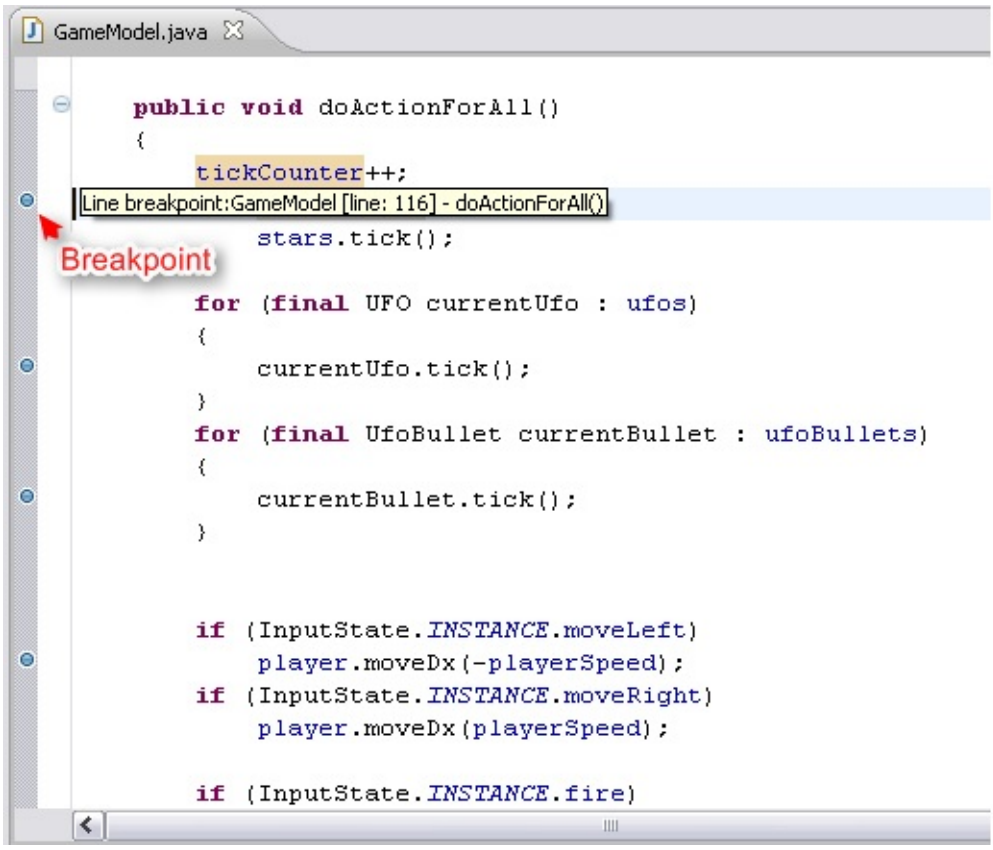


Abbildung 2-8 *Programmausschnitt mit vier gesetzten Breakpoints*

Um ein Programm in dieser speziellen Debugging-Ausführungsart ablaufen zu lassen und untersuchen zu können, muss es im Debug-Modus (F11) und nicht wie üblich per Run (CTRL+F11) gestartet werden. Im Debug-Modus wird die Ausführung des Programms an den zuvor im Editor gesetzten Breakpoints unterbrochen. In der folgenden Abbildung 2-9 ist dies für die Methode `doActionForAll()` gezeigt.

Im linken oberen Teil des Bildschirms ist die Aufrufhierarchie dargestellt. Rechts daneben sind zwei Views (als Karteikartenreiter) platziert, die sowohl die aktuellen Belegungen der Variablen als auch eine Übersicht über definierte Breakpoints bieten.

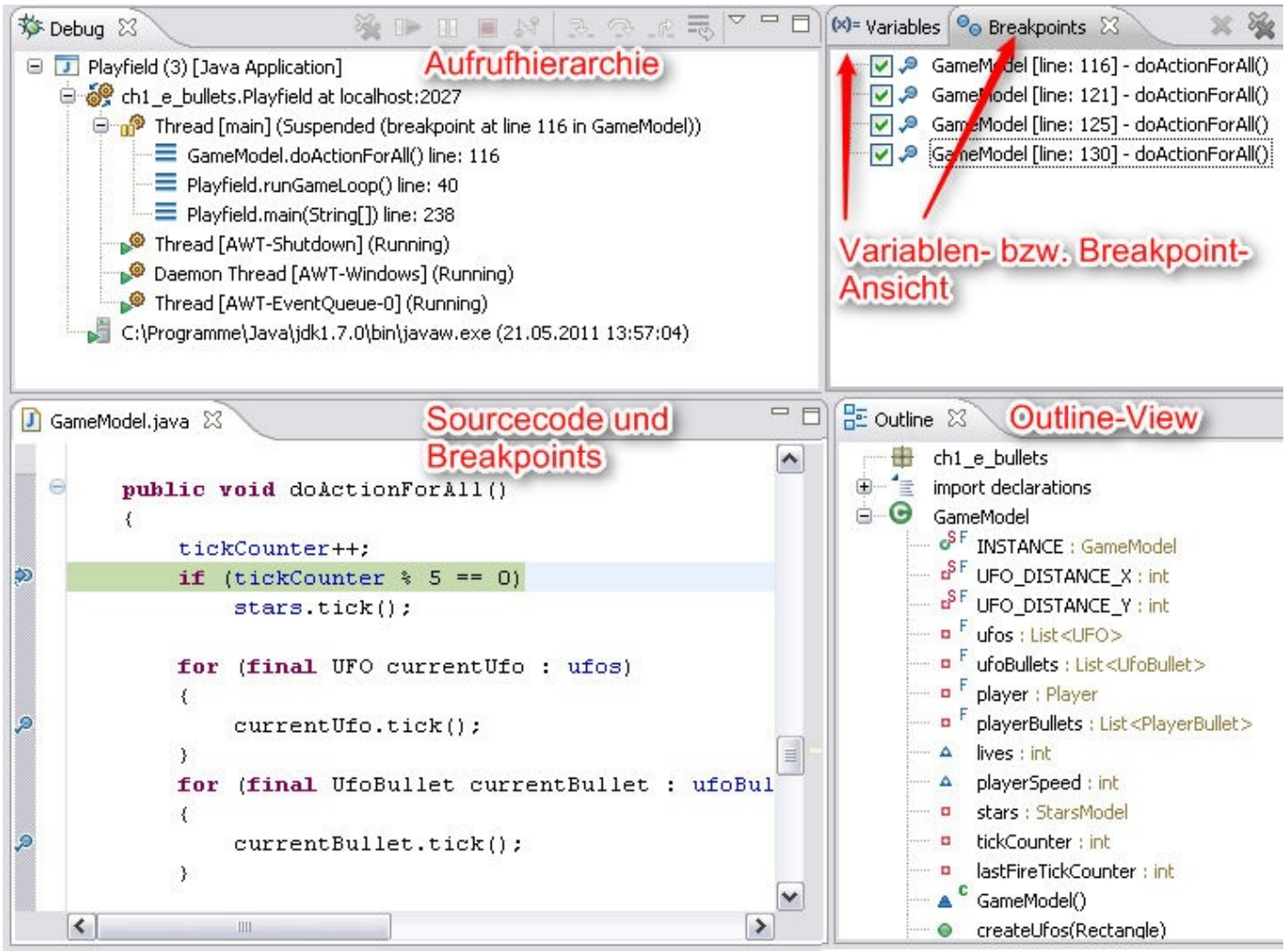


Abbildung 2-9 *Beispiel einer Debug-Session*

Im linken unteren Teil sehen wir den Programmausschnitt, in dem die Zeile markiert ist, in der das Programm unterbrochen wurde. Rechts daneben befindet sich die sogenannte Outline-View, die Informationen zu der hier untersuchten Klasse `GameModel` zeigt. Diese etwas ungewöhnliche Anordnung wurde hier lediglich zu Demonstrationszwecken gewählt, um die einzelnen Views in kompakter Form beschreiben zu können. Die Anordnung der Views lässt sich je nach Bedarf beliebig ändern und konfigurieren.

Aktionen beim Debugging

Wenn das Programm gerade angehalten ist (oder genauer: die JVM die Ausführung unterbrochen hat), können wir nun mit dem Debugging bzw. der Fehleranalyse beginnen. Uns stehen unter anderem folgende Optionen zur Verfügung:

- **RESUME (F8)** – Das Programm wird bis zum nächsten Breakpoint ausgeführt, falls es einen solchen im Ausführungspfad gibt. Im Beispiel würden wir dann auf dem Breakpoint der Zeile `currentUfo.tick();` landen.
- **STEP INTO (F5)** – Befinden wir uns in einer Zeile mit einem Methodenaufruf, so wird zur ersten Zeile der Methode – im Beispiel der aufgerufenen Methode `stars.tick()` – gesprungen und dort die Ausführung in der ersten Zeile

angehalten. [14] Enthält die Programmzeile keinen Methodenaufruf, dann stoppt der Debugger an der nächsten auszuführenden Zeile der aktuellen Methode oder dem letzten Befehl einer Methode.

- **STEP RETURN (F7)** – Wurde mit STEP INTO in eine Methode gesprungen, so kann man mit dieser Funktion das Debugging einer derart angesprungenen Methode aussetzen, d. h., die folgenden Zeilen werden ausgeführt, und der Debugger läuft bis direkt hinter die vorherige Aufrufstelle.
- **STEP OVER (F6)** – Es wird die aktuelle Programmzeile ausgeführt. Handelt es sich dabei um einen Methodenaufruf, so wird die entsprechende Methode nicht im Debugger angesprungen, sondern vollständig abgearbeitet. Die Programmausführung wird an der nächsten auszuführenden Zeile nach der aktuellen Zeile gestoppt oder an einem anderen Breakpoint, falls es einen im Aufrufpfad gibt.
- **RUN TO LINE (CTRL+R)** – Soll die Ausführung bis zu einer speziellen Zeile fortgesetzt werden, so können wir den Cursor auf die gewünschte Zeile bewegen und dann diese Funktion auswählen.
- **TERMINATE (CTRL+F2)** – Die Ausführung des Programms wird beendet.

Alternativ zu den in der Aufzählung angegebenen Tastaturkürzeln können wir die Aktionen mithilfe korrespondierender Buttons der Toolbar ausführen. In Abbildung 2-10 ist dies für die Aktion STEP INTO gezeigt. Im unteren rechten Teil ist die Darstellung der Variablenbelegung zum Ausführungszeitpunkt zu sehen. Wir können beobachten, ob sich alle Werte gemäß unseren Erwartungen verändern und sogar bei Bedarf einzelne Werte während der Programmausführung modifizieren.

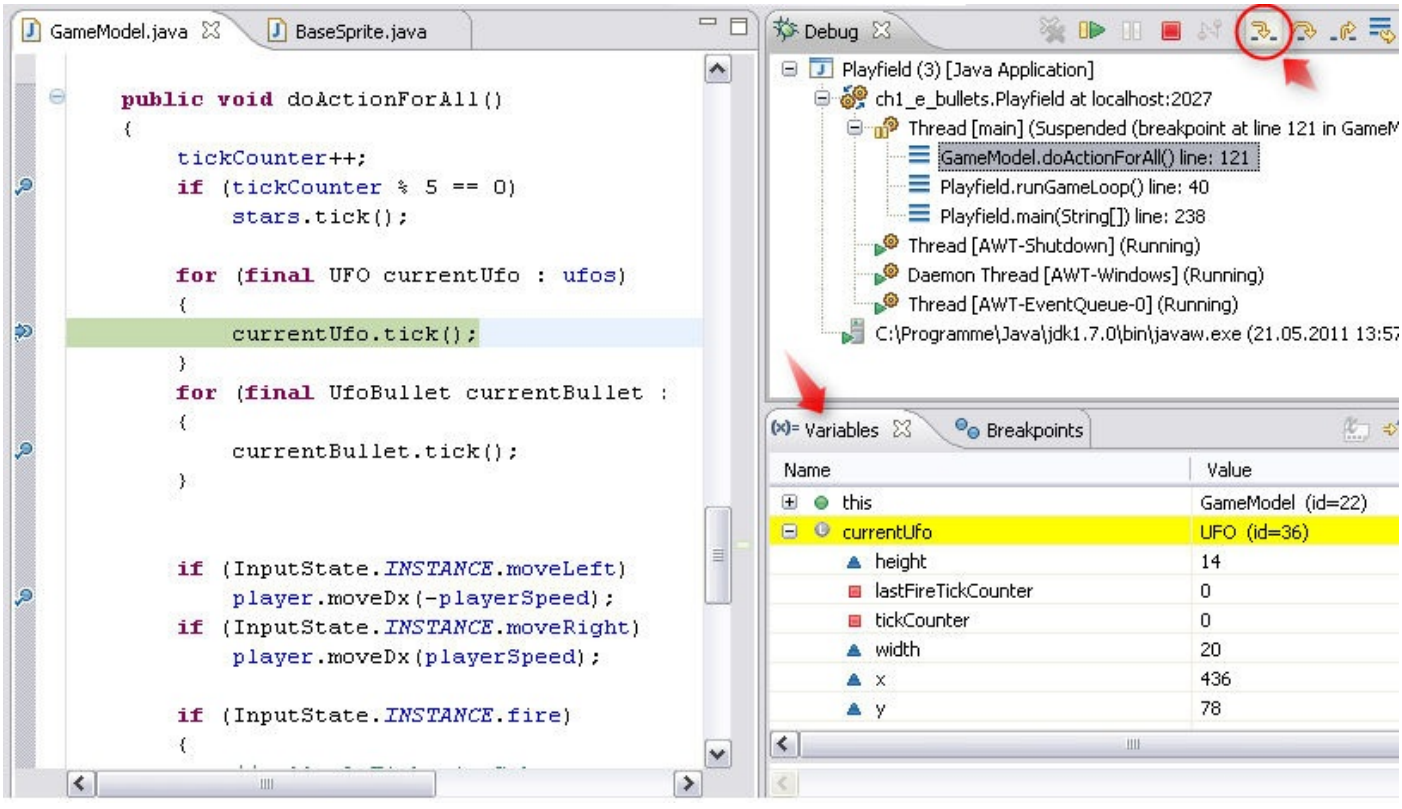


Abbildung 2-10 *Beispiel einer Debug-Session mit Auswahl von* STEP INTO

2.5.2 Remote Debugging

Die vorangegangenen Abschnitte haben einen Einstieg in die Thematik Debugging gegeben. Bisher haben wir aber nur solche Programme betrachtet, die direkt in unserer IDE laufen. Oftmals ist es jedoch wünschenswert, schon in Betrieb befindliche Programme – meistens solche, die bereits an Kunden oder die Testabteilung ausgeliefert wurden – mit einem Debugger zu untersuchen. Wenn ein Programm in einem Testsystem oder sogar beim Kunden im Einsatz ist, kann man Fehlern nicht mehr nachspüren, indem man das Programm direkt aus der IDE im Debug-Modus startet. Das ist schade, da eine Fehlersuche durch eine schrittweise Analyse des Programmablaufs leichter möglich ist. Durch einen Trick wird jedoch praktischerweise das Debugging separat gestarteter Java-Programme mit dem Eclipse-Debugger möglich. Voraussetzung dazu ist lediglich eine TCP/IP-Verbindung zum Zielsystem und eine entsprechende Parametrierung der JVM beim Start auf diesem System.

Parametrierung der JVM zum Remote Debugging

Die JVM bietet spezielle Parameter, die es erlauben, das in einer JVM ausgeführte Programm mithilfe eines externen Debuggers zu untersuchen. Dieser wird in der Regel in einer IDE ausgeführt und verbindet sich über ein Netzwerk mit der programmausführenden JVM. Dazu muss für diese JVM unter anderem einen Port für das Remote Debugging festgelegt werden, mit dem sich die IDE später verbinden kann. Zum Start einer programmausführenden JVM im Debug-Modus müssen folgende Parameter übergeben werden, wobei die Werte gegebenenfalls anzupassen sind:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777
```

Als Hilfestellung für möglicherweise erforderliche Anpassungen sind in der nachfolgenden Aufzählung die genutzten Optionen von `-agentlib:jdwp` beschrieben:[15]

- `transport=dt_socket` – Aktiviert die Kommunikation des Debuggers mit der JVM über Sockets. Normalerweise wird die IDE lokal ausgeführt und verbindet sich über ein Netzwerk mit einer programmausführenden JVM.
- `server=y/n` – Regelt, ob auf das Einklinken eines Debuggers gehorcht werden soll. Bei der Option `n` wird versucht, einen Debugger an der unter `address` angegebenen Adresse zu finden. Das ist in der Praxis eher ungewöhnlich.
- `suspend=y/n` – Bestimmt, ob die JVM mit der Ausführung wartet, bis sich ein Debugger mit ihr verbunden hat oder nicht. Die Einstellung `n` ist sinnvoll, wenn man sich in einer Testphase befindet oder nicht direkt beim Applikationsstart ein Remote Debugging durchführen möchte, sondern nur bei Bedarf.
- `address=7777` – Legt den Port fest, über den sich der Debugger und die JVM

verbinden. Es ist ratsam, hier einen leicht zu merkenden, immer gleichen Port zu nutzen (abgesehen vielleicht von Security-Bedenken).

Hinweis: Remote Debugging vor JDK 5

In der Praxis trifft man immer noch ältere Aufrufe ähnlich zu Folgendem an:

```
java -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777
```

Dieser Aufruf funktioniert problemlos auch mit neueren JDKs, besitzt aber eine leicht unterschiedliche Parametersyntax:

- `-Xdebug` – Startet die JVM mit Unterstützung von JVMDI (Java Virtual Machine Debug Interface). Dieses ist seit JDK 5 veraltet, und man sollte besser die zuvor beschriebene Kommandozeilenoption `-agentlib:jdwp` wählen.
- `-Xrunjdwp:transport=dt_socket, server=y, suspend=n, address=7777` – Diese Angabe ist nahezu identisch zu der neueren Variante. Der Unterschied liegt in der Syntax `-Xrunjdwp:` statt `-agentlib:jdwp=`.

Remote Debugging in Eclipse an einem Beispiel

An einem einfachen Beispiel wollen wir nachvollziehen, dass Remote Debugging recht leicht zu bewerkstelligen ist. Hier geht es wirklich nur darum, zu demonstrieren, wie man Remote Debugging ausführt. Es hat ansonsten keinen praktischen Nutzen!

Wir beginnen mit der zu debuggenden Klasse `RemoteDebuggingExample`. In deren simpler `main()`-Methode wird lediglich von 0 bis 49 gezählt, der aktuelle Wert auf der Konsole ausgegeben und danach jeweils 10 Sekunden Pause gemacht:

```
import java.util.concurrent.TimeUnit;

public class RemoteDebuggingExample
{
    public static void main(final String[] args) throws InterruptedException
    {
        for (int i = 0; i < 50; i++)
        {
            print(i);
        }
    }

    public static void print(final int i) throws InterruptedException
    {
        System.out.println("i=" + i);
        TimeUnit.SECONDS.sleep(10);
    }
}
```

Start des Programms mit JVM-Debug-Einstellungen

Der Start des Programms erfolgt auf dem Zielsystem (was für unseren einfachen Test

identisch mit dem eigenen Rechner sein kann) mit den bekannten JVM-Debug-Parametern:

```
java -agentlib:jdwp=transport=dt_socket,server=y,suspend=n,address=7777
      RemoteDebuggingExample
```

Statt eines Starts von der Kommandozeile bietet es sich an, in Eclipse eine Ausführungskonfiguration inklusive `main()`-Klasse zu erstellen. Das zeigt Abbildung 2-11.

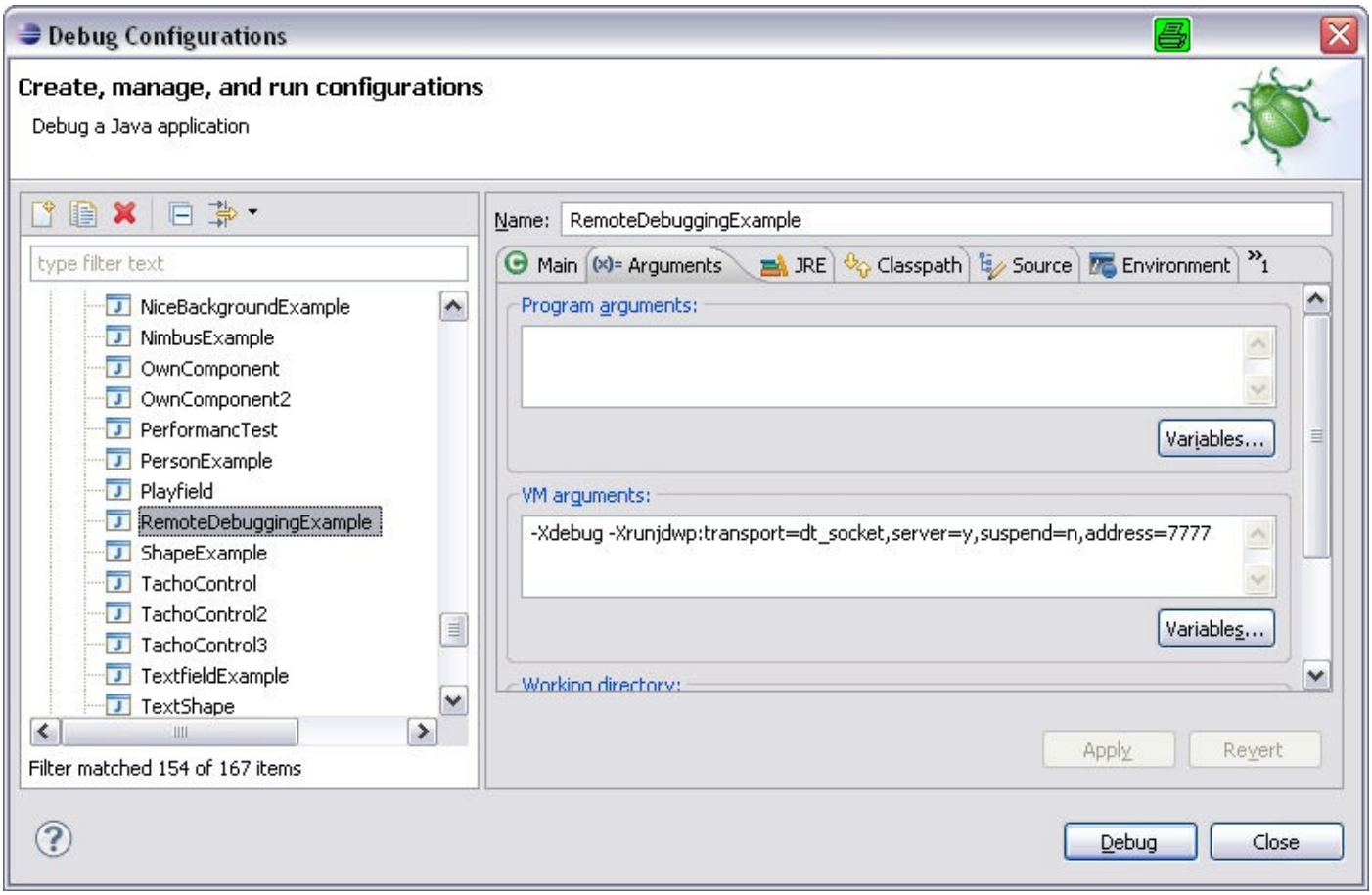


Abbildung 2-11 *Beispiel einer Einstellung zum Starten der Applikation*

Nachdem das Programm im Debug-Modus per Kommandozeile oder aus Eclipse heraus gestartet wurde, profitieren wir für die Beispielapplikation davon, dass eine Pause von 10 Sekunden recht groß gewählt ist. Diese Wartezeit erlaubt es uns, in Ruhe mithilfe des Eclipse-Debuggers eine Verbindung zu derjenigen JVM aufzubauen, die unser Beispielprogramm im Debug-Modus ausführt. Damit das Andocken des Debuggers an eine JVM möglich wird, müssen wir eine Ausführungskonfiguration »Remote Java Application« in Eclipse anlegen, wie dies im folgenden Abschnitt beschrieben ist.

Einstellungen zum Remote Debugging in Eclipse vornehmen

Um ein Programm remote zu debuggen, muss dafür eine Ausführungskonfiguration erstellt werden. Anschließend sind einige Parametrierungen vorzunehmen. Dazu wählt man in Eclipse das Menü RUN –> DEBUG CONFIGURATIONS…Im Konfigurationsdialog müssen die zuvor für die Applikation verwendeten Angaben zu

Port und Rechner eingegeben werden. Das ist beispielhaft in Abbildung 2-12 dargestellt.

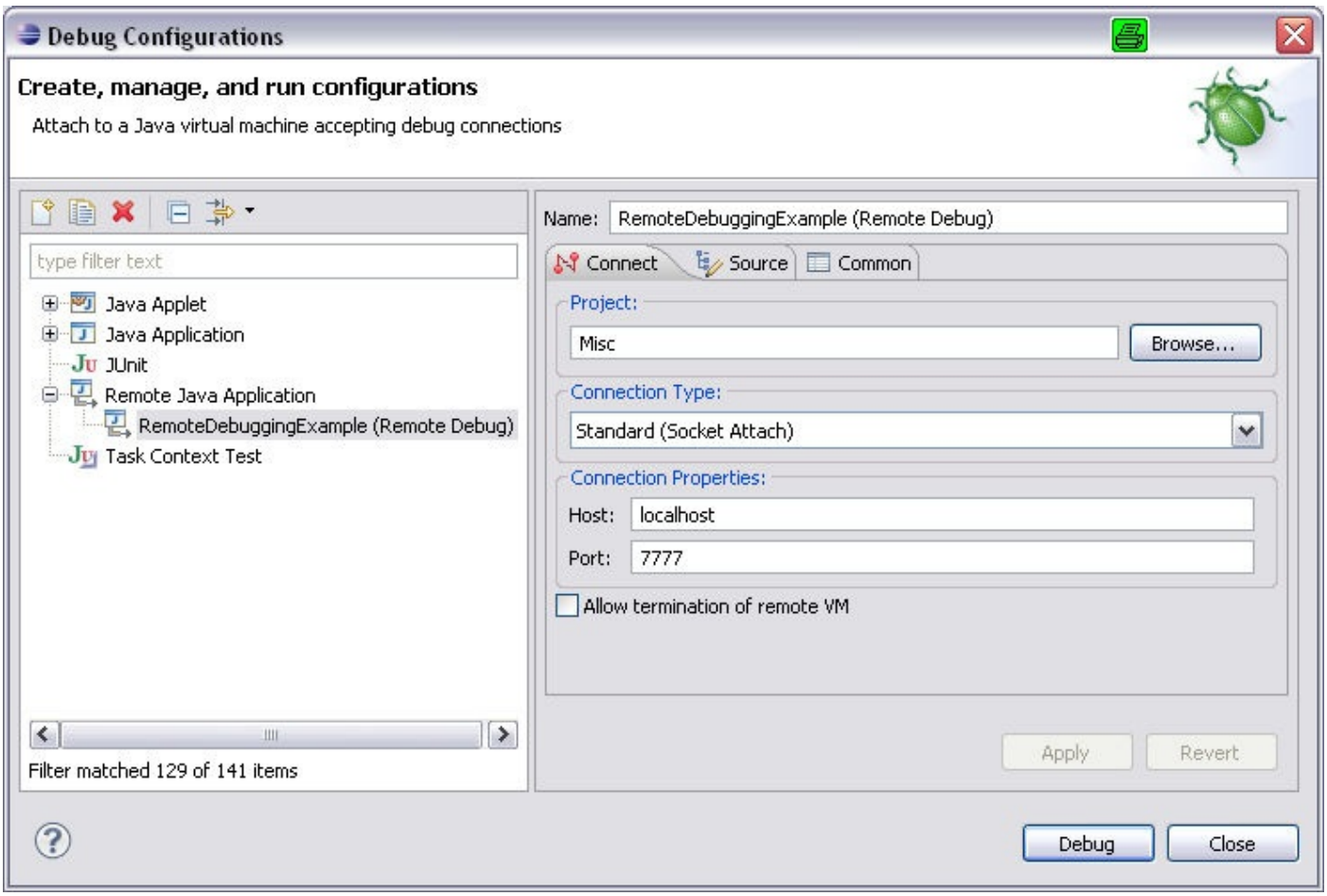


Abbildung 2-12 *Beispiel einer Einstellung zum Remote Debugging*

Hinweis: Remote Debugging vor JDK 5

Damit das Debugging sinnvoll durchgeführt werden kann, müssen im Tab SOURCE die Pfade zum Sourcecode des zu debuggenden Programms passend gewählt werden. Das heißt aber auch, dass im Standardfall für das Remote Debugging immer der korrekt passende Sourcecode vorliegen muss.

Remote Debugging des Programms

Nach diesen Vorbereitungen kann wie beim Debugging eines aus Eclipse heraus gestarteten Programms normal gearbeitet werden, d. h., es können im Sourcecode Breakpoints gesetzt und während der Programmausführung Variablen inspiziert werden. In diesem Beispiel lässt sich recht unspektakulär die Variable `i` beobachten. Zudem können wir ein wenig mit den Kommandos STEP INTO, STEP OVER usw. experimentieren, um ein erstes Gefühl dafür zu bekommen, dass Remote Debugging abgesehen von den Konfigurationsaufwänden tatsächlich so leicht ist wie lokales Debugging. Abbildung 2-13 zeigt eine Unterbrechung des obigen Programms nach dem 20. Durchlauf.

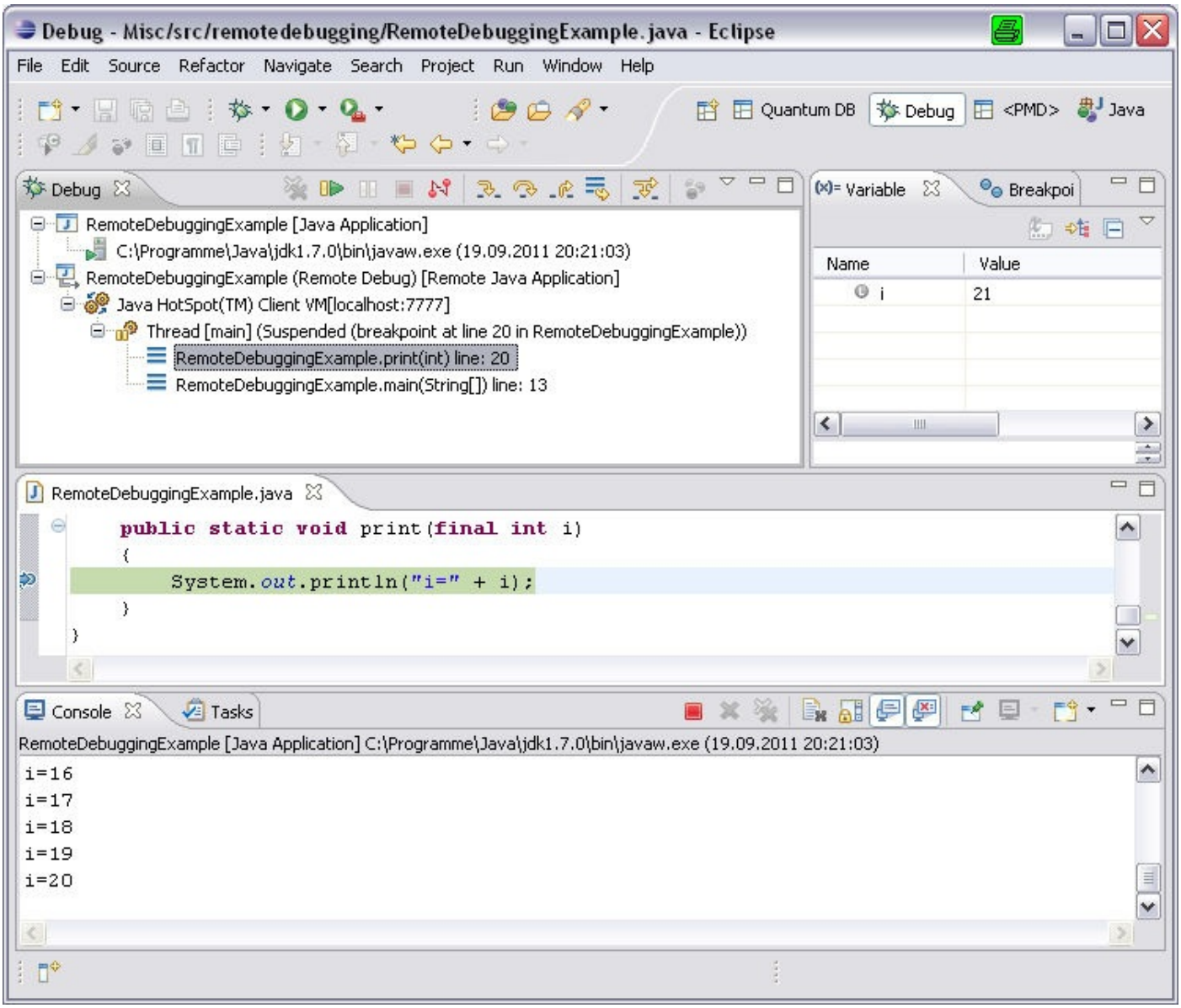


Abbildung 2-13 *Beispiel einer Remote Debugging Session*

Fazit

Dieser Abschnitt hat Ihnen einen einführenden Überblick über die Möglichkeiten zur Fehlersuche mithilfe eines Debuggers verschafft. Insbesondere der Eclipse-Debugger bietet noch weitreichendere Optionen – dort sind z. B. Breakpoints möglich, die nur bei Eintreten einer bestimmten Bedingung oder beim Auftreten einer bestimmten Exception aktiv werden, also im Normalfall den Programmablauf nicht unterbrechen.

Um das Werkzeug Debugger in der Praxis gewinnbringend einsetzen zu können, sollten Sie ein wenig damit experimentieren. Dadurch sind Sie in Zukunft besser gerüstet, wenn es einen hartnäckigen Fehler aufzuspüren gilt.

2.6 Deployment von Java-Applikationen

Nachdem wir nun wissen, wie wir unsere Projekte strukturieren und eine Historie der enthaltenen Dateien verwalten sowie auch durch Unit Tests und Debugging

funktionierende Stände produzieren können, werfen wir einen kurzen Blick auf die Auslieferung bzw. die Installation, neudeutsch: das ***Deployment***, von Applikationen.

Größere Programme bestehen aus einer Vielzahl an Klassen und anderen Dateien, wie Textressourcen oder Bildern. Zur Auslieferung eines solchen Programms an Kunden wäre es unpraktisch und vor allem auch fehleranfällig, alle diese Dateien einzeln übertragen zu müssen. Deutlich angenehmer und weniger fehleranfällig ist dagegen die Bereitstellung einer Java-Anwendung in Form einer einzigen Datei. Dazu kann man mehrere Dateien zu einem Java ARchive (***JAR***) zusammenfassen. Vorteilhaft ist dies auch für die Übertragung des Programms über das Netzwerk.

Nach dieser kurzen Motivation und Einführung möchte ich einen Blick auf einige Eigenschaften und Vorzüge von JAR-Dateien werfen:

- **Plattformunabhängigkeit** – Das JAR-Format ist plattformunabhängig und somit problemlos zwischen verschiedenen Plattformen austauschbar. Das stellt einen großen Vorteil im Vergleich zu nativen Bibliotheken dar.
- **Komfort** – JARs vereinfachen die Verteilung und Auslieferung von Software, weil nur eine Datei anstatt einer Menge von Dateien in unterschiedlichen Verzeichnissen bereitgestellt wird. Mithilfe spezieller Metainformationen, die in der sogenannten Manifest-Datei abgelegt sind, kann man das Programm auch direkt über die Kommandozeile starten und je nach Konfiguration im Betriebssystem ist es sogar möglich, das enthaltene Programm per Doppelklick zu starten.
- **Klassenbibliotheken** – Neben dem Einsatzgebiet, eine lauffähige Java-Applikation in Form eines JARs bereitzustellen, lassen sich auch mehrere Klassen als Klassenbibliothek in Form eines JARs anderen Programmen zur Verfügung stellen. Eine nutzende Applikation muss dann beim Kompilieren und bei der Ausführung die JAR-Datei in ihren `CLASSPATH` aufnehmen.
- **Performance** – Sowohl im Netzwerk als auch im Dateisystem ist das Kopieren bzw. Übertragen einer Vielzahl kleinerer Dateien aufwendiger und langsamer als das Verarbeiten einer größeren Datei. JARs erlauben somit eine optimalere Übertragung über das Netzwerk oder im Dateisystem.
- **Komprimierung** – Ein JAR ist standardmäßig im ZIP-Format komprimiert. Das macht Netzwerkübertragungen effizienter als ohne Komprimierung. Diese kann auch deaktiviert werden.
- **Sicherheit** – Durch Signierung mit einem digitalen Zertifikat kann man unerwünschte Manipulationen an JAR-Dateien aufdecken. In solchen Fällen kann man verhindern, das JAR auszuführen (Executable) bzw. zu laden (Bibliothek).

Tipp: JDK-Erweiterung

Wenn man zur Vereinfachung die Funktionalität aus einem bestimmten JAR allen

Java-Programmen, die mit der installierten JVM ausgeführt werden, zur Verfügung stellen möchte, ohne dass diese explizit auf das JAR verweisen, so kann man die JAR-Datei in das Verzeichnis `<JDK>\jre\lib\ext\` legen. In diesem Pfad sucht die JVM standardmäßig nach Klassen, bevor sie die Angaben aus dem `CLASSPATH` nutzt.

2.6.1 Das JAR-Tool im Kurzüberblick

Im JDK gibt es zum Verarbeiten, Erstellen, Inspizieren, Ausführen, Entpacken usw. von Java-Archiven ein Tool namens `jar`. Der Aufruf des Tools besitzt folgende Struktur:

```
jar [Aktionsoptionen] [Manifest] jarArchivName input1 [input2]
```

Achten Sie unbedingt darauf, keine Leerzeichen zwischen den Aktionsoptionen anzugeben, da das `jar`-Tool die Angaben ansonsten falsch interpretiert. Nachfolgend liste ich die wichtigsten Aktionsoptionen tabellarisch auf:

Tabelle 2-1 *Kommandozeilenoptionen des* `jar`*-Tools*

Kürzel und Aktion	Erklärung
`c` – create	Es wird aus den angegebenen Quellen (`input1`, `input2` usw.) ein neues JAR erzeugt. Quellen können nicht nur Dateien, sondern auch Verzeichnisse sein.
`t` – table of contents	Gibt den Inhalt des Archivs auf der Konsole aus.
`x` – extract	Extrahiert die angegebenen Dateien aus dem Archiv in das aktuelle Verzeichnis.
`u` – update	Aktualisiert (überschreibt oder erzeugt) die angegebene Datei in einem bestehenden Archiv.
`f` – file	Gibt an, dass der nächste Parameter die Archivdatei ist. Dies wird nahezu immer gewählt. Sonderbarerweise ist die Ausgabe auf die Konsole der Standard.
`v` – verbose	Man erhält eine ausführliche Protokollierung der durchgeführten Aktionen.
`m` – manifest	Erzeugt eine Manifest-Datei mit Metainformationen zu dem JAR – in Form einer Textdatei mit Schlüssel-Wert-Angaben.

Die Aktionen `c`, `t`, `x` und `u` können nicht miteinander kombiniert werden. Die Option `f` muss man dagegen fast immer angeben, weil fast nie eine Interaktion mit der Konsole gewünscht ist. Optional kann eine ausführliche Ausgabe durch Kombination mit der Option `v` erzielt werden. Falls Sie Unterstützung benötigen, bietet ein Aufruf ohne Argumente eine recht aussagekräftige Hilfe. Nach diesem Kurzüberblick schauen wir uns nun einzelne Kommandos etwas genauer an.

Generierung einer JAR-Datei

Zum Erzeugen einer JAR-Datei nutzt man das Kommando `c` mit folgender Syntax:

```
jar cvf jarFileName inputFileOrDir1 inputFileOrDir2 …
```

Die Optionen `cvf` stehen für Create, Verbose und File. Die Quellen können Java-Klassen oder Verzeichnisse sein. Die Angabe kann mit exaktem Namen erfolgen oder aber auch Wildcards (*) enthalten. Im Falle von Verzeichnissen wird deren Inhalt rekursiv dem JAR hinzugefügt.

Machen wir es konkret und verpacken ein kleines Browser-Spiel, das als Applet realisiert ist. Dieses enthält neben den Klassen auch Bilder und Töne hier aus den Verzeichnissen `images` und `sounds`:

```
> jar cvf MyGame.jar *.class images sounds
added manifest
adding: Breakout.class(in = 893) (out= 520)(deflated 41%)
adding: MainGame.class(in = 3616) (out= 2028)(deflated 43%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/explosion.png(in = 978) (out= 983)(deflated 0%)
adding: images/bonus.png(in = 839) (out= 844)(deflated 0%)
adding: sounds/(in = 0) (out= 0)(stored 0%)
adding: sounds/explosion.wav(in = 3912) (out= 1983)(deflated 50%)
adding: sounds/bonus.wav(in = 2839) (out= 1413)(deflated 50%)
```

Anhand der Ausgabe sehen wir neben der Protokollierung des erwarteten Hinzufügens von Dateien auch die Ausgabe von »`added manifest`« sowie die Kompressionsrate für die einzelnen Dateien. Basierend auf den unterschiedlichen Werten für `in` und `out` sowie der Angabe von `deflated` wird protokolliert, wie stark die Dateien komprimiert werden – tatsächlich sind die PNG-Bilder bereits von Hause aus komprimiert und werden durch die Kompression sogar minimal größer. Verbleibt noch das Manifest, das eine Sammlung von Metainformationen zu dem Archiv enthält. Später dazu mehr.

2.6.2 JAR inspizieren und ändern, Inhalt extrahieren

Neben dem Erzeugen kann man das `jar`-Tool auch dazu nutzen, sich den Inhalt einer JAR-Datei anzeigen zu lassen bzw. daran Modifikationen vorzunehmen oder daraus Dateien zu extrahieren. Statt dies über die Kommandozeile zu machen, bietet es sich an, dafür ein gängiges ZIP-Programm zu nutzen, etwa WinZIP, 7-ZIP oder WinRAR. Hier profitiert man davon, dass JAR-Dateien das ZIP-Format einsetzen.

Anzeigen des Inhalts

Zum Inspizieren des Inhalts einer JAR-Datei existiert das folgende Kommando `t`:

```
jar tf jar-file
```

Damit wird der Inhalt des JARs auf der Konsole ausgegeben. Detailliertere Informationen (etwa zu Dateigrößen usw.) erhält man durch die Angabe der Option `v`.

Ändern des Inhalts

Wenn einzelne Dateien nachträglich im JAR geändert werden sollen, so möchte man dazu natürlich nicht das Archiv vollständig neu erzeugen müssen. Der Inhalt einer JAR-Datei lässt sich mithilfe des Kommandos `u` modifizieren:

```
jar uf jar-file input-file(s)
```

Die übergebenen Datei(en) werden im JAR aktualisiert. Falls diese im JAR noch nicht existierten, werden sie hinzugefügt.

Nehmen wir an, wir hätten eine Änderung an der Datei `ReadMe.txt` vorgenommen und wollten diese ins JAR integrieren. Dazu schreiben wir:

```
jar uf MyGame.jar ReadMe.txt
```

Extrahieren des Inhalts

Zum Extrahieren des Inhalts einer JAR-Datei existiert das Kommando `x`:

```
jar xf jar-file [archived-file(s)]
```

Damit extrahiert man alle angegebenen Dateien aus dem JAR in das aktuelle Verzeichnis. Werden keine Dateien angegeben, so wird das JAR vollständig entpackt.

Als Anwendungsfall könnten wir uns vorstellen, dass man nochmals an der Textdatei `ReadMe.txt` eine Änderung vornehmen möchte. Dazu extrahieren wir die Datei aus dem JAR, ändern anschließend deren Inhalt und fügen sie wieder per Update in das Archiv ein.

```
jar xf MyGame.jar ReadMe.txt
…
jar uf MyGame.jar ReadMe.txt
```

2.6.3 Metainformationen und das Manifest

Wenn Sie bei der Erzeugung der JAR-Datei in Abschnitt 2.6.1 genau hingeschaut haben, dann wurde dabei auf der Konsole der Text »`added manifest`« ausgegeben. Bei der Generierung eines JARs wird durch das `jar`-Tool automatisch eine sogenannte ***Manifest-Datei*** namens `MANIFEST.MF` im Unterverzeichnis `META-INF` erstellt und dem JAR hinzugefügt. Dieses Manifest ist eine spezielle Datei und dient dazu, Metainformationen zu den Dateien, die im JAR zusammengefasst sind, bereitzustellen.

Zum besseren Verständnis werfen wir nun einen Blick auf die Default-Manifest-Datei, dann werden wir Klassen aus einem JAR starten und dazu eine Main-Klasse als Einstiegspunkt angeben.

Metainformationen in der Default-Manifest-Datei

Wir nutzen unser bisheriges Wissen und entpacken die Datei mit folgendem Aufruf:

```
jar xvf MyGame META-INF/MANIFEST.MF
```

Beachten Sie unbedingt, dass das `jar`-Tool penibel zwischen Groß- und Kleinschreibung unterscheidet, obwohl beispielsweise das Windows-Betriebssystem dies nicht tut. Durch das erfolgreiche Entpacken wird das Unterverzeichnis `META-INF` und darin die Datei `MAINFEST.MF` erzeugt. Öffnen wir die gerade entpackte

Datei, so enthält sie folgenden Inhalt mit (recht spärlichen) Metainformationen – wobei die nach `Created-By` angegebene Java-Version gegebenenfalls bei Ihnen abweicht:

```
Manifest-Version: 1.0
Created-By: 1.7.0_21 (Oracle Corporation)
```

Wie man leicht sieht, handelt es sich hier um eine einfache zeilenorientierte Textdatei mit der Angabe von Schlüssel-Wert-Paaren, die durch »: « getrennt sind.

Klassen aus JAR starten

Wie schon zuvor angedeutet, können wir eine Java-Anwendung als JAR bereitstellen. Wenn wir diese nun starten wollen, so kann man folgenden Aufruf nutzen:

```
java -cp <archiv.jar> <mypackage.Main-Class>
```

Dabei dient das JAR als Klassenbibliothek und wird explizit im `CLASSPATH` aufgeführt. Zudem wird die zu startende Klasse angegeben. Alternativ ist auch folgender Aufruf erlaubt, um eine Klasse aus einem JAR zu starten:

```
java -jar <archiv.jar>
```

Probieren wir es für unser Beispiel `MyGame.jar` folgendermaßen aus:

```
java -jar MyGame.jar
```

Statt des Programmstarts kommt es jedoch zu einem Problem und folgender Fehlermeldung »`kein Hauptmanifestattribut in MyGame.jar`«. Durch kurzes Nachdenken wird klar: Woher soll die JVM wissen, welche `main()`-Methode welcher Klasse gestartet werden soll? Diese Information kann in einem JAR über die Manifest-Datei beschrieben werden. Dazu kann man neben den Standardangaben, die beim Vorgang der Archivierung automatisch vom `jar`-Tool erzeugt werden, weitere Informationen, etwa zur `main`-Klasse und benötigten Bibliotheken, im Manifest hinterlegen.

Informationen im Manifest bereitstellen

Wir haben gesehen, dass die Manifest-Datei `MANIFEST.MF` automatisch vom `jar`-Tool erzeugt wird. Oftmals sollen dort weitere, über die Standardangaben hinausgehende Informationen hinterlegt werden. Dazu kann man eine Textdatei beliebigen Namens, meistens `manifest.txt` genannt, nutzen. Dort werden Schlüssel-Wert-Paare angegeben, etwa wie folgt:

```
Main-Class: de.javaprofi.helloworld.App
```

Diese Angaben werden vom `jar`-Tool bei der Erstellung der Manifest-Datei berücksichtigt, wenn man die Option `m` angibt:

```
jar cvfm MyGame.jar mainfest.txt classes/
```

Das so erzeugte JAR enthält nun die zuvor fehlenden Informationen zur `main`-Klasse. Rufen wir also Folgendes auf:

```
java -jar MyGame.jar
```

Doch das Programm startet immer noch nicht! Stattdessen kommt es zu folgendem Fehler: »`Hauptklasse de.javaprofi.helloworld.App konnte nicht gefunden oder geladen werden`«.

Das ist zunächst etwas merkwürdig. Inspizieren wir jedoch das JAR per `jar tf Mygame.jar`, so stellen wir fest, dass die `.class`-Dateien aus dem Unterordner `classes` exakt so (also auch inklusive des Verzeichnisnamens) in das Archiv übernommen wurden:

```
Manifest wurde hinzugefügt
classes/ wird hinzugefügt(ein = 0) (aus = 0)(0 % gespeichert)
classes/de/ wird hinzugefügt(ein = 0) (aus = 0)(0 % gespeichert)
classes/de/javaprofi/ wird hinzugefügt(ein = 0) (aus = 0)(0 % gespeichert)
classes/de/javaprofi/helloworld/ wird hinzugefügt(ein = 0) (aus = 0)(0 %
      gespeichert)
classes/de/javaprofi/helloworld/App.class wird hinzugefügt(ein = 561) (aus =
      349)(37 % verkleinert)
```

Dadurch stimmen die Package-Angaben im JAR und in den Java-Klassen nicht überein. Um dies zu verhindern, muss man dem `jar`-Tool beim Erzeugen eines JARs mitteilen, dass das Verzeichnis, in dem sich die Klassen befinden, nicht Bestandteil des archivierten Pfads werden soll. Dazu dient die Option `-C` zur Festlegung des Pfads zu den Klassen mit einer etwas ungewöhnlichen Angabe wie folgt:

```
jar cvfm MyGame.jar manifest.txt -C classes .
```

Nach dieser Modifikation lässt sich das Programm dann aus dem JAR starten.

2.6.4 Inspizieren einer JAR-Datei

Nachdem wir einiges zu JAR-Dateien und deren Manifest kennengelernt haben, wollen wir uns nun ansehen, wie wir diese Informationen mithilfe eines Java-Programms auslesen können. Neben der Klasse `java.io.File`, die hier die JAR-Datei im Dateisystem repräsentiert, kommen noch die Klassen `JarFile`, `Manifest` und `Attributes` aus dem Package `java.util.jar` zum Einsatz, die die programmatische Verarbeitung von Informationen aus JAR-Dateien erleichtern.

```
public static void main(final String[] args) throws IOException
{
   final String jarFileName = "src/ch02_profbuild/jarfiles/gradle-
example.jar";
   final File file = new File(jarFileName);
   System.out.println(file.getAbsolutePath());
   System.out.println("jarFileName='" + jarFileName + "'");

   // Informationsobjekte ermitteln
   final JarFile jarfile = new JarFile(jarFileName);
   final Manifest manifest = jarfile.getManifest();
   final Attributes attributes = manifest.getMainAttributes();

   // Informationen auslesen
   final String mainClassName =
attributes.getValue(Attributes.Name.MAIN_CLASS);
```

```
    System.out.println("mainClassName='" + mainClassName + "'");

    final String createdBy = attributes.getValue("Created-By");
    System.out.println("createdBy='" + createdBy + "'");

    final String manifestVersion = attributes.getValue(Attributes.Name.
          MANIFEST_VERSION);
    System.out.println("manifestVersion='" + manifestVersion + "'");
}
```

Listing 2.1 *Ausführbar als* **'INSPECTINGJAREXAMPLE'**

Im Listing sehen wir verschiedene Methoden, wie etwa `getManifest()` und `getMainAttributes()`, zum Zugriff auf die Attribute des Manifests. Zum Auslesen der Werte dient die Methode `getValue()`, der ein Schlüsselname übergeben wird. Verschiedene gebräuchliche Namen sind in `Attributes.Name` definiert. Starten wir das Programm INSPECTINGJAREXAMPLE, so kommt es zu folgenden Ausgaben

```
C:\Users\Micha\workspace\java-profi\src\ch02_profbuild\jarfiles\gradle-
example.
      jar
jarFileName='src/ch02_profbuild/jarfiles/gradle-example.jar'
mainClassName='ch02_profbuild.ResourceAccessExample'
createdBy='null'
manifestVersion='1.0'
```

Auf Daten bzw. Ressourcen aus JAR-Dateien zugreifen

Wir wissen nun, wie man Klassen und andere Ressourcen zu JARs zusammenfasst, Informationen dazu in einer Manifest-Datei hinterlegt und diese programmatisch wieder ausliest. Wenn wir eine Applikation in Form eines JARs generiert haben und nun deren `main`-Klasse starten, so stellt sich die Frage, wie man innerhalb der Applikation auf Daten zugreifen kann, die im JAR hinterlegt sind.

Nehmen wir an, die Spieleapplikation des Beispiels wäre als JAR verpackt und es soll auf die Bilder und Töne in den Unterverzeichnissen `images` bzw. `sounds` zugegriffen werden. Da diese Dateien aber innerhalb des JARs und nicht als Dateien im Dateisystem existieren, ist ein Zugriff nicht mithilfe einer `File`-Instanz möglich: Denn diese erlaubt keinen Zugriff auf die internen Strukturen des JARs. Für diesen Anwendungsfall gibt es in der Klasse `java.lang.ClassLoader` zwei spezielle Methoden, nämlich `getResource()` und `getResourceAsStream()`, wobei erstere ein Objekt vom Typ `java.net.URL` liefert. Die Methode `getResourceAsStream()` gibt ein `java.io.InputStream`-Objekt zurück. Als Alternative existieren beide Methoden auch in der Klasse `java.lang.Class`. Neben einem Zugriff auf Ressourcen innerhalb eines JARs kann man mit den Methoden auch auf Ressourcen zugreifen, die innerhalb des Dateisystems liegen.

```
public static void main(final String[] args) throws IOException
{
    final String filename = "images/Bad-Question-Message.png";

    // Variante mit ClassLoader (1) und Class (2)
```

```
    final URL imageUrl1 = ResourceAccessExample.class.getClassLoader().
                                                        getResource(filename);
// 1
    final URL imageUrl2 = ResourceAccessExample.class.getResource(filename);
// 2
    System.out.println("imageUrl1: " + imageUrl1);
    System.out.println("imageUrl2: " + imageUrl2);

    // Variante mit InputStream
    final InputStream is1 = ResourceAccessExample.class.
                            getClassLoader().getResourceAsStream(filename);
    final InputStream is2 = ResourceAccessExample.class.
                            getResourceAsStream(filename);
    // …
}
```

Listing 2.2 *Ausführbar als* **'RESOURCEACCESSEXAMPLE'**

Der Unterschied zwischen den im Listing gezeigten Varianten besteht darin, dass die Variante 1 basierend auf `ClassLoader` relativ zum Projektverzeichnis zugreift und Variante 2 relativ zu der `.class`-Datei der entsprechenden Klasse.

Um den Zugriff aus einem JAR zu demonstrieren, muss diese Klasse auch entsprechend gestartet werden, etwa wie folgt:

```
java -jar gradle-example.jar ResourceAccessExample
```

Dann kommt es zu folgenden Ausgaben, wenn sich die JAR-Datei im Verzeichnis `C:/jar-example` befindet:

```
imageUrl1: jar:file:/C:/jar-example/gradle-example.jar!/images/Bad-Question-
      Message.png
imageUrl2: jar:file:/C:/jar-example/gradle-example.jar!/ch02_profbuild/images/
      Bad-Question-Message.png
```

2.7 Einsatz eines IDE-unabhängigen Build-Prozesses

In Abschnitt 2.3 habe ich die Notwendigkeit einer Versionsverwaltung dargestellt, um gewünschte Versionsstände des Sourcecodes und anderer Dateien jederzeit zuverlässig gemäß einem älteren Stand wiederherstellen zu können. Dieser Abschnitt zeigt die Vorteile eines von der IDE unabhängigen Build-Prozesses auf. Schauen wir kurz, warum dies so wichtig ist. Programme bestehen gewöhnlich aus einer Vielzahl an Klassen, die meistens auch einige Abhängigkeiten zu externen Bibliotheken besitzen. Innerhalb der IDE lassen sich derartige Abhängigkeiten und zugehörige Pfade verwalten und Sourcen automatisch kompilieren – jedoch können hier lokale Einstellungen größeren Einfluss darauf haben, wie das Programm erstellt wird: Insbesondere variieren Einstellungen erfahrungsgemäß (zumindest ein wenig) von Entwickler-PC zu Entwickler-PC. Dadurch sind die Resultate beim Kompilieren nicht immer 100% reproduzierbar. Das ist für Auslieferungen oder spezielle Zwischenstände bzw. Versionen des Programms (auch ***Release*** genannt) nicht akzeptabel. Vielmehr sollte der Herstellungsprozess (***Build***) auf eine definierte Art und Weise mit einer eindeutigen Konfiguration und gemäß einem definierten Ablauf erfolgen. Dazu dient ein von der IDE unabhängiger Build-Prozess. Darüber hinaus

ist es wünschenswert, sowohl Unit Tests als auch andere Prüfungen als festen Bestandteil des Build-Laufs auszuführen, um ein Mindestmaß an Qualität sicherzustellen. Im Speziellen gilt dies, wenn man den Vorgang des Kompilierens und der Releaseerzeugung automatisieren möchte. Dafür nutzt man in der Regel ein Build-Tool.

Für Java stehen mit ***Ant***, ***Maven*** und ***Gradle*** populäre Tools bereit. Jedes davon besitzt eine unterschiedliche Art, wie es den Entwickler bei der Gestaltung des Build-Vorgangs unterstützt. Während mit Ant jeder (kleine) Arbeitsschritt einzeln beschrieben werden muss, vereinfachen Maven und Gradle den Build-Vorgang durch die Bereitstellung von Automatiken, basierend auf Konventionen und einer fixen Reihenfolge der Abläufe im Build-Prozess. Dadurch verliert man zwar etwas Flexibilität, jedoch wird andererseits auch viel Arbeit abgenommen, da die manuelle, mühsame und oftmals fehlerträchtige Beschreibung des Build-Laufs entfällt.[16]

Während für die ersten beiden Auflagen dieses Buchs noch Ant als Build-Tool genutzt wurde, kommt in dieser dritten Auflage nun Gradle zum Einsatz. Meine Wahl lässt sich besser verstehen, wenn man ein wenig Hintergrundwissen zu Ant und Maven besitzt. Daher werden wir in den folgenden Abschnitten Ant und Maven kurz beleuchten, um dann die Vorzüge von Gradle entsprechend schätzen zu können.

Motivation für ein IDE-unabhängiges Build-System

Stellen Sie sich noch einmal einen dieser hektischen Freitage vor. Es ist wieder für den späten Nachmittag eine Version des Programms, basierend auf einem älteren Stand, auszuliefern. Glücklicherweise haben Sie – spätestens nach Lektüre des Abschnitts über Versionsverwaltungen – eine solche Versionsverwaltung installiert. Die Tatsache, dass an der Applikation bereits weiterentwickelt wurde, stört Sie nicht, da Sie den benötigten älteren, mit einer Markierung versehenen Auslieferungsstand separat auschecken. Nach ein paar Stunden harter Arbeit sind die geforderten Bugfixes in das Programm integriert. Die Kompilierung in Ihrer Entwicklungsumgebung verläuft ohne Probleme. Auch die Ausführung der Unit Tests zeigt ein beruhigendes Grün. Bis jetzt sieht alles so aus, als ob die Auslieferung bald vorgenommen werden könnte. Um eine Auslieferung zu erzeugen, müssen die Sourcen allerdings mithilfe eines speziellen Build-Tools übersetzt werden. Nur wenn ein solcher Build-Vorgang erfolgreich war, erhält man ein neues Release. Eigentlich sollte jetzt nichts mehr schiefgehen, da der IDE-Build und die Unit Tests erfolgreich waren! Doch der Build schlägt fehl. Wie kann das sein?

Eine wahrscheinliche Ursache sind nicht vollständig eingecheckte Stände der eigenen Änderungen. Das stellt man jedoch nicht fest, wenn man das Build-Skript im eigenen Arbeitsverzeichnis ausführt. Für ein Release-Build sollte daher der aktuelle Repository-Stand in ein separates Verzeichnis ausgecheckt werden. In diesem wird dann das Build-Skript gestartet. Treten dabei Probleme auf, so wurden tatsächlich einige Änderungen nicht vollständig ins Repository eingespielt. Ansonsten erhält man

eine positive Rückmeldung und man kann eine Markierung setzen.

Ein weiteres mögliches Problem besteht darin, dass von der IDE und dem Build-Skript verschiedene Bibliotheken referenziert werden, etwa wenn eine neue Bibliothek in der IDE bekannt ist, aber noch nicht dem Build-Skript. Zu solchen Problemen kommt es schnell, wenn man die Abhängigkeiten in der IDE manuell verwaltet.[17]

Tipp: Startschwierigkeiten beim Einführen eines Build-Prozesses

Beim Einführen eines Build-Prozesses gibt es zum Teil einige Startschwierigkeiten: Möglicherweise verwendet die IDE einen anderen Java-Compiler (oder eine andere Version davon) als das Build-Skript. Dies kann immer dann passieren, wenn verschiedene JDKs und JREs installiert sind. Teilweise werden durch die IDE auch andere Pfade verwendet und so andere Versionen von Bibliotheken oder Klassen referenziert. Zudem können absolute statt relativer Pfade ein Problem darstellen. Als Tipp sollten Sie gleich bei Projektstart auf einen automatischen Build-Prozess Wert legen, um nachträglich Probleme zu vermeiden.

Grundlage für Nightly Builds bis hin zu Continuous Integration

Ein von der IDE-unabhängiger Build-Prozess befreit von Besonderheiten und Konfigurationseinstellungen der genutzten IDE und bietet vor allem auch die Möglichkeit, den Build-Vorgang auf einen speziellen Build-Server auszulagern. Auf dem Build-Server wird vor dem Start des Build-Vorgangs zunächst der aktuelle Stand aus der Versionsverwaltung in ein separates Verzeichnis ausgecheckt und dort dann ein Build-Lauf gestartet.

Erfolgt dieser Vorgang jede Nacht, so spricht man von einem ***Nightly Build***. Dadurch erhält man täglich eine Rückmeldung über den momentanen Stand der Entwicklung. Wünschenswert ist es, diesen Prozess durchaus auch mehrmals am Tag auszuführen. Einige neuere Vorgehensmodelle der Softwareentwicklung propagieren eine häufige Integration von Änderungen in den aktuellen Stand. Wird bei jeder Integration in das Repository nach kurzer Zeit automatisch ein Build-Vorgang angestoßen, so erhält man umgehend eine Rückmeldung über den Zustand des Systems und ob die letzten Änderungen möglicherweise Fehler enthalten. Tools, die eine solche ***kontinuierliche Integration*** (***Continuous Integration***) erlauben, sind Hudson[18] bzw. Jenkins[19].

Motivation für Ant, Maven und Gradle

Zu Recht kann man sich fragen, wozu immer noch weitere Build-Tools entwickelt wurden, wo es doch bereits kommandozeilenorientierte Tools wie make und andere gibt. Aber all diese haben ihre Nachteile, vor allem im Bereich der Wartbarkeit und

auch der Plattformunabhängigkeit. Glücklicherweise werden diese Unzulänglichkeiten von den heutzutage aktuellen Tools wie Ant, Maven, Gradle usw. adressiert. Deren Build-Dateien sind plattformunabhängig, besser lesbar und lassen sich mit jedem beliebigen Editor bearbeiten, wobei sich für Ant und Maven ein XML-Editor anbietet, um deren Build-Skripte strukturiert darzustellen. Für Gradle reicht ein x-beliebiger Texteditor, da die Build-Dateien in der Regel recht kurz, sehr gut lesbar und verständlich sind.

Vor einigen Jahren wurde für Java das Build-Tool Ant ersonnen. Durch dessen Einsatz kam es zu einer deutlichen Verbesserung und zur Standardisierung von Build-Umgebungen. Im Rahmen der Entwicklung verschiedener Projekte im Apache-Bereich stellte sich jedoch heraus, dass es zwar zu Erleichterungen durch Ant kam, jedoch immer noch einige Sonderfälle in den jeweiligen Projekten existierten, die einen Wechsel von Entwicklern zwischen Projekten erschwerten.

Neben dem Kompilieren und Testen sind eigentlich noch diverse weitere Schritte in verschiedenen Projekten (nahezu) gleich. Das möchte man nicht immer wieder erneut manuell in der Build-Datei beschreiben, wie dies bei Ant notwendig ist und was zu kleineren Inkonsistenzen führt. Statt einer solchen recht feingranularen Vorgabe des Build-Ablaufs verfolgt Maven ein anderes Konzept mit einer deklarativen Steuerung, was vieles vereinfacht: Die verschiedenen Schritte eines Build-Prozesses sind standardisiert, wodurch dieser Ablauf nicht jedes Mal erneut in den Build-Dateien festgelegt werden muss. Damit das Ganze allerdings funktionieren kann, müssen gewisse Konventionen eingehalten werden, wie beispielsweise die bereits vorgestellte einheitliche Projektstruktur in Abschnitt 2.2. Trotz seiner unbestreitbaren Vorzüge stellte sich aber auch für Maven mit der Zeit heraus, dass es einige Schwächen besitzt, sodass weitere Build-Tools entstanden, unter anderem Gradle.

Nach dieser Einführung schauen wir nun überblicksartig auf Ant und etwas genauer auf Maven bevor wir dann Gradle in größerer Detailtiefe betrachten werden.

2.7.1 Ant im Überblick

Das Build-Tool Ant (`http://ant.apache.org`) nutzt zur Beschreibung der Abläufe beim Build eine XML-codierte Datei, die standardmäßig `build.xml` genannt wird. Dort ist definiert, welche Aktionen vorhanden sind und ausgeführt werden können. Dazu werden Teilschritte (Targets) genutzt. Diese bestehen wiederum aus einer Folge von Kommandos (Tasks). Die genauen Details sind hier nicht wichtig (einige davon können in den älteren Auflagen dieses Buchs nachgeschlagen werden). Es geht lediglich darum, ein Verständnis dafür zu wecken, dass bei Ant viele einzelne Schritte sehr feingranular definiert werden. Dazu schauen wir uns den folgenden (gekürzten) Ausschnitt einer `build.xml`-Datei an:

```
<project name="Example" default="release" basedir=".">

    <!-SET SOURCE, BUILD AND RELEASE DIR ->
```

```
    <property name="source.dir" value="src" />
    <property name="build.dir" value="bin" />
    <property name="lib.dir" value="lib" />

    <!-SET CLASSPATH HERE ->
    <path id="project.classpath">
        <fileset dir="${lib.dir}" includes="**/*.jar" />
    </path>

    // …

    <!-COMPILE ALL SOURCES (AND TESTS) ->
    <target name="compile">
        <echo message="compile" />
        <javac srcdir="${source.dir}" destdir="${build.dir}"
source="${source}"
              compiler="${compiler}" debug="true" encoding="ISO-8859-1">
            <classpath>
                <path refid="project.classpath" />
            </classpath>
        </javac>

        <available file="${testsource.dir}" type="dir"
property="testsource.dir.
              present" />
        <antcall target="-compileTests" />
    </target>

    <!-CREATE A NEW RELEASE ->
    <target name="release" depends="compile">
        <!-create jar file ->
        <jar destfile="${release.dir}/${appname}.jar">

            <fileset dir="${build.dir}" excludes="**/*Test*.class" />
            <manifest>
                <attribute name="Main-Class" value="${mainclass}" />
            </manifest>
        </jar>

        <echo message="release created" />
    </target>

    // …
</project>
```

In der gezeigten Datei wird als Wurzelelement der XML-Knoten `<project>` definiert. Dessen Attribute enthalten den Namen des zu übersetzenden Projekts, das Startverzeichnis für alle Targets sowie ein Default-Target – hier `release`. Dieses erzeugt aus den kompilierten Klassen ein eigenständiges Release als JAR-Datei. Als Abhängigkeit sind mithilfe des Ant-Targets `compile` zunächst alle Klassen aus dem `src`- und dem `test`-Ordner zu übersetzen. Für einige zur Kompilierung benötigte Verzeichnisse des Projekts sind korrespondierende Properties definiert, etwa `source.dir`. Dadurch erreicht man eine bessere Lesbarkeit und reduziert Konfigurationsaufwände.

Nachteil 1: Immer wieder ähnliche Build-Dateien

Für nahezu jedes Projekt besteht ein Build-Ablauf immer wieder aus ziemlich ähnlichen Schritten, wie dem Kompilieren von Sourcecode und Testcode, dem Ausführen von Tests sowie der Paketierung als JAR. Auch wenn das Ganze im Listing nur ausschnittsweise gezeigt ist, so bekommt man doch eine recht gute Vorstellung davon, dass die Build-Files mit Ant alle recht ähnlich aufgebaut sind, dabei allerdings einige Unterschiede aufweisen. Es ist lästig und fehlerträchtig, die gleichen Schritte immer wieder erneut formulieren zu müssen.

Nachteil 2: Aufwendiges Dependency Management

Wenn man Ant nutzt oder mit Eclipse-Projekten arbeitet, so muss man die für ein Projekt benötigten Bibliotheken (JAR-Dateien) übers Netz herunterladen, in einem separaten `lib`-Verzeichnis des Projekts bereitstellen und insbesondere auch selbst verwalten. Gerade bei größeren Projekten mit vielen Abhängigkeiten wird das Vorgehen recht mühselig und fehleranfällig. Das gilt vor allem dann, wenn ein Projekt diverse Abhängigkeiten zu anderen Projekten besitzt, die wiederum externe Bibliotheken einbinden. Schnell wird dies kompliziert und erfordert einiges an Konfigurationsarbeit. Erschwerend kommt hinzu, dass alle Abhängigkeiten erneut überprüft und gegebenenfalls angepasst werden müssen, wenn ein Versionswechsel einer Bibliothek erforderlich ist. Möglicherweise sind als Folge davon dann andere Bibliotheken heranzuziehen oder es fallen einige weg.

Stellen wir uns beispielsweise vor, wir wollten eine Verbindung zu einer Datenbank aufbauen und dazu Hibernate nutzen. Gemäß der vorgestellten Verzeichnisstruktur sollten wir im Verzeichnis `lib` alle dazu benötigten Bibliotheken finden. Für Hibernate 3 wird etwa folgender Verzeichnisbaum in Ihrem Projekt benötigt:

```
lib
    hibernate3.jar
lib/jpa
    hibernate-jpa-2.0-api-1.0.0.Final.jar
lib/required
    antlr-2.7.6.jar
    commons-collections-3.1.jar
    dom4j-1.6.1.jar
    …
    slf4-api-1.6.1.jar
```

Nehmen wir an, Sie wollten nun auf die aktuellere Version 4 von Hibernate wechseln. Diese bringt neben Erweiterungen vor allem auch eine neue Verzeichnisstruktur und andere Abhängigkeiten von Fremdbibliotheken mit sich. Wenn dies alles von Hand aktuell gehalten werden muss, entsteht ein großer Aufwand und die Fehleranfälligkeit erhöht sich. Somit führt jeder Versionswechsel oder eine Änderung von Abhängigkeiten zu einem unguten Gefühl. Teilweise wird sogar auf ein Update verzichtet oder dieses lange hinausgezögert, einfach, um möglichen Problemen aus dem Weg zu gehen. Eine Verbesserung erreicht man, wenn man Apache Ivy zum Dependency Management einsetzt. Weitere Informationen finden Sie unter `http://ant.apache.org/ivy/`. Abhängigkeiten

beschreibt man dann wie folgt:

```
<dependencies>
  <dependency org="org.hibernate" name="hibernate-core" rev="4.3.6.Final"/>
  <dependency org="commons-lang" name="commons-lang" rev="2.0"/>
  <dependency org="junit" name="junit" rev="4.11"/>
</dependencies>
```

Fazit

Ant eignet sich für kleinere bis mittelgroße Projekte, die eine überschaubare Anzahl an externen Abhängigkeiten besitzen. Wenn sich diese jedoch häufiger ändern oder die Projektgröße zunimmt, empfiehlt sich Ant immer weniger.

Daher wollen wir uns nach einem kurzen Blick auf Maven anschließend mit Gradle als Build-Werkzeug befassen, das sich gleichermaßen für kleine und große Projekte eignet, seine Vorzüge aber insbesondere für größere, komplexere Projekte mit vielschichtigen Abhängigkeiten ausspielen kann.

2.7.2 Maven im Überblick

Nachdem wir einen Eindruck von Ant gewonnen haben, werfen wir in diesem Abschnitt einen Blick auf Maven, das unter `http://maven.apache.org/index.html` frei bezogen werden kann.

Erfahrungsgemäß besitzen viele Projekte doch recht ähnliche Anforderungen an einen Build-Lauf, nämlich Java-Klassen kompilieren, Tests ausführen und daraus ein Produkt (auch ***Artefakt*** genannt), z. B. in Form eines JARs, erzeugen. Wir haben bereits erkannt, dass es wünschenswert wäre, diesen Ablauf nicht für jedes Projekt erneut beschreiben zu müssen, wie dies für Ant notwendig ist. Maven räumt hier auf, indem es einen Standardablauf vorgibt, der für sehr viele Projekte geeignet ist, wodurch sich das Build-Management von Softwareprojekten vereinfacht. Maven setzt dazu stark auf ***Convention over Configuration***, d. h., es sind sinnvolle Standards definiert, und man muss nur selten steuernd eingreifen. Das basiert auf drei Eckpfeilern:

1. **Verzeichnislayout** – Es wird ein einheitliches Layout des Projektverzeichnisses erwartet, sodass immer klar ist, wo sich welche Dateien befinden und welche Verzeichnisse beim Build-Lauf angelegt werden.
2. **Abfolge im Build-Lauf** – Der Build-Lauf ist durch eine standardisierte, wohldefinierte Abfolge verschiedener Schritte im Build-Prozess festgelegt.
3. **Dependency Management** – Nahezu immer benötigt ein Projekt auch externe Bibliotheken oder Artefakte. Ein sogenanntes ***Repository*** dient dazu, versionierte Artefakte bereitzustellen, also Artefakte zu speichern und zu verwalten. Mithilfe von Repositories und von dort bereitgestellten Artefakten erleichtert Maven das Dependency Management enorm.

Aufbau des Projektverzeichnisses

Mithilfe von Maven wurden viele Dinge standardisiert und vereinfacht. Dafür sind gewisse Konventionen einzuhalten, etwa muss Maven wissen, in welchem Verzeichnis die zu kompilierenden Klassen und Tests zu finden sind. Die dazu benötigte einheitliche Projektstruktur wurde in Abschnitt 2.2 vorgestellt.

Build-Lauf, Project Object Model (POM) und Artefakte

Die Build-Ausführung ist bei Maven in verschiedene voneinander abhängige Phasen unterteilt. Für Java-Projekte existieren unter anderem folgende:

- `clean` – Führt Aufräumarbeiten aus, löscht temporäre oder im vorherigen Build-Lauf erzeugte Dateien und Verzeichnisse.
- `compile` – Kompiliert die Sourcen (aber nicht die Testklassen).
- `test` – Kompiliert die Testklassen und führt die die enthaltenen Testfälle aus.
- `package` – Dient zum Erstellen eines Artefakts, z. B. eines JARs. Dazu müssen zuvor die Sourcen kompiliert und die Tests ausgeführt werden.

Damit Maven weiß, was in jedem der aufgelisteten Schritte zu tun ist, lassen sich gewisse Vorgaben definieren. Dazu dient die Datei `pom.xml`. Dort wird das sogenannte Project Object Model (POM) eines Projekts definiert, das in XML codiert ist und festlegt, wie das zu erstellende Artefakt heißt, welche Versionsnummer es trägt, welche Abhängigkeiten von externen Bibliotheken existieren usw.

Eine Hello-World-Applikation könnte etwa folgende `pom.xml`-Datei besitzen:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org
      /2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/
        maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>de.javaprofi.helloworld</groupId>
  <artifactId>helloworld</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>helloworld</name>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Wie man sieht, ist die Build-Datei viel kürzer als diejenige für Ant und dabei sogar viel mächtiger, weil damit ein vollständiger Build-Lauf inklusive Packaging

ausgedrückt wird. Der Einsatz von XML erschwert es aber, den Nutzinhalt aus den visuell dominierenden Tags zu extrahieren. Darüber hinaus sind einige nicht sofort verständliche Begriffe, wie `groupId`, `artifactId` usw. enthalten, auf die ich nun eingehe.

Wichtige Begrifflichkeiten bei Maven

Um mit Maven produktiv zu arbeiten und die nachfolgenden Beispiele leichter nachvollziehen zu können, müssen wir zunächst ein paar elementare Begriffe kennenlernen.

Basierend auf der Standardverzeichnisstruktur wird durch einen Maven-Build in der Regel genau ein Artefakt erzeugt, etwa ein JAR, ZIP, WAR usw. Um Artefakte unterscheiden und versionieren zu können, werden drei Identifikationen genutzt:

1. `groupId` – Eine Gruppenkennung, ähnlich einer Package-Hierarchie oder eines Namespaces, z. B. `de.javaprofi.helloworld`
2. `artifactId` – Die Kennung eines Artefakts, z. B. `helloworld`
3. `version` – Eine Versionskennung, z. B. `3.8.1` [20]

Mithilfe dieser Informationen wird beim Build-Lauf der Name des Artefakts konstruiert. Darüber hinaus können die Angaben auch dazu genutzt werden, Abhängigkeiten des zu erstellenden Artefakts von anderen Artefakten, etwa den JARs einer externen Bibliothek, zu beschreiben. Dies wurde oben schon in der Sektion `dependencies` genutzt, um eine Abhängigkeit zu JUnit in der Version 4.11 zu definieren. Die Angabe von `<scope>test</scope>` erlaubt es, die Art der Abhängigkeit genauer festzulegen. Hier gilt, dass die Abhängigkeit nur in der Phase des Testens benötigt wird.

Abhängigkeiten und Dependency Management

Wir wollen nun einen Blick darauf werfen, wie man mit Maven Abhängigkeiten des eigenen Projekts von anderen Bibliotheken (Artefakten) beschreiben kann. Während das bei Ant noch aufwendig und fehlerträchtig von Hand verwaltet werden musste, wird von Maven eine leistungsfähige Automatik bereitgestellt, die darauf basiert, dass Abhängigkeiten deklarativ beschrieben werden. Dazu hinterlegt man im POM einen Abschnitt wie folgt (hier für eine externe Abhängigkeit zu JUnit in Version 4.11):

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>
</dependencies>
```

Für das Auflösen ist dann Maven zuständig und stellt dafür eine leistungsfähige Unterstützung bereit. Neben direkten Abhängigkeiten machen einem bei einer

händischen Verwaltung vor allem transitive Abhängigkeiten (also die Abhängigkeiten der direkten Abhängigkeiten) das Leben schwer. In diesem Beispiel benötigt JUnit 4.11 die Bibliothek Hamcrest 1.1. Hier profitiert man von Maven: Weil jedes Artefakt seine Abhängigkeitsinformation bei sich trägt, kann Maven transitive Abhängigkeiten automatisch auflösen, sodass diese nicht explizit aufgeführt werden müssen.

Abhängigkeiten mit Repositories auflösen

Wie eingangs erwähnt, dienen Repositories dazu, Artefakte zu speichern und zu verwalten. Auch Maven nutzt solch ein Repository. Praktischerweise existiert im Internet das Repository ***Maven Central***, in dem eine Vielzahl von Bibliotheken frei zugänglich sind.

Beim ersten Build-Lauf werden von Maven automatisch alle benötigten Abhängigkeiten (und gegebenenfalls auch benötigte Plugins) aus dem Internet nachgeladen. Um nachfolgende Build-Läufe zu beschleunigen und für folgende Build-Läufe eine Unabhängigkeit vom Internet zu schaffen, werden die heruntergeladenen Artefakte in einem lokalen Repository auf dem eigenen Rechner zwischengespeichert.

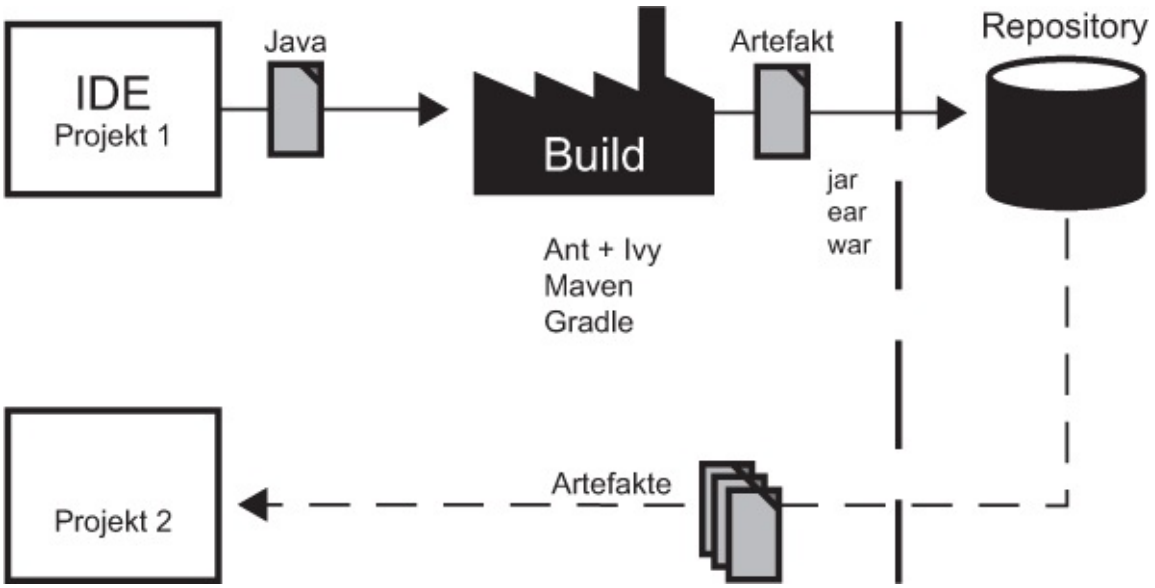


Abbildung 2-14 *Artefakterzeugung und -verwaltung beim Build-Prozess*

Maven am Beispiel

Mittlerweile haben wir alles an Informationen zusammen, um mit einem Beispiel beginnen zu können. Dabei werden Sie die Vorzüge von Maven erfahren und schätzen lernen. Es beginnt damit, dass Sie bereits für das Anlegen eines neuen Projekts inklusive Verzeichnisstruktur eine gute Unterstützung bekommen. Dadurch müssen Sie sich nicht immer wieder daran erinnern, wie denn die korrekte Verzeichnisstruktur aussieht.

Anlegen eines Projekts

Für ein Hello-World-Beispiel geben wir Folgendes ein:

```
mvn archetype:generate
    -DgroupId=de.javaprofi.helloworld
    -DartifactId=helloworld
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DinteractiveMode=false
```

Wenn Sie dieses Kommando zum ersten Mal ausführen, dauert das Ganze etwas länger, da sich Maven verschiedene Artefakte und andere benötigte Dateien aus dem Internet (vom zentralen Repository) besorgen und in einem lokalen Repository hinterlegen wird.

Schauen wir doch mal, was Maven als Reaktion auf das Kommando durchgeführt hat. Zunächst ist ein Verzeichnis `helloworld` entstanden. Dort finden wir wiederum eine Datei `pom.xml` und ein Verzeichnis `src` mit den Unterverzeichnissen `main/java` und `test/java`. Das entspricht der Standardverzeichnisstruktur aus Abschnitt 2.2:

```
helloworld
¦
¦   pom.xml
¦
+-src
    +-main
    ¦   +-java
    ¦       +-de
    ¦           +-javaprofi
    ¦               +-helloworld
    ¦                       App.java
    ¦
    +-test
        +-java
            +-de
                +-javaprofi
                    +-helloworld
                            AppTest.java
```

Man erkennt sehr schön die Separierung von Sourcen (`src/main/java`) und Testcode (`src/test/java`). Auch wurde eine Applikationsklasse `App.java` inklusive zugehörigem Test `AppTest.java` erstellt. Darüber hinaus wurde die Projektbeschreibung in Form der Datei `pom.xml` generiert.

Es ist schon recht beeindruckend, was Maven so alles automatisch angelegt hat. Das bietet einen guten Ausgangspunkt, um mit einem Java-Projekt zu starten. Einziger Wermutstropfen ist die dafür notwendige, jedoch etwas kryptische Aufruffolge von Parametern. Aber dafür lässt sich das Ergebnis sehen.

Erzeugen eines Artefakts

Wir wollen nun ein Artefakt mit Maven erzeugen, wechseln dazu in das gerade angelegte Verzeichnis `helloworld` und geben dann

```
mvn package
```

ein. Auch hier werden Sie – wie beim Kompilieren – bei der ersten Ausführung einige

Zeit warten müssen, da Maven nochmals diverse benötigte Dateien aus dem Internet nachlädt. Wenn der Build-Lauf abgeschlossen ist, wurde ein Verzeichnis `target` mit folgendem Inhalt angelegt:

```
helloworld
¦
+-target
    ¦   helloworld-1.0-SNAPSHOT.jar
    ¦
    +-classes
    ¦   +-de
    ¦       +-javaprofi
    ¦           +-helloworld
    ¦                   App.class
    ¦
    +-maven-archiver
    ¦       pom.properties
    ¦
    +-surefire
    +-surefire-reports
    ¦       de.javaprofi.helloworld.AppTest.txt
    ¦       TEST-de.javaprofi.helloworld.AppTest.xml
    ¦
    +-test-classes
        +-de
            +-javaprofi
                +-helloworld
                        AppTest.class
```

Wie man daran sieht, kompiliert Maven die Klassen in das Zielverzeichnis `classes`. Außerdem führt Maven Unit Tests aus und protokolliert deren Ergebnisse im Verzeichnis `surefire-reports`. Verläuft all dies ohne Fehler, so werden die Projektdateien in Form eines JARs bereitgestellt. Maven erzeugt ein lauffähiges ausführbares Programm, was wir nun wie folgt starten:

```
java -cp target/helloworld-1.0-SNAPSHOT.jar de.javaprofi.helloworld.App
```

Als Ausgabe erscheint der Text "`Hello World!`" auf der Konsole. Damit wollen wir unsere Kurzeinführung in Maven beenden und ziehen ein Fazit.

Fazit

Wir haben einen ersten Eindruck davon gewinnen können, wie Maven mithilfe von Convention over Configuration und durch die Vorgabe sinnvoller Voreinstellungen und Abläufe den Build im Gegensatz zu Ant deutlich vereinfacht.

Allerdings ist die Beschreibung des Projekts mithilfe einer `pom.xml` etwas geschwätzig und teilweise leicht unübersichtlich. Das gilt vor allem bei zunehmender Komplexität des Projekts und vielen Abhängigkeiten. Für den einen oder anderen erfordert Maven aufgrund der Abkehr von der befehlsbasierten Beschreibung des Builds zunächst einen etwas höheren Einarbeitungsaufwand als Ant. Jedoch besitzt Ant wiederum bekanntermaßen Schwächen, etwa den leicht uneleganten, imperativen Stil über Kommandos in XML-Syntax. Daher schauen wir nun auf das Build-Tool Gradle.

2.7.3 Builds mit Gradle

In diesem Abschnitt stelle ich einige Grundlagen zum Build-Tool Gradle vor. Wir werden dieses Wissen nutzen, um die Beispielapplikationen dieses Buchs unabhängig von der eingesetzten IDE kompilieren und starten zu können.

Installation

Gradle kann frei unter `http://www.gradle.org/downloads`[21] bezogen werden, wobei seit November 2014 die Version 2.2 aktuell ist, die bereits Java 8 unterstützt und daher für dieses Buch prädestiniert ist. Nachdem Sie den Download abgeschlossen haben, entpacken Sie das Archiv in einen beliebigen Ordner, z. B. `C:\tools\gradle`. Danach setzen Sie die Umgebungsvariable `GRADLE_HOME` auf dieses Installationsverzeichnis und ergänzen in der `PATH`-Variable den Pfad `%GRADLE_HOME%/bin`. Öffnen Sie eine Konsole und tippen Sie dort das Kommando `gradle` ein. Daraufhin sollte Gradle starten und in etwa folgende Ausgaben produzieren:

```
:help

Welcome to Gradle 2.2.

To run a build, run gradle <task>  …

To see a list of available tasks, run gradle tasks

To see a list of command-line options, run gradle –help

BUILD SUCCESSFUL

Total time:  2.046  secs
```

Erscheint diese Ausgabe, so haben Sie das Build-Tool Gradle erfolgreich installiert und wir können unsere Entdeckungsreise zu Gradle mit einem konkreten Beispiel beginnen. Falls Sie Gradle jedoch nicht starten können, so prüfen Sie nochmals die korrekten Angaben in den Umgebungsvariablen und konsultieren Sie die Gradle-Homepage.

Beispielprojekt

Wir werden gleich sehen, wie man ein Projekt mit Gradle per Hand aufsetzt, um dabei verschiedene Dinge zu lernen. Natürlich ist es ähnlich wie bei Maven auch mit Gradle möglich, sich ein Projekt samt Build-Skript erstellen zu lassen. Darauf geht der folgende Praxistipp »Einrichten eines initialen Projekts« kurz ein.

Bevor wir Builds mit Gradle kennenlernen werden, werfen wir zunächst einen Blick auf das zu verwaltende Beispielprojekt `GradleExample`. Die Verzeichnisstruktur folgt den Maven-Richtlinien. Im Source-Ordner `src/main/java` finden wir zwei Java-Klassen, für die es momentan noch keine

Ressourcen und Tests gibt und daher auch keine `resources`- und `src/test/java`-Unterverzeichnisse existieren.

```
GradleExample
¦
+-src
    +-main
        +-java
            +-de
                +-javaprofi
                    +-gradle
                            HelloGradle.java
                            HelloGradle2.java
```

Beide Java-Klassen besitzen eine extrem einfache Implementierung und führen nur eine Ausgabe auf der Konsole aus:

```
package de.javaprofi.gradle;

public class HelloGradle
{
    public static void main(final String[] args)
    {
        System.out.println(getMessage());
    }

    public static String getMessage()
    {
        return "Hello Gradle World";
    }
}
```

Die zweite Java-Klasse wird hier nicht gezeigt, da sie nahezu identisch ist und lediglich einen leicht abweichenden Text ausgibt.

Tipp: Einrichten eines initialen Projekts

Zum Einrichten eines initialen Projekts sind bei Gradle im Vergleich zu Maven abermals weniger Details und Angaben notwendig:

```
gradle init –type java-library
```

Dieses Kommando legt die Verzeichnisse für Sourcen und Test sowie die benötigten Build-Dateien mitsamt einem Verweis auf ein spezielles Repository zum Auflösen von Abhängigkeiten an.

Bis Gradle 2.0 erfolgten die Verweise auf Maven Central (`mavenCentral()`) und ab Gradle 2.1 wird auf JCenter (`jcenter()`) verwiesen, das ein Superset von Maven Central ist und somit alle dortigen Dependencies bereitstellt.

Das erste Build-Skript

Nun wollen wir unsere Java-Klassen mit Gradle kompilieren und daraus ein JAR bauen. Ähnlich wie bei Ant und Maven werden die Abläufe mithilfe einer speziellen Build-Datei (bzw. eines Build-Skripts) festgelegt, die standardmäßig `build.gradle`

heißt. Sie muss im Hauptverzeichnis des Projekts liegen, also für das Beispielprojekt im Verzeichnis `GradleExample`. In dem Build-Skript wird der Ablauf des Builds durch verschiedene Tasks mithilfe einer auf Groovy basierenden eigenen DSL (Domain Specific Language) beschrieben. Ein solcher Task[22] ist eine Aufgabenbeschreibung, etwa das Kompilieren von Sourcen oder das Ausführen von Tests.

Praktischerweise gibt es schon einige vordefinierte Tasks, die man über sogenannte Plugins in die eigene Build-Datei integrieren kann.[23] Basierend auf dem Plugin für Java schreiben wir die Build-Datei namens `build.gradle` folgendermaßen:

```
apply plugin: 'java'     // Hinweis: Einfache Anführungszeichen
```

Auch wenn es kaum zu glauben ist, diese eine Zeile stellt die vollständige Build-Datei dar, die neben dem Kompilieren und Erstellen eines JARs noch diverse weitere Funktionalität in Form vordefinierter Tasks bietet. Schauen wir uns die Möglichkeiten an.

Anzeige aller verfügbaren Tasks

Um die Installation zu prüfen, hatten wir das Kommando `gradle` ausgeführt. In der Konsolenausgabe wurde das Kommando

```
gradle tasks
```

angepriesen, um die verfügbaren Tasks anzuzeigen. Führen wir dieses Kommando aus, so wird folgende, beeindruckende Liste verfügbarer Tasks auf der Konsole ausgegeben (hier zur Lesbarkeit etwas mit … gekürzt und leicht umformatiert):

```
--------------------
All tasks runnable from root project
--------------------

Build tasks
-----
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that …
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles classes 'main'.
clean - Deletes the build directory.

jar - Assembles a jar archive containing the main classes.
testClasses - Assembles classes 'test'.

Build Setup tasks
------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Documentation tasks
-------
javadoc - Generates Javadoc API documentation for the main source code.
```

```
Help tasks
----
dependencies - Displays all dependencies declared in root project 'Gradle…
dependencyInsight - Displays the insight into a specific dependency in root …
help - Displays a help message
projects - Displays the sub-projects of root project 'GradleExample'.
properties - Displays the properties of root project 'GradleExample'.
tasks - Displays the tasks runnable from root project 'GradleExample'.

Verification tasks
------
check - Runs all checks.
test - Runs the unit tests.

…

BUILD SUCCESSFUL
```

Anhand der Ausgaben erkennt man, dass Gradle das aktuelle Verzeichnis als Projektverzeichnis ansieht und deswegen basierend auf dem Verzeichnisnamen von einem gleichnamigen Projekt `GradleExample` ausgeht.

Diese Ausgabe zeigt auch die Vielzahl an Tasks, die man für Build-Läufe von Java-Projekten benötigt. Bei Ant wäre für diese Funktionalität bereits eine sehr umfangreiche Build-Datei entstanden.

Kompilieren

Probieren wir einen Build-Lauf aus und tippen Folgendes ein:

```
gradle build
```

Gradle startet seine Arbeit und es kommt zu folgenden Konsolenausgaben:

```
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test UP-TO-DATE
:check UP-TO-DATE
:build

BUILD SUCCESSFUL
```

Die Ausgaben lassen die im Build-Lauf abgearbeiteten Schritte erahnen: Zunächst werden die Java-Dateien kompiliert (Task `compileJava`) und später zu einem JAR (Task `jar`) zusammengefasst. Zwischendrin sind einige Schritte als `UP-TO-DATE` gekennzeichnet, weil es zum einen keine Ressourcen zu verarbeiten gibt und zum anderen keine Tests vorhanden sind.

Während des Builds entsteht ein Verzeichnis `build`, in dem erzeugte Dateien landen. Im Unterverzeichnis `libs` entsteht ein JAR namens `GradleExample.jar`, welches die beiden kompilierten Klassen enthält.

```
GradleExample
¦
+-build
¦    +-classes
¦    ¦    +-main
¦    ¦        +-de
¦    ¦            +-javaprofi
¦    ¦                    +-gradle
¦    ¦                        HelloGradle.class
¦    ¦                        HelloGradle2.class
¦    ¦
¦    +-dependency-cache
¦    +-libs
¦    ¦        GradleExample.jar
¦    ¦
¦    +-tmp
¦        +-jar
¦                MANIFEST.MF
```

Neben dem Kompilieren ist es auch wünschenswert, Dokumentation zu generieren und Unit Tests auszuführen. Experimentieren wir noch ein wenig herum und lernen die dazu notwendigen Gradle-Kommandos kennen.

Hinweis: Inkrementelle Builds

Als Besonderheit erkennt Gradle, ob die Ausführung eines Tasks überhaupt notwendig ist oder ob dieser zuvor schon erfolgreich ausgeführt wurde. Dann protokolliert Gradle dies mit der Ausgabe von `UP-TO-DATE` und überspringt die Ausführung, wodurch sich Build-Läufe häufig zeitlich recht kurz halten lassen. Dabei wird auch geprüft, ob der generierte Output noch unverändert vorhanden ist und ob sich die Eingabe in den Task möglicherweise geändert hat.

Javadoc generieren

Schauen wir zunächst, wie einfach man eine Javadoc zu den Klassen erzeugen kann. Wir geben dazu Folgendes ein:

```
gradle javadoc
```

Dieses Kommando erzeugt für alle Klassen des Projekts eine Javadoc-Dokumentation im Unterverzeichnis `build/docs/javadoc`.

Ausführen von Unit Tests

Um das Programm zu testen, sollten die vorhandenen Unit Tests als fester Bestandteil des Build-Laufs ausgeführt werden. Werfen wir also noch einen Blick auf das Testen und prüfen die Funktionalität der Klasse `HelloGradle` durch folgenden Unit Test:

```
package de.javaprofi.gradle;

import static org.junit.Assert.*;
import org.junit.Test;
```

```
public class HelloGradleTest
{
    @Test
    public void testGetMessage()
    {
        assertEquals( "Hello Gradle World", HelloGradle.getMessage());
    }
}
```

Diese Testklasse speichern wir im Verzeichnis `src/test/java/de/javaprofi`. Zum Ausführen des Tests geben wir Folgendes ein:

```
gradle test
```

Statt der Ausführung unseres Tests kommt es jedoch unerwartet zu Kompilierfehlern. Die Ursache liegt darin, dass die benötigte JUnit-Bibliothek nicht im Build-Lauf verfügbar ist. Dazu müssen wir eine externe Abhängigkeit angeben.

Externe Abhängigkeit spezifizieren

Zur Erinnerung wird hier nochmals gezeigt, dass man Abhängigkeiten in Maven recht ausführlich in Form mehrerer XML-Tags in etwa wie folgt angeben muss:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.11</version>
    </dependency>
</dependencies>
```

Mit Gradle lassen sich Abhängigkeiten nahezu selbsterklärend angeben. Praktischerweise kann man sich auf das zentrale Maven-Repository beziehen. Wir spezifizieren die Abhängigkeit zu JUnit während des Kompilierens und der Testausführung wie folgt:

```
repositories
{
    jcenter()
}

dependencies
{
    testCompile group: 'junit', name: 'junit', version: '4.11'
    // Variante: testCompile 'junit:junit:4.11'
}
```

Unschwer ist zu erkennen, dass diese Form viel kürzer und klarer ist. Außerdem wird der Artefaktname direkt sichtbar. Die Art der Notation hilft dabei, Fehler zu vermeiden, insbesondere solche, die bei XML in Zusammenhang mit öffnenden und schließenden Klammern, Tags usw. stehen.

Tipp: Abhängigkeiten definieren

Für einen Build können verschiedene Abhängigkeiten existieren, die sich – wie

zuvor gesehen – recht intuitiv angeben lassen. Dabei kann für jede Abhängigkeit festgelegt werden, welchen Gültigkeitsbereich sie besitzt. Das Ganze wird durch verschiedene Schlüsselwörter in der Sektion `dependencies` der Build-Datei spezifiziert. Abhängigkeiten können beim Kompilieren (`compile`), beim Ausführen (`runtime`), beim Kompilieren von Tests (`testCompile`) und beim Ausführen der Tests (`testRuntime`) existieren und durch die zuvor in Klammern angegebenen Schlüsselworte festgelegt werden. Standardmäßig sind alle Abhängigkeiten von `testRuntime` eine Obermenge von `testCompile`, was wiederum alle Abhängigkeiten von `compile` enthält. Demnach bildet `compile` die Basis und auch die kleinste Menge von Abhängigkeiten.

Führen wir nun erneut das Kommando `gradle test` aus, so werden die Abhängigkeiten aufgelöst und dazu die JUnit-Bibliotheken aus dem Internet nachgeladen. Anschließend werden die vorhandenen (automatisch aus dem Verzeichnis `src/test/java` ermittelten) Unit Tests ausgeführt und ein Test-Report generiert– sowohl in XML als auch HTML. Einen Eindruck vermittelt Abbildung 2-15.

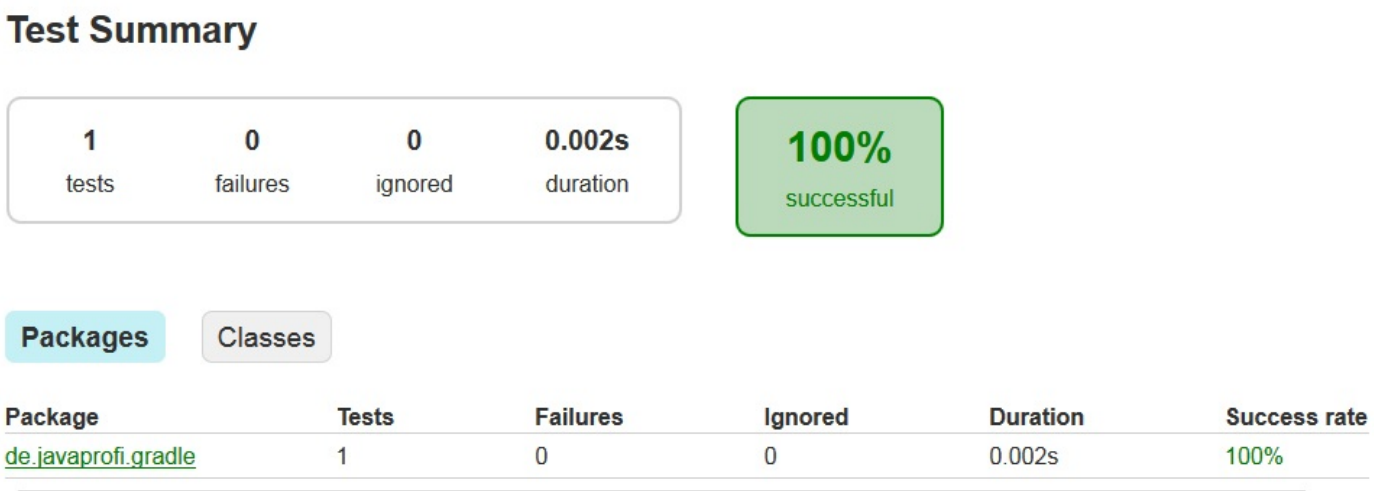


Abbildung 2-15 *Von JUnit als HTML erzeugter Bericht*

Sonderbehandlung fehlschlagender Tests

In der Regel sollte ein fehlgeschlagener Test zum Fehlschlagen des Build-Prozesses führen. Eine abweichende Handhabung ist beispielsweise dann nützlich, wenn zwar normalerweise der erfolgreiche Abschluss der Unit Tests für alle Builds verbindlich ist, jedoch während einer Prototypentwicklung einige Tests temporär fehlschlagen dürfen. Man kann dies über die Angabe von `ignoreFailure` steuern:

```
test
{
    ignoreFailures = true
}
```

JAR erstellen

Zum Erzeugen eines Artefakts kann man den `jar`-Task nutzen. Auch hier geschieht in der Regel vieles automatisch. Sollen spezielle Attribute in der Manifest-Datei gesetzt werden (vgl. Abschnitt 2.6), so kann man dies einfach wie folgt spezifizieren:

```
jar
{
    manifest
    {
        attributes ( "Implementation-Title" : "<title>",
                     "Implementation-Version": version )
    }
}
```

Oftmals enthält das JAR auch eine zu startende `main`-Klasse. Dazu schreibt man Folgendes:

```
jar
{
    manifest
    {
        attributes ( "Main-Class" : "de.javaprofi.gradle.HelloGradle" )
    }
}
```

Teilweise enthält ein JAR aber auch mehrere Klassen mit `main()`-Methoden. Die Frage ist nun, wie man die `main()`-Methode einer beliebigen, gewünschten Klasse aus einem JAR ausführen kann. Dazu lernen wir nun die Definition eines eigenen Tasks kennen.

Eigene Tasks definieren – Abhängigkeiten zwischen Tasks

Bis hierher haben wir bereits diverse Funktionalität allein durch Gradle-Basics realisieren können, jedoch ist es mit Gradle-Bordmitteln nicht direkt möglich, die `main()`-Methoden verschiedener Klassen ausführen zu können. Diese Funktionalität benötigen wir aber beispielsweise zum Starten der Beispielprogramme dieses Buchs: Die einzelnen JARs zu den Kapiteln dieses Buchs enthalten in der Regel eine Vielzahl startbarer Demo-Applikationen.

Zur Vervollständigung für unseren Build-Lauf nutzen wir das Schlüsselwort `task`, um einen eigenen Task zu definieren, den wir zum Starten von `main()`-Methoden beliebiger Klassen verwenden. Den neuen Task lassen wir auf dem vordefinierten Typ `JavaExec` basieren und erweitern diesen um die benötigte Funktionalität bzw. hier die Angaben zu `CLASSPATH` und `main`-Klasse wie folgt:

```
task helloGradle(type: JavaExec) { // <- öffnende Klammer muss aufgrund von
                                   // Groovy-Besonderheiten hier stehen
    dependsOn jar

    classpath = files("${buildDir}/libs/GradleExample.jar")
    main = "de.javaprofi.gradle.HelloGradle" // Achtung: ohne .class
}
```

Wie man bei der Definition sieht, existieren gewisse Abhängigkeiten der Tasks untereinander. Im obigen Beispiel soll beim Aufruf von `gradle helloGradle`

zunächst der Task `jar` abgearbeitet werden, um die Applikation zu bauen. Die Reihenfolge zwischen Tasks lässt sich durch das Schlüsselwort `dependsOn` festlegen.[24] Damit wird definiert, dass etwa das Kompilieren vor dem Testen erfolgt oder aber das Bereinigen von Verzeichnissen vor dem Kompilieren.

Während die Angabe der `main`-Klasse noch recht intuitiv ist, aber ohne `.class` erfolgen muss, bedarf es bei der Angabe des zu nutzenden `CLASSPATH` noch eines kleinen Hinweises. Dort finden wir die Methode `files()`. Sie ist Bestandteil der Gradle DSL und erlaubt es, eine Menge von Dateien zu spezifizieren. In diesem Fall ist es genau eine Datei, nämlich `GradleExample.jar` im Unterverzeichnis `${buildDir}/libs`, wobei hier ein Platzhalter für das Build-Verzeichnis genutzt wird.

Tipp: Unvollständige Task-Namen

Bei den ersten Fingerübungen mit Gradle kann eine Eigenschaft zunächst irritieren: Zur Ausführung von Tasks muss nicht immer der vollständige Name des Tasks angegeben werden. Stattdessen reicht es aus, nur so viel anzugeben, dass der Task dadurch eindeutig identifiziert werden kann. Im obigen Beispiel könnte man statt `gradle helloGradle` auch kürzer `gradle hello` schreiben und dies würde ebenso funktionieren.

Code-Checker-Tool einbinden

Mit recht wenig Aufwand haben wir schon einen vollständigen Build-Lauf beschreiben können. Als Schmankerl möchte ich noch ein paar Sourcecode-Prüfungen ergänzen. Dazu nutzen wir das Tool PMD, für welches Gradle ein Plugin bereitstellt, sodass wir lediglich folgende Zeile in unserer Datei `build.gradle` ergänzen müssen:

```
apply plugin:  'pmd'
```

Führen wir den Build aus, entstehen im Verzeichnis *build/reports/pmd* einige Report-Dateien. Allerdings zeigen diese keine Verstöße. Das liegt schlichtweg daran, dass PMD noch keine Regeln konfiguriert hat. Wir fügen wie folgt ein paar Regeln ein:

```
pmd
{
    toolVersion = '5.1.1'
    ruleSets = ["java-basic", "braces", "design"]
    ignoreFailures = true
}
```

Führen wir nun erneut einen Build aus, so werden uns Verstöße gemeldet sowie das Verzeichnis, wo wir eine detaillierte HTML-Seite dazu finden:

```
:pmdMain
2 PMD rule violations were found. See the report at:
file:///C:/Users/min/Deskto
p/GradleExample/build/reports/pmd/main.html
/pmd/main.html
```

```
:pmdTest
```

Als erster Einstieg in die Sourcecode-Prüfung soll der Aufruf von PMD hier genügen. Später in Abschnitt 19.4 behandle ich das Thema ausführlicher.

Hinweis: Gradle 2 integriert verschiedene neue Tool-Versionen

Neben dem Support für Java 8 wurden die mit Version 2 von Gradle mitgelieferten Tool-Plugins einem Update unterzogen, unter anderem folgende:

- Checkstyle: 5.7
- Findbugs: 2.0.3 (mit Gradle 2.1 ist es Findbugs 3.0.0)
- PMD: 5.1.1

IDE-Projekte erzeugen

Bisher haben wir noch keine Integration in die IDE betrachtet. Weil die Verwaltung und Bearbeitung von Sourcecode mithilfe von IDEs komfortabler ist, wäre es wünschenswert, wenn wir basierend auf dem Projektstand entsprechende Dateien für unsere IDE generieren könnten. Tatsächlich ist dies mit Gradle sowohl für Eclipse als auch IntelliJ IDEA möglich.

Eclipse-Projekt erzeugen und einbinden

Um das Gradle-Plugin zu integrieren, ergänzen wir folgende Angabe in der Build-Datei:

```
apply plugin:  'eclipse'
```

Als Folge stehen uns mit `eclipse` und `cleanEclipse` zwei weitere Tasks für Gradle zur Verfügung. Wenn wir den Task `gradle eclipse` ausführen, dann werden Eclipsespezifische Dateien, nämlich `.project` und `.classpath`, erzeugt. Darüber hinaus kümmert sich Gradle um das Dependency Management und lädt benötigte JAR-Dateien herunter und bindet diese als Referenced Libraries korrekt in Eclipse-Projekte ein.

Um ein wie eben erzeugtes Projekt in Eclipse zu importieren, wählen Sie das Kontextmenü IMPORT –> GENERAL –> EXISTING PROJECTS INTO WORKSPACE.

Tipp: Integration in Eclipse

Zu Eclipse kompatible Verzeichnisstruktur Bisher haben wir das Maven-Standardverzeichnislayout genutzt. Gradle ist aber diesbezüglich flexibel. Mit nur ein wenig Konfigurationsaufwand kann man auch Projekte mit Gradle bauen, die eine abweichende Verzeichnisstruktur besitzen, etwa eine, die dem Eclipse-Standardverzeichnislayout entspricht. Bevor allerdings Anpassungen erfolgen, sollte

man sich überlegen, ob man nicht eher das Eclipse-Projekt so konfiguriert, dass es sich an den Maven/Gradle-Standard hält.

Zur Konfiguration nutzt man die Angabe `sourceSets`. Damit wird eine Menge von Dateien beschrieben, die kompiliert werden sollen. Zudem kann man über die Angabe `output.classesDir` das Ausgabeverzeichnis konfigurieren:

```
apply plugin: 'java'

sourceSets
{
    main
    {
        java
        {
            srcDir = ['src']
            output.classesDir = ['bin']
        }
    }
}
```

Eclipse-Plugin Wir haben eben gesehen, dass man die Eclipse-Projekt-Dateien mit Gradle generieren lassen kann. Vielfach ist es praktisch, die einzelnen Tasks einer Gradle-Build-Datei direkt in Eclipse und nicht nur aus der Kommandozeile ausführen zu können. Dazu existiert unter anderem das Plugin NODEECLIPSE, das Sie im Eclipse Marketplace finden und installieren können.

Multi-Project-Builds

Je größer Softwareprojekte werden, desto mehr bietet es sich an, diese in Subprojekte aufzuteilen. Das führt zu einer besseren Modularisierung und man kann (unerwünschte) Abhängigkeiten besser kontrollieren. Die Strukturierung geschieht in etwa wie folgt:

```
MultiProject
¦
+-Subproject_1
+-Subproject_2
+-Subproject_3
```

Hier ist angedeutet, dass das Hauptprojekt in drei Subprojekte aufgegliedert ist. Eine solche Dreiteilung findet man etwa bei einer Schichtenarchitektur mit GUI, Servicebzw. Modellschicht und Datenzugriffsschicht.[25] Dabei hängt das GUI von der Serviceschicht ab und diese wiederum von der Datenzugriffsschicht.

Wenn wir dies mit einem Gradle-Build beschreiben wollen, dann enthält das Root-Projektverzeichnis lediglich die Dateien `build.gradle` und `settings.gradle`. In Letzterer werden die Subprojekte wie folgt aufgelistet:

```
include 'gui', 'model', 'dao'
```

Die Hauptdatei enthält einen Abschnitt `subprojects`, in dem die Gemeinsamkeiten für alle Subprojekte aufgeführt werden:

```
subprojects
```

```
{
    apply plugin: 'java'

    repositories
    {
        mavenCentral()
    }
}
```

Außerdem wird für jedes Subprojekt ein eigenes Verzeichnis angelegt und dort findet sich wiederum eine `build.gradle`-Datei. Diese ist dann so aufgebaut, wie es zuvor in diesem Kapitel beschrieben wurde – allerdings gibt es Varianten, sodass nicht jeweils eine separate `build.gradle`-Datei pro Subprojekt zwingend existieren muss.

```
MultiProject
¦
¦   build.gradle
¦   settings.gradle
¦
+-gui
¦   ¦   build.gradle
¦   ¦
¦   +-src
¦       +-main
¦           +-java
¦               +-de
¦                   +-gui
¦                           PersonGui.java
¦
+-model
¦   ¦   build.gradle
¦   ¦
¦   +-src
¦       +-main
¦           +-java
¦               +-de
¦                   +-model
¦                           Person.java
¦                   +-service
¦                           PersonService.java
¦
+-dao
    …
```

Man kann dann im Hauptprojekt `gradle build` eingeben und die Subprojekte werden dann gebaut. In jedem Subprojekt entsteht als Folge – wie zuvor für normale Projekte auch – im Verzeichnis `build` eine JAR-Datei.

Zwischenfazit

Ausgehend von der maximal einfachsten Build-Datei, bestehend aus einer Zeile, haben wir schrittweise den Build um sinnvolle weitere Funktionalität ergänzt. Dabei haben wir sowohl das Verwalten von externen Abhängigkeiten sowie das Ausführen von Analysetools oder Dokumentationswerkzeugen als auch das Aufbereiten von deren Ergebnissen kennengelernt. Auch der Abgleich mit den Projektdateien der

genutzten IDE wurde kurz thematisiert.

2.7.4 Vergleich von Ant, Maven und Gradle

Ich habe mit Ant, Maven und Gradle drei Build-Tools vorgestellt. Obwohl ich für neue Projekte den Einsatz von Gradle empfehle, gibt es sicherlich auch Argumente für den Einsatz von Ant oder Maven (oder sogar eines ganz anderen Build-Tools). Um die Wahl zu erleichtern, möchte ich hier die spezifischen Stärken von Ant, Maven und Gradle kurz gegenüberstellen.

Stärken von Ant

Ant eignet sich besonders, aus folgenden Gründen:

- Es erfordert wenig Einarbeitungsaufwand – zumindest für kleinere Projekte.
- Abläufe lassen sich recht feingranular per XML beschreiben.

Stärken von Maven und Gradle

Der Einsatz von Maven oder Gradle hat folgende Vorteile:

- Es erleichtert die Einarbeitung und Orientierung, da eine einheitliche Projektstruktur genutzt wird. Auch neue Mitarbeiter finden sich so schnell zurecht, denn alles ist relativ vertraut und man muss nicht ständig von Projekt zu Projekt umdenken.
- Es fördert eine Vereinheitlichung durch Convention over Configuration, wodurch Konfigurationsaufwände nur dort anfallen, wo vom Standard abgewichen wird.
- Es erlaubt, Abhängigkeiten von Fremdbibliotheken deklarativ zu beschreiben und automatisch durch das Build-Tool auflösen zu lassen (insbesondere auch transitiv, also die von einer Bibliothek benötigten anderen Bibliotheken).
- Es vereinfacht die Generierung von Dokumentation und Reports zu Sourcecode-Qualität durch Aufruf von Tools wie Javadoc, Checkstyle, PMD, FindBugs usw. Darauf gehe ich etwas genauer in Abschnitt 19.4 im Rahmen der Beschreibung zu Coding Conventions ein.

Fazit

In diesem Abschnitt haben Sie das Handwerkszeug kennengelernt, um den Sourcecode und die Unit Tests verschiedener Beispielapplikationen in den nachfolgenden Kapiteln kompilieren und ausführen zu können.

Weiter wurde motiviert, dass ein Softwareprojekt stets mit einem vernünftigen

Build-Management gekoppelt sein sollte. Alle zuvor genannten Tools bieten sich für diese Aufgabe an. Für kleine Projekte kann man durchaus auch noch Ant nutzen. Je größer das Projekt wird, desto mehr profitiert man aber von den Automatiken von Maven und Gradle. Letzteres vereinfacht den Build-Prozess auch für kleine Projekte und erfordert lediglich minimalen Einarbeitungsaufwand, sodass für dieses Buch die Wahl auf Gradle gefallen ist.

2.8 Weiterführende Literatur

Dieses Kapitel hat einen einführenden Überblick darüber gegeben, wie Sie sich eine professionelle Arbeitsumgebung einrichten. Weitere Ideen zum Ausbau einer Arbeitsumgebung, die die »agile« Entwicklung unterstützt, finden Sie in den folgenden Büchern und Onlinequellen, wobei die meistens davon detaillierte Informationen zu den Themen Versionsverwaltungen und Build-Automatisierung liefern.

- »**Agile Java-Entwicklung in der Praxis**« von Michael Hüttermann [47]
- »**CVS: Versionskontrolle und Quellcode-Management**« von Jennifer Vespermann [88]
- »**Pragmatisch Programmieren: Versionsverwaltung mit CVS**« von Andrew Hunt und David Thomas [50]
- »**Versionskontrolle mit Subversion**« von Ben Collins-Sussman, Brian W. Fitzpatrick und C. Michael Pilato [16]
- »**Pragmatic Version Control Using Subversion**« von Mike Mason [63]
- »**Git**« von René Preißel und Bjørn Stachmann [74]
- »**Mercurial: The Definitive Guide**« von Bryan O'Sullivan [73]
- »**Ant: The Definitive Guide**« von Steve Holzner [43]
- »**Pragmatisch Programmieren: Projekt-Automatisierung**« von Mike Clark [14]
- »**Gradle**« von Joachim Baumann [3]
- »**Gradle In Action**« von Benjamin Muschko [68]
- »**Git**« `http://git-scm.com/book/de`
- »**Mercurial**« `http://hgbook.red-bean.com/`
- »**Maven**« `http://books.sonatype.com/mvnref-book/reference/`

3 Objektorientiertes Design

Dieses Kapitel gibt einen Einblick in den objektorientierten Entwurf von Software. Die zugrunde liegende Idee der Objektorientierung (OO) ist es, ***Zustand*** (Daten) mit ***Verhalten*** (Funktionen auf diesen Daten) zu verbinden. Prinzipiell lässt sich diese Idee in jeder Programmiersprache realisieren. Sprachen wie z. B. C++ und Java unterstützen die objektorientierten Konzepte von Hause aus. In diesem Kapitel möchte ich Ihnen die Grundgedanken der Objektorientierung und die für gutes Design verfolgten Ziele ***Kapselung***, ***Trennung von Zuständigkeiten*** und ***Wiederverwendbarkeit*** vermitteln.

Abschnitt 3.1 stellt zunächst einige Grundbegriffe und -gedanken vor und verdeutlicht diese am Entwurf eines Zählers. Das Beispiel ist bewusst einfach gewählt. Nichtsdestotrotz kann man daran bereits sehr gut den OO-Entwurf mit seinen Tücken nachvollziehen. Die gewonnenen Erkenntnisse helfen, die OO-Grundlagen besser zu verstehen. Um Objektorientierung jedoch als Konzept zu erfassen und das ***Denken in Objekten und Zuständigkeiten*** zu verinnerlichen, muss man auch Fallstricke und Grenzen kennen. Dazu zeigt Abschnitt 3.1.3 ein paar Irrwege der »funktionalen Objektierung«[1]. Darunter versteht man, dass häufig fälschlicherweise von Objektorientierung gesprochen wird, nur weil man Klassen und Objekte verwendet. Allerdings umfasst Objektorientierung viel mehr als das Programmieren mit Klassen und den intensiven Einsatz von Vererbung. Deshalb erläutere ich in Abschnitt 3.2 einige technische Grundlagen, wie Interfaces und abstrakte Klassen, bevor wir mit diesem Wissen in Abschnitt 3.3 das Thema Vererbung erneut aufgreifen und es kritisch diskutieren. Zudem betrachten wir Situationen von »explodierenden« Klassenhierarchien und Wege, die Probleme zu lösen. In Abschnitt 3.4 thematisieren wir unter anderem Read-only-Interfaces und Immutable-Klassen als fortgeschrittenere OO-Techniken. Danach lernen wir in Abschnitt 3.5 verschiedene Prinzipien guten OO-Entwurfs als Hilfestellung auf dem Weg zu einem guten Design kennen. Dazu werden einige bekannte Designrichtlinien wie etwa das Law of Demeter sowie die SOLID-Prinzipien vorgestellt.

Mit JDK 5 wurden generische Typen, auch ***Generics*** genannt, eingeführt, um typsichere Klassen und insbesondere typsichere Containerklassen realisieren zu können. Eine wichtige Rolle zum Verständnis beim Überschreiben von Methoden, bei der Nutzung von Arrays und insbesondere beim Einsatz von Generics spielt der Begriff Varianz und verschiedene Formen davon. Diese werden in Abschnitt 3.6 vorgestellt. Abschließend bietet Abschnitt 3.7 einen Einstieg in das Thema Generics.

3.1 OO-Grundlagen

Die objektorientierte Softwareentwicklung setzt eine spezielle Denkweise voraus. Kerngedanke dabei ist, den Programmablauf als ein Zusammenspiel von Objekten

und ihren Interaktionen aufzufassen. Dabei erfolgt eine Anlehnung an die reale Welt, in der Objekte und ihre Interaktionen ein wesentlicher Bestandteil sind. Beispiele dafür sind Personen, Autos, CD-Spieler usw. All diese Dinge werden durch spezielle Merkmale und Verhaltensweisen charakterisiert. Auf die Software übertragen realisieren viele einzelne Objekte durch ihr Verhalten und ihre Eigenschaften die benötigte und gewünschte Programmfunktionalität. Bevor ich diese Gedanken für den Entwurf eines Zählers wieder aufgreife, möchte ich zuvor einige Begriffe definieren, die für ein gutes Verständnis dieses und der folgenden Kapitel eine fundamentale Basis darstellen.

3.1.1 Grundbegriffe

Klassen und Objekte

Sprechen wir beispielsweise über ein spezielles Auto, etwa das von Hans Mustermann, so reden wir über ein konkretes ***Objekt***. Sprechen wir dagegen abstrakter von Autos, dann beschreiben wir eine Klasse. Eine ***Klasse*** ist demnach eine Strukturbeschreibung für Objekte und umfasst eine Sammlung von beschreibenden ***Attributen*** (auch ***Membervariablen*** genannt) und ***Methoden***, die in der Regel auf diesen Daten arbeiten und damit Verhalten definieren. Ein Objekt ist eine konkrete Ausprägung (***Instanz***) einer Klasse. Die Objekte einer Klasse unterscheiden sich voneinander durch ihre Adresse im Speicher und eventuell in den Werten ihrer Attribute.

Objektzustand

Der ***Objektzustand*** beschreibt die momentane Wertebelegung aller Attribute eines Objekts. Er sollte in der Regel nur durch Methoden des eigenen Objekts verändert werden, um Veränderungen leichter unter Kontrolle zu halten.

Schnittstelle bzw. Interface

Häufig werden die Begriffe Schnittstelle und ***Interface*** analog zueinander genutzt. Mit beidem sind diejenigen Methoden einer Klasse gemeint, die von anderen Klassen aufgerufen werden können. Schnittstellen definieren ein »Angebot von Verhalten«. Java kennt das Schlüsselwort `interface` zur expliziten Definition der Schnittstelle einer Klasse in Form einer ***Menge von Methoden ohne Implementierung***.[2] Man spricht hier von ***abstrakten Methoden***. Eine Klasse, die das Interface implementiert, muss diese abstrakten Methoden mit Funktionalität füllen.

Abstrakte Klassen

Ähnlich wie Interfaces erlauben ***abstrakte Klassen*** die Vorgabe einer Schnittstelle in Form von Methoden. Im Unterschied zu Interfaces können in abstrakten Klassen

nicht nur Methoden implementiert werden,[3] sondern auch Attribute definiert werden, um eine Basisfunktionalität anzubieten. Methoden, die noch keine Implementierung besitzen, müssen dann mit dem Schlüsselwort `abstract` gekennzeichnet werden. Aufgrund der unvollständigen Implementierung können keine Objekte abstrakter Klassen erzeugt werden und die Klasse muss ebenfalls mit dem Schlüsselwort `abstract` versehen werden. Dadurch sind abstrakte Klassen nicht mehr instanziierbar. Das gilt übrigens auch für den Spezialfall, dass keine einzige Methode, sondern nur die Klasse `abstract` definiert ist.

Typen

Durch Klassen und Interfaces werden eigenständige ***Datentypen*** oder kurz Typen mit Verhalten beschrieben, etwa eine Klasse `Auto`. Im Unterschied dazu existieren ***primitive Datentypen***, die lediglich Werte ohne Verhalten darstellen. Beispiele hierfür sind die Zahlentypen `int`, `float` usw.

Realisierung

Das Implementieren eines Interface durch eine Klasse wird ***Realisierung*** genannt und durch das Schlüsselwort `implements` ausgedrückt.Eine Realisierung eines Interface durch eine Klasse bedeutet, dass sich ein Objekt dieser Klasse so verhalten kann, wie es das Interface vorgibt. Daher spricht man von ***Typkonformität*** oder auch von einer ***»behaves-like«-Beziehung*** bzw. ***»can-act-like«-Beziehung***. Wichtig dafür sind zum einen sprechende Methodennamen sowie zum anderen eine aussagekräftige Dokumentation der jeweiligen Methode. Für ein gelungenes Design reicht es also nicht aus, dass ein Interface nur aus Methodendeklarationen besteht!

Deklaration und Definition

Eine ***Deklaration*** beschreibt bei Variablen deren Typ und ihren Namen, etwa `int age`. Bei Methoden entspricht eine Deklaration dem Methodennamen, den Typen der Übergabeparameter sowie angegebenen Exceptions. Diese Elemente beschreiben die ***Signatur*** einer Methode. Von einer ***Definition*** einer Variablen spricht man, wenn ihr bei der Deklaration ein Wert zugewiesen wird. Die Definition einer Methode entspricht der Methodenimplementierung.

Info: Signatur einer Methode

Unter der Signatur einer Methode versteht man die Schnittstelle, die aus dem Namen, der Parameterliste (Anzahl, Reihenfolge und Typen der Parameter) und optional angegebenen Exceptions besteht. Betrachten wir folgende Methode `open()`:

```
boolean  open (final String filename, final int retries) throws IOException
{
    // …
```

In Java gehören weder die Implementierung, der Rückgabewert noch die Namen der Parameter und die `final`-Schlüsselworte zur Signatur, sondern nur Folgendes:

```
open (String, int) throws IOException
```

Jetzt fragen Sie sich vielleicht, warum das so ist. Nehmen wir dazu folgende weitere Methodensignatur einer `open()`-Methode an:

```
open(File, boolean) throws IOException
```

Erfolgt ein Aufruf an die Methode `open()`, so kann der Compiler basierend auf den Typen der Parameterliste die korrekte Methode auswählen. Der Rückgabewert trägt nicht zur Unterscheidung beim Aufruf bei, weil er nicht zwangsweise ausgewertet werden muss (in der Regel aber sollte). Somit ist er nicht Bestandteil der Signatur.

Referenzen

Definiert man in Java eine Variable vom Typ einer Klasse, so stellt diese nicht das Objekt selbst dar, sondern nur eine Referenz auf das Objekt. Eine solche Variable ist ein Verweis, um das Objekt zu erreichen[4] und mit dessen Daten und Methoden zu arbeiten. Abbildung 3-1 zeigt dies für eine Referenzvariable `myBrother`, die auf ein Objekt vom Typ `Person` verweist und somit Zugriff auf dessen Attribute (hier `name` und `age`) sowie dessen (nicht dargestellte) Methoden erlaubt.

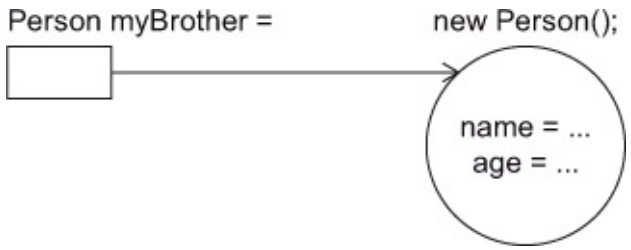


Abbildung 3-1 *Objekterzeugung und -referenzierung*

Die folgende Abbildung 3-2 zeigt die Referenzierung desselben Objekts durch mehrere Referenzvariablen, hier `myBrother` und `otherPerson`.

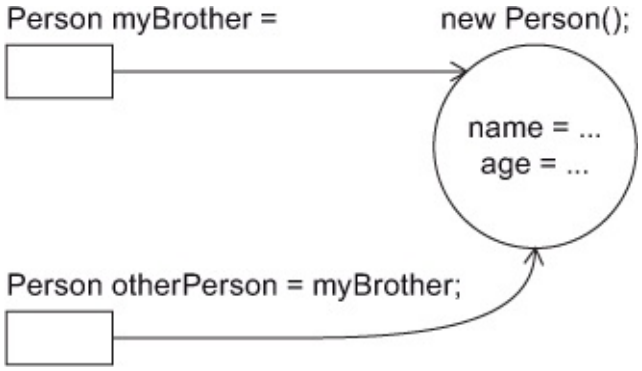


Abbildung 3-2 *Mehrfache Referenzierung eines Objekts*

Bereits beim bloßen Betrachten erahnt man mögliche Auswirkungen von Änderungen von Attributen: Wird etwa der Wert des Attributs `name` für das durch `myBrother` referenzierte `Person`-Objekt verändert wird, so wirkt sich das natürlich auch in dem durch `otherPerson` referenzierten Objekt aus. Das ist in diesem Beispiel noch

leicht verständlich, verweisen doch beide Referenzen auf *dasselbe* Objekt. In der Praxis sind die Beziehungsgeflechte zwischen Objekten oftmals komplexer, und somit lassen sich Referenzierungen und Auswirkungen von Änderungen an den Daten eines Objekts nicht immer so einfach nachvollziehen wie hier.

Lebenszyklus von Objekten und Speicherfreigabe

Der Lebenszyklus eines Objekts beginnt bei dessen Konstruktion. Danach kann es mit anderen Objekten interagieren. Irgendwann ist ein Objekt an sein »Lebensende« gelangt. Dies ist der Fall, wenn keine Referenzvariable mehr auf dieses Objekt verweist. Folglich kann es von keiner Stelle im Programm mehr erreicht und angesprochen werden. Der genaue Zeitpunkt des »Ablebens« und der damit verbundenen Speicherfreigabe kann in der JVM nicht exakt bestimmt werden. Das liegt daran, dass die Aufräumarbeiten und die Speicherfreigabe durch einen speziellen Mechanismus zur Speicherbereinigung, die sogenannte ***Garbage Collection***, erfolgen. Die konkreten Zeitpunkte der Ausführung sind nicht vorhersehbar, da diese von der gewählten Aufräumstrategie und des gerade belegten und derzeit noch freien Speichers abhängig sind. Details dazu beschreibt Abschnitt 8.5.

Sichtbarkeiten

Beim objektorientierten Programmieren unterscheidet man verschiedene Sichtbarkeiten, die festlegen, ob und wie andere Klassen auf Methoden und Attribute zugreifen dürfen. Java bietet folgende vier Sichtbarkeiten:

- `public` – von überall aus zugreifbar
- `protected` – Zugriff für abgeleitete Klassen und alle Klassen im selben Package
- Package-private oder default (kein Schlüsselwort[5]) – nur Klassen aus demselben Package haben Zugriff darauf
- `private` – nur die Klasse selbst und alle ihre inneren Klassen haben Zugriff

Kapselung

Die ***Kapselung*** von Daten (***Information Hiding***) stellt einen Weg dar, die Attribute eines Objekts vor dem direkten Zugriff und der direkten Manipulation durch andere Objekte zu schützen. Anstatt dass eine Klasse ihre Attribute `public` definiert und damit potenziellen Nutzern direkten Zugriff erlaubt, empfiehlt es sich, die Sichtbarkeit der Attribute auf `protected`, Package-private oder `private` zu reduzieren und Zugriffsmethoden (`get()` - und `set()` -Methoden) anzubieten. Diese werden im OO-Sprachjargon auch als ***Accessors*** oder ***Mutators*** bezeichnet.

Durch die Definition von Sichtbarkeitsregeln erreicht man eine Strukturierung und Kapselung: Die Klasse kann durch die Zugriffsmethoden die Möglichkeiten zum

Zugriff auf Attribute selbst steuern oder einschränken. Außerdem können verschiedene Sichtbarkeiten für Methoden vergeben werden, wodurch internes Verhalten nicht nach außen sichtbar wird, sondern nur gewünschte verhaltendsdefinierende Methoden, wie die nachfolgend beschriebenen Business-Methoden.

Objektverhalten und Business-Methoden

Das Verhalten der Instanzen einer Klasse ist durch die bereitgestellten Methoden definiert. Diese Methoden weisen in der Regel unterschiedliche Abstraktionsgrade und verschiedene Sichtbarkeiten auf. Einige Arbeits- und Hilfsmethoden verwenden viele Implementierungsdetails und arbeiten direkt mit den Attributen der Klasse. Sie sollten bevorzugt `private` oder Package-private definiert werden. Meistens realisieren nur wenige der Methoden einer Klasse »High-Level-Operationen« mit einem hohen Abstraktionsgrad. Diese verhaltensdefinierenden Methoden bilden in der Regel die Schnittstelle der Klasse nach außen. Derartige Methoden nennt man auch ***Business-Methoden***. Sie verstecken die komplexen internen Vorgänge. Auf diese Weise erreicht man folgende Dinge:

- **Abstraktion** und **Datenkapselung** – Implementierungsdetails werden versteckt.
- **Klare Abhängigkeiten** – Wenn Zugriffe durch andere Klassen nur über die Business-Methoden erfolgen, existieren wenige und klare Abhängigkeiten.
- **Austauschbarkeit** und **Wiederverwendbarkeit** – Werden die Business-Methoden durch ein Interface beschrieben, so kann ein Austausch der Realisierung (im besten Fall) sogar ohne Rückwirkung auf Nutzer erfolgen.

Tipp: Aufrufhierarchien von Methoden

Um den Sourcecode übersichtlich zu halten, sollten sich öffentliche Methoden aus Methodenaufrufen anderer Sichtbarkeiten zusammensetzen und wenig Implementierungsdetails zeigen. Sofern Methoden keine Methoden höherer Sichtbarkeit aufrufen (also etwa eine private Methode keine öffentliche Methode aufruft), erreicht man eine Aufrufhierarchie und eine Art klasseninterne Schichtenarchitektur. Diese Strukturierung hilft insbesondere dann, wenn man Datenänderungen verarbeiten und an andere Klassen kommunizieren muss. Ohne die Einhaltung dieses Hinweises kommt es schneller zu einem Chaos.

Kohäsion

Beim objektorientierten Programmieren verstehen wir unter ***Kohäsion*** den inneren Zusammenhang, also in welchem Maße eine Klasse tatsächlich genau eine Aufgabe oder einen Aufgabenbereich erfüllt. Je höher die Kohäsion, desto besser realisiert eine Klasse lediglich eine spezielle Funktionalität. Bei hoher Kohäsion erfolgt eine gute Trennung von Zuständigkeiten. Klassen mit hoher Kohäsion können normalerweise

gut kombiniert werden, um neue Funktionalitäten zu realisieren. Dies hilft bei der Wiederverwendbarkeit. Je niedriger auf der anderen Seite die Kohäsion ist, desto mehr wird unterschiedliche Funktionalität innerhalb einer Klasse realisiert. Analog können wir von Kohäsion einer Methode reden. Sie beschreibt, wie genau abgegrenzt die Funktionalität der Methode ist.

Info: Orthogonalität und Wiederverwendung

In der Informatik spricht man von ***Orthogonalität***, wenn man eine ***freie Kombinierbarkeit*** unabhängiger Konzepte – hier Methoden und Klassen – erreicht. Eine derartige Implementierung zu erstellen, erfordert allerdings einiges an Erfahrung. Häufig sind Methoden und Klassen daher in der Praxis eben leider nicht immer so gestaltet, dass sie nur genau eine Aufgabe erfüllen. Wenn man dann eine benötigte Teilfunktionalität einer Methode oder Klasse verwenden möchte, und weil man nur diese eine Aufgabe in Form eines Methodenaufrufs nicht bekommen kann, werden als Abhilfe die entsprechenden Zeilen kopiert, statt eine Methode oder Klasse mit der gewünschten Funktionalität herauszulösen. Eine derartige Sourcecode-Duplizierung (Copy-Paste-Wiederverwendung) führt häufig zu einer schwachen Kohäsion und sollte daher möglichst vermieden werden.[a]

[a] Wird eine ganze Methode in eine andere Klasse kopiert, kann sie durchaus für sich eine hohe Kohäsion aufweisen. Allerdings wird die Kohäsion auf Klassenebene dadurch geringer, weil nun zwei Klassen dasselbe tun (können). Das ist wartungsunfreundlich.

Assoziationen

Stehen Objekte in Beziehung zueinander, so spricht man ganz allgemein von einer ***Assoziation***. Diese Beziehung kann nach der Zusammenarbeit wieder aufgelöst und es können neue Assoziationen zu anderen Objekten aufgebaut werden.

Man untergliedert bei Assoziationen noch feiner in Aggregation und Komposition. Eine ***Aggregation*** verstärkt die Beziehung zwischen den Objekten, es entsteht eine ***Ganzes-Teile-Beziehung***. Insbesondere sind die verbundenen Objekte nicht mehr gleichwertig, sie können aber unabhängig voneinander existieren. Von ***Komposition*** spricht man dann, wenn diese Ganzes-Teile-Beziehung noch stärker ausgeprägt ist: Ein enthaltenes Teilobjekt kann ohne das Ganze nicht selbstständig existieren. Ganzes-Teile-Beziehungen findet man häufig in dem Zusammenhang, dass ein Objekt, also das Ganze, eine Menge von anderen Objekten, die Teile, referenziert. Je nach Modellierungsabsicht stellt die in Abbildung 3-3 dargestellte Beziehung zwischen einer Musik-CD und einigen MP3-Liedern eine Aggregation bzw. eine Komposition dar.

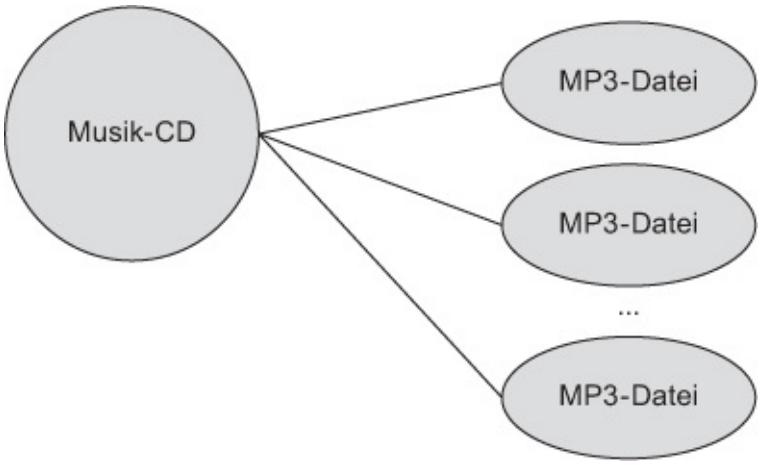


Abbildung 3-3 *Aggregation / Komposition*

Wenn man dies als Aggregation betrachtet, so existieren die MP3-Lieder unabhängig von der CD, z. B. als Dateien einer Play-Liste. Bei der Komposition sind die MP3-Lieder Bestandteil der CD in Form von Bits und Bytes auf dem Datenträger selbst.

Kopplung

Unter ***Kopplung*** verstehen wir, wie stark Klassen miteinander in Verbindung stehen, also den Grad ihrer Abhängigkeiten untereinander. Einen großen Einfluss hat, welche Attribute und Methoden sichtbar und zugreifbar sind und wie dieser Zugriff erfolgt. Zwei Klassen sind stark miteinander gekoppelt, wenn entweder viele feingranulare Methodenaufrufe der jeweils anderen Klasse erfolgen oder aber auf Attribute der anderen Klasse direkt zugegriffen wird. Viele Methodenzugriffe zur Veränderung des Objektzustands deuten auf das Designproblem mangelnder Kapselung hin. Bei Unachtsamkeit pflanzt sich diese fort und führt dazu, dass jede Klasse viele Details anderer Klassen kennt und mit diesen eng verbunden (stark gekoppelt) ist. Man kann dann von »Objekt-Spaghetti« sprechen. Häufig ist starke Kopplung durch mangelnde Trennung von Zuständigkeiten, also eine niedrige Kohäsion, begründet.

Ziel einer Modellierung ist es, eine möglichst lose Kopplung und damit geringe Abhängigkeiten verschiedener Klassen untereinander zu erreichen. Durch eine gute Datenkapselung sowie die Definition von Objektverhalten in Form von Business-Methoden erreicht man dies: Durch eine gute Kohäsion definiert man klare Zuständigkeiten, wodurch Klassen möglichst eigenständig und unabhängig von anderen werden. Mithilfe einer guten Kapselung wird dieser Effekt dahingehend verstärkt, dass weniger Realisierungsdetails für andere Klassen sichtbar sind. Als Folge kann eine kleine Schnittstelle mit wenigen Methoden zur Kommunikation mit anderen Klassen definiert werden.

Tipp: Vorteile loser Kopplung und hoher Kohäsion

Durch den Einsatz von loser Kopplung lassen sich nachträgliche Änderungen meistens einfacher und ohne größere Auswirkungen für andere Klassen umsetzen. Eine starke Kopplung führt dagegen oft dazu, dass Änderungen in vielen Klassen notwendig werden. ***Gutes OO-Design besteht darin, jede Klasse so zu gestalten,***

dass deren Aufgabe eindeutig ist, jede Aufgabe nur durch eine Klasse realisiert wird und die Abhängigkeiten zwischen Klassen möglichst minimal sind.

Vererbung

Eine spezielle Art, neue Klassen basierend auf bestehenden Klassen zu definieren, wird ***Vererbung*** genannt. Diese wird durch das Schlüsselwort `extends` ausgedrückt: Die neu entstehende Klasse erweitert oder übernimmt (erbt) durch diesen Vorgang das Verhalten und die Eigenschaften der bestehenden Klasse und wird ***abgeleitete*** oder ***Sub***klasse genannt. Die wiederverwendete Klasse bezeichnet man als ***Basis***-, ***Ober***- oder ***Super***klasse. Eine Subklasse muss dann in ihrer Implementierung lediglich die Unterschiede zu ihrer Basisklasse beschreiben und nicht komplett neu entwickelt werden. Vererbung ermöglicht also die ***Wiederverwendung*** bereits existierender Funktionalität und erleichtert die Wartung, da bei einer Fehlersuche und -korrektur weniger Sourcecode zu schreiben, zu pflegen und zu analysieren ist. Das ist ein Vorteil gegenüber dem ***Copy-Paste-Ansatz***. Dieser erreicht Wiederverwendung dadurch, dass Teile des Sourcecodes (im Extremfall manchmal sogar ganze Klassen) kopiert und entsprechend je nach Bedarf modifiziert werden. Bereits anhand dieser Beschreibung erahnt man, dass beim Copy-Paste-Ansatz im Rahmen von Erweiterungen oder Korrekturen oftmals Änderungen recht verstreut in vielen Klassen durchzuführen sind. Beim Einsatz von Vererbung und der Extraktion bzw. Definition passender Basis- und Subklassen sind Änderungen lediglich an wenigen Stellen, im Idealfall sogar nur einer Stelle, notwendig.

Durch Vererbung entsteht eine ***Klassenhierarchie***. Wenn man diese gedanklich in Richtung Subklassen durchläuft, spricht man von einer ***Spezialisierung***, und der Weg in Richtung Basisklassen wird ***Generalisierung*** genannt (vgl. Abbildung 3-4).

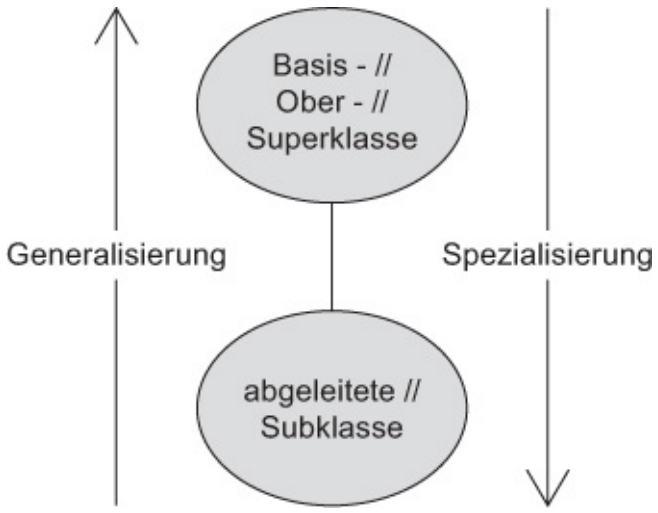


Abbildung 3-4 *Generalisierung und Spezialisierung*

Vererbung sollte nur dann eingesetzt werden, wenn die sogenannte ***»is-a«-Beziehung*** erfüllt ist. Diese besagt, dass ***Subklassen tatsächlich eine semantische Spezialisierung ihrer Basisklasse darstellen*** und dadurch das Verhalten sowie die Eigenschaften der Basisklasse besitzen. Wenn statt einer Basisklasse überall auch eine Spezialisierung davon eingesetzt werden kann, dann spricht man von ***Substituierbarkeit*** oder auch

von dem ***Substitutionsprinzip*** und insbesondere dem LISKOV SUBSTITUTION PRINCIPLE (LSP). Dieses lernen wir in Abschnitt 3.5.3 kennen und erfahren dort, dass die Einhaltung der »is-a«-Beziehung bereits eine wesentliche Grundlage ist.

Wird Vererbung wirklich nur dann eingesetzt, wenn tatsächlich die ***»is-a«-Beziehung*** erfüllt ist, so profitiert man davon, dass beim Aufbau einer ***Klassenhierarchie*** durch Ableitung lediglich die Unterschiede zum vererbten Verhalten und Zustand definiert werden müssen. Erweiterungen werden durch neue Methoden und Attribute realisiert. Für Änderungen am bestehenden Verhalten müssen die abgeleiteten Klassen in der Lage sein, die Methoden der Basisklasse zu verändern. Dies wird durch die Technik ***Overriding*** erreicht. Ganz wichtig ist die Abgrenzung zum sogenannten ***Overloading***. Damit ist gemeint, dass mehrere Methoden mit gleichem Namen, aber unterschiedlicher Signatur innerhalb einer Klasse definiert sind. Da selbst erfahrene Entwickler die Techniken Overriding und Overloading gelegentlich nicht immer richtig einsetzen, möchte ich diese im Anschluss detaillierter beschreiben.

Wird jedoch schon die Forderung nach semantischer Spezialisierung nicht beachtet, sondern Vererbung dazu eingesetzt, um aus der Basisklasse benötigte Funktionalität zu übernehmen, dann handelt es sich um eine sogenannte ***Implementierungsvererbung***. Diese ist zu vermeiden, weil damit zwar technisch, aber nicht semantisch ein Subtyp definiert wird. Als Konsequenz kann man Objekte einer Subklasse dann konzeptuell nicht mehr als Objekte der Basisklasse betrachten. Abbildung 3-5 zeigt ein Positiv- und ein Negativbeispiel.

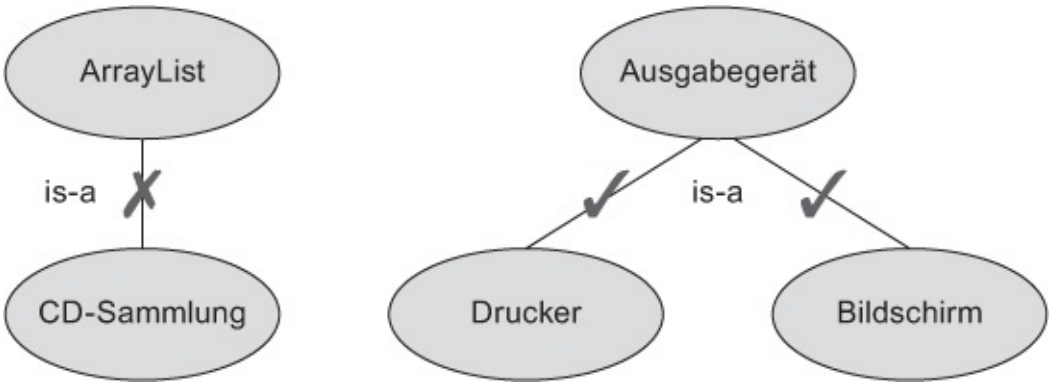


Abbildung 3-5 *Vererbung und »is-a«-Beziehung*

Overriding

Mit Überschreiben bzw. ***Overriding*** ist das ***Redefinieren von geerbten Methoden*** gemeint. Dazu wird der Methodenname übernommen und in der Subklasse eine neue Implementierung der Methode bereitgestellt. Dadurch können Subklassen Methoden modifizieren oder sogar komplett anders definieren, um Änderungen im Verhalten auszudrücken. Dabei darf weder die Signatur der Methode geändert noch die Sichtbarkeit eingeschränkt werden – eine Erweiterung ist dagegen erlaubt.

Overloading

Unter Überladen oder ***Overloading*** versteht man die Definition von Methoden gleichen Namens innerhalb der gleichen Klasse, aber mit unterschiedlicher Parameterliste. Im Gegensatz zum Overriding besteht demnach kein Zusammenhang mit Vererbung, sondern es wird eine ***Vereinfachung der Schreibweise*** adressiert. Durch Overloading kann man den Namen von Methoden ähnlicher Intention vereinheitlichen. Betrachten wir als Beispiel eine Klasse `Rectangle` mit folgenden Methoden:

```
drawByStartAndEndPos(int x1, int y1, int x2, int y2)}
drawByStartPosAndSize(Point pos, Dimension size)}
```

Die Handhabung der Klasse wird vereinfacht, wenn man sich als Anwender nur einen Methodennamen zum Zeichnen merken muss. In diesem Fall könnte man folglich den Namen auf `draw()` verkürzen und zwei Methoden mit unterschiedlichen Parameterlisten anbieten. Die JVM wählt beim Aufruf automatisch aufgrund der übergebenen Parameter bzw. deren Anzahl oder Typen die passende Methode.

So praktisch das Ganze auch manchmal ist, so sind dem Overloading doch Grenzen gesetzt. Damit Overloading möglich ist, müssen Methoden unterschiedliche Parameterlisten besitzen. Für die folgenden beiden Methoden gilt das nicht und somit ist hier kein Overloading möglich:

```
drawByStartAndEndPos(int x1, int y1, int x2, int y2)
drawByStartPosAndSize(int x1, int y1, int width, int height)
```

Sub-Classing und Sub-Typing

Spezialisierung ist sowohl zwischen Klassen als auch zwischen Interfaces möglich. Beides wird durch das Schlüsselwort `extends` ausgedrückt. Zwischen Klassen wird durch Spezialisierung ein Vererben von Verhalten erreicht, d. h., eine Klasse ist eine spezielle Ausprägung einer anderen Klasse, übernimmt deren Verhalten und fügt eigene Merkmale und Verhaltensweisen hinzu. Hier spricht man von ***Sub-Classing***. Eine Spezialisierung eines Interface erweitert die Menge der Methoden eines anderen Interface. In diesem Fall spricht man von ***Sub-Typing***. Für Klassen spricht man häufiger der Einfachheit halber auch von Sub-Typing. Dies erleichtert die Diskussion, denn eine Vererbung zwischen Klassen ist streng genommen sowohl Sub-Classing als auch Sub-Typing. Das Implementieren eines Interface ist jedoch nur Sub-Typing.

Polymorphie

Variablen eines Basistyps können beliebige davon abgeleitete Spezialisierungen referenzieren. Das wird als Vielgestaltigkeit oder ***Polymorphie*** bezeichnet. Polymorphie basiert auf dem Unterschied zwischen dem ***Kompiliertyp*** und dem ***Laufzeittyp***. Der Kompiliertyp ist der zur Kompilierzeit bekannte Basistyp. Der Laufzeittyp entspricht der konkret verwendeten Spezialisierung. Betrachten wir eine Klassenhierarchie der Klassen `Base`, `Sub` und `SubSub` sowie deren Methoden `doIt()` und `doThat()`. Die Klassenhierarchie ist in Abbildung 3-6 dargestellt.

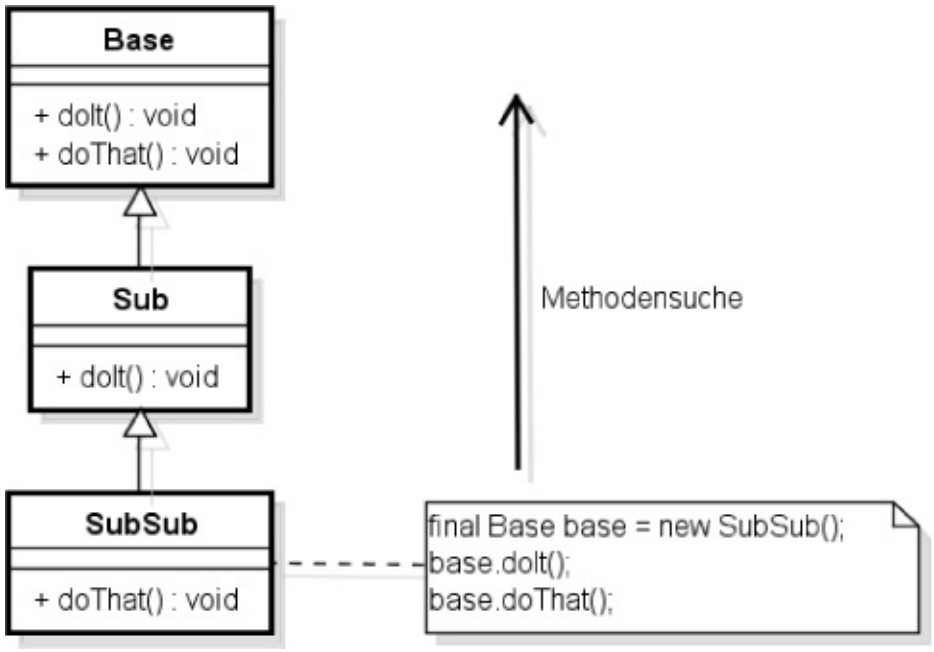


Abbildung 3-6 *Polymorphie und dynamisches Binden*

Ein Beispiel für Polymorphie ist, dass die Variable `base` den Kompiliertyp `Base` besitzt und während der Programmausführung den Laufzeittyp `SubSub`. Damit Polymorphie funktioniert, muss immer die spezialisierteste Methode eines Objekts verwendet werden, d. h. die Implementierung der spezialisiertesten Subklasse, die diese Methode anbietet: Zur Bestimmung der auszuführenden Methode wird dazu startend bei dem Laufzeittyp nach einer passenden Methode gesucht. Für den Aufruf von `doIt()` startet die Suche daher in der Klasse `SubSub`. Dort wird die JVM allerdings nicht fündig, sodass die Suche sukzessive weiter nach oben in der Vererbungshierarchie fortgesetzt wird, bis eine Methodendefinition gefunden wird. In diesem Beispiel ist dies für `doIt()` in der Klasse `Sub` der Fall. Das beschriebene Verfahren zum Auffinden der auszuführenden Methode wird ***dynamisches Binden*** (***Dynamic Binding***) genannt.

Achtung: Polymorphie – Einfluss von Kompilier- und Laufzeittyp

Eine Quelle für Flüchtigkeitsfehler ist die Annahme, dass die polymorphen Eigenschaften von Objektmethoden auch für Attribute und statische Methoden gelten würden. Attribute und statische Methoden sind nicht polymorph: Es wird immer auf die durch den Kompiliertyp sichtbaren Methoden bzw. Attribute zugegriffen. Dieses wird leicht übersehen und kann zu falschem Systemverhalten führen.

Explizite Typumwandlung – Type Cast (Cast)

Unter einer expliziten Typumwandlung, einem sogenannten ***Type Cast*** oder kurz ***Cast***, versteht man eine »Aufforderung« an den Compiler, eine Variable eines Typs in einen anderen angegebenen Typ umzuwandeln. Solange man dabei in der Typhierarchie nach oben geht, ist diese Umwandlung durch die »is-a«-Beziehung

abgesichert. Möchte man jedoch einen Basistyp in einen spezielleren Typ konvertieren, so geht man in der Ableitungshierarchie nach unten. Eine derartige Umwandlung wird als ***Down Cast*** bezeichnet und ist durch die »is-a«-Beziehung nicht abgesichert. Betrachten wir dies an einem Beispiel für eine Methode `doSomethingWithSub(Object)`, die einen Cast vornimmt und mit einer Instanz vom Typ `Sub` bzw. `String` aufgerufen wird:

```
// Dieser Aufruf ist problemlos möglich
doSomethingWithSub(new Sub());

// Dieser Aufruf mit einem String führt zu einem Fehler
doSomethingWithSub("This will fail");

// …

// Diese Methode sollte man mit Referenzen vom Typ Sub aufrufen
void doSomethingWithSub(final Object obj)
{
    final Sub sub = (Sub) obj; // Unsicherer Down Cast: Object -> Base -> Sub
    // …
}
```

Nur für den Fall, dass die übergebene Referenz `obj` ein Objekt vom Typ `Sub` (oder eine Spezialisierung davon) enthält, ist der Cast erfolgreich. Ansonsten kommt es zu einer Inkompatibilität von Typen, wodurch eine `java.lang.ClassCastException` ausgelöst wird. Dies ist für den zweiten Methodenaufruf des Beispiels der Fall.

Solche Down Casts sollte man daher möglichst vermeiden oder zumindest durch eine explizite Typprüfung mit `instanceof` versehen:

```
if (obj instanceof Sub)    // Absicherung des Down Cast: Object -> Base -> Sub
{
    final Sub sub = (Sub) obj;
    // …
}
else
{
    //  Problem: Wie soll man auf diese Situation reagieren?
}
```

Mit `instanceof` kann eine Umwandlung abgesichert werden, es stellt sich dann aber die Frage, wie auf alle nicht erwarteten Typen reagiert werden sollte. In der Regel kann sinnvollerweise nur eine Fehlermeldung ausgegeben werden.

Parameterübergabe per Call-by-Value / Call-by-Reference

Beim Aufruf von Methoden existieren verschiedene Strategien, wie die Werte von Parametern übergeben werden. Man unterscheidet zwischen der Übergabe eines Werts (***Call-by-Value***) und der Übergabe per Referenz (***Call-by-Reference***). Entgegen der weitverbreiteten Meinung, in Java würden beide Arten der Parameterübergabe unterstützt, und zwar Call-by-Value für Parameter primitiver Typen und Call-by-Reference für Referenzparameter, wird tatsächlich in beiden

Fällen Call-by-Value genutzt.

Betrachten wir diese in Java umgesetzte Form und ihre Auswirkungen genauer: Alle primitiven Typen werden in Java als Wert übergeben, genauer als bitweise Kopie des Werts. Dadurch existiert keine Verbindung zwischen dem Wert außerhalb der Methode und dem Wert des Parameters. Deshalb sind Änderungen an den Parameterwerten nur lokal innerhalb der Methode sichtbar, nicht jedoch für aufrufende Methoden.

Bei der Übergabe von Referenzvariablen als Parameter könnte man intuitiv auf die Idee kommen, die Übergabe würde per Referenz erfolgen, wie dies etwa bei C++ möglich ist. Bei dieser Form der Übergabe sind alle Änderungen an einem derartig übergebenen Parameter, die innerhalb der Methode stattfinden, auch außerhalb sichtbar. In Java gilt dies allerdings nicht! Auch die Übergabe von Referenzvariablen als Parameter erfolgt als Kopie – allerdings wird hierbei eine bitweise Kopie des Werts der Referenzvariablen erstellt und nicht eine Kopie des referenzierten Objekts. Das hat folgende Konsequenzen: Innerhalb der aufgerufenen Methode kann man zwar das durch die Referenzvariable referenzierte Objekt durch ein anderes ersetzen, diese Änderung ist aber für die aufrufende Methode nicht sichtbar: Die Referenzvariable verweist nach Aufruf der Methoden auf dasselbe Objekt wie zuvor. ***Allerdings ist es möglich, in der aufgerufenen Methode den Zustand des referenzierten Objekts zu verändern. Dadurch können aber Änderungen am Zustand des referenzierten Objekts auch außerhalb der Methode sichtbar sein*** – ähnlich wie man dies für Call-by-Reference kennt. Man spricht hier von der ***Referenzsemantik*** von Java. Schauen wir zur Klärung auf die folgende Methode `changeSomething(String[])`. Diese erhält ein Array von Strings. Zwar kann die Referenz auf das Array in der Methode nicht verändert werden, sehr wohl aber dessen Inhalt:

```
public class ReferenceSemanticsExample
{
    private static void changeSomething(String[] names)
    {
        // Änderungen auch im Original-Array
        names[0] = "Michael";
        names[1] = "changed this entry";

        // Keine Auswirkung auf das Original-Array
        names = new String[] { "Nearly Empty" };
    }

    public static void main(final String[] args)
    {
        final String[] names = { "Test1", "Test2", "!!!" };

        changeSomething(names);

        System.out.println(Arrays.toString(names));
    }
```

Listing 3.1 *Ausführbar als* **'REFERENCESEMANTICSEXAMPLE'**

Führen wir das Programm ReferenceSemanticsExample aus, so werden die ersten

beiden Einträge des Arrays geändert. Damit ergibt sich folgende Ausgabe:

```
[Michael, changed this entry, !!!]
```

Außerdem wird durch diese Ausgabe offensichtlich, dass die Zuweisung der Referenz innerhalb der Methode sich nicht in der `main()`-Methode auswirkt, Wertänderungen einzelner Elemente dagegen schon. Das gilt gleichfalls für Datenstrukturen aus dem Collections-Framework, etwa für Listen und Mengen. Dieser Sachverhalt sollte einem immer bewusst sein, da sich ansonsten leicht Fehler einschleichen, die schwierig zu finden sind. Call-by-Value bezieht sich in Java wirklich ausschließlich auf den Wert der Referenz bzw. des primitiven Typs. Auf weitere Auswirkungen der Referenzsemantik werde ich insbesondere in Abschnitt 3.4.1 detaillierter eingehen.

3.1.2 Beispielentwurf: Ein Zähler

Nachdem nun die Grundbegriffe beim objektorientierten Entwurf bekannt sind, werden diese im Folgenden anhand eines Beispiels vertieft. Eine grafische Anwendung soll um eine Auswertung der Anzahl gezeichneter Linien und Rechtecke erweitert werden. Dazu ist ein Zähler als Klasse zu entwerfen, der folgende Anforderungen erfüllt:

1. Er lässt sich auf den Wert 0 zurücksetzen.
2. Er lässt sich um eins erhöhen.
3. Der aktuelle Wert lässt sich abfragen.

Mit diesen scheinbar einfach umzusetzenden Anforderungen eines Kollegen »User« lassen wir die zwei exemplarischen Entwickler »Schnell-Finger« und »Überleg-Erst-Einmal« diese Aufgabe realisieren. Schauen wir uns an, wie die beiden arbeiten.

Bevor wir beiden Entwicklern bei der Arbeit zusehen, betrachten wir den Nutzungskontext. Der Anwendungscode ist bereits folgendermaßen rudimentär und nicht besonders elegant mit einem `instanceof`-Vergleich implementiert:

```
// TODO: 2 Zähler initialisieren

for (final GraphicObject graphicObject : graphicObjects)
{
   graphicObject.draw();

   if (graphicObject instanceof Line)
   {
        // TODO: lineCounter erhöhen
   }
   if (graphicObject instanceof Rect)
   {
        // TODO: rectCounter erhöhen
   }
}

// TODO: Zähler auslesen und ausgeben
```

Beim Zeichnen soll abhängig vom Typ der grafischen Objekte ein korrespondierender Zähler inkrementiert werden. Die eigentliche Funktionalität bleibt zunächst unimplementiert, da noch auf die konkrete Realisierung der Klasse `Counter` gewartet wird. Deren Einsatzstellen sind mit `TODO`-Kommentaren markiert.

Hinweis: Anmerkungen zu `instanceof`-Vergleichen

`instanceof`-Vergleiche sind vielfach ein Anzeichen schlechten Programmierstils. Meistens liegt ein Verstoß gegen das sogenannte OPEN CLOSED PRINCIPLE (OCP) vor – einem der SOLID-Prinzipien (siehe Abschnitt 3.5.3). Diesen Nachteil wollen wir für dieses Beispiel akzeptieren, da hier der Fokus auf dem objektorientierten Entwurf eines Zählers liegt.

Entwurf à la »Schnell-Finger«

Herr »Schnell-Finger« überlegt nicht lange und startet sofort seinen Texteditor. Dabei hat er bereits ein paar Ideen. Kaum ist das Editor-Fenster geöffnet, legt er los. Klingt alles recht einfach, also wird kurzerhand die folgende Klasse erstellt:

```
public class Counter
{
    public int count = 0;

    public Counter()
    {
    }

    public void setCounter(int count)
    {
        count = count;
    }
}
```

So, damit ist er schon fertig. Die Variable `count` ist öffentlich, kann also von überall abgefragt werden. Zum Verändern dient außerdem die Methode `setCounter(int)`. Rücksetzen erfolgt durch Übergabe von 0. Zufrieden speichert er die Klasse. Am nächsten Tag fragt der Kollege »User«, wie weit die Klasse `Counter` wäre, und bekommt die Antwort: »Längst fertig!« Der Kollege »User« schaut sich die Klasse an und sagt: »Naja, richtig objektorientiert ist das nicht. Es fehlt an Kapselung: Du veröffentlichst dein Attribut und bietest Zugriffsmethoden nur unvollständig an und der Defaultkonstruktor ist auch überflüssig.« Etwas verärgert über den pingeligen Kollegen macht sich Herr »Schnell-Finger« nochmal an die Arbeit und kommt nach kurzer Zeit zu folgender Umsetzung:

```
public class Counter
{
    private int count =  0;

    public int getCounter()
    {
        return count;
```

```
    }

    public void setCounter(int count)
    {
        count = count;
    }
}
```

Die Klasse `Counter` ist laut Herrn »Schnell-Finger« nun einsatzbereit. Daher wollen wir diese zum Zählen von grafischen Figuren nutzen und fügen sie in das zuvor vorgestellte Anwendungsgerüst an den durch `TODO`-Kommentare markierten Stellen ein. Dadurch entsteht folgender Sourcecode:

```
// 2 Zähler initialisieren
final Counter lineCounter = new Counter();
final Counter rectCounter = new Counter();

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        // lineCounter erhöhen
        lineCounter.setCounter(lineCounter.getCounter() + 1);
    }

    if (graphicObject instanceof Rect)
    {
        // rectCounter erhöhen
        rectCounter.setCounter(rectCounter.getCounter() + 1);
    }
}

// Zähler auslesen und ausgeben
System.out.println("Number of Lines: " + lineCounter.getCounter());
System.out.println("Number of Rects: " + rectCounter.getCounter());
```

Wirklich zufrieden ist der Kollege »User« mit der Klasse `Counter` nicht. Der Sourcecode sieht irgendwie nicht rund aus. Und tatsächlich kommt es auch bei einem Test trotz vieler gemalter Linien und Rechtecke zu folgenden, unerwarteten Ausgaben:

```
Number of Lines: 0
Number of Rects: 0
```

Der Anwendungscode scheint fehlerfrei. In der Implementierung der Methode `setCounter(int)` steckt jedoch ein Flüchtigkeitsfehler: Der Übergabeparameter heißt genauso wie das Attribut und ist nicht `final`. Dadurch erfolgt unbemerkt eine Zuweisung an sich selbst. Das lässt sich leicht wie folgt korrigieren:

```
public void setCounter(final int newCount)
{
    this.count = newCount;
}
```

Entwurf à la »Überleg-Erst-Einmal«

Kollege »Überleg-Erst-Einmal« liest die Anforderungen und skizziert ein kleines UML-Klassendiagramm von Hand auf einem Blatt Papier, wie in Abbildung 3-7 gezeigt.

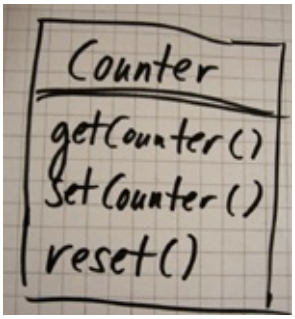


Abbildung 3-7 *Die Klasse* `Counter`

Sein Entwurf sieht zum Verarbeiten des Zählers die Methoden `getCounter()` und `setCounter()` vor. Zum Rücksetzen des Wertes dient die Methode `reset()`. Da er bereits einige Erfahrung im Softwareentwurf hat, weiß er, dass die erste Lösung meistens nicht die beste ist. Um seinen Entwurf zu prüfen, überlegt er sich ein paar Anwendungsfälle und spielt diese im Kopf durch. Später kann er daraus Testfälle definieren. Zudem schaut er nochmals auf die gewünschten Anforderungen. Dort steht: Der Zähler soll um eins erhöht werden. Folglich entfernt er die `setCounter()`-Methode und führt eine `increment()`-Methode ein. Da er bereits jetzt an Dokumentation denkt, nutzt er ein UML-Tool, um die Klasse dort zu konstruieren. Abbildung 3-8 zeigt das Ergebnis.

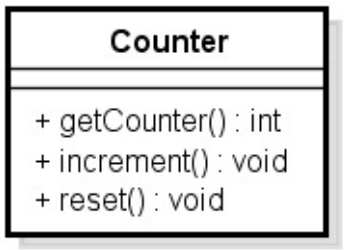


Abbildung 3-8 *Die Klasse* `Counter`*, 2. Version*

Als unbefriedigend empfindet er aber noch eine Inkonsistenz in der Namensgebung bezüglich des Methodennamens `getCounter()`. Der Name ist missverständlich: Hier wird kein `Counter`-Objekt zurückgeliefert, sondern dessen Wert. Namen wie z. B. `getValue()` bzw. `getCurrentValue()` wären damit aussagekräftiger und verständlicher. Eine weitere Alternative ist, auf das Präfix `get` im Namen zu verzichten. Damit ergeben sich `value()` und `currentValue()` als mögliche Methodennamen. In diesem Fall entscheidet er sich für letztere Variante, da sich die Namen dann besser lesen lassen (vgl. Abbildung 3-9). Der Einsatz des Präfixes `get` würde allerdings den lesenden Charakter der Zugriffsmethode auf einen Blick sichtbar machen.[6]

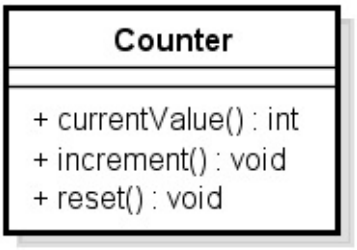


Abbildung 3-9 *Die Klasse* `Counter`*, 3. Version*

Während Herr »Überleg-Erst-Einmal« noch an der Implementierung arbeitet, kommt der Kollege »User« vorbei und berichtet, dass er die Lösung von Herrn »Schnell-Finger« als zu sperrig empfindet. Daher würde er gern eine weitere Lösung begutachten und in seine Anwendung einbauen. Die Implementierung der Klasse `Counter` ist zwar noch nicht vollständig abgeschlossen. Deren Schnittstelle, das sogenannte ***API*** (***Application Programming Interface***), d. h. die angebotenen Methoden, sind durch das UML-Diagramm aber bereits vollständig definiert. Daher kann der Applikationscode wie folgt auf das API der Klasse `Counter` von Herrn »Überleg-Erst-Einmal« angepasst werden:

```
final Counter lineCounter = new Counter();
final Counter rectCounter = new Counter();

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        lineCounter.increment();
    }

    if (graphicObject instanceof Rect)
    {
        rectCounter.increment();
    }
}

System.out.println("Number of Lines: " + lineCounter.currentValue());
System.out.println("Number of Rects: " + rectCounter.currentValue());
```

Der resultierende Sourcecode ist deutlich besser lesbar. Die Zeilen mit dem Aufruf der Methode `increment()` sind nun selbsterklärend, und die Kommentare konnten daher ersatzlos entfallen. Schauen wir abschließend auf den von Kollege »Überleg-Erst-Einmal« erstellten Sourcecode seiner Variante der Klasse `Counter`:

```
public class Counter
{
    private int value = 0;

    public int currentValue()
    {
        return value;
    }

    public void increment()
    {
```

```
            value++;
        }

        public void reset()
        {
            value = 0;
        }
}
```

Vergleich der beiden Lösungen

Nachdem wir die zwei verschiedenen Arten der Entwicklung exemplarisch kennengelernt haben, wollen wir die erreichten Lösungen abschließend miteinander vergleichen und so weitere Erkenntnisse zum objektorientierten Programmieren gewinnen bzw. vertiefen. Wir betrachten dazu die beiden Implementierungen nun bezüglich ihres Grads an Objektorientierung sowie Erweiter- und Wiederverwendbarkeit.

Grad der Objektorientierung

Die Lösung von Herrn »Schnell-Finger« hat durch einige Iterationen und vor allem Review-Kommentare von Kollegen einen brauchbaren Zustand erreicht. Allerdings macht diese Lösung ausschließlich von `get()`-/`set()`-Methoden Gebrauch, ohne aber Verhalten zu definieren. Bei ersten OO-Gehversuchen besteht die Gefahr, dass nahezu alle Methoden zu derartigen reinen Zugriffsmethoden ohne Objektverhalten verkümmern. Hier herrscht meiner Ansicht nach das größte Missverständnis bezüglich der Objektorientierung vor: ***Ein intensiver Gebrauch von `get()`- und `set()`-Methoden ist kein Zeichen von guter Objektorientierung, sondern deutet häufig vielmehr auf einen fragwürdigen OO-Entwurf hin***. Solche Klassen tendieren dann dazu, kein Verhalten mehr zu kapseln – stattdessen wird dieses stark von nutzenden Klassen oder Methoden gesteuert bzw. in diesen realisiert. Dadurch besteht die Gefahr, dass Attribute derart von außen manipuliert werden, dass sich der Objektzustand in einer unerwarteten (möglicherweise ungewünschten) Art und Weise verändert. Dies können wir im Zählerbeispiel von Herrn »Schnell-Finger« sehr schön daran sehen, wie die Inkrementierung des Zählers realisiert ist. Tatsächlich könnte der Zähler durch die angebotene `setCounter(int)`-Methode auf beliebige Werte gesetzt werden, also erhöht oder erniedrigt werden. Das Inkrement wird also im Applikationscode programmiert. Jede weitere Applikation müsste wiederum diese Funktionalität des Zählers selbst realisieren. Diese Implementierung ist demnach vollständig an der Anforderung vorbei entwickelt, da lediglich ein besserer Datencontainer realisiert wird. Kurz gesagt: Die Klasse hat keine Business-Methoden. Damit verletzt diese Art des Entwurfs zum einen den Gedanken der Definition von Verhalten durch Objekte und zum anderen das Ziel der Wiederverwendbarkeit. Beim `get()`-/`set()`-Ansatz kann somit oftmals keine (oder zumindest kaum) Funktionalität wiederverwendet werden.

Die Klasse `Counter` von Herrn »Überleg-Erst-Einmal« realisiert dagegen ein

logisches Modell. Hier stehen die technischen Details nicht im Vordergrund, sondern man konzentriert sich auf das Erfüllen einer Aufgabe. Design und Implementierung stellen die Funktionalität eines Zählens bereit und verhindern feingranulare Zugriffe auf interne Variablen. Dies verstärkt die Kapselung, da die Methoden `increment()` und `currentValue()` Implementierungsdetails gut verbergen. Es ist dadurch von außen unmöglich, den Zähler um beliebige Werte zu erhöhen oder zu erniedrigen. Diese Umsetzung besitzt zudem eine höhere Kohäsion und damit eine bessere Abschirmung bei anstehenden Änderungen als der `get()`-/`set()`-Ansatz.

Tipp: Logisches Modell und Business-Methoden

Durch das Realisieren eines logischen Modells wird der entstehende Sourcecode in der Regel deutlich besser verständlich und menschenlesbar, da man Konzepten und nicht Programmanweisungen folgt. Veränderungen am Objektzustand werden dabei durch eine Reihe von verhaltensdefinierenden ***Business-Methoden*** implementiert.

Grad der Wiederverwendbarkeit und Erweiterbarkeit

Um die Auswirkungen der beiden Entwürfe bezüglich Wiederverwendbarkeit und Erweiterbarkeit zu untersuchen, nehmen wir an, dass wir den Zähler um einen Überlauf bei einer bestimmten Schwelle erweitern und die Anzahl der Überläufe protokollieren wollten. Dies könnte man etwa für eine Spieleapplikation dazu nutzen, um nach 100 aufgesammelten Bonuselementen ein weiteres Leben zu erhalten.

Die Erweiterung müsste bei Einsatz der Realisierung von Herrn »Schnell-Finger« von jeder Applikation selbst implementiert werden. Das kann schnell sehr aufwendig werden, zu dupliziertem oder sehr ähnlichem Sourcecode führen und später in einem Wartungsalbtraum enden. Diesen Weg wollen wir nicht weiter betrachten, da wir (glücklicherweise) eine Alternative haben.

Überlegen wir also, wie wir den Zähler von Herrn »Überleg-Erst-Einmal« zur Realisierung der gewünschten Funktionalität verwenden können. Wir erinnern uns an die Möglichkeit der Vererbung und realisieren eine Klasse basierend auf der Klasse `Counter`. Soll eine Applikation nachträglich diese erweiterte Variante des Zählers nutzen, so ist nur genau eine Stelle zu ändern, nämlich die Konstruktion des Zählers. Das besitzt zudem den Vorteil, dass keine Fortpflanzungen der Änderungen in die nutzenden Applikationen stattfinden – natürlich abgesehen von denjenigen, die die zusätzliche Funktionalität, also den Zählerstand des Überlaufzählers, nutzen wollen.

Realisierung eines Zählers mit Überlauf

Wir implementieren nun eine Klasse `CounterWithOverflow` basierend auf der Klasse `Counter`. Dazu erweitern wir die Basisklasse um eine Konstante `COUNTER_MAX`, ein zusätzliches Attribut `overflowCounter` und eine Zugriffsmethode auf dessen Wert. Das Zählen, Rücksetzen und einiges anderes delegieren wir an die Basisklasse durch Aufruf der entsprechenden Methoden mithilfe

des Schlüsselworts `super`. Lediglich die `increment()`-Methode muss selbst realisiert werden. Dort prüfen wir auf einen Überlauf und führen gegebenenfalls ein Rücksetzen des Zählers und eine Erhöhung des Überlaufzählers durch. Das Ganze kann man wie folgt realisieren:

```
public class CounterWithOverflow extends Counter
{
    private static final int COUNTER_MAX = 100;

    private final Counter overflowCounter = new Counter();

    public int overflowCount()
    {
        return overflowCounter.currentValue();
    }

    public void reset()
    {
        super.reset();
        overflowCounter.reset();
    }

    public void increment()
    {
        if (currentValue() == COUNTER_MAX-1)
        {
            super.reset();
            overflowCounter.increment();
        }
        else
        {
            super.increment();
        }
    }
}
```

Tatsächlich hatte ich diese Klasse zunächst mit einem Überlaufzähler als `int`-Wert realisiert. Viel natürlicher und objektorientierter ist aber die gezeigte Variante, die Überläufe wiederum mithilfe der Klasse `Counter` zählt.

Fazit

Anhand dieses Beispiels lässt sich sehr schön erkennen, dass neben dem puren Einsatz von Klassen und Objekten vor allem auch die Definition von Verhalten und Zuständigkeiten die objektorientierte Programmierung ausmachen. Dabei ist es hilfreich, zusammengehörende Funktionalität innerhalb einer Klasse zu bündeln (***Kohäsion***) sowie gleichzeitig Implementierungsdetails zu verbergen (***Kapselung***). In komplexeren Entwürfen ergeben sich weitere Vorteile durch den sinnvollen Einsatz von Vererbung und die Konstruktion passender Klassenhierarchien.

Klares Ziel beim objektorientierten Entwurf sollte es sein, Klassen mit Verhalten zu definieren. Der massive Einsatz von `get()`-/`set()`-Methoden widerspricht dem und ist somit vielfach wenig objektorientiert. Dies eignet sich vor allem für Datenbehälter, also Klassen, die eigentlich kein eigenes Verhalten definieren und

deren Instanzen damit eher als Hilfsobjekte betrachtet werden sollten.

3.1.3 Vom imperativen zum objektorientierten Entwurf

Nachdem wir einige OO-Grundlagen beim Entwurf einer Zähler-Klasse und dabei auch mögliche Tücken kennengelernt haben, wollen wir nun den Einfluss verschiedener Vorgehensweisen und Entwurfsstile auf die Entwicklung objektorientierter Programme betrachten.

Bevor der Einsatz der objektorientierten Programmierung modern wurde, hat man vielfach den algorithmischen oder imperativen Ansatz beim Softwareentwurf eingesetzt: Hierbei werden Befehle in Prozeduren und Funktionen zusammengefasst, die wiederum das Programm ergeben. Der Informatikmethode »***teile und herrsche***« (***divide and conquer***) folgend, werden komplexe Algorithmen in kleinere Teile zerlegt. Auch wenn das Vorgehen für den imperativen Ansatz angemessen ist, entstehen merkwürdige Entwürfe, wenn man versucht, diese Vorgehensweise auf die Objektorientierung zu übertragen.

Entwurf mit verschiedenen Entwurfsstilen

Zum Vergleich verschiedener Entwurfsstile soll eine Methode `drawFigure()` zum Zeichnen von grafischen Objekten auf einer Zeichenfläche implementiert werden. Es sollen verschiedene Typen grafischer Elemente (Punkt, Linie und Rechteck) dargestellt werden können. Im Folgenden betrachten wir drei verschiedene Arten der Realisierung in Pseudocode. Das erste Beispiel implementiert die Lösung imperativ, das zweite arbeitet mit Objekten und das dritte ist schließlich objektorientiert und verdeutlicht die Vorteile der Techniken Vererbung und Polymorphie.

Imperativer Entwurf

Das folgende Listing zeigt einen Entwurf, wie er von einem Kollegen kommen könnte, der in der imperativen Programmierwelt zu Hause ist. In der Klasse `PaintingArea` realisiert er eine Methode `drawFigure()`, die alle gewünschten Typen grafischer Elemente zeichnen kann:

```
public class PaintingArea extends JComponent
{
    public static final int POINT_TYPE = 0;
    public static final int LINE_TYPE = 1;
    public static final int RECT_TYPE = 2;

    void drawFigure(int graphicsObjectType, int x1, int y1, int x2, int y2)
    {
        if (graphicsObjectType == POINT_TYPE)
        {
            drawPoint(x1, y1);
```

```
        }
        if (graphicsObjectType == LINE_TYPE)
        {
            drawLine(x1, y1, x2, y2);
        }
        if (graphicsObjectType == RECT_TYPE)
        {
            drawRect(x1, y1, x2, y2);
        }
    }
    // …
```

Diese Methode bekommt sämtliche zum Zeichnern der Figuren benötigten Parameter übergeben: Anhand einer `int`-Variablen `graphicsObjectType` wird geprüft, welcher Typ einer grafischen Figur gezeichnet werden soll. Je nach identifiziertem Typ wird eine spezielle Methode mit einer Auswahl der zum Zeichnen benötigten Parameter aufgerufen. Zum Zeichnen eines Punkts sind zwar die Werte `x2` und `y2` uninteressant, müssen aber trotzdem an die `drawFigure()`-Methode übergeben werden.

Entwurf mit Objekten, aber nicht objektorientiert

Ein Entwurf, wie er von einem bezüglich Objektorientierung unerfahrenen Entwickler kommen könnte, arbeitet zwar mit Objekten, nutzt jedoch nicht die Vorteile gemeinsamer Basisklassen und der Polymorphie. Positiv ist allerdings, dass die Objekte das Zeichnen mit Methoden selbst erledigen und dass auch die dazu benötigten Koordinaten als Attribute gespeichert werden. Es findet demnach eine Definition von Verhalten sowie eine Datenkapselung statt. Diese Art des Entwurfs befreit die Klasse `PaintingArea` und die `drawFigure()`-Methode von einigen Implementierungsdetails. Besonders auffällig ist dies anhand der benötigten Übergabeparameter. Es wird lediglich noch das zu zeichnende Objekt benötigt. Um die korrekte Methode zum Zeichnen aufzurufen, muss hier wieder zunächst der Typ des grafischen Objekts festgestellt werden. Dies geschieht hier nicht anhand eines `int`-Werts, sondern basierend auf dem Laufzeittyp des jeweiligen Objekts. Dieser wird über `instanceof` geprüft und auf die entsprechenden Klassen gecastet. Dann kann die spezifische Methode zum Zeichnen aufgerufen werden:

```
void drawFigure(final Object graphicsObject)
{
    if (graphicsObject instanceof Point)
    {
        ((Point) graphicsObject).drawPoint();
    }
    if (graphicsObject instanceof Line)
    {
        ((Line) graphicsObject).drawLine();
    }
    if (graphicsObject instanceof Rect)
    {
        ((Rect) graphicsObject).drawRect();
    }
}
```

Hier erkennen wir einen »händischen Nachbau« von Polymorphie. Ein Variante davon ist, überladene Methoden zum Zeichnen anzubieten, wie dies im folgenden Praxistipp beschrieben wird.

Achtung: Künstliches Overriding und Overloading

In der Praxis kann es schnell zu Missverständnissen bezüglich der Wirkungsweise von Overriding und Overloading kommen. Zum Teil sieht man statt Overriding den Einsatz von Overloading. Es wird dann versucht, ein Verhalten wie beim Overriding nachzubilden. Dies kann schnell unverständlich und unübersichtlich werden.

Betrachten wir das konkret am Beispiel der Klasse `PaintingArea`. Nehmen wir an, diese würde einige überladene `draw()`-Methoden zum Zeichnen von Punkten, Linien und Rechtecken anbieten. Zur Vereinfachung für Nutzer wird eine Methode `drawFigure(Object)` eingeführt. Ziel dabei ist es, ein beliebiges `Figure`-Objekt übergeben zu können und die Wahl der passenden, spezialisierten `draw()`-Methode automatisch erledigen zu lassen:

```
public void drawFigure(final Object obj)
{
    draw(obj); // Hoffnung, eine der überladenen Methoden aufzurufen
}

public void draw(final Point point)
{
    point.drawPoint();
}

public void draw(final Line line)
{
    line.drawLine();
}

// …
```

Obwohl man intuitiv denken könnte, dass der Aufruf von `drawFigure(Object)` bei der Übergabe eines Parameters vom Typ `Rect` die Methode `draw(Rect)` aufruft, führt diese Zeile vielmehr zu einem Kompilierfehler: Der Compiler bemängelt das Fehlen der Methode `draw(Object)`.

Selbst wenn wir als Abhilfe eine solche Methode einführen, sind trotzdem sämtliche `draw()`-Methoden lediglich überladen und damit Polymorphie hier nicht wirksam. Um dennoch die gewünschte typabhängige Auswahl der passenden Methode zu erreichen, sieht man dann teilweise die folgende Abhilfe:

```
public void draw(final Object obj)
{
    if (obj instanceof Point)
        draw((Point) obj);

    if (obj instanceof Line)
        draw((Line) obj);

    // …
}
```

Zunächst wird eine Typprüfung vorgenommen und davon abhängig ein Cast ausgeführt. Anschließend erfolgt ein Aufruf der für diesen Typ überladenen `draw()`-Methode. Dies wirkt bereits optisch merkwürdig. Zudem ist auch das Design weder elegant noch flexibel erweiterbar. Einerseits blähen die `instanceof`-Prüfungen und die nachfolgenden Casts den Sourcecode auf und erschweren so die Lesbarkeit. Andererseits leidet die Erweiterbarkeit dadurch, dass während der Implementierungsphase bereits alle zu behandelnden Klassen bekannt sein müssen. Das Problem lässt sich einfach und elegant durch Einführen einer gemeinsamen Basisklasse lösen.

Objektorientierter Entwurf

Beim OO-Entwurf führen wir eine abstrakte Basisklasse `BaseFigure` ein und leiten alle grafischen Objekte von dieser ab. Die Basisklasse definiert eine abstrakte `draw()`-Methode. Alle konkreten Subklassen müssen diese implementieren. Dort wird der Algorithmus zum Zeichnen entsprechend ihrem Typ realisiert. Über dynamisches Binden wird von der JVM automatisch die passende Methode gewählt. Es ergibt sich folgende Implementierung:

```
public void drawFigure(final BaseFigure baseFigure)
{
    baseFigure.draw();
}
```

Vergleich der Realisierungen

Beim Betrachten der drei zuvor beschriebenen Umsetzungen erkennt man, dass die objektorientierte Lösung die kürzeste und eleganteste ist, da hier aufgrund der Polymorphie die expliziten Typprüfungen entfallen können. Sowohl der imperative als auch der mit Objekten arbeitende Ansatz können darauf nicht verzichten, denn nur so kann dort die Unterscheidung getroffen werden, welches grafische Element gezeichnet werden soll. Die beiden Realisierungen ähneln einander auf den ersten Blick. Allerdings bietet der Entwurf mit Objekten einen entscheidenden Vorteil gegenüber dem imperativen Entwurf — er delegiert das Zeichnen an die einzelnen Objekte, statt dies in der Zeichenfläche zu realisieren.

Leider sieht man Sourcecode mit expliziten Typprüfungen viel häufiger, als man denken sollte. Das ist problematisch, weil man sich dadurch auf alle zu verarbeitenden Klassen festlegt. Spätere Erweiterungen erfordern Änderungen in der Typprüfung und in der Realisierung. Beide Lösungen verstoßen demzufolge gegen das OPEN CLOSED PRINCIPLE (OCP), eines der fünf SOLID genannten Prinzipien für guten OO-Entwurf, die wir später in Abschnitt 3.5.3 kennenlernen werden.

Die objektorientierte Lösung nutzt die Vorteile der Polymorphie. Dadurch können sogar Figuren gezeichnet werden, deren Klassen zum Zeitpunkt der Implementierung der Klasse `PaintingArea` noch unbekannt sind, solange diese Figurenklassen von der gemeinsamen Basisklasse `BaseFigure` abgeleitet sind.

Mögliche Probleme bei Erweiterungen

Stellen wir uns vor, wir wollten nun auch Polygone oder Kreise zeichnen. Bei der imperativen und der mit Objekten arbeitenden Realisierung müssten wir für jede neue grafische Figur eine weitere Fallunterscheidung in Form einer `if`-Anweisung einfügen, wie dies der folgende Sourcecode exemplarisch für Polygone zeigt:

```
void drawFigure(int graphicsObjectType, int x1, int y1, int x2, int y2, int
x3,
      int y3, /*…*/ int xn_1, int yn_1, int xn, int yn)
{
    if (graphicsObjectType == POINT_TYPE)
    {
        drawPoint(x1, y1);
    }

    // usw.

    if (graphicsObjectType == POLYGON_TYPE)
    {
        // Zeichne Polygon durch mehrere Linien
        drawLine(x1, y1, x2, y2);
        drawLine(x2, y2, x3, y3);
        // …
        drawLine(xn_1, yn_1, xn, yn);
    }
}
```

Im imperativen Fall müssten wir zudem die Parameterliste im Extremfall stark erweitern, um alle benötigten Informationen für alle Arten von Figuren übergeben zu können. Dabei können leicht Situationen auftreten, in denen Parameter für andere Figurtypen schlicht unbenutzt und damit überflüssig sind oder sich sogar widersprechen. Ansatzweise haben wir dies bereits für das Parameterpaar `x2`, `y2` und das Zeichnen von Punkten kennengelernt. Noch schlimmer ist, dass wir Erweiterungen in der Klasse `PaintingArea` vornehmen müssten, die das Zeichnen der neu eingeführten grafischen Objekte realisieren. Unter Umständen haben wir keinen Zugriff auf den Sourcecode dieser Klasse oder wir dürfen das API nicht mehr ändern, insbesondere wenn schon eine breite Nutzerbasis existiert.[7] In beiden Situationen sind Erweiterungen nur noch extrem schwer möglich, etwa durch so hässliche Tricks, wie neue Figuren durch bestehende Operationen zeichnen zu lassen. Wir wollen uns gar nicht vorstellen, was passieren würde, wenn wir auf diese Art Kreise oder gefüllte Figuren zeichnen müssten.

Der objektbasierte Entwurf hat zumindest keine Probleme mit den Unzulänglichkeiten bei der Parameterübergabe. Alle Definitions- und Implementierungsdetails, z. B. Ecken, Stützpunkte oder Abmessungen und Zeichenroutinen, werden verborgen. Demnach erspart uns die Kapselung schon viele Probleme der imperativen Lösung.

Wie schon erwähnt, verstoßen beide Lösungen gegen das OPEN CLOSED PRINCIPLE (OCP) (vgl. Abschnitt 3.5.3) und erfordern einiges an Aufwand bei Erweiterungen. Die objektorientierte Lösung folgt dagegen dem OCP und ist absolut

einfach zu erweitern: Die Klasse `PaintingArea` bleibt einfach so, wie sie ist. Jedes neue grafische Objekt von Typ `BaseFigure` erfordert dann lediglich noch eine Realisierung.

3.1.4 Diskussion der OO-Grundgedanken

Nachdem wir mittlerweile einige Erkenntnisse bezüglich OO gewonnen haben, möchte ich die zu Beginn dieses Kapitels kurz vorgestellten Grundgedanken des OO-Entwurfs aufgreifen und diese mit dem neu gewonnenen Verständnis betrachten.

Datenkapselung und Trennung von Zuständigkeiten

Die Datenkapselung, also das Verbergen von Informationen, stellt ein Kernkonzept der objektorientierten Programmierung dar und ermöglicht, die Daten eines Objekts vor Änderungen durch andere Objekte mithilfe von Zugriffsmethoden zu schützen. Zugriffsmethoden erleichtern auch eine Konsistenzprüfung eingehender Werte und eine zuverlässige Synchronisierung für Multithreading. Des Weiteren ermöglicht die Datenkapselung, die Speicherung der Daten eines Objekts unabhängig von dem nach außen bereitgestellten Interface zu ändern.

Aufgrund der obigen Argumentation ist in vielen Fällen der Einsatz von Zugriffsmethoden dem direkten Zugriff auf Attribute vorzuziehen – aber es gibt auch Ausnahmen! Die Forderung nach Zugriffsmethoden sollte man nicht dogmatisch umsetzen: Für Package-interne Klassen oder lokal innerhalb einer anderen Klasse definierte Hilfsklassen ist es häufig akzeptabel, auf Zugriffsmethoden zu verzichten und direkt auf Attribute zuzugreifen. Dies gilt allerdings nur dann, wenn absehbar ist, dass

1. diese Klasse nicht außerhalb des eigenen Packages verwendet wird,
2. keine größeren strukturellen Änderungen (viele neue Attribute, geänderte Assoziationen usw.) zu erwarten sind und
3. keine anderen Rahmenbedingungen, wie etwa Thread-Sicherheit, gegen eine direkte Nutzung und die dadurch reduzierte Kapselung sprechen.

Wir wollen uns die Vorteile der Datenkapselung anhand eines Beispiels ansehen: Es ist eine Klasse zur Modellierung von Personen und zugehörigen Adressdaten zu erstellen. Im Folgenden betrachten wir drei verschiedene Umsetzungen, die als UML-Klassendiagramm in Abbildung 3-10 dargestellt sind.

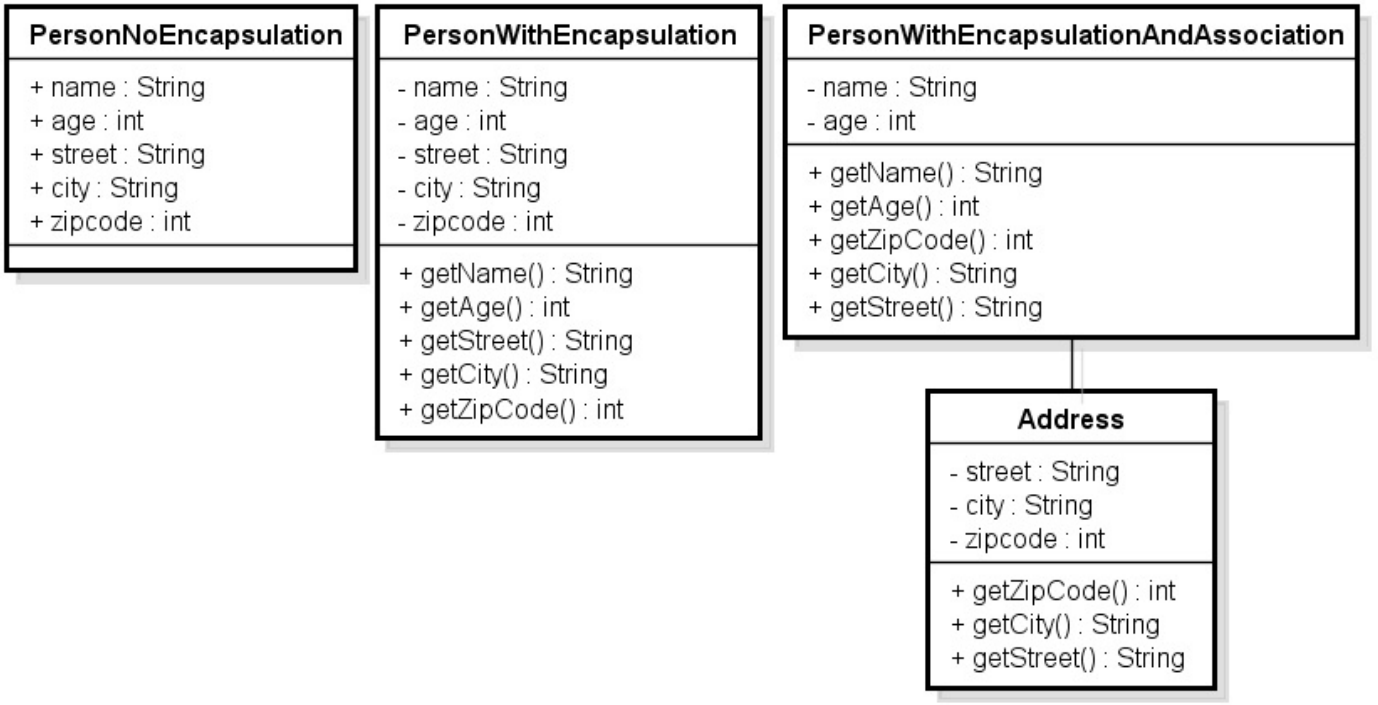


Abbildung 3-10 *Kapselung am Beispiel unterschiedlicher Personenklassen*

Die bisher gezeigten UML-Diagramme zeigten intuitiv verständliche Elemente. Hier sind jedoch bereits einige Feinheiten enthalten: Die Zeichen `'+'` und `'-'` dienen zur Beschreibung der Sichtbarkeiten `public` und `private`. Attribute können im mittleren Kasten und Methoden im unterer Kasten angegeben werden. Eine Verbindungslinie zwischen zwei Klassen beschreibt eine Assoziation. Weitere Details zur UML finden Sie auf der Webseite zum Buch `www.dpunkt.de/java-profi`.

Eine Realisierung ohne Kapselung ist mit der Klasse `PersonNoEncapsulation` gezeigt. Öffentlich zugängliche Attribute speichern die Informationen zu Name, Alter und Adressdaten. Zudem wird auf Zugriffsmethoden verzichtet. Dadurch müssen andere Klassen direkt auf die öffentlichen Attribute zugreifen. Schnell kommt der Wunsch auf, die Adressinformationen in eine eigenständige Klasse `Address` auszulagern, um eine bessere Trennung von Zuständigkeiten zu erreichen und die Kohäsion der einzelnen Klassen zu vergrößern. Weil andere Klassen bisher direkt auf die jetzt auszulagernden Attribute zugegriffen haben, gestalten sich selbst Änderungen an den Implementierungsdetails, hier das Herauslösen einer Klasse `Address`, als schwierig. Es sind nämlich als Folge (massive) Änderungen in allen einsetzenden Klassen notwendig. ***Voraussetzung für weniger Aufwand bei Änderungen ist demnach, dass der Zugriff auf Attribute gekapselt erfolgt und die Attribute möglichst*** **`private`** ***deklariert sind.***

Wäre die ursprüngliche Klasse analog zur Klasse `PersonWithEncapsulation` mit Kapselung realisiert worden, so wäre ein solcher Auslagerungsschritt wesentlich einfacher möglich. Es wäre lediglich eine Klasse `Address` zu definieren und alle zugehörigen Methodenaufrufe aus der Klasse `Person` an diese weiter zu delegieren. Die Klasse `PersonWithEncapsulationAndAssociation` setzt dies um und behält das nach außen veröffentlichte API bei. Daher müssen keine Anpassungen in anderen Klassen erfolgen. Anhand dieses Beispiels wird klar, dass die Datenkapselung

sehr wichtig ist. Anpassungen bleiben dann meistens lokal auf eine Klasse begrenzt, ohne Änderungen in anderen Klassen zu verursachen.

Achtung: API-Design – Einfluss von Kapselung und Kopplung

Kapselung und Kopplung haben einen großen Einfluss auf den Entwurf von APIs:

- Findet keine Datenkapselung statt, so sind alle öffentlichen Attribute nach außen für andere Klassen sichtbar und damit Bestandteil des APIs.
- Trotz Datenkapselung wird in Zugriffsmethoden der Rückgabedatentyp und damit häufig auch der Datentyp des korrespondierenden Attributs veröffentlicht. Im Idealfall handelt es sich jedoch nur um einen primitiven Typ oder ein Interface. Dann gibt es keine Abhängigkeiten von den Implementierungsdetails der realisierenden Klasse.

Es kann zu Problemen führen, wenn sich Änderungen an internen Daten im API bemerkbar machen und Änderungen in nutzenden Klassen erforderlich sind. Das gilt vor allem für öffentliche Schnittstellen, insbesondere das veröffentliche und durch andere Programme verwendete API, wenn man allgemeingültige oder zentrale Komponenten realisiert. Um in solchen Fällen überhaupt Modifikationen durchführen zu können, muss man alle Anwender der eigenen Klasse kennen und dafür sorgen, dass die notwendigen Folgeanpassungen in den benutzenden Programmen durchgeführt werden. Wird eine Klasse jedoch von diversen externen Klassen verwendet, die man nicht im Zugriff hat, so kann man nachträglich Änderungen zur Verbesserung des Designs nicht mehr ohne Folgen durchführen: Einige Programme lassen sich dann nicht mehr kompilieren oder es gibt Laufzeitfehler.

Diese kurze Diskussion sensibilisiert für die Probleme, ein gutes API zu entwerfen. Wie schwer dies trotz hoher Kompetenz sein kann, stellt man am JDK mit den vielen als veraltet markierten Methoden fest. Dies kann über den Javadoc-Kommentar `@deprecated`, die Annotation `@Deprecated` oder besser sogar durch beides geschehen. Derartige Methoden sollten in neu erstelltem Sourcecode nicht mehr verwendet werden.

Vererbung und Wiederverwendbarkeit

Mithilfe von Vererbung kann durch den Einsatz von Basisklassen und das Nutzen gemeinsamer Funktionalität ein übersichtlicheres Design erreicht und so mehrfach vorhandener Sourcecode vermieden werden. Allerdings sollte man Vererbung auch immer mit einer gewissen Vorsicht einsetzen. Sie ist zwar ein Mittel, um bereits modelliertes Verhalten vorhandener Klassen ohne Copy-Paste-Ansatz wiederzuverwenden und zu erweitern, aber nicht mit dem alleinigen Ziel, keinen Sourcecode zu duplizieren. Eine Vererbung, die lediglich dem Zweck dient, bereits existierende Funktionalität aus bestehenden Klassen zu verwenden, führt fast immer zu unsinnigen oder zumindest zweifelhaften Designs und wird

Implementierungsvererbung genannt. Für die entstehenden Klassen gilt des Öfteren dann keine wirkliche Subtyp-Beziehung mehr, wodurch es zum Teil zu Überraschungen bei deren Einsatz kommen kann.

Durch mehrere Klassen verwendbare Funktionalität sollte in eine separate Hilfsklasse ausgelagert und dann per ***Delegation***[8] statt Vererbung angesprochen werden.

3.1.5 Wissenswertes zum Objektzustand

Nachdem wir in den vorangegangenen Abschnitten bereits einiges über den objektorientierten Entwurf erfahren haben, möchte ich hier das Thema Objektzustand vertiefen.

Bereits bekannt ist, dass man unter dem momentanen Objektzustand die aktuelle Belegung der Attribute eines Objekts versteht. Die Menge der Objektzustände eines konkreten Objekts umfasst sämtliche möglichen Belegungen von dessen Attributen. Die gültigen Zustände beschränken sich meistens auf eine (kleine) Teilmenge daraus. Ausgehend von einem oder mehreren Startzuständen sollten nur solche Zustandsübergänge möglich sein, die wiederum zu einem gültigen Objektzustand führen. Dieser Wunsch wird in Abbildung 3-11 verdeutlicht. Dort sind einige gültige Zustände und Zustandsübergänge visualisiert. Der deutlich größere Zustandsraum ist als graues Rechteck dargestellt. Die damit beschriebenen Zustände sind alle möglich, aber nicht jeder verkörpert einen sinnvollen Objektzustand; einige davon sind explizit als `Invalid 1`, `Invalid 2` bzw. `Invalid n` dargestellt. Bei einem `Person`-Objekt könnten dies etwa negative Werte für Alter oder Größe sein.

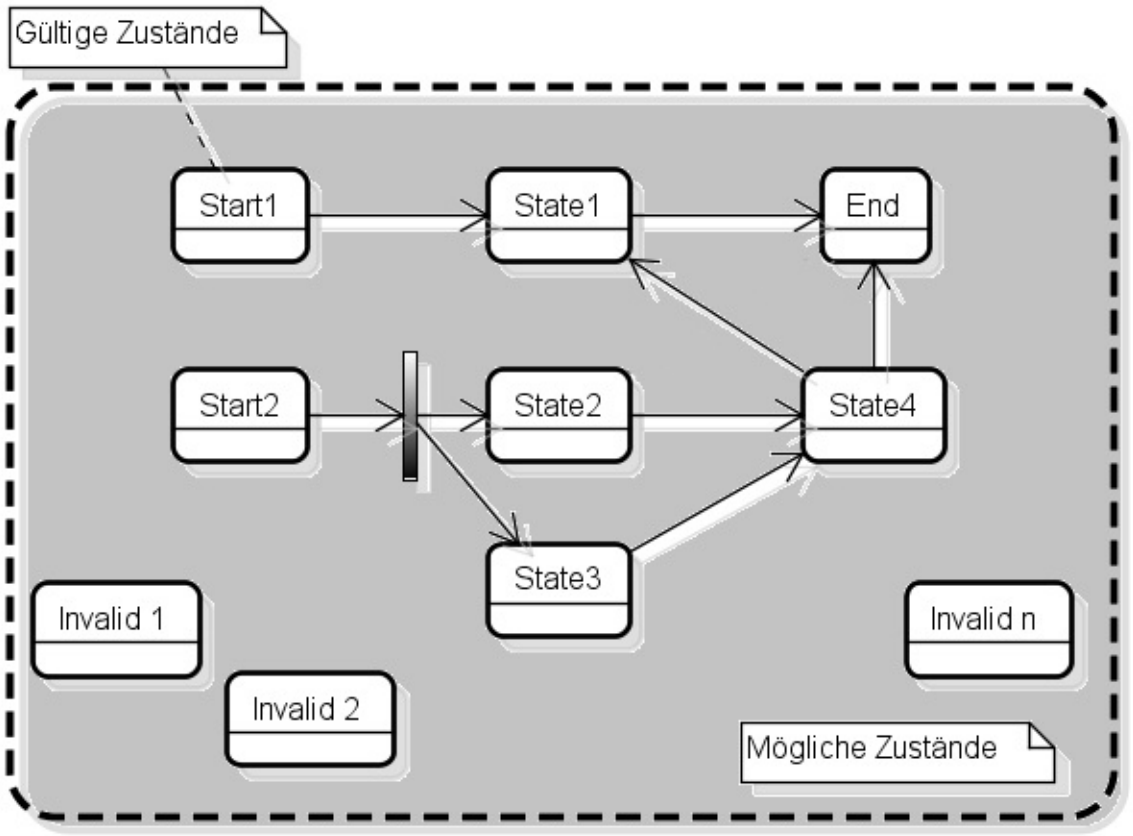


Abbildung 3-11 *Gültige Objektzustände und Zustandsraum*

Die Forderung nach gültigen Objektzuständen ist gar nicht so leicht zu erfüllen. Tatsächlich wird dies sogar in vielen Programmen vernachlässigt. Vielfach wird der erlaubte Zustandsraum nicht geprüft oder ist nicht einmal festgelegt. Dadurch besitzen Objekte zum Teil beliebige, wahrscheinlich auch ungültige Zustände. Dies löst häufig unerwartete Programmreaktionen aus, führt zu Berechnungsfehlern oder Abstürzen und erhöht den Aufwand beim Testen. Um die genannten Probleme zu verhindern, müssen wir ein besonderes Augenmerk auf all diejenigen Methoden werfen, die Änderungen am Objektzustand auslösen. Eine Voraussetzung dazu ist es, einen gültigen Ausgangszustand unter anderem durch die Prüfung der Eingabeparameter sicherzustellen. Ungültige Parameterwerte sollten zurückgewiesen werden, um eine sichere Ausführung von Methoden zu gewährleisten. Zudem müssen durchgeführte Berechnungen wiederum gültige Objektzustände erzeugen. Das Ganze wollen wir ein klein wenig formaler betrachten, bevor wir es an einem Beispiel verdeutlichen.

Invarianten, Vor- und Nachbedingungen

Sogenannte Vor- und Nachbedingungen sowie Invarianten spielen bei der Programmverifikation und dem Beweis der Korrektheit von Programmen in der theoretischen Informatik eine wichtige Rolle. In der Praxis kann man auch ohne Kenntnis der Details von diesen profitieren und ungültige Objektzustände vermeiden.

Mithilfe einer ***Invariante*** kann man Aussagen über Teile des Objektzustands treffen. Damit beschreibt man *unveränderliche* Bedingungen, die vor, während und

nach der Ausführung verschiedener Programmanweisungen gültig sein sollen. Ein Beispiel für eine solche Forderung ist, dass das Alter einer Person nicht negativ sein darf.

Über sogenannte ***Vorbedingungen*** lässt sich eine zur Ausführung einer Methode notwendige Ausgangssituation beschreiben: Die Vorbedingung einer Suchmethode könnte etwa sein, dass Elemente vorhanden und sortiert sind. Wenn diese Vorbedingung erfüllt ist, so wird nach Ausführung der Methode als Ergebnis die Position des gesuchten Elements zurückgeliefert, oder aber -1, wenn kein solches existiert. Bedingungen nach Ausführung eines Programmstücks werden durch sogenannte ***Nachbedingungen*** formuliert.

Info: Design by Contract

Der Einsatz von Invarianten, Vor- und Nachbedingungen erinnert an die Forderungen des von Bertrand Meyer beim Entwurf der Programmiersprache Eiffel erfundenen »***Design by Contract***«. Idee dabei ist es, Vor- und Nachbedingungen bei jedem Methodenaufruf abzusichern. Dies soll verhindern, dass eine Methode mit ungültigen Werten aufgerufen wird. Auf diese Weise versucht man, die Konsistenz des Programmzustands sicherzustellen sowie ein mögliches Fehlverhalten des Programms auszuschließen. Beim Verlassen einer Methode wird durch Nachbedingungen sichergestellt, dass nur erwartete, gültige Werte als Rückgabe möglich sind. Weitere Informationen finden Sie unter `http://www.eiffel.com/developers/design_by_contract.html`.

Der Objektzustand am Beispiel

Im folgenden Beispiel eines grafischen Editors werden wir Vor- und Nachbedingungen nutzen, um sicherzustellen, dass die Koordinaten grafischer Figuren einem Rasterpunkt entsprechen. Nehmen wir an, es soll ein Raster mit einem Abstand von 10 Punkten modelliert werden.

Wir entwerfen dazu eine Klasse `GridPosition`, die ihre Koordinaten in den zwei Attributen `x` und `y` vom Typ `int` speichert. Gültige Punkte, und damit Belegungen der Attribute, sollen immer auf den genannten Rasterpunkten liegen. Die Wertebelegungen der Attribute vom Typ `int` beschreiben den Zustandsraum, d. h. alle potenziell möglichen Objektzustände. Die Menge der erlaubten Zustände ist jedoch wesentlich geringer. Das gilt im Speziellen für dieses Beispiel und lässt sich auf viele Fälle in der Praxis übertragen.

Die Klasse `GridPosition` bietet zwei Business-Methoden, die Objektverhalten beschreiben – dies sind `addOffset(int, int)` und `setSamePosition(int)`. Zur Demonstration der Gefahr von Inkonsistenzen im Objektzustand sind öffentliche `set()`-Methoden für die Attribute `x` und `y` definiert.

```
public final class GridPosition
{
```

```
    private static final int GRID_SIZE = 10;

    // Gewünschte Invariante: x, y liegen immer auf einem Raster der Größe 10
    private int x = 0;
    private int y = 0;

    public void addOffset(final int dx, final int dy)
    {
        // Vorbedingung: x, y auf einem beliebigen Rasterpunkt
        checkOnGrid(x,y);

        x += snapToGrid(dx);
        y += snapToGrid(dy);

        // Nachbedingung: x, y immer noch auf einem Rasterpunkt
        checkOnGrid(x,y);
    }

    public void setSamePosition(final int position)
    {
        // Vorbedingung: x, y auf einem beliebigen Rasterpunkt
        checkOnGrid(x,y);

        x = snapToGrid(position);
        y = snapToGrid(position);

        // Nachbedingung: x = y und immer noch auf einem Rasterpunkt
        checkOnGrid(x,y);
    }

    private static void checkOnGrid(final int x, final int y)
    {
        if (x % GRID_SIZE != 0 || y % GRID_SIZE != 0)
            throw new IllegalStateException("invalid position, not on grid");
    }

    private int snapToGrid(final int value)
    {
        return value - value % GRID_SIZE;
    }

    public int getX()                   { return x; }
    public int getY()                   { return y; }
    public void setX(final int x)       { this.x = x; }  // problematisch!
    public void setY(final int y)       { this.y = y; }  // problematisch!
}
```

Dem aufmerksamen Leser fällt die ungewöhnliche Formatierung der Zugriffsmethoden (`get()`-/`set()`-Methoden) auf. Die hier angewendete kompakte Schreibweise wird in Abschnitt 19.3.1 diskutiert. Nur so viel vorab, sie kann die Lesbarkeit für derart einfache Zugriffsmethoden mit sehr wenig Sourcecode, bevorzugt einer Anweisung, erhöhen.

Veränderungen am Objektzustand

Nach der Konstruktion eines `GridPosition`-Objekts entspricht der Startzustand der Wertebelegung $x = y = 0$. Wie bereits erwähnt, sollten öffentliche Methoden ein

Objekt aus einem gültigen Objektzustand in einen anderen gültigen versetzen. Hier gilt:

- Jeder Aufruf von `addOffset(dx, dy)` führt zu einem Zustandswechsel bei dem Folgendes gilt: $x = x + dx^*$ und $y = y + dy^*$.
- `setSamePosition(pos)` wechselt in den Zustand $x = y = pos^*$.[9]

In der Klasse `GridPosition` werden jedoch zwei öffentliche `set()`-Methoden angeboten, in denen keine Korrektur der Eingabewerte stattfindet. Rufen andere Klassen diese `set()`-Methoden statt der Business-Methoden auf, so können beliebige Werte für die Attribute `x` und `y` gesetzt werden. Diese liegen höchstwahrscheinlich nicht auf dem gewünschten Raster und stellen somit auch keinen gültigen Objektzustand dar. Die gewünschte Invariante kann somit nicht sichergestellt werden.

Tipp: Probleme durch öffentliche `set()`-Methoden

In der Praxis findet man aufgrund fehlender (oder zu weniger) Business-Methoden eine Vielzahl öffentlicher `set()`-Methoden. Durch deren Aufruf ist es aber möglich, den Zustand eines Objekts (auf unerwartete Weise) zu verändern. Häufig wird dann von Aufrufern Objektverhalten durch eine Abfolge von `get()`/`set()`-Methoden realisiert. Dieser Weg an den Business-Methoden vorbei, erschwert die Einhaltung eines gültigen Objektzustands ungemein. Um Probleme zu vermeiden, sollten `set()`-Methoden in ihrer Sichtbarkeit eingeschränkt oder, falls möglich, durch Business-Methoden ersetzt werden. Eine Anleitung dazu liefert das Refactoring ERSETZE MUTATOR- DURCH BUSINESS-METHODE in Abschnitt 17.4.4. In der Klasse `Grid-Position` wurden bereits Business-Methoden bereitgestellt, die `set()`-Methoden allerdings (noch) nicht entfernt.

Objektzustände während der Objektinitialisierung

Manchmal kommt es während der Objektinitialisierung zu ungewünschten, vermeidbaren oder sogar ungültigen Zwischenzuständen. Dies gilt insbesondere, wenn lediglich ein Defaultkonstruktor und diverse `set()`-Methoden angeboten werden.

Die Grundlage für die Diskussion bildet folgende Klasse `SimpleImage`. Sie verwendet einen Namen, eine Breite und eine Höhe sowie die eigentlichen Bilddaten und speichert diese Werte in vier privaten Attributen. Der Zugriff ist durch öffentliche `get()` - und `set()`-Methoden möglich – hier nur für das Attribut `name` gezeigt:

```
public final class SimpleImage
{
    private String name;
    private int width;
    private int height;
```

```
    private byte [] imageData;

    public SimpleImage()
    {}

    public void setName(final String name)
    {
        this. name = name;
    }

    public String getName()
    {
        return this. name;
    }

    // weitere Getter und Setter …
}
```

Das von der Klasse angebotene API bietet keine Unterstützung zur Sicherstellung und Wahrung eines gültigen Objektzustands. Vielmehr ist dazu erforderlich, dass nach der Objektkonstruktion alle Attribute durch Aufruf von `set()`-Methoden sukzessive mit gültigen Werten initialisiert werden. Betrachten wir den Ablauf anhand des Einlesens aus einer Datei in der folgenden Methode `createSimpleImageFromFile()`:

```
public SimpleImage createSimpleImageFromFile() throws IOException
{
    final SimpleImage simpleImage = new SimpleImage();

    // ACHTUNG: Nur zur Demonstration zu früh registrieren
    storedImages.add(simpleImage);

    final String imageName = readNameFromFile();
    simpleImage.setName(imageName);          // Name setzen

    try
    {
        final int imageWidth = readWidthFromFile();
        simpleImage.setWidth(imageWidth);    // Breite setzen

        final int imageHeight = readHeightFromFile();
        simpleImage.setHeight(imageHeight);  // Höhe setzen
    }

    catch (final NumberFormatException e)
    {
        // ACHTUNG: Keine Fehlerbehandlung zur Demonstration
    }

    final String imageData = readImageDataFromFile();
    simpleImage.setImageData(imageData);     // Daten setzen

    return simpleImage;
}
```

Ähnliche Programmausschnitte trifft man in der Praxis immer wieder an. Was daran problematisch ist, wollen wir nun analysieren:

1. **Ungültiger Objektzustand** – Es werden ungültige Objektzustände sichtbar. Im

Beispiel kann dies dadurch geschehen, dass andere Programmteile die Liste der Bilder `storedImages` zu früh – vor Abschluss der Objektinitialisierung eines neu hinzugefügten Bilds – auslesen.

2. **Teilinitialisierter Objektzustand** – Treten Fehler beim Einlesen aus der Datei oder beim Verarbeiten der eingelesenen Daten auf, so verbleibt das Objekt in einem teilinitialisierten Objektzustand, der höchstwahrscheinlich keinem gültigen Objektzustand entspricht.
3. **Unnötige Zugriffsmethoden** – Es müssen viele, möglicherweise bei anderem Design unnötige Zugriffsmethoden für Attribute bereitgestellt werden. Dies verhindert eine Realisierung, die nur in den wirklich notwendigen Teilen veränderlich ist.
4. **Öffentliche Zugriffsmethoden** – Öffentlich definierte Zugriffsmethoden erlauben beliebige Änderungen durch andere Objekte. Daraus können inkonsistente Objektzustände resultieren.

Ungültiger Objektzustand

Im Beispiel wird zunächst ein `SimpleImage`-Objekt über einen Defaultkonstruktor erzeugt. Vorausgesetzt es werden gültige Eingabedaten an die vier `set()` -Methoden übergeben, so führt dies auch zu vier Änderungen im Objektzustand. Würde ein anderes Objekt genau das neu erzeugte `SimpleImage`-Objekt aus der Liste `storedImages` auf Veränderungen untersuchen (vgl. Abschnitt 18.3.7 zum Entwurfsmuster BEOBACHTER), so sind im Extremfall vier Änderungen in kurzer Abfolge nacheinander beobachtbar. Erst die Letzte davon erzeugt einen gültigen Objektzustand.

Teilinitialisierter Objektzustand

Nehmen wir an, einer der eingelesenen Werte für Breite und Höhe wäre keine Zahl und damit ungültig. Die fehlende Fehlerbehandlung (der leere `catch`-Block) verhindert, dass dieses Problem zum Aufrufer propagiert wird. Stattdessen kommt es zu einer unbemerkten Inkonsistenz im Objektzustand: Das neu konstruierte `SimpleImage`-Objekt besitzt nach dem Einlesen der korrekten Werte von Name und Bilddaten falsche Werte für Breite oder Höhe.

Unnötige Zugriffsmethoden

Alle Attribute der Klasse `SimpleImage` sind über `set()` -Methoden veränderbar. Ein Teil dieser Methoden wurde lediglich eingeführt, um ein sukzessives Setzen der Werte zu ermöglichen. Für die Nutzung der Klasse ist wahrscheinlich, dass Höhe, Breite und Bilddaten voneinander abhängig sind und nur miteinander als Einheit zu verändern sein sollten. Statt einzelner `set()` -Methoden sollte man daher eine Business-Methode `changeImageData(int, int, byte[])` anbieten. Zudem

sollte der Name des Bildes unveränderlich sein. Dies wird durch die Deklaration des Attributs `name` als `final` erreicht:

```
public final class SimpleImage
{
   private final String name;
   private int width;
   private int height;
   private byte[] imageData;

   public SimpleImage(final String name, final int width, final int height,
                      final byte[] imageData)
   {
      this.name = name;
      changeImageData(width, height, imageData);
   }

   public void changeImageData(final int width, final int height,
                               final byte[] imageData)
   {
      this.width = width;
      this.height = height;
      this.imageData = imageData;
   }

   public String getName()
   {
      return this.name;
   }

   // weitere get()-Methoden …
}
```

Die genannten Änderungen machen die Klasse besser lesbar. Wichtiger ist aber, dass die Abhängigkeiten der Attribute untereinander nun deutlich zu erkennen sind und nur durch eine Methode in der öffentlichen Schnittstelle beschrieben werden. Diese Business-Methode ist auch die richtige Stelle, um Änderungen an andere Objekte zu kommunizieren.

Betrachten wir, wie sich das neue API der Klasse `SimpleImage` im Einsatz auswirkt. Statt der direkten `set()`-Aufrufe erfolgt zunächst eine Zwischenspeicherung in lokalen Variablen. Erst nachdem alle Daten erfolgreich eingelesen werden konnten, wird ein `SimpleImage`-Objekt konstruiert. Das Auslesen und Verarbeiten der in der Liste `storedImages` gespeicherten Elemente bereitet nun keine Probleme mehr – es werden nur gültige Zustände nach außen sichtbar. Selbst die fehlende Behandlung eines Parsing-Fehlers hat bei dieser Art der Realisierung deutlich weniger negative Auswirkungen. Es wird dann einfach kein Objekt erzeugt. Folgendes Listing zeigt das neue API im Einsatz:

```
public SimpleImage createSimpleImageFromFile() throws IOException
{
   final String imageName = readNameFromFile();

   try
   {
       final int imageWidth = readWidthFromFile();
```

```
        final int imageHeight = readHeightFromFile();
        final String imageData = readImageDataFromFile();

        final SimpleImage simpleImage = new SimpleImage(imageName, imageWidth,
                                                        imageHeight,
                                                        imageData.getBytes());
        return simpleImage;
    }
    catch (final NumberFormatException ex)
    {
       log.error("failed to create SimpleImage object from file", ex);
    }

    // Alternativ: return null
    throw new IOException("illegal content. failed to parse width/height", ex);
}
```

Falls Zugriffsprobleme beim Lesen aus der Datei auftreten, wird eine `IOException` ausgelöst. Als Folge wird kein `SimpleImage`-Objekt erzeugt. Darüber hinaus gibt es noch den Spezialfall zu behandeln, dass die hinterlegten Angaben zu Breite und Höhe keine Zahlenwerte darstellen, was eine `NumberFormatException` auslöst. Wir wollen dieses Detail jedoch nicht direkt an den Aufrufer der Methode weiterleiten. Daher wrappen wir die Exception in eine `IOException` und geben dort die Information über das Problem mit. Alternativ könnte man den Wert `null` zurückliefern. Das würde aber aufseiten des Aufrufers eine Spezialbehandlung erfordern, weshalb wir hier die zuvor beschriebene Variante wählen.

Öffentliche Zugriffsmethoden

Alle Attribute der eingangs gezeigten Klasse `SimpleImage` waren über öffentliche `set()`-Methoden veränderlich. Damit konnten Klienten[10] beliebige Änderungen am Objektzustand durchführen. Im vorherigen Schritt haben wir bereits die Anzahl öffentlicher, den Objektzustand beeinflussender Methoden verringert. Als Folge konnten sämtliche `set()`-Methoden entfallen. Eine Zustandsänderung ist nur noch durch Aufruf der Methode `changeImageData(int, int, byte[])` möglich. Erfolgt innerhalb dieser Methode eine Parameter- und Konsistenzprüfung, so kann das Objekt von anderen Klassen nicht mehr in einen ungültigen Objektzustand versetzt werden. Dies gilt jedoch nur für Singlethreading. Für Multithreading müssten dazu noch eine geeignete Synchronisierung erfolgen, um Inkonsistenzen sicher auszuschließen.

Weitere Komplexitäten durch Multithreading

Methodenaufrufe werden nicht als Ganzes abgearbeitet, sondern in Form vieler kleiner Schritte. Dadurch kommt es während der Abarbeitung der Anweisungen teilweise zu ungültigen oder unerwarteten Zwischenzuständen im Objektzustand. Abbildung 3-12 zeigt dies für den Aufruf `addOffset(50, 30)`. Dort wird kurzzeitig ein Zustand `x=50; y=0` eingenommen, der aber logisch keinem Methodenaufruf entspricht. Erst das Setzen von `y=30` führt dazu.

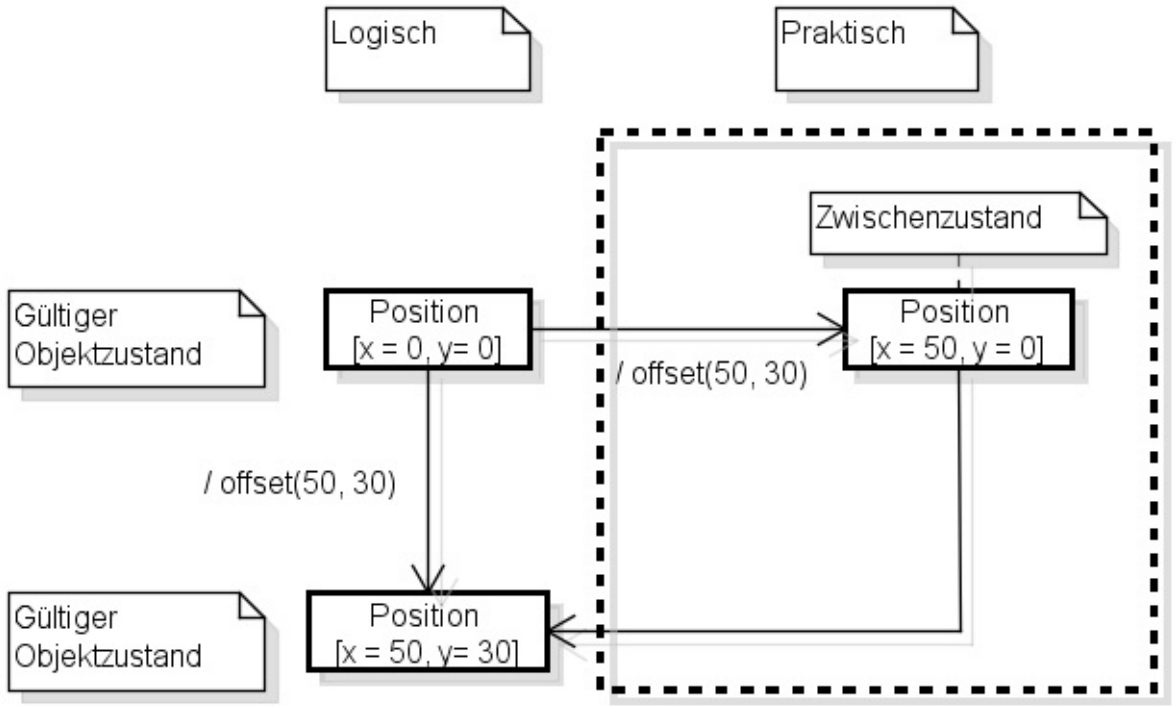


Abbildung 3-12 *Objektzustandsübergänge*

Diese Zwischenzustände lassen sich systembedingt nicht verhindern – sind bei Singlethreading jedoch nicht beobachtbar. Für Multithreading muss dafür gesorgt werden, dass diese Zwischenzustände für andere Objekte nicht zugreifbar sind und sich ein Objekt nach der Ausführung einer Methode wieder in einem gültigen Objektzustand befindet. Einen wichtigen Beitrag dazu leisten sowohl Datenkapselung als auch die Definition von kritischen Bereichen. Darauf gehe ich nun etwas genauer ein.

Datenkapselung

Mithilfe von Datenkapselung kann man vermeiden, dass andere Objekte jederzeit einen Blick in die »Innereien« eines anderen Objekts werfen können, und somit auch verhindern, dass möglicherweise mit ungültigen Zwischenzuständen weitergearbeitet wird. Im ungünstigsten Fall sind sogar externe, verändernde Zugriffe möglich, wodurch das Objekt die Kontrolle über seinen Objektzustand abgibt. Das ist nahezu immer zu vermeiden. Bei Datenkapselung verhindern die `private` definierten Attribute die zuvor beschriebenen Probleme.

Kritische Bereiche

Um bei Multithreading unerwünschte Zwischenzustände durch Unterbrechungen von Berechnungen zu vermeiden, kann man kritische Bereiche nutzen. Das ist notwendig, weil ein aktiver Thread nahezu jederzeit in seiner Abarbeitung unterbrochen und ein anderer Thread aktiviert werden kann. Geschieht ein solcher Thread-Wechsel während der Abarbeitung einer Methode, sind eventuell noch nicht alle Zuweisungen an die Attribute erfolgt und ein anderer Thread sieht einen Zwischenzustand. Kapitel 7 behandelt diese Problematik ausführlich.

3.2 Grundlegende OO-Techniken

Bis hierher haben wir einige wichtige Einblicke in das objektorientierte Design bekommen. Dieser Abschnitt geht nochmals etwas genauer auf grundlegende OO-Techniken ein, die wir für den Entwurf größerer Softwaresysteme als Basisbausteine benötigen: Schnittstellen, Basisklassen und deren Kombination. All dies hilft dabei, Funktionalität zu kapseln und Komponenten (bestehend aus verschiedenen Klassen, Interfaces und Packages) voneinander unabhängig zu halten.

Abschnitt 3.2.1 behandelt die Definition von Schnittstellen. Durch explizit definierte Schnittstellen, auch Interfaces genannt, kann man ein »Angebot von Verhalten« festlegen, indem man eine Menge von Methoden angibt. In Abschnitt 3.2.2 lernen wir zunächst, wie und warum Basisklassen entstehen, und dann, wozu diese dienen. Danach stelle ich eine spezielle Form von Basisklassen vor, nämlich abstrakte Basisklassen, die sich dadurch auszeichnen, dass sie nicht instanziierbar sind. Abstrakte Basisklassen können als Stellvertreter für konkrete Subklassen agieren, die das gewünschte, spezifische Verhalten realisieren. Für sich allein eingesetzt können die beiden Techniken Interface und abstrakte Basisklasse bereits hilfreich sein, aber sie lassen sich darüber hinaus gewinnbringend kombinieren. Darauf geht Abschnitt 3.2.3 genauer ein.

3.2.1 Schnittstellen (Interfaces)

Wenn Aufrufe an Methoden anderer Klassen erfolgen sollen, so existiert eine sogenannte Use-Beziehung: Eine Klasse nutzt eine andere. Dabei gibt es einen Aufrufer und einen Bereitsteller von Funktionalität. Man spricht dann zum Teil auch von Client und Server. Im einfachsten Fall ergibt sich die Schnittstelle, die ein Client von einem Server nutzen kann, implizit über die öffentlichen Methoden. Manchmal ist es jedoch zweckmäßiger, Methoden zu gruppieren und diesen Funktionsblöcken einen eigenen Namen zu geben. Dies ist durch die nun besprochene Technik der Interfaces möglich.

Einführung

Am Beispiel eines Kunden (Client), der bei einem Pizza-Online-Lieferservice (Server) Bestellungen aufgeben kann, lernen wir die Definition einer Schnittstelle mithilfe des Schlüsselworts `interface` kennen. Wir schreiben Folgendes:

```
// Zeigt drei Varianten der Angabe von Zugriffsmodifiern
public interface IOnlinePizzaService
{
    public long registerNewCustomer(String name, Address address);
    abstract boolean orderPizza(long customerId, Pizza pizza);
    public abstract DeliveryInformation deliverTo(long customerId);
}
```

Ganz nebenbei lernen wir eine Besonderheit von Interfaces in Java kennen: Unabhängig von der expliziten Angabe der gezeigten Zugriffsmodifier, besitzen die Methoden alle die Sichtbarkeit `public` und sind zudem `abstract`. Weitere Details nennt der folgende Praxistipp.

Tipp: Implizite Zugriffsmodifier

In Java sind alle in Interfaces definierten …

- **Methoden** immer mit den Zugriffsmodifiern `public` und `abstract` definiert.
- **Variablen** implizit als `public`, `static` und `final` definiert. Diese sind dadurch als Konstanten nach außen sichtbar.

Die obigen Zugriffsmodifier können explizit im Sourcecode angegeben werden, und zwar einzeln oder gemeinsam, und sind unabhängig davon immer vorhanden. Alle anderen Zugriffsmodifier sind in Interfaces nicht erlaubt. Mit Java 8 wird mit Defaultmethoden ein neues Sprachfeature eingeführt, das in Interfaces genutzt werden kann. Details dazu entnehmen Sie bitte Abschnitt 11.2.

Interfaces als »Angebot von Verhalten«

Weil die in Interfaces aufgelisteten Methoden so etwas wie einen Vertrag beschreiben, können sich sowohl Client als auch Server darauf stützen. Dies ermöglicht eine Entkopplung von Realisierung und Spezifikation. Diese Entkopplung ist die Basis für viele Entwurfsmuster, wie ITERATOR, DEKORIERER, PROXY, NULL-OBJEKT u. v. m. (vgl. Kapitel 18).

Die Definition eines Interface hat zwei Facetten: Zum einen wird für Clients festgelegt, welche Methoden sie aufrufen können. Zum anderen wird für einen Server bestimmt, welche Methoden durch ihn implementiert werden müssen, um das Interface zu erfüllen. Übertragen auf das Beispiel des Pizza-Online-Lieferservice bedeutet dies Folgendes: Für den Kunden ist der reale Pizza-Betrieb, also die konkrete Ausprägung, nicht von Interesse. Er benötigt für seine Bestellungen lediglich die Schnittstelle, z. B. die Internetseite (und eine Möglichkeit, diese zu erhalten[11]). Damit der Pizza-Betrieb seine Leistungen online zur Verfügung stellen kann, muss dieser wiederum nur wissen, welche Funktionalität er zur Erfüllung der Schnittstelle bereitstellen muss. Möglicherweise können auch andere Pizza-Betriebe später die geforderte Schnittstelle erfüllen und ihre Dienste dann online anbieten.

Bevor Sie damit beginnen, jeweils ein Interface für Ihre Klassen einzuführen, bedenken Sie aber bitte Folgendes: Ein Interface ist ***kein Selbstzweck*** und es stellt per se auch kein gutes Design dar. Wie oben schon angedeutet, helfen Interfaces bei der ***Definition von Verhalten*** und ***Entkopplung von Komponenten***. Innerhalb eigener Packages ist ein solcher Abstraktionsschritt eher selten vonnöten.

Wissenswertes zur Definition von Interfaces

Nach diesen kurzen Vorbetrachtungen kommen wir nochmals auf Wissenswertes zur Definition einer Schnittstelle zurück.

Wie schon bekannt, wird durch ein Interface keine Realisierung vorgegeben, sondern nur einige ***Methoden ohne Implementierung***, d. h. ***abstrakte Methoden***. Beim Betrachten des obigen Interface kommen wir noch zu einer wichtigen Erkenntnis: Sowohl Nutzer als auch Implementierer verwenden lediglich die angegebene Menge an Methoden. Um beiden das Verständnis der bereitzustellenden Funktionalität und den Einsatz zu erleichtern, sollten die Methodennamen in Interfaces besonders sprechend und klar gewählt werden. Diese Forderung ist im zuvor gezeigten Interface `IOnlinePizza-Service` schon recht gut erfüllt. Trotzdem ist nicht vollständig klar, was die Methoden im Detail tun sollen und was die Eingabe- und Rückgabewerte bedeuten. Schauen wir uns exemplarisch eine für die Methode `registerNewCustomer(String, Address)` verbesserte Variante des Interface an:

```
public interface IOnlinePizzaService
{
    /**
     * create a new customer with the given name and delivery address
     *
     * @param name – the name of the customer to create
     * @param deliveryAddress – the customer's address
     *
     * @return the id of the newly created customer
     * @throws DuplicateCustomerException if a customer with the same
     *         name already exists
     */
    long registerNewCustomer(String name, Address deliveryAddress)
                             throws DuplicateCustomerException;

    boolean orderPizza(long customerId, Pizza pizza);
    DeliveryInformation deliverTo(long customerId);
}
```

In diesem Beispiel wurde aus Gründen der Übersichtlichkeit nur eine knappe Dokumentation für eine Methode gezeigt. Aber schon daran erkennt man, wie hilfreich dies ist. In der Praxis ist für alle Methoden eines Interface eine gute Dokumentation zwingend notwendig, damit Nutzer und Implementierer eines Interface die gleiche Vorstellung von dem bereitgestellten bzw. bereitzustellenden Verhalten haben. Das gilt selbst für ähnlich einfache Interfaces.

Abgrenzung von Interfaces und Vererbung

Sowohl das Implementieren eines Interface als auch Vererbung sind Techniken, um Funktionalität zu beschreiben bzw. bereitzustellen. Während Vererbung eine Spezialisierung ausdrückt, liegt bei Interfaces der Fokus auf dem Angebot oder Ausüben von Funktionalität bzw. der Beschreibung einer ***Verhaltensweise*** (auch ***Rolle*** genannt). Es wird eine ***»can-act-like«-Beziehung*** oder auch ***»provides«-Beziehung***

realisiert.

Mit einem ***Interface*** wird für ein Objekt beschrieben, ***was es kann***. Im Gegensatz dazu wird mit ***Vererbung*** für ein Objekt beschrieben, ***was es ist***. Insbesondere sollte dabei die »is-a«-Beziehung eingehalten sein.

Mehrfachvererbung der Definition von Verhaltensweisen

Während in Java jede Klasse nur genau eine Basisklasse besitzen kann und somit Mehrfachvererbung von Zustand oder Verhalten durch mehrere Basisklassen nicht möglich ist, sind Klassen nicht darauf festgelegt, nur ein einziges Interface zu implementieren.

Eben erwähnte ich, dass mithilfe eines Interface für ein Objekt beschrieben wird, was es kann. Dieses Können einer Klasse muss jedoch nicht nur auf eine Funktionalität, Verhaltensweise oder Rolle beschränkt sein, sondern kann durchaus mehrere umfassen. Somit können Klassen mehrere voneinander unabhängige Funktionalitäten oder Verhaltensweisen gemäß »can-act-like« ausüben.

Namensgebung von Interfaces

Teilweise ist es hilfreich, dass Interfaces als solche anhand ihres Namens erkennbar sind. Im JDK wird vielfach die Endung `-able` verwendet, um die »can-act-like«-Beziehung zu betonen, also die Beschreibung von Rollen. Nicht in jedem Fall macht diese Namensgebung tatsächlich Sinn und ist anwendbar. Während bei `java.lang.Runnable` recht klar wird, dass eine Schnittstelle vorgegeben wird, gilt dies für das Interface `javafx.event.EventHandler` zur Behandlung von GUI-Events in JavaFX nicht mehr. Anhand der für letzteres Interface gewählten Namensgebung ist keine Unterscheidbarkeit zu Klassen gegeben. Mitunter ist es aber wünschenswert, bereits am Namen des Typs einer Referenzvariable zwischen Interface und Klasse unterscheiden zu können, um so zu erkennen, dass man lose Kopplung einhält und nicht versehentlich eine konkrete Klasse referenziert.

Für selbst definierte Interfaces wird in diesem Buch in der Regel das Präfix `I` verwendet, um die Interface-Eigenschaft zu betonen. Dieses Vorgehen findet man z. B. auch in der Programmierung von SWT (Eclipse RCP) oder als Standard in der Microsoft-Welt. Der Einsatz hängt stark davon ab, ob man eine Unterscheidung zwischen Interfaces und Klassen direkt durch den gewählten Namen sichtbar machen möchte. Die Kennzeichnung ist also tendenziell eine Frage von persönlichen Vorlieben oder durch Coding Conventions vorgegeben (vgl. Kapitel 19).

Bewertung

Der Einsatz von Interfaces …

+ entkoppelt Spezifikation und Realisierung: Dadurch kennen sich nutzende und

Funktionalität bereitstellende Klassen nicht direkt, sondern kommunizieren nur über das Interface.

+ erleichtert den Austausch einer konkreten Implementierung: Ein Beispiel dafür ist das Interface `List` aus dem Collections-Framework mit seinen Realisierungen `ArrayList`, `LinkedList` und `Vector`. Nutzende Klassen dürfen sich daher nicht auf spezielle Implementierungen oder Implementierungsdetails verlassen. Für Listen darf man also weder einen konstanten Aufwand beim indizierten Zugriff auf Elemente annehmen (`ArrayList`) noch einen konstanten Aufwand beim Hinzufügen am Ende der Liste (`LinkedList`). Details dazu finden Sie in Abschnitt 5.1.5.

+ ermöglicht die spezifische Definition einer oder mehrerer wohldefinierter Schnittstellen. Eine Klasse kann dann je nach Anwendungsfall die eine oder andere Rolle spielen, abhängig davon, über welches Interface sie angesprochen wird.

+ erleichtert den Überblick über die angebotenen Methoden, da diese in Interfaces zentral definiert werden. Verzichtet man auf die Definition von Interfaces, so bilden alle öffentlichen Methoden die Schnittstelle einer Klasse. Diese ist einerseits nicht immer offensichtlich, da die öffentlichen Methoden im Sourcecode verstreut (bzw. durchmischt mit Methoden geringerer Sichtbarkeiten) sind, und andererseits wenig spezifisch, da sich eben alle öffentlichen Methoden ohne semantische Gruppierung in der Klasse befinden. Die Sammlung von Methoden in einem Interface bietet außerdem den Vorteil, dass man leicht einen Überblick darüber gewinnt, ob das Interface gut geschnitten ist (d. h., es besteht aus sinnvollen Methoden mit sprechenden Namen, die eine spezifische Funktionalität abdecken) oder eben nicht. Für folgendes Beispiel ist das offensichtlich nicht der Fall:

```
interface IUniversalDiskPizzaAndMore
{
    void scanDisk( Drive drive) throws IOException;

    Iterable<Customer> getAllCustomers();

    boolean orderPizza( long customerId, Pizza pizza);
}
```

○ bietet keinen Zugriff auf Konstruktoren realisierender Klassen und keine Möglichkeit zur Erzeugung von Realisierungen,[12] weil nur Methoden deklariert werden.

○ erfordert die Realisierung öffentlicher Methoden – nämlich derjenigen im Interface.

○ erhöht leicht die Komplexität. Die Abstraktion von Funktionalität über ein Interface wird häufig erst dann benötigt, wenn es mehr als eine Realisierung gibt (oder sicher geben wird) oder spezielle Funktionalitäten extrahiert werden sollen.

3.2.2 Basisklassen und abstrakte Basisklassen

In diesem Abschnitt schauen wir kurz darauf, warum und wie Basisklassen entstehen, und motivieren mögliche Einsatzgebiete für abstrakte Basisklassen.

Basisklassen und ihre Vorteile

Bekanntermaßen entstehen Basisklassen beim Entwurf dann, wenn man erkennt, dass Klassen eine oder mehrere Gemeinsamkeiten besitzen und diese dann in eine Basisklasse ausgelagert werden. Dabei durchläuft man in etwa folgende vier Schritte, um gemeinsame Funktionalität in einer Basisklasse zusammenzuführen:

1. Identifiziere potenzielle Klassen als Kandidaten mit genügend Gemeinsamkeiten.
2. Erstelle eine Hierarchie mit gemeinsamer Basisklasse (wodurch die Klassen dann zu Subklassen werden).
3. Lagere die Gemeinsamkeiten geeignet in die Basisklasse aus.
4. Entferne diese Gemeinsamkeiten aus den Subklassen.

Dieses Vorgehen ist insofern praktisch, weil es erlaubt, Code-Duplikation zu vermeiden, die ohne gemeinsame Basisklasse durch die mehrfache Realisierung von Funktionalität in den jeweiligen Klassen entstehen würde. Ein weiterer Grund für die Einführung einer gemeinsamen Basisklasse ist, dass man mehrere Klassen einheitlich behandeln möchte, ohne die Details der Spezialisierungen kennen zu müssen. Die Gemeinsamkeit bildet dann die Schnittstelle und möglicherweise auch ein wenig gemeinsames Verhalten.

Am Beispiel von grafischen Figuren lassen sich das Ganze gut verdeutlichen. Stellen wir uns eine Applikation vor, die Figuren zeichnen soll. Diese werden jeweils als eigenständige Klassen modelliert, die als Basisfunktionalität Methoden zum Zeichnen anbieten. Weil aber keine Konvention für die dazu zu verwendenden Methodennamen existierte und verschiedene Entwickler die Klassen realisiert haben, findet man leicht unterschiedliche Namen, etwa neben `draw()` auch die Varianten `drawLine()`, `drawRect()` usw. Das ist für einige Figurenklassen in Abbildung 3-13 gezeigt.

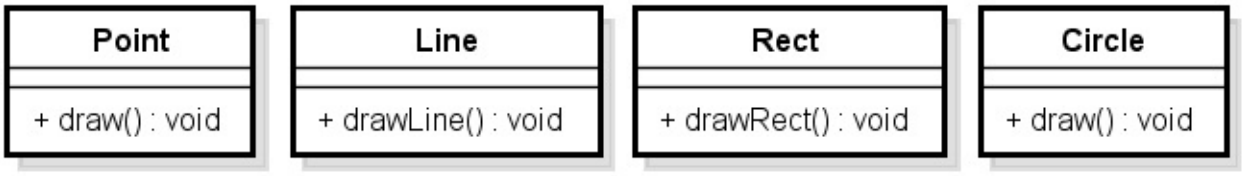


Abbildung 3-13 *Klassen für grafische Figuren (ohne Basisklasse)*

Durch die unterschiedlichen Signaturen sowie keine gemeinsame Basisklasse wird die Handhabung der Figuren für Klienten umständlich und nicht intuitiv: Die Methodennamen unterscheiden sich von Klasse zu Klasse. Um diesen Missstand zu beheben, führen wir eine Basisklasse `Figure` mit einer `draw()`-Methode ein und

vereinheitlichen in den Subklassen die Methodennamen, was für Konsistenz sorgt. Außerdem lässt sich die von Subklassen gemeinsam genutzte Funktionalität in der Basisklasse zentral definieren.

Einsatzgebiet abstrakter Basisklassen

Bei der Implementierung der Basisklasse `Figure` stoßen wir jedoch auf ein Problem: Wie sollen wir die Methode `draw()` implementieren? Was soll diese zeichnen? Diese Fragen stellen sich, weil die Klasse das abstrakte Konzept einer Figur modelliert, aber keine konkrete Ausprägung. Ohne Kenntnis abstrakter Basisklassen und abstrakter Methoden würde man, wohl die `draw()`-Methode mit leerem Methodenrumpf implementieren, weil die Basisklasse `Figure` nichts Sinnvolles zeichnen kann. Dies ist aber wenig hilfreich,[13] da es natürlicher ist, *eben keine* Implementierung anzubieten, wenn eine Methode nicht sinnvoll realisiert werden kann, wie hier die `draw()`-Methode.

Genau für solche Fälle eignen sich abstrakte Basisklassen. Diese erlauben es, Methodensignaturen vorzugeben, die von Subklassen zu implementieren sind. Das geschieht mithilfe abstrakter Methoden, die durch das Schlüsselwort `abstract` gekennzeichnet sind, keine Implementierung besitzen und somit dann durch Subklassen realisiert werden müssen. Im Gegensatz zu den Methoden in Interfaces können abstrakte Methoden neben der Sichtbarkeit `public` auch die Sichtbarkeit `protected` besitzen. In letzterem Fall blähen sie die öffentliche Schnittstelle nicht auf und können eine Art internes Interface für Subklassen vorgeben.

Im Beispiel wird die Klasse `Figure` zu einer abstrakten Klasse und in `BaseFigure` oder alternativ `AbstractGraphicsFigure` umbenannt.[14] Die Realisierung des Zeichnens erfolgt dann in den Subklassen, die genau wissen, wie die durch sie repräsentierten Figuren gezeichnet werden. Abbildung 3-14 zeigt das Klassendiagramm.

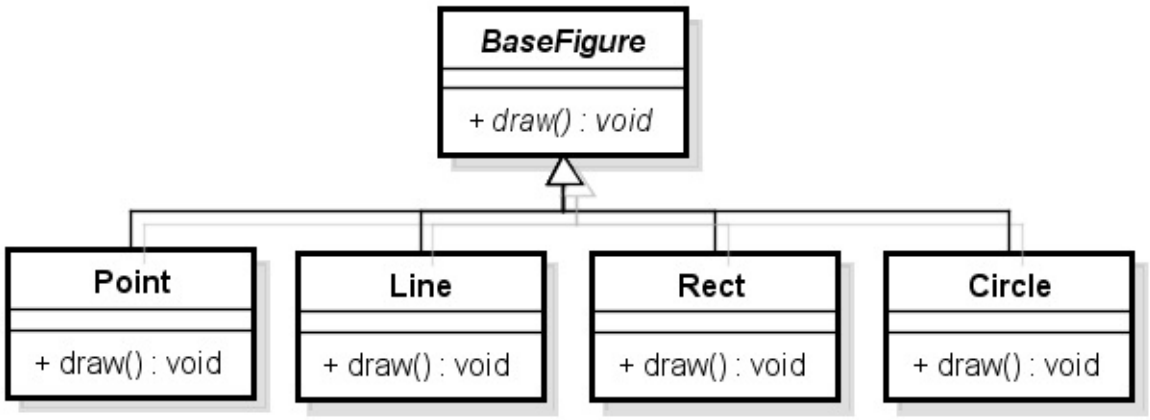


Abbildung 3-14 *Grafische Figuren mit abstrakter Basisklasse*

Eben haben wir gesehen, dass im Rahmen einer Generalisierung derart allgemeine Klassen entstehen, dass dort nicht jede Methode implementiert werden kann. In diesem Fall spricht man von ***abstrakten Basisklassen***. Davon abgeleitete Klassen

können dann die abstrakten Methoden passend realisieren. Ist eine Klasse vollständig implementiert, so spricht man von einer ***konkreten Klasse*** bzw. einer ***konkreten Realisierung***.

Es gibt jedoch den Spezialfall einer abstrakten Klasse, die keine abstrakten Methoden besitzt. Eine solche Klasse wird explizit über das Schlüsselwort `abstract` speziell gekennzeichnet, sodass die Instanziierung dieser Basisklasse verhindert wird.

Bewertung

Abstrakte Basisklassen …

+ helfen bei einer konsistenten Implementierung von Funktionalitäten. Subklassen bringen nur an speziellen, dafür vorgesehenen Stellen ihre Realisierungen ein.
+ verringern Sourcecode-Duplikation und dadurch bedingten Wartungsaufwand.
+ fördern die Erweiterbarkeit, weil beim Erzeugen neuer Subklassen in der Regel nur wenige Anpassungen erfolgen müssen.
- haben den Nachteil, dass sie dazu verleiten, Vererbung unüberlegt einzusetzen und dadurch eine tiefe Klassenhierarchie zu erzeugen. Oftmals ist der Einsatz von Vererbung jedoch nicht der richtige Designweg: Vererbung sollte nur dann eingesetzt werden, wenn eine »is-a«-Beziehung vorliegt (vgl. Abschnitt 3.3.2).

3.2.3 Interfaces und abstrakte Basisklassen

Die Kombination von Interfaces und abstrakten Basisklassen bietet sich an, um die Vorteile beider Techniken zu vereinen, etwa wenn einige Klassen einer Klassenhierarchie bereits über ein Interface abstrahiert werden und diese Klassen gemeinsame Funktionalität besitzen, weshalb sich eine Basisklasse anbietet.

Beispiel

Wir greifen das Beispiel der grafischen Figuren wieder auf und stellen uns dabei vor, dass wir ein Framework für grafische Applikationen erstellen wollen. Daran kann man die nützliche Kombinierbarkeit von Interfaces und abstrakten Basisklassen verdeutlichen. Mithilfe des Interface `IGraphicsFigure` wird die von den Figuren bereitgestellte Funktionalität festgelegt und durch die Klasse `AbstractGraphicsFigure` werden gewisse Funktionalitäten bereits realisiert. Dies ist in Abbildung 3-15 gezeigt.

Gebräuchliche Anwendung in der Praxis

Neben dem eben beschriebenen Anwendungsfall bietet sich eine Kombination aus Interface und abstrakter Basisklasse insbesondere dann an, wenn Klienten aus anderen

Packages auf die Funktionalität des eigenen Packages zugreifen und man eine lose Kopplung erreichen möchte. Dabei verfolgt man das Ziel, den externen Klassen keinen Zugriff auf die Details der genutzten internen Klassen zu gewähren. In diesem Fall sollte man die Funktionalität für Aufrufer über Interfaces bereitstellen und dazu für alle extern genutzten Klassen jeweils ein öffentliches Interface anbieten.[15] Zudem definiert man die meisten Klassen mit der Sichtbarkeit Package-private, um deren Implementierung für Package-externe Klassen unzugänglich zu machen und diese nach außen zu verbergen. So hält man die Klienten unabhängig von den Details realisierender Klassen und erzielt eine lose Kopplung zwischen den Klassen in unterschiedlichen Packages.

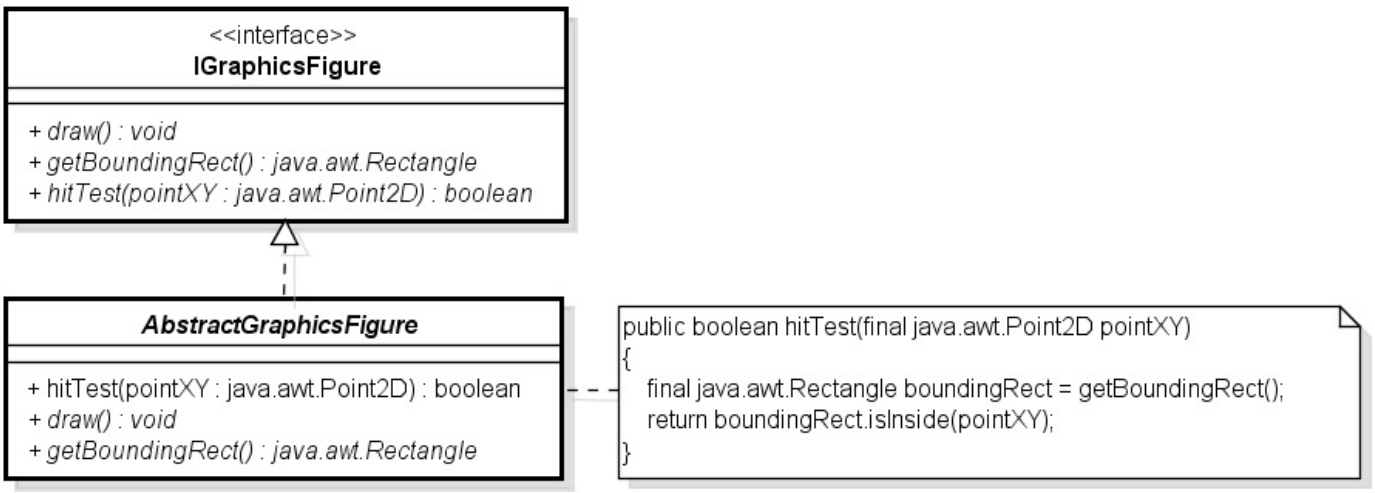


Abbildung 3-15 *Einfache Kombination von Interface und abstrakter Basisklasse*

Nehmen wir an, dass innerhalb des Packages mehrere Varianten der Realisierung eines Interface bereitgestellt werden sollen. Wenn diese Implementierungen alle gemeinsame Funktionalität besitzen, dann empfiehlt es sich, zusätzlich zum Interface eine abstrakte Basisklasse einzusetzen – natürlich ist auch möglich, dass es einzelne Realisierungen des Interface geben kann, die sich nicht auf die abstrakte Basisklasse stützen. Diese Interna bleiben für Klienten transparent, da diese lediglich über das Interface kommunizieren. Abbildung 3-16 zeigt diesen Fall für ein Interface `IService`, eine abstrakte Basisklasse `AbstractBaseService` sowie zwei konkrete Realisierungen.

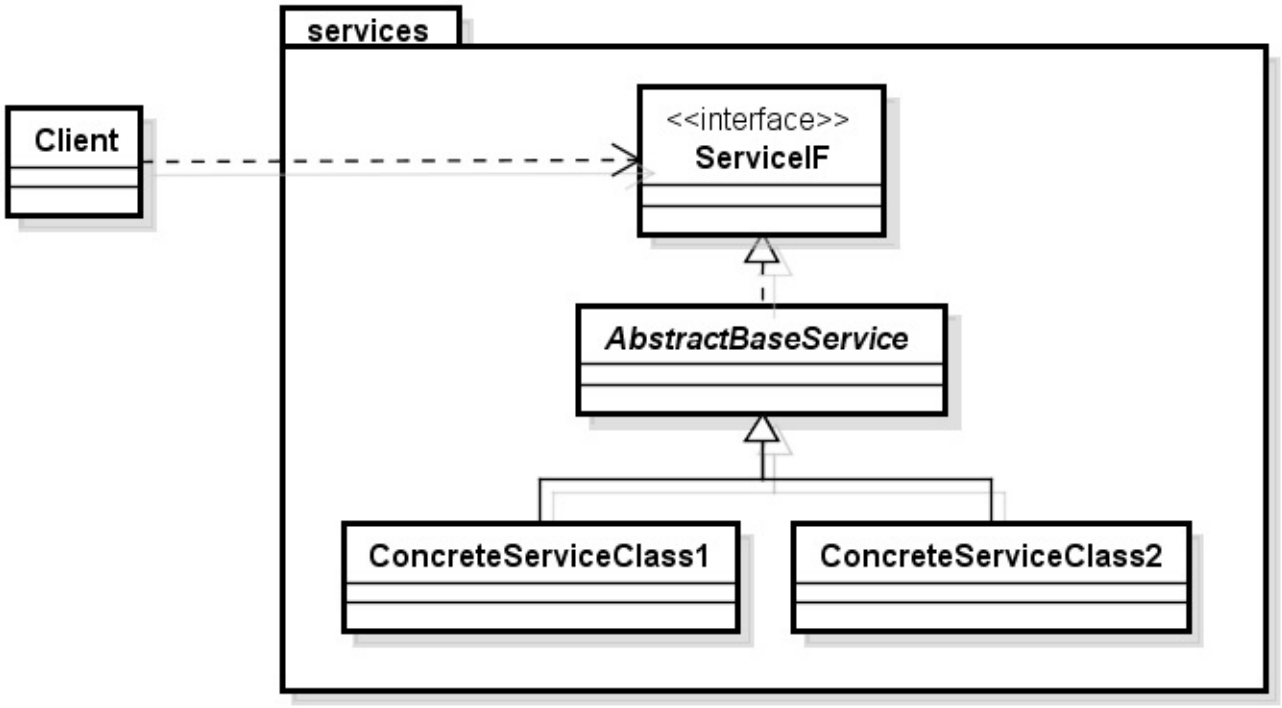


Abbildung 3-16 *Komplexere Kombination von Interface und abstrakter Basisklasse*

Bewertung

Eine Kombination von Interfaces und abstrakter Basisklasse …

+ ermöglicht sowohl eine Kapselung der Funktionalität über Interfaces sowie eine sinnvolle Vorgabe von Basisfunktionalität durch eine abstrakte Basisklasse.
+ ermöglicht, dass Subklassen nur noch an speziellen, gewünschten Stellen ihre Erweiterungen einbringen müssen.
+ erlaubt es oftmals, die konkreten Implementierungen austauschen zu können.
+ ist insbesondere für Frameworks sinnvoll und empfehlenswert. Ein schönes Beispiel ist das Collections-Framework, wo dieses Schema sauber umgesetzt wird.

○ erfordert nur einen geringen Mehraufwand in der Implementierung.

○ erhöht allerdings (geringfügig) die Komplexität. Ein Einsatz sollte demzufolge nicht ohne Grund und insbesondere eher selten innerhalb eines Packages erfolgen, da die Nachvollziehbarkeit durch die Indirektionen und Abstraktionen leiden kann.

3.3 Wissenswertes zu Vererbung

Wir haben nun schon einiges über den OO-Entwurf und mögliche Fallstricke gelernt. Bisher haben wir vor allem die positiven Seiten von Vererbung betrachtet. Dass diese auch Nachteile besitzt, habe ich bereits angedeutet. Es erfordert einige Erfahrung, diese Probleme zu erkennen, zu verstehen und zu beheben.

Beim Erweitern einer bestehenden Klassenhierarchie können Probleme auftreten, wenn man dazu unüberlegt Vererbung einsetzt. Dies diskutiert Abschnitt 3.3.1. Werden Eigenschaften oder verschiedene feingranulare Funktionalitäten in Form einzelner Subklassen definiert, kann eine Modellierung über Vererbung problematisch sein und besser mithilfe von Delegation geregelt werden. Das stellt Abschnitt 3.3.2 vor.

3.3.1 Probleme durch Vererbung

In diesem Abschnitt möchte ich anhand eines Beispiels darstellen, welche Probleme beim Erweitern einer bestehenden Klassenhierarchie auftreten können.[16]

Nehmen wir an, wir hätten bereits eine erste Programmversion einer Simulation einer biologischen Umwelt mit verschiedenen Lebensräumen als Klassen implementiert. Dazu sei eine Klasse `Simulation` definiert. Eine weitere Klasse `Environment` beschreibt die Lebensräume für verschiedene zu simulierende Tierarten. Diese werden mithilfe der abstrakten Basisklasse `Animal` realisiert. Als Tierarten sind dann beispielsweise Füchse, Hasen u. v. m. als konkrete Subklassen (`Fox`, `Rabbit` usw.) modelliert.

Da diese Tiere als Gemeinsamkeit alle eine Größe, ein Gewicht usw. besitzen, sind diese Eigenschaften durch Attribute in der Basisklasse `Animal` beschrieben. Des Weiteren bewegen sich unsere Tierarten durch Laufen fort. Dies wird daher durch eine Methode `walk()` realisiert. Abbildung 3-17 zeigt das zugehörige Klassendiagramm.

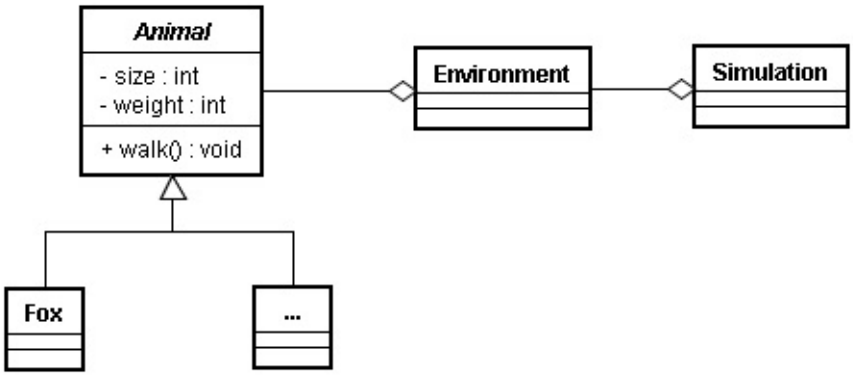


Abbildung 3-17 *Die Klasse* `Animal` *und ihre Vererbungshierarchie*

Zur Simulation einer Umwelt werden diverse Spezialisierungen der Klasse `Animal` erzeugt und danach wiederholt deren Methoden, etwa `walk()`, aufgerufen. Wie so häufig in der Praxis anzutreffen, kommen beim Betrachten und Testen der ersten Version einer Software neue Ideen und dadurch Änderungs- oder Erweiterungswünsche auf. Zeigen wir das Programm einem Ornithologen, so fällt diesem auf, dass unsere Simulation ohne Vögel unvollständig ist. Ein Meeresbiologe könnte etwa einwenden, dass Fische unberücksichtigt sind. Beide Erweiterungswünsche führen in der ursprünglichen Modellierung zu folgenden Problemen:

1. Vögel können zwar in der Regel laufen, aber ihre natürliche Fortbewegungsart ist meistens das Fliegen – Ausnahmen sind z. B. die Vogelarten Strauß und Pinguin.
2. Für Fische ist die Methode `walk()` eine Fehlmodellierung. Wenn man in dieser Methode eine schwimmende Fortbewegung realisieren würde, wäre das Design unverständlich. Nachfolgende Bearbeiter der Klasse werden darunter leiden und Sie möglicherweise dafür verfluchen.

Umgang mit Erweiterungen

Sollen nun die genannten Erweiterungen umgesetzt werden, so müssen wir dabei die zuvor genannten Kritikpunkte beachten. Damit das Klassendesign sauber und logisch strukturiert bleibt, wird die Basisklasse `Animal` wie folgt geändert: Die Basisklasse erhält eine allgemeinere Methode `move()` zur Modellierung der Fortbewegung. Diese nimmt keine Festlegung der tatsächlichen Bewegungsart vor.[17] Außerdem werden nun weitere Basisklassen einer zweiten Gliederungsebene erstellt, etwa die Klassen `Fish`, `Mammal` und `Bird` zur Modellierung der Tierkategorien Fisch, Säugetier und Vogel. In diesen Basisklassen können wir jeweils die Methoden `swim()`, `walk()` und `fly()` zur Fortbewegung gemäß der Tierkategorie definieren, ohne diese Methoden bereits auf dieser Abstraktionsebene implementieren zu wollen. Das liegt daran, dass die konkrete Fortbewegung von einer konkreten Tierart abhängt. Für Klienten soll es nicht von Interesse sein, dass diese Zwischenschicht von Basisklassen in das Design eingezogen wurde. Vielmehr ist es das Ziel, eine Abstraktion von den konkreten Klassen mithilfe der Methode `move()` zu gewährleisten. Basierend auf dieser Argumentation müssen die Methoden `swim()`, `walk()` und `fly()` in ihrer Sichtbarkeit eingeschränkt werden, aber auch ein Überschreiben in Subklassen erlauben, was die Sichtbarkeit `protected` ermöglicht. Das Klassendiagramm ist in Abbildung 3-18 dargestellt.

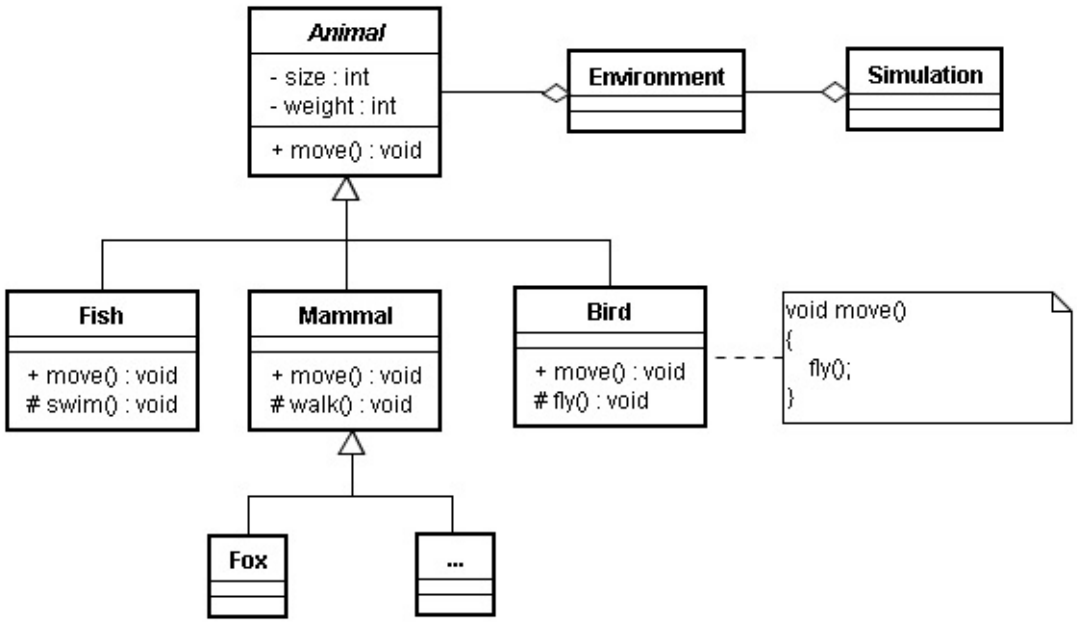


Abbildung 3-18 *Variante 2 der Klasse* `Animal` *und ihrer Vererbungshierarchie*

Obwohl diese Modellierung bereits deutlich besser als die ursprüngliche Umsetzung ist, gibt es immer noch einige Schwachstellen:

1. Wie zuvor angemerkt, können Vögel häufig nicht nur fliegen, sondern auch laufen bzw. hüpfen. Außerdem können einige Vogelarten überhaupt nicht fliegen. Klienten nutzen derzeit als Abstraktion die Basisklasse `Bird` und deren Methode `move()`, die per Standard das Fliegen modelliert. Wollte man damit auch Laufvögel, wie Sträuße mit der Klasse `Ostrich`, adäquat modellieren, so hätte man ein Problem. Eine mögliche Lösung bestünde darin, die Vererbungshierarchie in die zwei Basisklassen `FlyingBird` und `NonFlyingBird` aufzuspalten. Damit sorgt man zwar für mehr Unterscheidbarkeit, aber es lässt sich immer noch nicht abbilden, dass die meistens Vogelarten sowohl (bevorzugt) fliegen als auch laufen oder hüpfen können. Zudem entsteht dadurch bereits eine recht tiefe Vererbungshierarchie. Eine solche erschwert häufig Wartungsarbeiten und Änderungen. Und die Realität ist noch komplexer: Wie integrieren wir die vorwiegend schwimmenden und tauchenden Pinguine und die korrespondierende Klasse `Penguin`?
2. Auch die Säugetierklasse `Mammal` wirft ein Modellierungsproblem auf: Solange man damit ausschließlich Landsäugetiere modelliert, ist alles in Ordnung. Was ist aber mit Walen? Schließlich kann ein Wal nicht laufen. Müssen wir hier wiederum ein `WalkingMammal` und ein `SwimmingMammal` einführen? Je nach Anwendungsfall ist das wohl notwendig. Und es würde noch komplexer, wenn wir Fledermäuse als fliegende Säugetiergattung modellieren müssten.

Anhand dieser Beschreibung wird klar, dass die objektorientierte Modellierung mit Vererbung nicht immer leicht ist. Im Besonderen gilt, dass in Basisklassen nur die Funktionalitäten modelliert werden sollten, die allen möglichen Subklassen gemein sind.

Wissenswertes zu Vererbung und Wiederverwendbarkeit

An diesem Beispiel erkennen wir, dass Vererbung nicht immer eine einfache Wiederverwendung ermöglicht, wie dies leider viel zu häufig in der Einsteigerliteratur zum objektorientierten Programmieren dargestellt wird. Vielmehr wird deutlich, dass Vererbung gut überlegt eingesetzt werden sollte, da sie ansonsten Probleme bereiten kann: Schnell entsteht eine schlechte Klassenstruktur, die einem die Möglichkeiten raubt, die Vererbung sonst bietet. Betrachten wir das Ganze im Folgenden detaillierter.

Der korrekte Zuschnitt von Basisklassen ist stark von dem zu modellierenden System abhängig. Für unsere Simulation einer biologischen Umwelt gilt z. B.: Spielen Vögel oder Fische eine Rolle? Müssen Sonderfälle wie Laufvögel berücksichtigt werden? Aufgrund der unterschiedlichen Anforderungen ist eine Wiederverwendbarkeit durch Vererbung zwischen verschiedenen Applikationen daher nur recht selten möglich. Innerhalb einer Applikation kann durch Vererbung

allerdings eine Duplizierung von Funktionalität vermieden werden. Oftmals ist der Mehrwert der Wiederverwendung jedoch weniger ausgeprägt ist, als er gerne dargestellt wird.

Des Weiteren ziehen Änderungen in Basisklassen teilweise Anpassungen in deren Subklassen nach sich. Das hat weitreichende Konsequenzen: Wurden Basisklassen eingangs ungünstig gewählt, so können sich Korrekturen daran in vielen Subklassen auswirken. Das lässt für das OO-Design wiederum folgende Schlüsse zu:

- ***Basisklassen müssen mit besonderer Sorgfalt entworfen werden.*** Das ist dadurch begründet, dass Änderungen in Basisklassen ansonsten diverse Folgeänderungen in deren Subklassen verursachen. Änderungen in den Subklassen besitzen dagegen meist recht lokale und gut überschaubare Auswirkungen.
- ***Umfangreiche (tiefe oder breite) Ableitungshierarchien sind ein Indikator für eine gute Wiederverwendung.*** Einher geht aber – wie gerade besprochen – auch eine erhöhte Anfälligkeit für Folgeänderungen in Subklassen bei Änderungen in deren Basisklassen. Deswegen ist eine komplexe Ableitungshierarchie immer auch kritisch zu betrachten und zu hinterfragen. Im Beispiel erkennt man, dass trotz der gemeinsamen Basisklassen tatsächlich recht wenig Funktionalität wiederverwendet werden konnte. Stattdessen muss durch die Betrachtung der ganzen Sonderfälle fast alle Funktionalität in den Subklassen realisiert werden.

Hinweis: Vererbung, Objektmodellierung und Wiederverwendbarkeit

Ein Modell stellt eine Vereinfachung (Abstraktion) eines realen Gegenstands oder Systems dar und definiert die für den jeweiligen Anwendungsfall benötigten Aspekte. Daraus entsteht dann der Entwurf einer Klasse mit ihren Attributen und Methoden. Es kommt leicht zu dem Missverständnis und der unerfüllbaren Hoffnung, die für einen Anwendungsfall modellierten (Basis-)Klassen für alle möglichen Applikationen wiederverwenden zu können. Das ist aber nur eher selten möglich, da die jeweiligen Applikationen in der Regel andere Verhaltensweisen und Attribute der modellierten Objekte benötigen.

Betrachten wir das konkret für die Modellierung einer Person als Klasse. Für einen Onlineshop sind Name, Anschrift und eventuell Bonität wichtige Kriterien. In einer Patientenverwaltung eines Hausarztes werden neben Name und Anschrift eher Patientendaten, wie Größe, Gewicht, Vorerkrankungen usw., als wichtige Aspekte der Modellierung berücksichtigt werden. Für eine Klasse `Person` gelten in den beiden genannten Systemen also deutlich unterschiedliche Anforderungen. Diese mit nur einer Klasse zu beschreiben, wäre wahrscheinlich nicht sinnvoll. Allerdings könnte man sich eine Modellierung vorstellen, die eine gemeinsame Basisklasse `Person` aus den Anforderungen extrahiert. Diese bekommt die für beide Anwendungen gebräuchlichen Attribute wie Name und Anschrift. Für den Onlineshop könnte man eine Subklasse `Kunde` von der Basisklasse `Person` ableiten und um weitere Attribute, etwa die oben geforderte Bonität, ergänzen. Ebenso

würde man bei der Patientenverwaltung vorgehen. Dort könnte eine Subklasse `Patient` als Spezialisierung der Basisklasse `Person` definiert und um Attribute zu dem Patienten sowie seinen Krankendaten ergänzt werden.

Abstrakte vs. konkrete Basisklassen

Das Beispiel zur Modellierung der Tierkategorien und die durch unsere Erweiterungen entstandenen Basisklassen werfen die folgende Fragestellung auf: Wann sind Basisklassen abstrakt zu definieren und wann nicht? Meistens sollten Basisklassen abstrakt definiert werden. In diesem Beispiel ist das für die Basisklasse `Animal` geschehen. Vermutlich sollten auch die Basisklassen `Fish`, `Mammal` und `Bird` abstrakt definiert werden. Das hängt davon ab, ob unsere Simulation tatsächlich Objekte dieser Basistypen erzeugen und in der Simulation verwenden können soll. Als Faustregel gilt, dass Basisklassen so lange abstrakt definiert werden sollten, wie durch sie lediglich allgemeine Konzepte modelliert werden und noch keine konkreten Implementierungen für alle Methoden vorgegeben werden können.

Da die Simulation mit einzelnen Individuen spezieller Subklassen der Tierkategorien arbeiten soll, führen wir einen letzten Korrekturschritt an unserem Modell durch: Wir definieren die Basisklassen `Fish`, `Mammal` und `Bird` abstrakt (Abbildung 3-19).

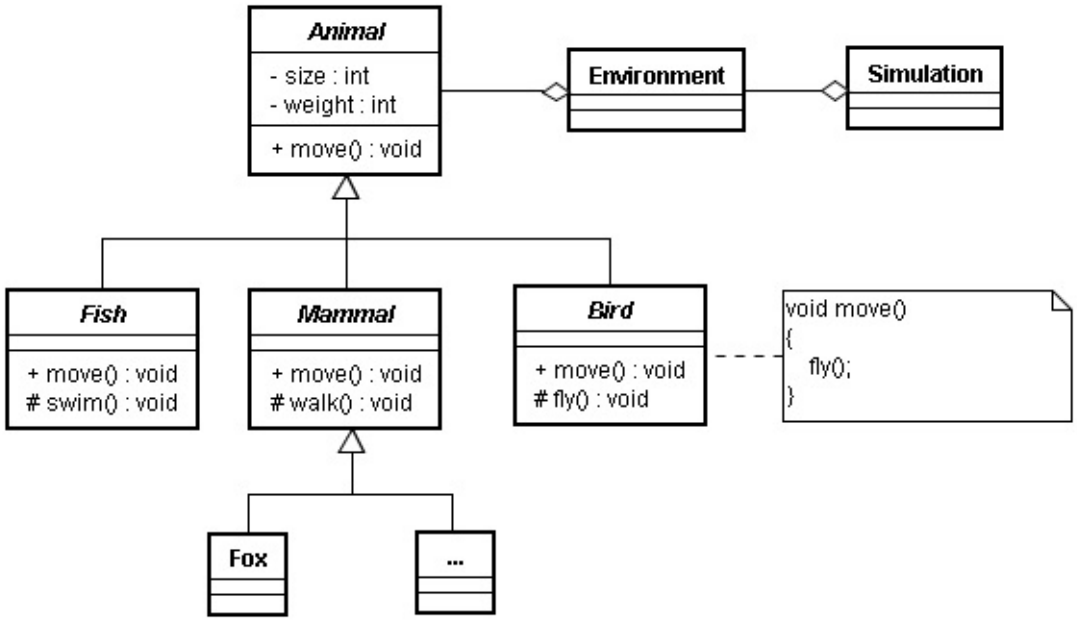


Abbildung 3-19 *Variante 3 der Klasse* `Animal` *und ihrer Vererbungshierarchie*

Dieses Beispiel der Modellierung von Tieren für eine Simulationssoftware zeigt anschaulich, dass Basisklassen in der Regel abstrakt definiert werden sollten. Man verhindert dadurch die Instanziierung von Klassen, die unvollständige Konzepte darstellen. Oder anders formuliert: ***Konkrete Basisklassen sollten sehr aufmerksam betrachtet und nur in gut begründeten Ausnahmefällen verwendet werden.***

Fazit

Nutzt man Vererbung als Instrument der Wiederverwendung ohne viel Nachdenken, so kann dies zu unübersichtlichen Designs führen. Abschließend bleibt die Erkenntnis, dass die früher vielfach als Allheilmittel beim OO-Entwurf angesehene Technik der Vererbung durchaus nicht unproblematisch ist. Tatsächlich ist sie oftmals weit weniger praktikabel zur Wiederverwendung als häufig angenommen bzw. propagiert.

Wie jedes Werkzeug kann man auch Vererbung ungeschickt einsetzen oder Unsinn damit anstellen. Schauen wir im folgenden Abschnitt auf weitere Problemfälle tiefer oder breiter Vererbungshierarchien. Ganz allgemein kann man hier aber bereits Folgendes feststellen: ***Sie sollten einen prüfenden Blick auf Ihre Klassenstruktur werfen, sobald der Ableitungsbaum drei oder mehr Hierachiestufen erreicht oder einzelne Verzweigungen mehr als etwa 5-10 Klassen besitzen.*** Im Zusammenhang mit Frameworks (etwa Swing, Spring) sind solche Werte allerdings durchaus normal. Diese Richtwerte werde ich in Abschnitt 19.4.1 bei der Beschreibung verschiedener Metriken aufgreifen.

3.3.2 Delegation statt Vererbung

Betrachten wir im Folgenden zunächst, welche Folgen es hat, wenn man für jede neu zu modellierende Eigenschaft eine neue Klasse definiert, die eine andere Klasse erweitert. Anschließend sehen wir uns an, was passiert, wenn man auf diese Art verschiedene Verhaltensweisen eines Objekts, sogenannte ***Rollen***, modelliert. Eine Person könnte etwa die Rolle Student, aber auch (gleichzeitig) die Rolle Hotelgast spielen.

Klassenexplosion durch Modellierung von Eigenschaften

Am Beispiel einer Klasse `Window` analysieren wir die Modellierung von Eigenschaften durch Vererbung. Aufgabe ist es, diese Klasse um zusätzliche Funktionalität (Titelleiste, Rahmen usw.) zu erweitern. Eine mögliche Realisierung erstellt für jede Eigenschaft eine Subklasse als Spezialisierung. Dies zeigt Abbildung 3-20.

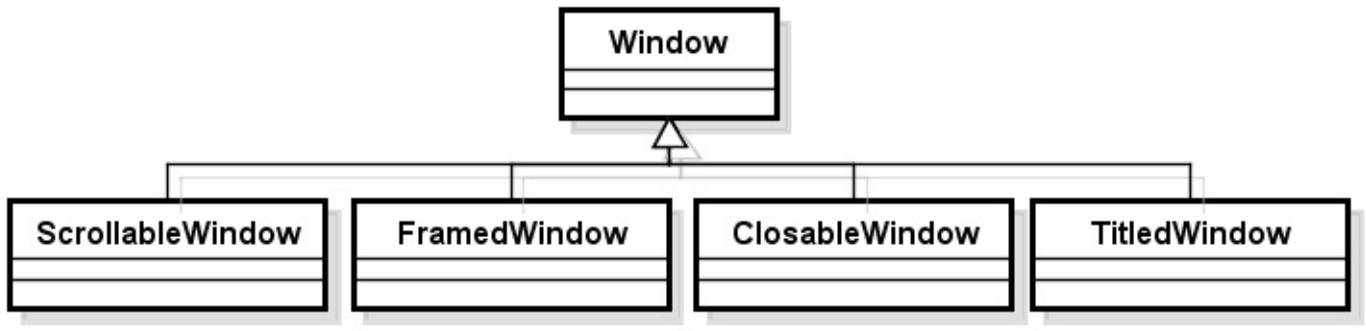


Abbildung 3-20 *Modellierung von Eigenschaften durch Vererbung*

Diese flache und breite Vererbungshierarchie wirkt zunächst nur unelegant.

Problematisch wird dies in dem Moment, in dem man verschiedene Eigenschaften kombinieren möchte. Dadurch werden viele weitere Ableitungen erforderlich und es kommt zu einer kombinatorischen Explosion der Klassen, die vielfach mit einer missverständlichen Namensgebung einhergeht. In Abbildung 3-21 wird dies verdeutlicht.

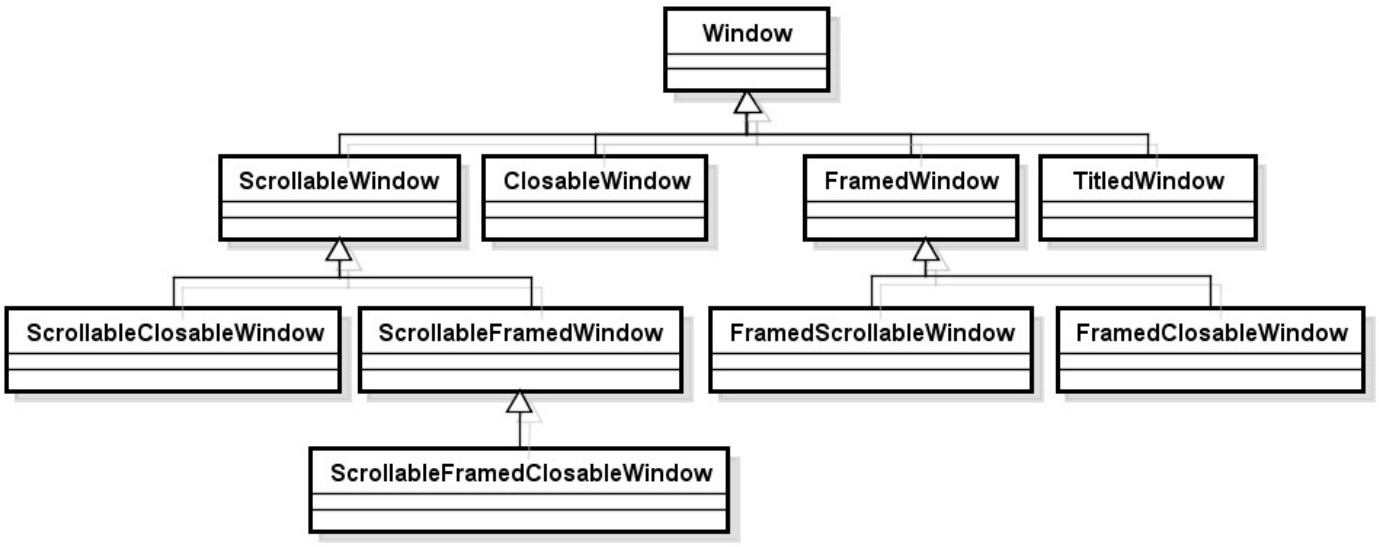


Abbildung 3-21 *Klassenexplosion durch Modellierung von Eigenschaften*

Es stellt sich umgehend die Frage, ob es tatsächlich Unterschiede bei verschiedenen Ableitungsreihenfolgen gibt: Ist ein `ScrollableFramedWindow` semantisch verschieden zu einem `FramedScrollableWindow`? Und wenn ja, wo liegen die Unterschiede? Spätestens dann, wenn man sich diese Frage stellt, erkennt man, dass man mit dem Einsatz von Vererbung hier an die Grenzen einer derartigen Modellierung stößt.

Eine mögliche Lösungsidee ist, statt Vererbung boolesche Variablen zur Auswahl gewünschter Funktionalitäten zu verwenden. Funktionalitäten aus den Spezialisierungen werden dazu in die Basisklasse verlagert. Dort erfolgt dann, abhängig von dem jeweiligen booleschen Wert, ein Aufruf an die zu aktivierende Funktionalität. Diese Lösung stößt, ebenso wie der Einsatz von Vererbung, schnell an Grenzen, insbesondere, wenn viele oder komplexe Eigenschaften modelliert werden sollen. Dann degeneriert die Basisklasse zu einer »One-for-All«-Funktionssammlung von Optionen. Die Komplexität der Abfragen steigt zunehmend, insbesondere wenn nur spezielle Kombinationen von Eigenschaften erlaubt sind.

Eine Abhilfe bietet der Einsatz des DEKORIERER-Musters (vgl. Abschnitt 18.2.3). Damit kann Funktionalität dynamisch, ohne Vererbung hinzugefügt und beliebig miteinander kombiniert werden. Man verhindert dadurch einerseits eine »explodierende« Klassenhierarchie und erreicht andererseits eine gute Trennung von Zuständigkeiten: Jede Dekoriererklasse trägt den jeweiligen Funktionsanteil bei. Die Gesamtfunktionalität entsteht aus der Kombination der verwendeten Dekoriererklassen. Allerdings gibt es auch einen kleinen Haken: Während mit booleschen Flags Funktionalität sogar dynamisch deaktiviert werden kann, ist dies mit Dekorierern nahezu unmöglich.

Dekorierer eignen sich besonders, wenn die modellierten Eigenschaften orthogonal zueinander, d. h. voneinander unabhängig sind. In der Praxis hängen Eigenschaften und Funktionalitäten häufig voneinander ab und schließen sich im Extremfall sogar gegenseitig aus. Es ist dann Aufgabe des Entwicklers, nur erlaubte Kombinationen von Dekorierern zu verwenden. Dies kann man wiederum durch den Einsatz geeigneter Erzeugungsmuster (vgl. Abschnitt 18.1), etwa FABRIKMETHODE und ERBAUER, lösen.

Klassenexplosion durch Modellierung von Rollen

Anhand der Basisklasse `Person` betrachten wir nun, welche Folgen die Modellierung von Rollen durch Vererbung haben kann. Spezialisierungen modellieren dann häufig lediglich zeitweilig ausgeübte Rollen. Dadurch kommt es zu mehreren Problemen. Einerseits müssen alle möglichen Kombinationen von Rollen im Voraus in Form von Klassen realisiert werden. Andererseits bedingt ein Rollenwechsel den Wechsel des Typs und wirft die Frage auf, wie ein Objekt seinen Typ wechseln soll.[18]

Analysieren wir kurz mögliche Probleme durch die Kombination unterschiedlicher Rollen. Bei der Umsetzung durch Vererbung kann es, wie zuvor bei der Modellierung von Eigenschaften, zu einer kombinatorischen Explosion von Klassen kommen. Nehmen wir beispielsweise eine Person, die studiert und in einem Cafe als Kellner arbeitet. Ein studierender Cafe-Kellner, der in einem Hotel übernachten möchte, ist durch Vererbung kaum sinnvoll abzubilden. Man stelle sich vor, was passiert, wenn das Studium beendet oder der Job gewechselt wird. Man möchte sicher nicht noch weitere Klassen einführen, um auch diese neuen Rollen mit allen bisherigen Kombinationen darstellen zu können. Abbildung 3-22 deutet eine kombinatorische Explosion an.

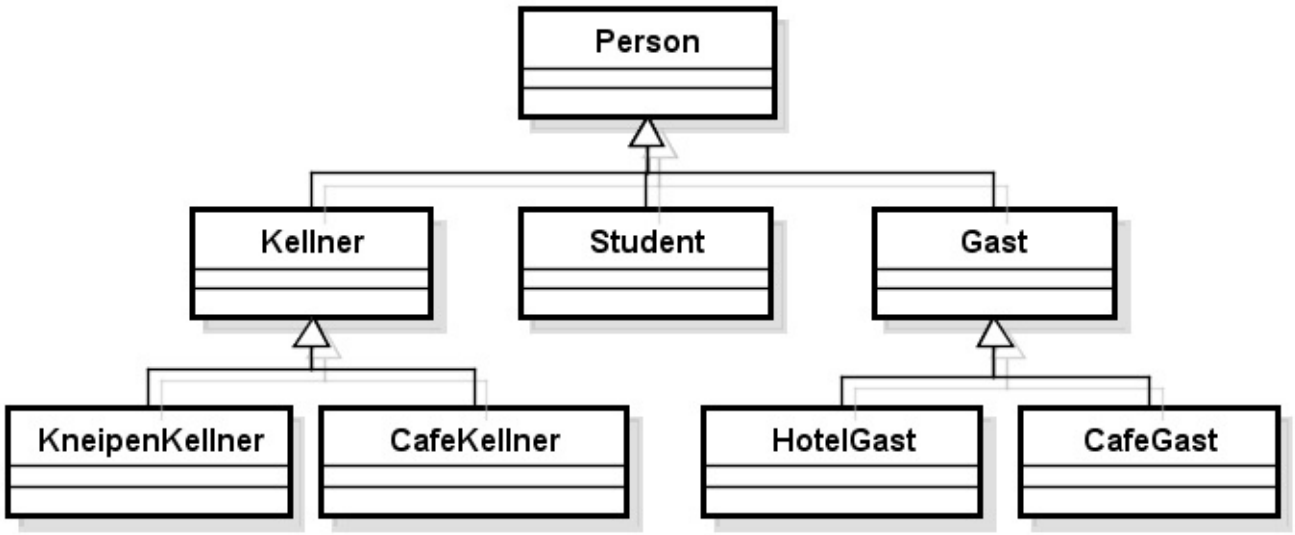


Abbildung 3-22 *Klassenexplosion durch Modellierung von Rollen*

Es ist wahrscheinlich, dass einige Kombinationen von Rollen widersprüchlich sind, etwa die eines zu Fuß gehenden Radfahrers. Im Extremfall passen dann einige Methoden einer Spezialisierung nicht mehr in das Konzept. Es kann kein sinnvolles

Verhalten als Methodenimplementierung definiert werden. In einem solchen Fall kann zwar eine `java.lang.UnsupportedOperationException` ausgelöst werden, allerdings ist das meistens nur eine Notlösung und weist auf ein vorliegendes Designproblem hin und man sollte die Klassenhierarchie überdenken, Klassen und Interfaces anders schneiden oder die »unerfüllbare« Methode in ein separates Interface auslagern.

Da mit Vererbung lediglich statische Eigenschaften ausgedrückt werden können, muss jede einzelne Rolle und jede Kombination als Klasse realisiert werden. Um unterschiedliche Rollen auszuführen, muss ein Objekt als Folge eines Rollenwechsels auch seinen Typ wechseln. Das kann ein Objekt jedoch nicht selbst durchführen. Es wird demnach eine steuernde Einheit benötigt, die die passenden Instanzen erzeugt und verwaltet. Eine weitere Schwierigkeit besteht darin, die Instanz einer neuen Rolle weiterhin mit der ursprünglichen Instanz in Verbindung zu bringen.

Wie man sieht, lauern einige Fallstricke bei der Modellierung von Rollen durch Vererbung. Im Gegensatz zur Realisierung von Eigenschaften ist als Abhilfemaßnahme der Einsatz des DEKORIERER-Musters ungünstig, da zusätzliche Funktionalität in diesem Fall nicht transparent hinzugefügt, sondern explizit angesprochen werden soll. Es bietet sich ein anderes Vorgehen zur Lösung an: Die einzelnen Vererbungsbeziehungen werden jeweils analysiert und können anhand der Regeln aus folgender Aufzählung umgewandelt werden.

Eine Vererbungsbeziehung zwischen Klassen, die …

- sich mit »***can-act-like***« beschreiben lässt, wird durch das Einführen eines speziellen ***Interface*** aufgelöst, das die statische Rolle oder Funktionalität abbildet, die zur Kompilierzeit bekannt ist. Bei Bedarf kann dann per Delegation auf eine Klasse zugegriffen werden, die die Rolle implementiert.
- eine dynamische Eigenschaft, also eine Rolle, modelliert, lässt sich durch Vererbung nur schwierig adäquat ausdrücken. Auch das Implementieren eines Interface ist unpassend. Stattdessen wird die Beziehung umgekehrt und mit ***Delegation*** wird auf die ursprüngliche Basisklasse und deren Funktionalität zugegriffen.

Beide Möglichkeiten werden in Abbildung 3-23 dargestellt. Die Studentenrolle modellieren wir als statische Rolle und verwenden ein Interface `IStudent`. Das Agieren als `Kellner` oder `Gast` wird dynamisch als Rolle abgebildet.

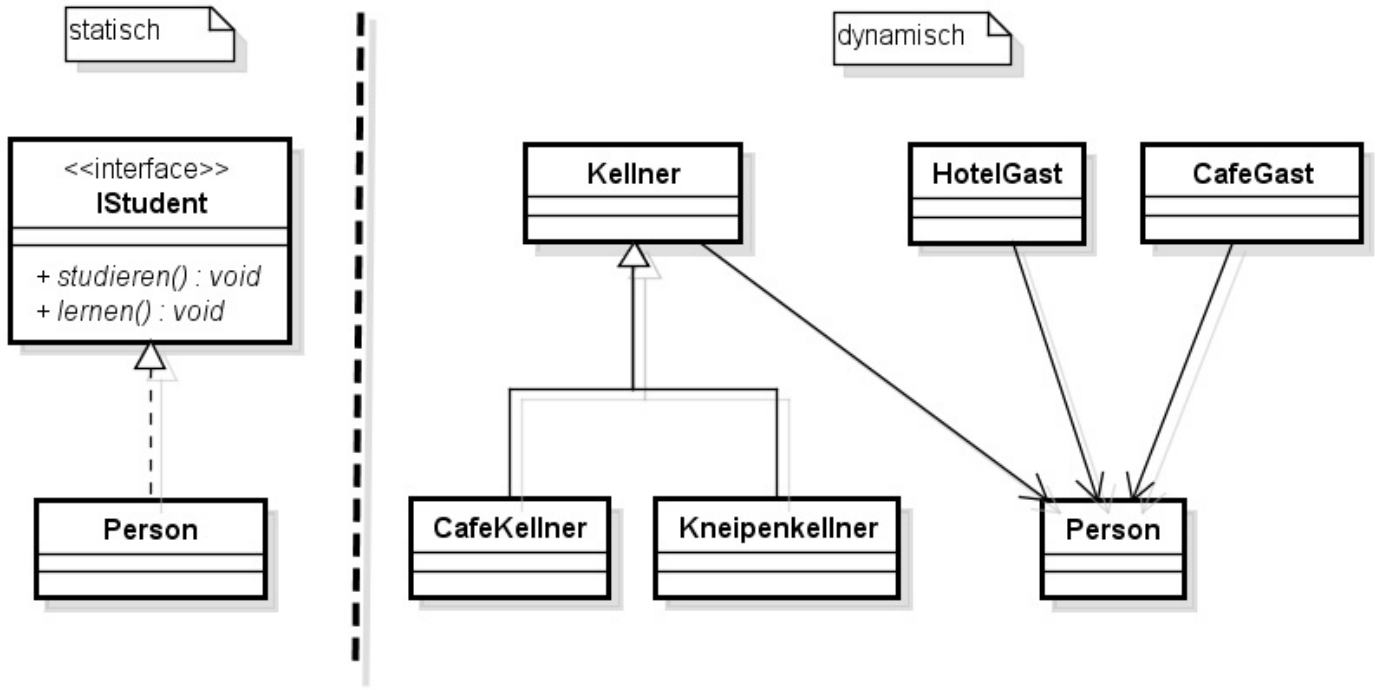


Abbildung 3-23 *Delegation statt Vererbung*

Bewertung

Der Einsatz von Delegation …

+ erlaubt, dass Objekte ihr Verhalten zur Laufzeit ändern. Ableitung bietet nur eine statische Verbindung zwischen Klassen und durch diese mögliche Rollen.
+ verhindert, dass Klassen aus dem modellierten System von rein technischen Utility-Klassen erben. Nutzt man Vererbung statt Delegation, so kann es zu Problemen bei Veränderungen in Basisklassen kommen: Mögliche Änderungen an Utility-Klassen wirken sich dann im modellierten System aus. Das zeugt von schlechtem Design.
- erzeugt für jede delegierte Methode etwas Overhead. Je nach Anzahl der vorhandenen Methoden kann der Sourcecode sich dadurch stark »aufblähen«. Man implementiert im Extremfall fast ausschließlich Delegationen an Methoden anderer Objekte; durch das Objekt selbst wird nahezu kein eigenes Verhalten mehr realisiert.

3.4 Fortgeschrittenere OO-Techniken

Dieser Abschnitt wendet die bereits vorgestellten grundlegenden OO-Techniken an, um nützliche Bausteine für den Entwurf größerer Softwaresysteme zu realisieren. Ein Baustein sind sogenannte READ-ONLY-INTERFACEs, die einen ausschließlich lesenden Zugriff auf Objektdaten erlauben. Das ist Thema von Abschnitt 3.4.1. Der Schutz vor Modifikationen wird mit der Technik der IMMUTABLE-KLASSE logisch fortgeführt und sichert die Unveränderlichkeit des Objektzustands. Darauf geht Abschnitt 3.4.2

ein. Eigenschaften von Klassen lassen sich durch sogenannte MARKER-INTERFACEs beschreiben, die in Abschnitt 3.4.3 erläutert werden.[19] Aufzählungen gruppieren verschiedene semantisch zusammengehörende Werte. Einige Realisierungsmöglichkeiten werden in Abschnitt 3.4.4 diskutiert. Abschließend stellt Abschnitt 3.4.5 sogenannte VALUE OBJECTS (oder auch DATA TRANSFER OBJECTS genannt) vor, die verschiedene Attribute oder Parameter zu neuen Einheiten gruppieren, wodurch sich Parameterlisten übersichtlich halten und Aufrufe vereinfachen lassen.

3.4.1 Read-only-Interface

Das mit der Technik READ-ONLY-INTERFACE verfolgte Ziel ist die Definition eines separaten Interface, das ausschließlich Lesezugriff auf die Attribute einer Klasse anbietet. Im Prinzip ist diese Technik nur ein Spezialfall eines Interface, und zwar genau mit der Einschränkung auf Lesezugriffe, um den Objektzustand vor Modifikationen zu schützen. In meinen Designs nutze ich gern das Postfix `RO`, um die Eigenschaft readonly explizit auszudrücken. Abbildung 3-24 stellt dies für die Klasse `Figure` und das Interface `IFigureRO` dar.

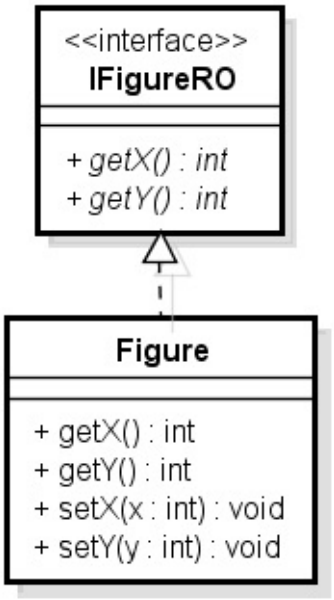


Abbildung 3-24 *Read-only-Interface und zugehörige Implementierung*

Bewertung

Ein Read-only-Interface ….

+ hilft, Änderungen am Objektzustand durch andere Klassen zu verhindern. Dies ist lediglich für Attribute primitiver Datentypen sicher möglich. Bei Rückgabe von Referenzvariablen besteht die Gefahr unerwarteter Änderungen. Darauf geht der folgende Abschnitt detailliert ein.
+ kann immer dann sinnvoll sein, wenn eine Klasse über Package-Grenzen hinweg ausschließlich lesend angesprochen werden soll.
+ sorgt für geringere Komplexität und höhere Robustheit, da Änderungen nur innerhalb der Klasse oder des Packages erfolgen.
○ kann Modifikationen nur dann sicher verhindern, wenn ausschließlich Referenzen auf Read-only-Interfaces, unveränderliche Klassen (vgl. Abschnitt 3.4.2) oder aber (losgelöste, tiefe) Kopien angeboten werden.

Probleme trotz Read-only-Interface durch die Referenzsemantik von Java

In Java können durch `get()`-Methoden sowohl Werte primitiver Typen als auch Referenzen auf Objekte zurückgeliefert werden. Auf diesen können beliebige Methoden[20] – im Speziellen auch solche, die den Zustand verändern – aufgerufen werden. Liefern `get()`-Methoden Referenzen auf Collections zurück, so ist die Situation noch problematischer, da sowohl die Collections als auch deren Elemente verändert werden können.

Wir nutzen ein Beispiel, weil das Ganze noch etwas kompliziert klingt. Der Datencontainer `ClubMembers` verwaltet `Person`-Objekte und bietet ein Read-only-Interface `IClubMembersRO` zum Datenzugriff an (vgl. Klassendiagramm in Abbildung 3-25).

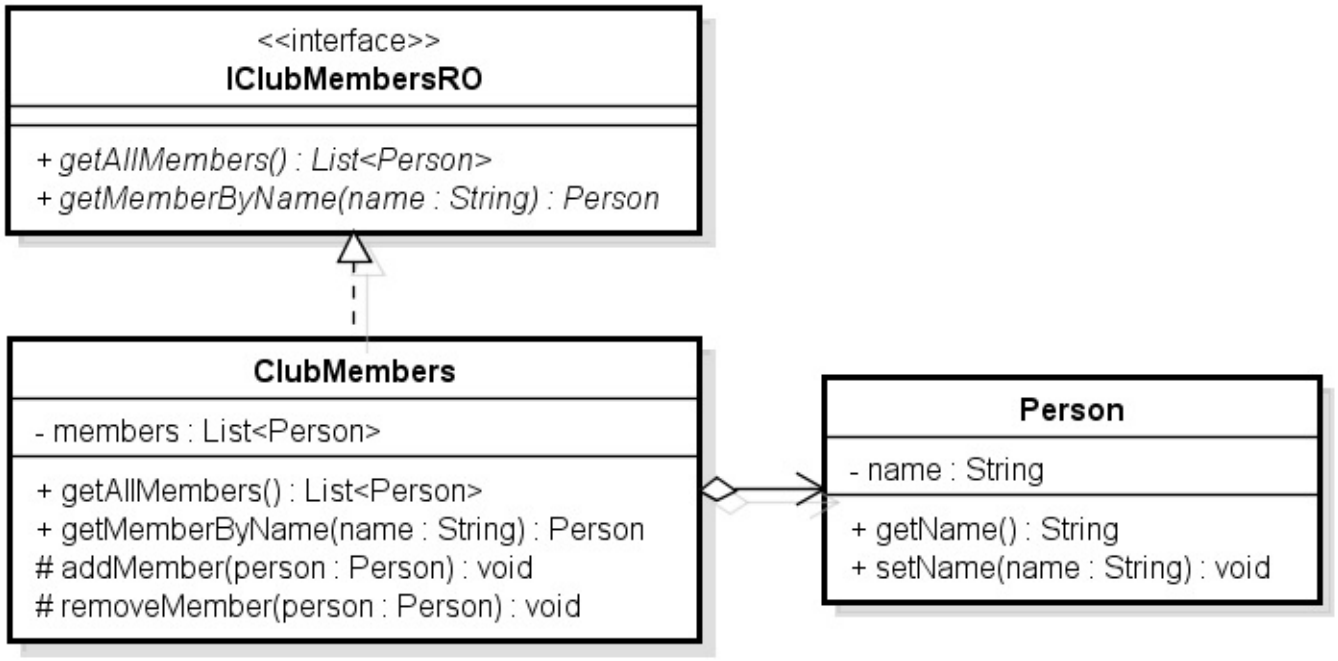


Abbildung 3-25 *Probleme trotz Read-only-Interface*

Hierbei lernen wir mit der Definition `List<Person>` ein bisher nicht vorgestelltes Sprachelement von Java kennen: die sogenannten ***Generics***. Für dieses Beispiel reicht es, zu wissen, dass damit typsichere Containerklassen realisiert werden können: Die Schreibweise `List<Person>` erlaubt in einer so definierten Liste die Speicherung von Referenzen vom Typ `Person` (und auch von Subtypen). Abschnitt 3.7 stellt das Thema Generics ausführlicher vor.

Selbst wenn man – wie hier mit `IClubMembersRO` – ein Read-only-Interface einsetzt, sind Änderungen am Objektzustand der Klasse `ClubMembers` indirekt möglich:

- Die Methode `getMemberByName()` bietet Zugriff auf ein Objekt der Klasse `Person`. Dort können Modifikationen erfolgen.
- Die Methode `getAllMembers()` bietet Zugriff auf die interne Liste.

Probleme in der Zugriffsmethode `getMemberByName()`

Die nur lesende Methode `getMemberByName()` liefert eine Referenz auf ein Objekt der Klasse `Person` zurück. Aufrufer können durch einen Aufruf von `set()`-Methoden somit Änderungen an den Werten der Attribute bewirken. Es könnte etwa durch einen Aufruf von `setName()` zu einem unerwarteten Namenswechsel kommen: Aus Herrn Meyer könnte Herr Müller werden usw. Abbildung 3-26 macht dies deutlich.

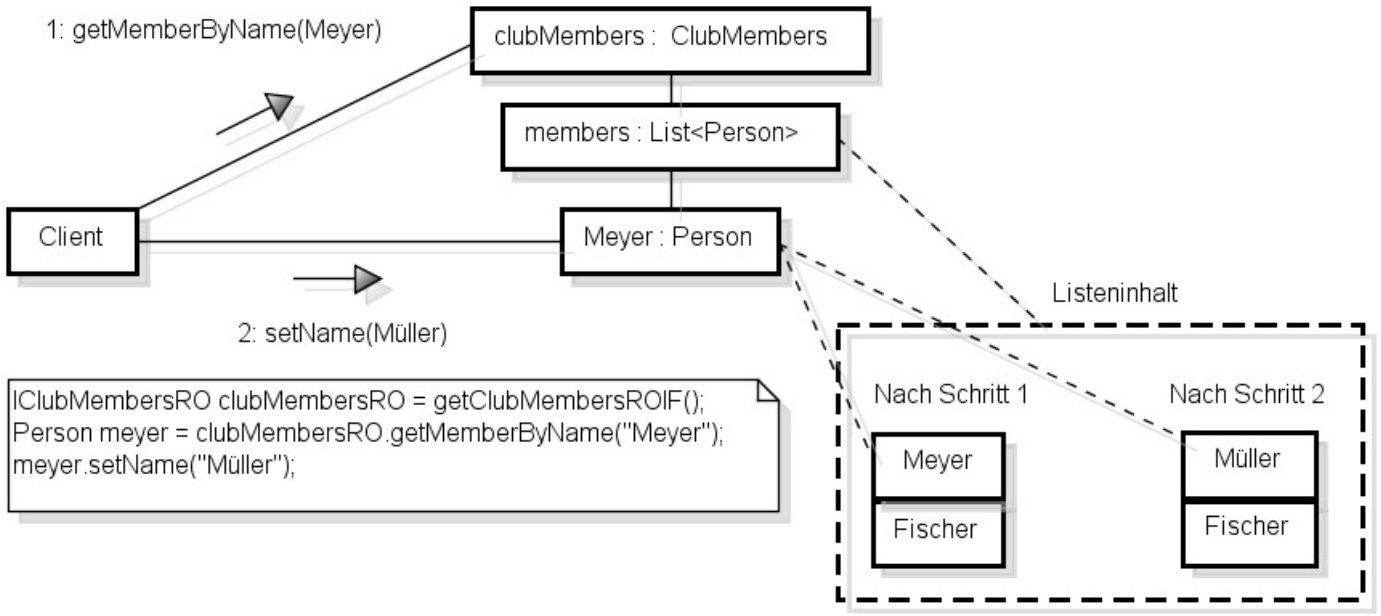


Abbildung 3-26 *Mögliche Probleme nach Aufruf von* `getMemberByName()`

Probleme in der Zugriffsmethode `getAllMembers()`

Die Zugriffsmethode `getAllMembers()` ist problematisch, weil sie Zugriff auf interne Daten der Klasse `ClubMembers` ermöglicht:

```
// Achtung: Problematische Realisierung
public List<Person> getAllMembers()
{
    return members;
}
```

Klienten können mit der zurückgegebenen `List<Person>`-Referenz `members` folglich Änderungen am internen Objektzustand vornehmen und am Interface der Klasse `ClubMembers` vorbei programmieren: Beispielsweise können `Person`-Objekte eingefügt werden, ohne die dafür vorgesehenen Methode `addMember(Person)` aufzurufen. Es wäre mithilfe der zurückgelieferten `List<Person>`-Referenz sogar möglich, die komplette Mitgliederliste durch einen Aufruf der Methode `clear()` zu löschen. Anschließend könnten über den Aufruf von `add(Person)` die Personen »Schulze« und »MisterX« in den Club aufgenommen werden. In Abbildung 3-27 ist dieser Ablauf dargestellt. Durch die Rückgabe von Collection-Referenzen kann der Objektzustand also (unerwartet) beliebig von außen geändert werden, ohne dass das `ClubMembers`-Objekt etwas davon »bemerkt«![21] Weiterhin existieren die zuvor angesprochenen Probleme für herausgereichte Referenzen auf `Person`-Objekte.

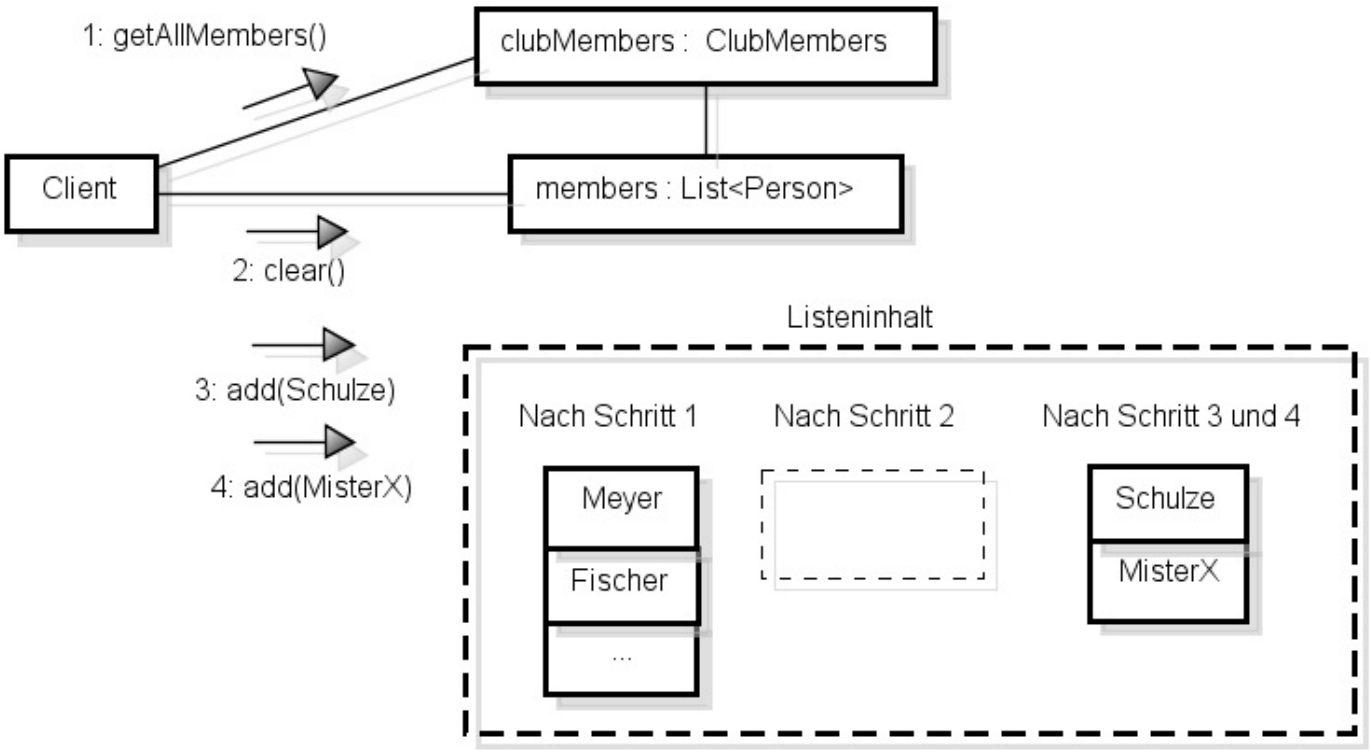


Abbildung 3-27 *Mögliche Probleme nach Aufruf von* `getAllMembers()`

Lösungsmöglichkeiten für die »Schlupflöcher«

Das vorherige Beispiel zeigt, dass der Einsatz eines Read-only-Interface Änderungen am internen Zustand eines Objekts bei Rückgabe von Referenzen nicht sicher verhindern kann. Das widerspricht jedoch dem verfolgten Ziel der Unveränderlichkeit. Was kann man gegen unbeabsichtigte Änderungen unternehmen?

Wenn man Referenzen zurückliefert, empfiehlt es sich, die Technik der Read-only-Interfaces zu nutzen. Schauen wir nun, was dies konkret für Objekte und Collections für das vorgestellte Beispiel bedeutet.

Lösungsmöglichkeiten für Objekte

Zunächst wird für die Klasse `Person` ein Read-only-Interface `IPersonRO` eingeführt. Änderungen können dann unterbunden werden, wenn die Methode `getMemberByName()` dieses Interface zurückliefert. Dadurch erhält ein Aufrufer lediglich eine Read-only-Sicht auf `Person`-Objekte. Abbildung 3-28 zeigt das zugehörige Klassendiagramm.

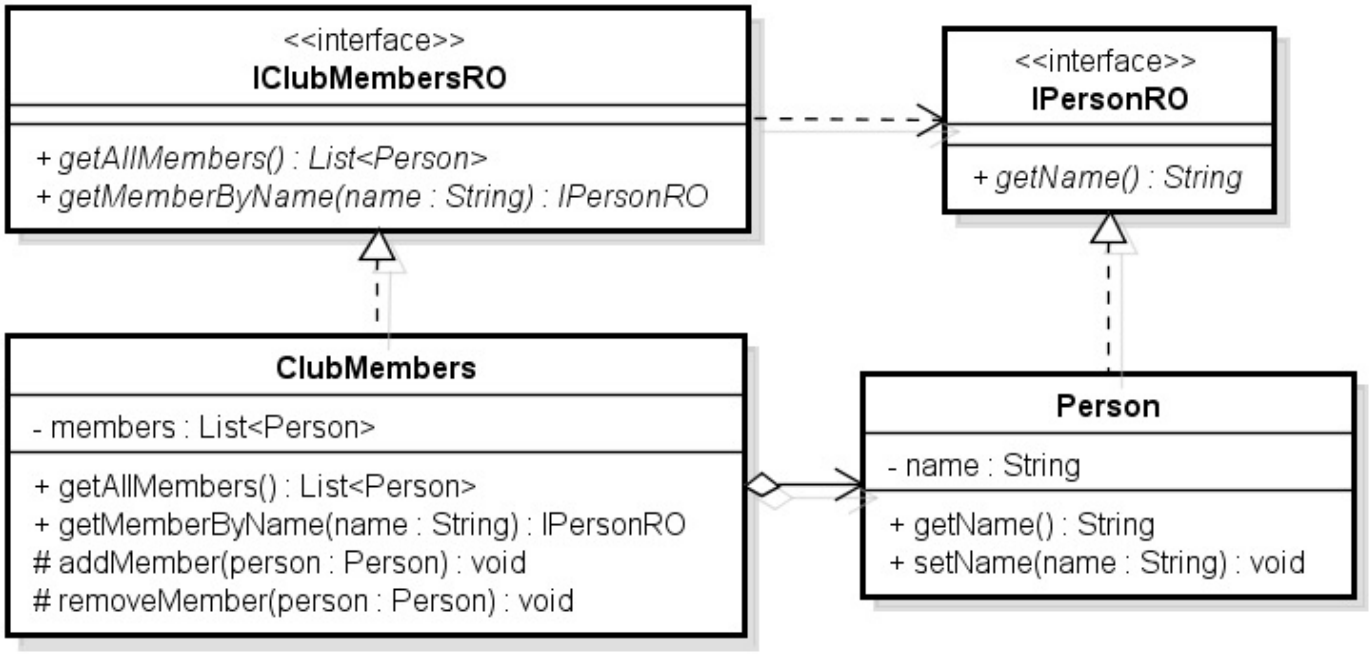


Abbildung 3-28 *Einführen eines Read-only-Interface* `IPersonRO`

Achtung: Probleme trotz Read-only-Interface durch »böse« Casts

Wenn Klienten böswillig wären, könnten diese durch einen Down Cast auf einen Subtyp aus Read-only-Interfaces wieder `Person`-Objekte referenzieren:

```
// NIEMALS MACHEN, NUR ZUR DEMONSTRATION
final IPersonRO readOnlyPerson = clubMembers.getMemberByName("Müller");
final Person modifiablePerson = (Person)readOnlyPerson;
```

Dieses Beispiel sollte reichen, um zu zeigen, weshalb Casts vermieden werden sollten. Ergänzend sollte man die Sichtbarkeit von Klassen auf Package-private einschränken: Package-externe Klienten besitzen dann keinen Zugriff auf die Klasse, und das verhindert automatisch derartige Missbrauchsversuche.

Einfache Lösungsmöglichkeiten für Collections

Für Collections sind mehrere Dinge zu beachten: Zum einen müssen wir verhindern, dass Änderungen an der Collection an sich vorgenommen werden. Zum anderen wollen wir Modifikationen an den gespeicherten Daten möglichst verhindern.

Um zu vermeiden, dass die Collection an sich verändert wird, nutzen wir die Methode `unmodifiableList()` aus dem Collections-Framework:

```
public List<Person> getAllMembersAsUnmodifiable()
{
    // Read-only-Collection
    return Collections.unmodifiableList(members);
}
```

Dabei werden die Elemente der ursprünglichen Liste nicht kopiert, sondern es wird eine unveränderliche Sicht auf die übergebene Liste bereitgestellt: Lesezugriffe

werden an die ursprüngliche Liste weiter delegiert, Schreibzugriffe lösen Exceptions aus.

Zum Schutz vor Modifikationen kann man alternativ eine Kopie der Collection anfertigen und diese zurückgeben.[22] Aufrufer können dann Änderungen an der Zusammensetzung der Kopie vornehmen, ohne dadurch Rückwirkungen auf die Zusammensetzung der ursprünglichen Collection auszulösen:

```
public List<Person> getAllMembersAsCopy()
{
    // Kopie
    return new ArrayList<Person>(members);
}
```

Diese Variante der Kopie kann vorteilhaft sein, wenn Klienten mit den zurückgelieferten Daten beliebig weiterarbeiten möchten.

Allerdings ermöglichen beide genannten Techniken nur den Schutz der Datenstruktur selbst. Die dort gespeicherten Elemente sind weiterhin veränderlich.

Aufwendigere Lösungsmöglichkeiten für Collections

Wollen wir verhindern, dass die Daten verändert werden, so gibt es dafür zwei mögliche Lösungen: Entweder werden für alle Elemente ausschließlich Read-only-Referenzen zurückgegeben oder es werden sogenannte ***tiefe Kopien*** erstellt (siehe folgenden Praxistipp).

Wir werden hier Ersteres umsetzen. Basierend auf dem Read-only-Interface `IPersonRO` für die Klasse `Person`, könnte man intuitiv Folgendes schreiben:

```
public List<IPersonRO> getAllMembers()
{
    // Compile-Error: Cannot cast from List<Person> to List<IPersonRO>
    return (List<IPersonRO>)Collections.unmodifiableList(members);
}
```

So wünschenswert diese Schreibweise auch ist, so führt sie doch zu Problemen bei der Typprüfung und löst einen Kompilierfehler aus. Der Grund ist, dass für Generics eine `List<Person>` nicht zuweisungskompatibel zu einer `List<IPersonRO>` ist. Details werden später in Abschnitt 3.7 behandelt. Hier können wir als Abhilfe folgende Methode `getAllMembersRO()` einführen:

```
public List<IPersonRO> getAllMembersRO()
{
    final List<IPersonRO> readOnlyMembers = new ArrayList<>(members);

    // Read-only-Collection auf Read-only-Daten
    return Collections.unmodifiableList(readOnlyMembers);
}
```

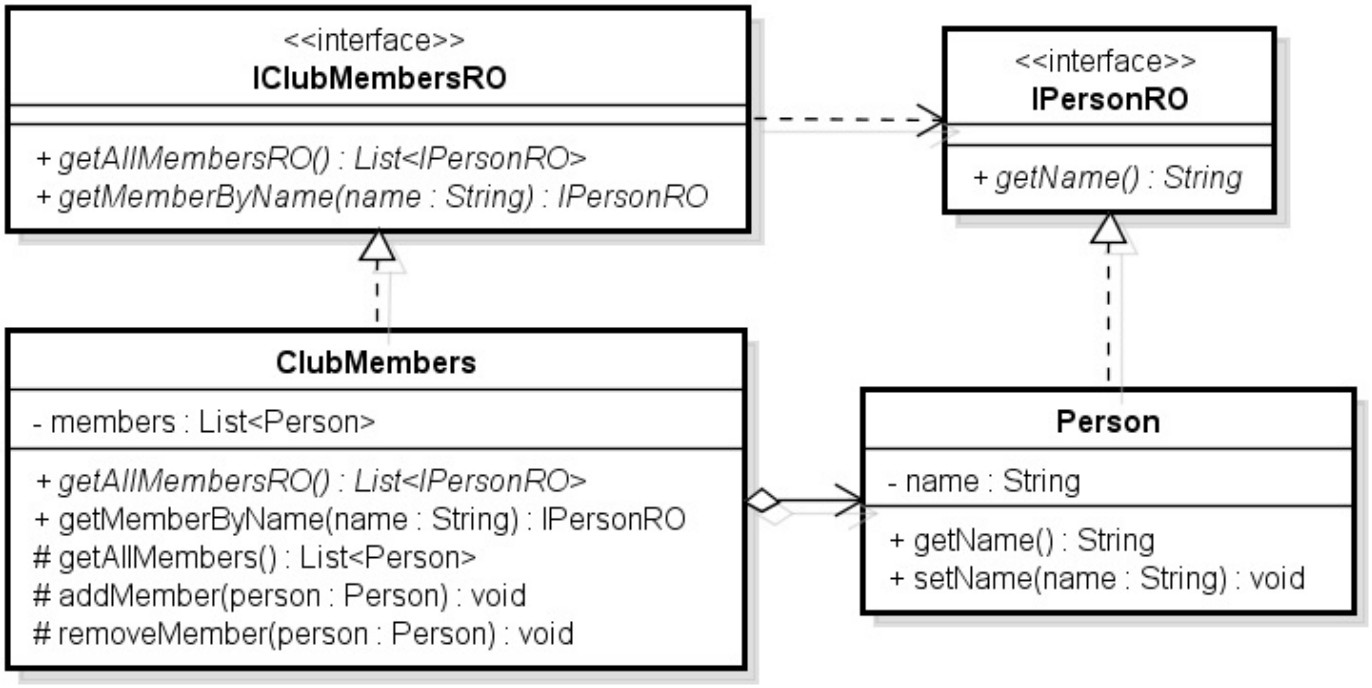


Abbildung 3-29 *Einsatz der Methode* `getAllMembersRO()`

Weiterhin kann man die ursprünglich öffentliche Methode `getAllMembers()` in ihrer Sichtbarkeit auf `protected` einschränken. Damit gewährt man Subklassen weiterhin Schreibzugriffe, Klienten können jedoch nur noch lesend über das Interface `IClubMembersRO` zugreifen. Abbildung 3-29 zeigt dies im Kontext der anderen Klassen.

Tipp: Kopierstrategien

Eine sogenannte ***flache Kopie*** (***Shallow Copy***) erzeugt eine exakte, bitweise Kopie eines Originals. Es entstehen sowohl für primitive Elemente als auch für Referenzvariablen Wertkopien. Für primitive Typen ist das nicht problematisch. Referenzen von Kopie und Original verweisen so jedoch auf die gleichen Daten, sodass spätere Änderungen am Original folglich auch in der Kopie sichtbar sind (und umgekehrt). Man kann dies vermeiden, wenn man eine sogenannte ***tiefe Kopie*** (***Deep Copy***) erstellt. Dazu werden alle enthaltenen Objekte rekursiv kopiert. Dies ist vom Entwickler selbst zu programmieren und kann aufwendig werden, wenn viele Objekte referenziert werden.

3.4.2 Immutable-Klasse

Ziel beim Einsatz der nun beschriebenen Technik IMMUTABLE-KLASSE ist es, den Objektzustand nach der Objekterzeugung unveränderlich zu halten. Eine wichtige Voraussetzung besteht darin, alle Attribute `private` und `final` zu deklarieren. Dies drückt aus, dass nach der initialen Zuweisung bei der Konstruktion keine Veränderungen stattfinden sollen, und schützt zudem vor versehentlichen Änderungen innerhalb der Klasse selbst. Dazu müssen allerdings bereits zur Konstruktion alle benötigten Daten vorliegen und an Attribute zugewiesen werden.

Dadurch bietet die Klasse dann nur noch lesenden Zugriff auf ihre Daten. Ein Beispiel zeigt Abbildung 3-30.

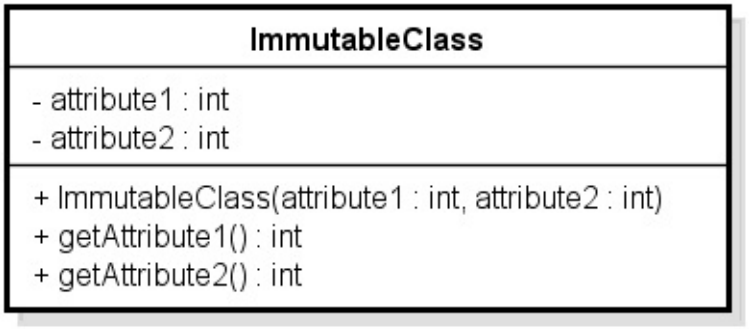


Abbildung 3-30 *Klassendiagramm für eine Immutable-Klasse*

Wird eine Veränderbarkeit unter gewissen Bedingungen dennoch gewünscht oder benötigt, so kann man dies durch einen Trick erreichen: Statt die Werte der Attribute zu modifizieren, wird ein neues Objekt mit passender Wertebelegung erzeugt.[23]

Am Beispiel der folgenden Klasse `ImmutablePosition` verdeutliche ich das zuvor Beschriebene anhand der Methode `offset(int, int)`. Diese erzeugt für jede Positionsänderung neue `ImmutablePosition`-Objekte:

```
public final class ImmutablePosition
{
    private final int x;
    private final int y;

    public ImmutablePosition(final int x, final int y)
    {
        this.x = x;
        this.y = y;
    }

    public int getX()     { return x; }
    public int getY()     { return y; }

    public ImmutablePosition offset(final int xOffset, final int yOffset)
    {
        return new ImmutablePosition(x + xOffset, y + yOffset);
    }
}
```

Umgang mit veränderlichen gespeicherten Werten

Das einleitende Beispiel lässt vermuten, dass es leicht wäre, eine Klasse unveränderlich zu machen. Für Attribute primitiver Datentypen reicht es tatsächlich bereits aus, diese `final` zu deklarieren und während der Konstruktion zu initialisieren. Es gibt einige Spezialfälle für die dies nicht gilt, etwa immer dann, wenn über die sogenannten »Escaping References«, die wir später in Kapitel 16 kennenlernen werden, andere Objekte bereits Zwischenzustände von Wertebelegungen sehen können.

Werden in Attributen jedoch Referenzen auf andere Klassen, insbesondere auf

Containerklassen, gespeichert, so ist es nicht so einfach möglich, Unveränderlichkeit zu erreichen. Die Gründe wurden bereits zuvor in der Beschreibung der Read-only-Interfaces ausführlich dargestellt. Schauen wir nun, was notwendig ist, um eine Klasse unveränderlich zu machen, wenn auch Referenzattribute auf veränderliche Klassen existieren.

Schutz von Referenzattributen

Nehmen wir an, wir wollten eine Zeitspanne beschreiben. Die folgende Klasse `PeriodOfTime` modelliert diese und nutzt dazu zwei Attribute `start` und `end` vom Typ `Date`:

```
public final class PeriodOfTime
{
    private final Date start;
    private final Date end;

    public PeriodOfTime(final Date start, final Date end)
    {
        Objects.requireNonNull(start, "start must not be null");
        Objects.requireNonNull(end, "end must not be null");

        if (start.after(end))
            throw new IllegalArgumentException(start + " must be <= " + end);

        this.start = start;
        this.end = end;
    }

    public Date getStart()    { return start; }
    public Date getEnd()      { return end;   }
}
```

Bei dieser Realisierung sind mehrere Dinge erwähnenswert. Beginnen wir mit den Prüfungen der Konstruktorparameter. Dazu wird die Methode `requireNonNull(T, String)` aus der mit JDK 7 eingeführten Utility-Klasse `java.util.Objects` genutzt. Ansonsten scheint diese Implementierung ähnlich zu der zuvor gezeigten Klasse `ImmutablePosition` zu sein und somit dem beschriebenen Muster der Unveränderlichkeit zu folgen. Es gibt aber Unterschiede. Betrachten wir dies im Detail: Die Implementierung der Klasse `PeriodOfTime` prüft im Konstruktor die Gültigkeit der Parameter und damit auch die Einhaltung der gewünschten Invariante *start* ≤ *end*. Die Attribute sind zudem `final`. Allerdings führt dies lediglich zu einer Unveränderlichkeit der `Date`-Referenzen. Aber die dort gespeicherten Werte sind *veränderlich*, da die Klasse `Date` selbst veränderlich ist: Sie besitzt `set()`-Methoden für ihre Attribute. Das folgende Listing zeigt die Veränderlichkeit exemplarisch anhand des Aufrufs der Methode `setTime(700)` für die Instanz `end`:

```
final Date start = new Date(1000);
final Date end = new Date(2000);
final PeriodOfTime period = new PeriodOfTime(start, end);

// Invariante (start <= end) von PeriodOfTime wird durch setTime(700) zerstört
```

```
end.setTime(700);
```

Möchte man Änderungen im Objektzustand sicher ausschließen, so muss man übergebene Referenzen und deren Speicherung in Attributen entkoppeln. Dazu erzeugt man aus den an den Konstruktor übergebenen `Date`-Objekten neue Instanzen wie folgt:

```
public class PeriodOfTime(final Date start, final Date end)
{
    Objects.requireNonNull(start, "start must not be null");
    Objects.requireNonNull(end, "end must not be null");

    if (start.after(end))
        throw new IllegalArgumentException(start + " must be <= " + end);

    this.start = new Date(start.getTime());
    this.end = new Date(end.getTime());
}
```

Nun sind die Attribute geschützt: Wenn Änderungen an den übergebenen `Date`-Objekten durchgeführt werden, führt dies nicht mehr zu Änderungen im Objektzustand. Es existieren aber weitere Schlupflöcher:[24] Da die Zugriffsmethoden `getStart()` und `getEnd()` Referenzen auf die internen `Date`-Objekte liefern, ist eine Veränderung von außen tatsächlich immer noch möglich. Um auch dies zu unterbinden, müssen die herausgereichten Referenzen wie folgt von den internen Daten abgekoppelt werden:

```
public Date getStart()   { return new Date(start.getTime()); }
public Date getEnd()     { return new Date(end.getTime()); }
```

Schutz von Collections

Selbst wenn wir Referenzen vor Veränderungen schützen, kann es noch zu Änderungen am Objektzustand kommen: Werden Referenzen auf Containerklassen gehalten, so ist zusätzlicher Aufwand nötig, um die Unveränderlichkeit (möglichst weitgehend) herzustellen – eine vollständige Unveränderlichkeit erhält man nur, wenn man tiefe Kopien erstellt. Wir werden am Ende dieses Abschnitts eine schwächere Abkopplung in Form von Kopien der eingehenden Collections realisieren.

Wir betrachten nun den Schutz von Collections und greifen das Beispiel der Klasse `ClubMembers` wieder auf, und zwar konkret für das Attribut `members`, das die Mitglieder des Clubs in Form einer Liste von `Person`-Objekten speichert. Nehmen wir an, es wäre ein Konstruktor wie folgt eingeführt worden:

```
public final class ClubMembersUnsafeCollectionExample
{
    // Sowohl die Referenz als auch die Zusammensetzung sind veränderlich
    private List<Person> members = new ArrayList<>();

    public ClubMembersUnsafeCollectionExample(final List<Person> members)
    {
        this.members = members;
    }
```

```
}
```

Diese in der Praxis relativ häufig zu sehende Umsetzung birgt einige Risiken. Leider sieht man mitunter darüber hinaus noch eine `set()`-Methode, die ebenfalls eine Referenzzuweisung durchführt wie folgt:

```
public void setMember(final List<Person> newMembers)
{
    // Achtung: ungeprüfte Zuweisung!
    this.members = newMembers;
}
```

Wie bereits bei den Read-only-Interfaces erwähnt, wirken sich bei der Speicherung von Referenzen auf Collections spätere Änderungen anderer Objekte auf den eigenen Objektzustand aus. Das ist natürlich für Immutable-Klassen nicht erwünscht. Ähnliches gilt aber meistens auch für »normale« Klassen, weil deren Objektzustand dann von außen veränderlich ist. Es existiert ein weiteres Problem, weil eine `null`-Referenz übergeben werden kann, was Laufzeitfehler durch `NullPointerException`s auslösen kann. Wir korrigieren das Beispiel wie folgt:

```
public final class ClubMembersSafeCollectionExample
{
    // Referenz fix, Inhalt veränderlich
    private final List<Person> members;

    public ClubMembersSafeCollectionExample(final List<Person> newMembers)
    {
        Objects.requireNonNull("parameter 'newMembers' must not be null!");

        // Sorgt für Unabhängigkeit von Änderungen im Original
        this.members = new ArrayList<>(newMembers);
    }
}
```

Im Listing sehen wir, dass die Referenz auf die Liste `final` ist und mit einer `ArrayList<Person>` initialisiert wird. Es erfolgt keine Zuweisung der Referenz der übergebenen Liste, sondern es wird deren Inhalt in eine interne Liste übertragen. Somit wirken sich Änderungen an der Zusammensetzung des Originals nicht im neu konstruierten Objekt aus.

Bewertung

Immutable-Klassen …

+ verhindern Änderungen am Objektzustand und sorgen für weniger Komplexität und mehr Robustheit der eigentlichen Anwendung.
+ benötigen keine Synchronisation von Zugriffen auf Attribute bei Multithreading, da nach der Objektkonstruktion keine Wertänderungen mehr erfolgen (können).
- erfordern bei Attributen von Referenztypen und insbesondere von Collectiontypen deutlich mehr Aufwand zum Schutz von gespeicherten Daten vor Veränderungen als bei der Nutzung primitiver Datentypen.

- führen selbst bei kleinen Änderungen am Objektzustand immer zur Konstruktion neuer Objekte. Dadurch kommt es zu ständig wechselnden Referenzen. Falls diese in anderen Klassen zwischengespeichert werden, können wiederum Probleme auftreten. In einem solchen Fall hat man zwar keine Zugriffsprobleme, aber trotzdem inkonsistente Sichten durch veraltete Daten.

3.4.3 Marker-Interface

Im vorherigen Abschnitt haben wir verschiedene Methoden kennengelernt, Eigenschaften oder Verhaltensweisen ohne den Einsatz von Vererbung auszudrücken. Eine weitere Möglichkeit dazu bietet die Technik MARKER-INTERFACE. Mithilfe eines solchen Interface kann man Metainformationen über eine Klasse zur Verfügung stellen. Man kann sich dies in etwa wie ein konstantes statisches boolesches Attribut einer Klasse vorstellen, das eine Eigenschaft ausdrückt. Seit JDK 5 sollte man dafür allerdings besser die Technik der Annotations nutzen, die ich später in Abschnitt 8.2 beschreibe.

Um wirklich nur eine Markierung vorzunehmen, enthält ein Marker-Interface weder Methoden noch Konstanten: Es ist also leer. Wenn eine Klasse ein Marker-Interface implementiert, so wird damit gekennzeichnet, dass eine spezielle Eigenschaft erfüllt werden soll. Abbildung 3-31 zeigt dies am Beispiel des Marker-Interface `java.io.Serializable`.

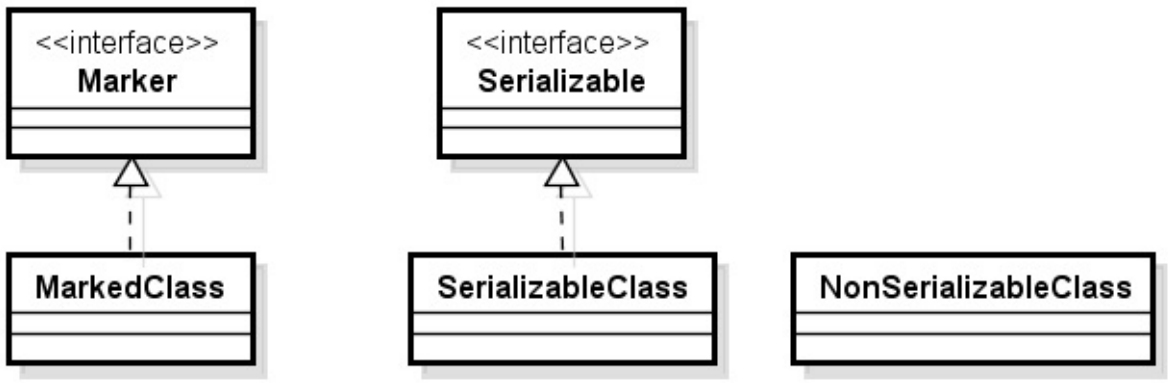


Abbildung 3-31 *Klassendiagramm des Marker-Interface*

Die Abbildung zeigt eine Klasse `SerializableClass`, die das genannte Marker-Interface erfüllt, und sich damit von der in die JVM integrierten Serialisierungsautomatik verarbeiten lässt, d. h. in eine bzw. aus einer Folge von Bytes umgewandelt werden kann. Die weiterhin gezeigte Klasse `NonSerializableClass` erfüllt das Interface nicht und kann daher von der Automatik nicht verarbeitet werden (vgl. Abschnitt 8.3).

Bewertung

Der Einsatz von Marker-Interfaces …

+ erlaubt es, markierte Eigenschaften allein basierend auf dem Typ abfragen zu

können, also ohne, dass eine Instanz einer Klasse benötigt wird.

o erfordert einen Aufruf von `instanceof`, um die Existenz einer markierten Eigenschaft zu prüfen. Für das Beispiel der Serialisierung wäre etwa eine Abfrage über eine Methode `isSerializable()` klarer.

- garantiert nicht, dass sich eine implementierende Klasse auch entsprechend der angegebenen Eigenschaft verhält, weil es ja nur eine Markierung ist. Bei einem normalen Interface würde man zumindest die Implementierung der dort definierten Methoden durch den Compiler prüfen können.

3.4.4 Konstantensammlungen und Aufzählungen

Unter Aufzählungen versteht man Sammlungen semantisch zusammengehörender Werte. Statt der Definition von Konstanten sieht man manchmal jedoch den Einsatz der jeweils benötigten konkreten Werte als ***Literale***[25]. Man spricht auch von ***Magic Numbers***. Deren Nutzung erschwert in der Regel die Verständlichkeit und Wartbarkeit. Als Abhilfe ist es wünschenswert, die konstanten Werte semantisch zu gruppieren und zentral zu definieren. Dadurch kann man die Konstanten über deren Namen innerhalb von anderen Klassen ansprechen. Dazu sind verschiedene Realisierungen gebräuchlich:

- Definition einiger Konstanten in der verwendenden Klasse oder in einer separaten Konstantensammlungsklasse
- Definition eines eigenen Aufzählungstyps gemäß dem ENUM-Muster
- Definition eines eigenen Aufzählungstyps mit dem Schlüsselwort `enum`

Da man alle genannten Formen der Realisierung immer wieder antrifft, sollen hier kurz mögliche Auswirkungen ihres Einsatzes besprochen werden. Seit JDK 5 sollte man bevorzugt das Schlüsselwort `enum` verwenden, um Aufzählungen zu definieren. Nur in älteren Java-Versionen benötigte man dafür Hilfskonstrukte wie das ENUM-Muster.

Konstantensammlungen

Häufig sieht man eine einfache Umsetzung von Aufzählungen in einer Klasse, die lediglich mit öffentlichen Konstanten arbeitet. In folgendem Beispiel sind in der Klasse `ErrorStateAsIntConstants` einige Fehlerwerte als `int`-Konstanten definiert:

```
public final class ErrorStateAsIntConstants
{
    public static final int OK                = 0;
    public static final int INVALID_POSITION  = 3;
    public static final int INPUT_BUFFER_FULL = 8;

    private ErrorStateAsIntConstants()
```

```
    {} // Vermeide Konstruktion dieser Klasse
}
```

Diese Realisierung weist zwei Nachteile auf. Zum einen sind derart definierte Konstanten weder typsicher noch überraschungsfrei: Überall, wo die Werte dieser Konstanten zu übergeben sind, könnten auch beliebige andere `int`-Werte verwendet werden, und nicht nur diejenigen, für die Konstanten definiert sind: Statt der Konstanten `OK` könnte auch der weniger aussagekräftige, aber entsprechende Zahlenwert `0` verwendet werden.

Aufzählungstypen gemäß dem ENUM-Muster

Um die geschilderten Probleme reiner Konstantensammlungen zu adressieren, wurde das ENUM-Muster erdacht. Hierbei wird ein neuer, eigenständiger Datentyp in Form einer Klasse definiert. Diese dient nicht nur als Namensraum und Sammelstelle für Konstanten, sondern definiert vor allem einen eigenständigen Typ und zugehörige Aufzählungskonstanten als Objekte.

In folgenden Beispiel werden die zuvor als `int`-Werte implementierten Konstanten nun als statische, öffentliche Objekte vom Typ `ErrorState` definiert. Die Klasse ist `final`, um eine Ableitung zu verbieten. Eine Definition zusätzlicher Konstanten dieses Typs an anderer Stelle als dieser Klasse wird durch den privaten Konstruktor verhindert.

```
public final class ErrorState
{
    public static final ErrorState OK          = new ErrorState(0, "Ok");
    public static final ErrorState INVALID_POS = new ErrorState(3,
                                                   "Ungültige Positionsangabe");
    public static final ErrorState BUFFER_FULL = new ErrorState(8,
                                                   "Empfangspuffer voll");

    private final int              value;
    private final String           description;

    private ErrorState(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }

    public int getValue()              { return value; }
    public String getDescription()     { return description; }
}
```

Zur Modellierung der Konstanten werden hier die Attribute `value` und `description` verwendet. Im Attribut `value` wird der Wert der `int`-Konstante gespeichert und in `description` ein beschreibender Text. Neben den eingangs genannten Vorteilen besitzen derart definierte Aufzählungen folgende Beschränkungen:

1. **Fehlende Sortierreihenfolge** – Die Klasse `ErrorState` definiert zunächst lediglich eine lose Sammlung von Konstanten ohne eine Ordnung. In vielen Fällen

ist dies bereits ausreichend, wenn es nur um Typsicherheit geht. Die Realisierung einer Sortierreihenfolge muss selbst implementiert werden, etwa mithilfe des Interface `Comparable` (vgl. Abschnitt 5.1.8) und der Methode `compareTo()`. Dadurch kann eine Reihenfolge für die Konstanten festgelegt werden.

2. **Fehlende Serialisierbarkeit** – Sollen die Aufzählungskonstanten in einen Stream serialisiert werden, so muss die Aufzählungsklasse das Interface `Serializable` (vgl. Abschnitt 8.3) implementieren und zudem einen inhaltlichen Vergleich durch Überschreiben der Methode `equals()` (vgl. Abschnitt 4.1.2) realisieren.

Beide Nachteile können durch etwas Implementierungsaufwand behoben werden. Sinnvoller ist aber der Einsatz des Schlüsselworts `enum`, wie dies nun vorgestellt wird.

Aufzählungstypen mit dem Schlüsselwort `enum`

Werden Aufzählungen mithilfe des Schlüsselworts `enum` definiert, so besitzen die Konstanten die zuvor genannten Nachteile nicht, sondern sind »out-of-the-Box« vergleichbar und serialisierbar, da sie von Hause aus die Interfaces `Comparable` und `Serializable` implementieren. Außerdem besitzen sie eine Reihenfolge gemäß ihrer Definition (dem Auftreten im Sourcecode), die durch die Methode `ordinal()` abgefragt werden kann.

Folgendes Listing zeigt die Definition der Fehlerwerte als `enum`-Aufzählung `ErrorStateEnum`. Diese enthält, wie die zuvor selbst erstellte Lösung in Form einer Klasse, zwei eigene Attribute `value` und `description`:

```
public enum ErrorStateEnum
{
    OK(0, "Ok"),
    INVALID_POS(3, "Ungültige Positionsangabe"),
    BUFFER_FULL(8, "Empfangspuffer voll");

    private final int    value;
    private final String description;

    private ErrorStateEnum(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }

    public int getValue()              { return value; }
    public String getDescription()     { return description; }
}
```

Kombination von Eigenschaften mithilfe der Klasse `EnumSet`

Bis hierher haben wir lediglich Aufzählungen betrachtet, deren Werte sich gegenseitig ausgeschlossen haben. Zum Teil beschreiben Aufzählungswerte aber voneinander unabhängige Eigenschaften und sollen kombinierbar sein. Das gilt etwa für Darstellungsattribute eines Zeichensatzes. Eine Implementierung für diese könnte

wie folgt aussehen, wenn wir keine `enum`-Aufzählun verwenden, sondern lediglich `int`-Konstanten:

```
public final class FontAttributes
{
    public static final int NORMAL    = 0;
    public static final int BOLD      = 1;
    public static final int ITALIC    = 2;
    public static final int UNDERLINE = 4;

    private FontAttributes()
    {} // Vermeide Konstruktion dieser Klasse
}
```

Zur Kombination von Eigenschaften müssen diese bitweise mit ODER (Operator ' | ') verknüpft werden. Zur Sicherstellung der Eindeutigkeit bei der Kombination von Eigenschaften sind für die Werte Zweierpotenzen zu wählen, die in ihrer binären Darstellung jeweils einem gesetzten Bit entsprechen. Möchte man ermitteln, ob eine Eigenschaft gewählt ist, müssen die Werte durch ein bitweises UND (Operator ' & ') geprüft werden. Folgendes Listing zeigt dies in drei Varianten (V1, V2 und V3):

```
public static void main(final String args[])
{
    final int fontStyles = BOLD | ITALIC;

    // Variante 1: Prüfe Attribut und zeige Implementierungsdetails
    final boolean isUnderline = (fontStyles & UNDERLINE) == UNDERLINE; // V1

    // Variante 2+3: Prüfe Attribute, Abstraktion von Implementierungsdetails
    final boolean isBold = isBold(fontStyles);                          // V2
    final boolean isItalic = isAttributeEnabled(fontStyles, ITALIC);    // V3

    System.out.println("isUnderline " + isUnderline);
    System.out.println("isBold " + isBold);
    System.out.println("isItalic " + isItalic);
}

// Variante 2
private static boolean isBold(final int fontStyles)
{
    return (fontStyles & BOLD) == BOLD;
}

// Variante 3
private static boolean isAttributeEnabled(final int fontStyles,
                                          final int attributeValue)
{
    return (fontStyles & attributeValue) == attributeValue;
}
```

Listing 3.2 *Ausführbar als* **'FONTATTRIBUTESEXAMPLE'**

Derartige Prüfungen werden schnell unübersichtlich, wenn man sie, wie für das Attribut `isUnderline` gezeigt, selbst programmiert (Variante 1). Dies macht sich besonders negativ bemerkbar, wenn viele Abfragen unterschiedlicher Eigenschaften erfolgen sollen. Die Lesbarkeit des Sourcecodes leidet und die Gefahr für Fehler steigt. Es bietet sich an, die Prüfung der Eigenschaften in Methoden auszulagern, wie dies im Beispiel für die Methode `isBold()` (Variante 2) umgesetzt ist. Dadurch lässt

sich nutzender Sourcecode besser lesen. Sind allerdings viele Eigenschaften zu prüfen, so steigt auch die Anzahl trivialer Prüfmethoden. Das Programm nimmt an Umfang zu, ohne viel mehr Funktionalität bereitzustellen. Es bietet sich dann eine allgemeinere Prüfmethode ähnlich zu der gezeigten `isAttributeEnabled()` an (Variante 3).

Einfacher und eleganter lässt sich die Aufgabenstellung realisieren, wenn man die Darstellungsattribute als `enum`-Aufzählung `FontAttributesEnum`[26] realisiert:

```
enum FontAttributesEnum
{
    BOLD, ITALIC, UNDERLINE;
}
```

Durch Einsatz der generischen Klasse `EnumSet<E>` kann man auf effiziente Weise Mengen von `enum`-Aufzählungswerten vom Typ `E` verwalten.[27] Zudem können wir auf die Definition des zuvor benötigten Werts `NORMAL` verzichten, da sich dieser als leere Menge darstellen lässt. Wir schreiben das Beispiel wie folgt um:

```
public static void main(final String args[])
{
    final EnumSet<FontAttributesEnum> fontStyles = EnumSet.of(BOLD, ITALIC);

    final boolean isUnderline = fontStyles.contains(UNDERLINE);
    final boolean isBold = fontStyles.contains(BOLD);

    System.out.println("isUnderline " + isUnderline);
    System.out.println("isBold " + isBold);

    // nützliche weitere Methoden
    System.out.println("All " + EnumSet.allOf(FontAttributesEnum.class));
    System.out.println("None " + EnumSet.noneOf(FontAttributesEnum.class));
}
```

Listing 3.3 *Ausführbar als* **'FONTATTRIBUTESENUMEXAMPLE'**

Im Listing wurden verschiedene der im Folgenden beschriebenen FABRIKMETHODEN (vgl. Abschnitt 18.1.2) der Klasse `EnumSet<E>` eingesetzt. Diese ermöglichen es, Mengen von `enum`-Aufzählungswerten zusammenzustellen:

- `EnumSet<E> of(E first, E... others)` – Erstellt eine Menge, die aus den angegebenen Werten besteht.
- `EnumSet<E> range(E from, E to)` – Beschreibt eine Menge, die alle Werte enthält, die zwischen den beiden angegebenen Werten liegen, auch die angegebenen Grenzwerte.
- `EnumSet<E> allOf(Class<E> elementType)` – Erstellt eine Menge, die alle Werte des angegebenen Typs enthält.
- `EnumSet<E> noneOf(Class<E> elementType)` – Konstruiert eine typsichere leere Menge vom Typ `EnumSet<E>`.

Sprachelemente Varargs und Klasseninformationen

Trotz der Kürze des Beispiels sind verschiedene Dinge erwähnenswert: Neben der neu kennengelernten Klasse `EnumSet<E>` wurden zwei bisher in diesem Buch nicht beschriebene Sprachelemente genutzt, die ich nun hier erläutere.

Zum einen ist dies die sogenannte variable Argumentliste (kurz ***Varargs***), die beliebig viele Übergabeparameter eines Typs erlaubt. Syntaktisch wird dies durch drei Punkte nach der Typangabe, etwa `String...` in einer Methodensignatur ausgedrückt. Das bedeutet hier also 0 bis n Elemente vom Typ `String`. Diese Notation wird im Beispiel etwa beim Aufruf der Methode `EnumSet.of(E, E...)` eingesetzt und erfordert die Angabe eines Elements des Enum-Typs `E`, hier also vom Typ `FontAttributesEnum`, gefolgt von beliebig vielen weiteren Elementen des gleichen Typs.

Zum anderen haben wir die Angabe von Informationen zum Typ beim Aufruf von `EnumSet.allOf(Class<E>)` genutzt, mit deren Hilfe wir alle Elemente vom Typ `FontAttributesEnum.class` als `EnumSet<FontAttributesEnum>` erhalten. Wie im Beispiel gesehen, kann man auf diese Typinformationen über die Notation `<Typ>.class`, etwa `String.class` oder `String[].class`, zugreifen und erhält dann ein Objekt vom Typ `Class<E>`. Ganz allgemein wird jeder Typ in Java durch ein `Class<E>`-Objekt beschrieben und bietet darüber Zugriff auf die Metainformationen von Klassen: Man kann etwa Methoden und Attribute sowie deren Sichtbarkeiten abfragen. Außerdem kann man den Typ einer Instanz mithilfe der Methode `getClass()` ermitteln und diese Rückgabe mit dem Typ anderer Objekte vergleichen.

3.4.5 Value Object (Data Transfer Object)

Das Impementierungsmuster VALUE OBJECT[28] oder auch (DATA TRANSFER OBJECT) folgt der Idee, einen Container für mehrere Attribute oder Parameter bereitzustellen, um diese zu neuen Strukturen zusammenzufassen. Dies ist beispielsweise immer dann sinnvoll, wenn man Parameter in einer Methodensignatur oder Attribute in einer Klasse semantisch gruppieren. Zudem wird die ursprünglich benötigte Anzahl an Parametern bzw. Attributen reduziert. Je nach Einsatzort und -zweck besitzt dieses Muster diverse Namen: Fasst es verschiedene Parameter in einer Methodensignatur zusammen, so spricht man häufig auch von einem ***Parameter Value Object*** oder kurz ***Parameter Object***. Im Bereich von Client-Server-Architekturen kennt man diese Art der Realisierung unter dem Namen ***Data Transfer Object*** (DTO). Hier sorgt es dafür, dass die Übertragung von Daten nicht feingranular, Attribut für Attribut, sondern in einem Rutsch erfolgen kann. Das bietet den Vorteil, dass nur einmalig statt mehrfach die Kosten für Netzwerkzugriffe zu zahlen sind, und macht sich positiv bemerkbar: Insbesondere verbessert es die Performance.

Als Beispiel betrachten wir hier eine Klasse `Person`, die zwei Value Objects referenziert, die die Adressdaten und die Kontaktinformationen (Telefon, E-Mail usw.) beinhalten. Abbildung 3-32 zeigt dies.

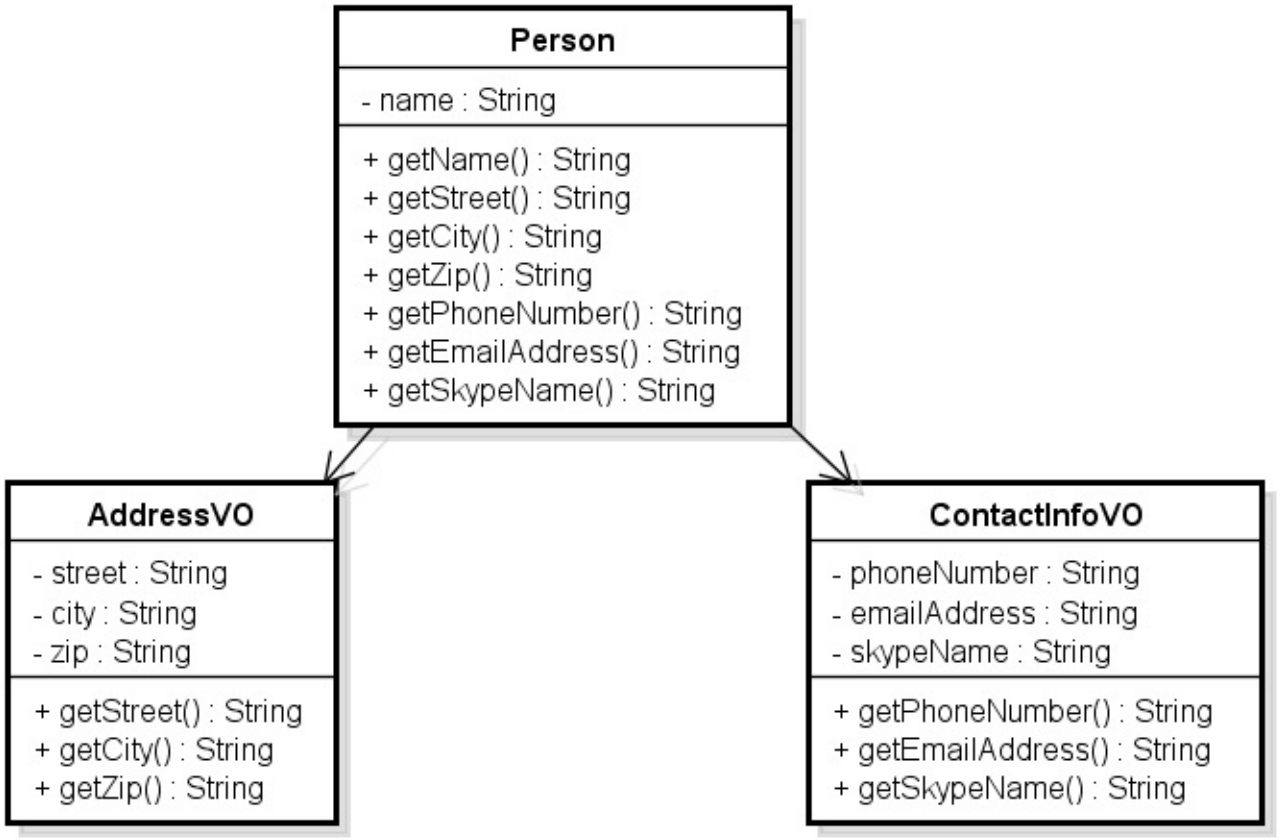


Abbildung 3-32 *Beispiele für Realisierungen von Value Objects*

Anmerkungen

Häufig ist es wünschenswert und sinnvoll, semantisch zusammengehörende Werte zu Value Objects / Data Transfer Objects zu gruppieren. In einigen Fällen kann die semantische Klammer auch nur darin bestehen, die Übergabe oder Datenspeicherung zu kapseln. Es wird kein eigenes Verhalten in Form von Business-Methoden angeboten, sondern lediglich ein »dummer« Container für Daten. Daher erlauben feingranulare `get()` - und `set()`-Methoden den individuellen Zugriff auf die gespeicherten Werte. Die zuvor angeführte Kritik an `get()` - und `set()`-Methoden greift hier nicht, da kein Verhalten im Sinne von Business-Funktionalität modelliert wird. Wird ein Value Object als innere Klasse implementiert, so kann auf Zugriffsmethoden verzichtet werden.

Bewertung

Der Einsatz eines Value Object / Data Transfer Object …

+ erlaubt es, einen eigenen Datencontainer zu realisieren, dessen Verwendung Details versteckt, da weniger Attribute und Parameter zum Einsatz kommen.

+ fasst Daten zusammen und kann für klarere Methodensignaturen sorgen.

○ modelliert nur einen Ausschnitt aus einem größeren Objekt und repräsentiert zudem die Wertebelegung zu einem speziellen Zeitpunkt. Ein Konsistenzabgleich

zwischen unterschiedlichen Ständen kann schwierig sein.

Tipp: Vermeide Hilfsmethoden in DTOs

Manchmal ist man versucht, Hilfsmethoden in einem DTO zu definieren. Dieser Versuchung sollte man jedoch widerstehen. Zum einen widerspricht dies dem Gedanken, einfache Datencontainer-Objekte zur Verfügung zu stellen. Zum anderen führt dies zu einem noch viel entscheidenderen Problem, wenn ein Transport per Netzwerk benötigt wird.

Die Übertragung von DTOs über ein Netzwerk wird stark vereinfacht, wenn diese das Interface `Serializable` implementieren. Dadurch können sie von der JVM automatisch in eine Folge von Bytes umgewandelt werden. Dabei wird eine spezielle Versionsnummer benutzt, um die Typen der Objekte zu identifizieren. Diese Versionsnummer kann explizit als Attribut `serialVersionID` im Sourcecode angegeben werden. Häufig ist dies aber nicht der Fall. Dann wird automatisch beim Kompilieren eine solche Versionsnummer berechnet und beim Serialisieren verwendet. In die Berechnung gehen unter anderem die in einer Klasse definierten Methoden ein. Jede neu eingeführte Methode verändert damit diese automatisch ermittelte Versionsnummer und führt zu Inkompatibilitäten zu älteren Versionen des DTO, ohne dass sich die strukturelle Zusammensetzung (die Attribute) des Containers geändert hätte. Details zur Serialisierung beschreibt Abschnitt 8.3.

Es ist demnach nicht nur aus der Sicht der Trennung von Daten und Verarbeitung, sondern auch aus Gründen der Kompatibilität sinnvoll, Hilfsmethoden in Toolkitoder Utility-Klassen zu definieren.

3.5 Prinzipien guten OO-Designs

Bis hierher sollten Sie mittlerweile einen ersten Eindruck davon gewonnen haben, was gute objektorientierte Entwürfe ausmacht und welche Probleme einem mitunter dabei begegnen können.

Bevor wir uns gleich intensiver mit gutem Design beschäftigen, möchte ich zuvor verdeutlichen, was denn schlechtes Design ist und woran man es erkennen kann.

Was ist schlechtes Design? Woran erkennt man es?

Sicherlich ist Ihnen beim Überarbeiten von Programmen schon das eine oder andere Problem begegnet. Ein Zeichen für schlechtes Design ist häufig, wenn …

- der Sourcecode wenig verständlich ist, unter anderem verursacht durch ***ungünstige Namensgebung*** oder eine ***fehlende Dokumentation*** komplizierterer Stellen.
- ***unnötige Komplexität*** oder ein extrem flexibles Design mit hohem

Variantenreichtum existiert, obwohl diese Extras (teilweise) nicht benötigt werden.

- sich Erweiterungen oder ***Modifikationen nur schwierig durchführen lassen*** und zum Teil ***große Auswirkungen*** auch in anderen Teilen des Sourcecodes besitzen.
- das zu lösende Problem anhand der Implementierung nicht abgeleitet werden kann, etwa weil es an semantischer Strukturierung fehlt oder ***kein klares Layout des Sourcecodes*** vorliegt.
- nur ***wenige Tests*** existieren und Testbarkeit kaum gegeben ist, z. B. weil ***kaum für sich testbare Klassen existieren*** und Objekte stark voneinander abhängen, sodass lediglich schwierig testbare Objekthaufen oder Verklumpungen vorhanden sind.
- ***diverse Fehler*** bereits bekannt sind und wahrscheinlich noch viele weitere versteckt lauern. Das erkennt man an Folgendem: Behebt man einen Fehler, so stößt man umgehend auf einen weiteren. Behebt man auch diesen, so kommt es zu Problemen an ganz anderer Stelle. Das System ist so weit verrottet, dass nur noch ein Aufräumkraftakt oder aber ein vollständiges Redesign helfen.

Gutes Design

In den Beispielen der vorangegangenen Abschnitte wurde das Thema Design vor allem aus dem Blickwinkel der Praxis betrachtet. Nun wollen wir das Ganze etwas formalisieren und uns dazu fragen, welche Regeln oder Vorgehensweisen man befolgen sollte, um die zuvor geschilderten Negativpunkte schlechter Designs möglichst zu vermeiden. Wie entstehen also wartbare und gut strukturierte Programme und was zeichnet gutes objektorientiertes Design aus? Und wie erreicht man Eigenschaften wie Flexibilität, Erweiterbarkeit, Wartbarkeit und Verständlichkeit. Verschiedene Merkmale zur Qualität, u. a. Zuverlässigkeit und Benutzbarkeit, finden Sie in der Norm ISO 9126.

Darüber haben sich schon diverse erfahrene Leute Gedanken gemacht und ihre Erkenntnisse zu verschiedenen Regeln, Leitsätzen und Prinzipien vereinigt. Beim objektorientierten Design bietet es sich an, aus diesem Erfahrungsschatz zu lernen und einiges davon zu kennen und einzuhalten.

3.5.1 Geheimnisprinzip nach Parnas

Gemäß dem ***Geheimnisprinzip*** (***Information Hiding***) sollten Klienten einer Komponente zu deren Nutzung keine Kenntnis über die internen Details besitzen (müssen). Schon 1972 wurde das Prinzip durch Parnas formuliert, damals noch für Module. Auf die heutige Zeit und Java übertragen, bedeutet dies, dass jede Komponente (Klasse, Package) eines Programms ihre Implementierungsdetails vor anderen Komponenten verbergen sollte. Daraus ergibt sich die Forderung, nur

diejenigen Bestandteile einer Komponente nach außen zugänglich zu machen, die für andere Komponenten zur Zusammenarbeit wirklich relevant sind. Für Klassen erreicht man dies, indem das Objektverhalten lediglich über öffentliche Business-Methoden bereitgestellt wird. Somit kann der Objektzustand von außen nur über diese Business-Methoden modifiziert und abgefragt werden. Für Attribute und Methoden ermöglicht eine stark eingeschränkte Sichtbarkeit, Implementierungsdetails möglichst geheim zu halten.

Befolgt man das Geheimnisprinzip, so kann man oftmals die einzelnen Bestandteile einer Software als Blackbox mit definierter Schnittstelle ansehen.

Bewertung

Wenn man das Geheimnisprinzip anwendet, so …

+ entsteht ein eher lose gekoppeltes System mit einer guten Modularisierung.
+ sollten sich einzelne Teile leichter unabhängig voneinander testen lassen. Auch die Anzahl der benötigten Tests reduziert sich, da Aufrufe nur über die öffentliche Schnittstelle erfolgen und Zustandsänderungen besser kontrolliert werden können.
+ können Details der Implementierung ohne Rückwirkungen auf Nutzer geändert werden. Beispielsweise können Daten entweder als Attribut gehalten, bei Bedarf berechnet oder aus einer externen Quelle gelesen werden.[29]
+ erleichtert dies das Verständnis und die Nachvollziehbarkeit. Allein basierend auf der öffentlichen Schnittstellen sollte die Funktionalität ermittelbar sein.
+ lassen sich neue Subklassen leichter implementieren, da in diesen für gewöhnlich nur wenige Anpassungen erfolgen müssen.
- ist ein minimaler Mehraufwand zur Implementierung von Zugriffsmethoden zur Datenkapselung und für deren Aufruf notwendig.

3.5.2 Law of Demeter

Beim ***Gesetz von Demeter*** (Law of Demeter, kurz: ***LoD***) geht es darum, die Kopplung auf ein verständliches und wartbares Maß zu reduzieren. Da das noch recht abstrakt ist, betrachten wir zunächst einige Negativbeispiele, die Verstöße gegen das LoD zeigen.

Negativbeispiele

Jeder kennt sicher Aufrufkonstrukte, die mehrere Methodenaufrufe wie folgt mithilfe der .-Notation miteinander verknüpfen:

```
getPreferencesService().getDimension(MAIN_WINDOW_ID).setWidth(700);
```

```
if (getCommandProcessor().getPool().getSize() >= MAX_POOL_SIZE)
{
    // warning
}
else
{
    // process command
}
```

Dieses Beispiel zeigt, dass das eigene Objekt durch die Aufrufe recht tief in die Interna anderer Objekte eingreift bzw. diese abfragt. Dabei werden diverse Annahmen getroffen, etwa darüber, dass die Komponenten alle zugreifbar und korrekt initialisiert sind. Weil man dies nicht immer voraussetzen kann, sollten gegebenenfalls `null`-Prüfungen vor den Zugriffen erfolgen. Diese beeinträchtigen die Lesbarkeit, weil sie den Sourcecode aufblähen und somit schwieriger nachvollziehbar und schlussendlich schlechter wartbar machen. Es gibt aber noch einen viel schwerwiegenderen Nachteil bei der Missachtung des LoD: Beim Einsatz der gezeigten Aufrufketten erfordern Änderungen an den Details genutzter Klassen nahezu zwangsläufig auch Änderungen in der eigenen Klasse. Schauen wir uns dies genauer an und abstrahieren dazu ein wenig und gehen von folgendem Aufruf aus:

```
getObjA().getObjB().methodC();
```

Das sieht noch recht harmlos aus, aber betrachten wir die Konsequenzen möglicher Änderungen. Es gibt Probleme, wenn …

- etwas im Design geändert wird und das Objekt `objB` nicht mehr durch Objekt `objA` bereitgestellt wird.
- die Methode `methodC()` verändert, umbenannt oder in eine andere Klasse verschoben oder gelöscht wird.

All dies erfordert auch Modifikationen an der eigenen Klasse. Das kann aufwendig werden und sowohl die Wartbarkeit als auch die lose Kopplung beeinträchtigen.

Die Auswirkungen sind besonders störend, wenn die Aufrufketten über Systemoder Package-Grenzen hinweg gehen. Möglicherweise kann man die dadurch referenzierten Klassen nicht anpassen oder die Implementierer dieser Klassen wissen von Ihnen als Nutzer gar nichts. Schauen wir nun darauf, wie die Einhaltung des Gesetzes von Demeter uns vor derartigen Problemen schützen soll.

Regeln beim Gesetz von Demeter

Das Gesetz von Demeter ist unter dem plakativen Begriff »***Don't talk to strangers***« bekannt und definiert Regeln zur Gestaltung von Aufrufen. Es wird gefordert, möglichst nur diejenigen Methoden oder Attribute anderer Klassen in Aufrufen zu nutzen, die direkt (ohne Indirektionen) bekannt und zugreifbar sind. Nutze nur …

1. Methoden der eigenen Klasse,

2. Methoden von Objekten, die als Parameter übergeben werden,

3. Methoden von Objekten, die die Klasse selbst erzeugt, oder

4. Methoden assoziierter Klassen.

Neben diesen Regeln hilft zur Anschauung folgende Abbildung 3-33 mit drei Objekten. Das eigene Objekt `Me` referenziert ein Objekt `A` und dieses wiederum ein Objekt `B`. Gemäß dem LoD darf das eigene Objekt nur Methoden aus Objekt `A` aufrufen, nicht jedoch solche von Objekt `B`.

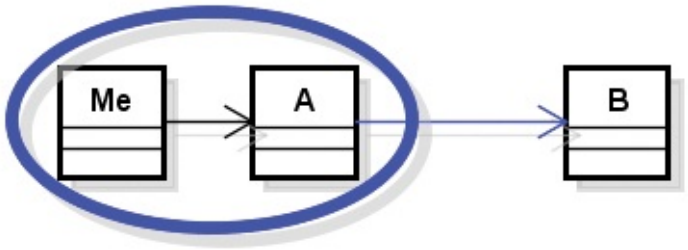


Abbildung 3-33 *Law of Demeter*

Beispiel für die Befolgung der Regeln

Um ein Gespür für die Regeln des LoD zu bekommen, betrachten wir eine Klasse `Person`, die ein `Address`-Objekt erzeugt und ein optionales `Company`-Objekt referenziert. Im Listing sind verschiedene Methodenaufrufe markiert, die die obigen Regeln einhalten. Beispiele für Verstöße sind in Kommentarzeilen dargestellt:

```
// Getter und Setter aus Gründen der Übersichtlichkeit nicht gezeigt
class Person
{
    final String name;
    final boolean isAdult;
    int age;
    Address homeAddress = new Address()
    Company company = null;

    Person(final String name, final int age)
    {
        this.name = name;
        this.age = age;

        // Regel 1
        this.isAdult = isOlderThan(18);
    }

    // Regel 1
    boolean isOlderThan(final int desiredAge)
    {
        return getAge() > desiredAge;
    }

    boolean sameAge(final Person other)
    {
        // Regel 2
        return getAge() == other.getAge();
    }

    boolean livesIn(final City city)
```

```
    {
        // Regel 3
        return getAddress().getCity().equals(city);
        // Verstoß: getAddress().getCity().getZipCode() == city.getZipCode()
    }

    boolean isManager()
    {
        // Regel 4
        return (company != null && company.isInManagingPosition(name));
        // Verstoß: company.getStuffMembers().isManager(name)
    }
}
```

Bewertung

Wenn man das Gesetz von Demeter einhält, so …

+ entsteht ein eher lose gekoppeltes System mit wenigen Abhängigkeiten, was die Wiederverwendbarkeit erleichtert.
+ lassen sich einzelne Klassen leichter ändern und diese Änderungen pflanzen sich mit geringerer Wahrscheinlichkeit im System fort.
+ lassen sich einzelne Klassen leichter und unabhängig von anderen testen.
- müssen zum Teil weitere Methoden in das Interface der eigenen Klasse aufgenommen werden, weil diverse Methoden nun nicht mehr über Indirektionen angesprochen werden dürfen.

3.5.3 SOLID-Prinzipien

SOLID ist ein Akronym und beschreibt fünf von Robert C. Martin vorgestellte Prinzipien guten OO-Entwurfs, die sich insbesondere mit Design beschäftigen:

- **S** – SINGLE RESPONSIBILITY PRINCIPLE – Jede Klasse sollte möglichst nur genau für eine Aufgabe zuständig sein.
- **O** – OPEN CLOSED PRINCIPLE – Dieses Prinzip besagt, dass Klassen offen für Erweiterungen sein sollen, aber geschlossen gegenüber Änderungen.
- **L** – LISKOV SUBSTITUTION PRINCIPLE – Hierbei geht es um Ersetzbarkeit von Basisklassen durch Subklassen.
- **I** – INTERFACE SEGREGATION PRINCIPLE – Entwerfe Interfaces, sodass sie gut für Klienten passen. Bevorzuge mehrere feingranulare Interfaces gegenüber einem oder wenigen umfangreichen Interfaces.
- **D** – DEPENDENCY INVERSION PRINCIPLE – Vermeide direkte Abhängigkeiten auf konkrete Klassen.[30]

Single Responsibility Principle

Das ***Single Responsibility Principle*** (SRP) besagt, dass eine Klasse (möglichst) ***genau eine klar definierte Aufgabe erfüllen soll*** und es folglich nur einen (oder wenige Gründe) für Änderungen geben sollte.[31] Umgangssprachlich könnte man sagen, wer viele Dinge auf einmal tut, dem gelingen selten alle gut. Die Wahrscheinlichkeit ist recht hoch, dass keine der Aufgaben richtig erfüllt wird. Zudem besitzen komplexe Klassen oftmals zu viele und zum Teil unerwünschte Abhängigkeiten auf andere Klassen.

Wird das SRP eingehalten, so erzielt man (in der Regel) eine hohe Kohäsion, also einen hohen Zusammenhalt einer Klasse. Die Ausrichtung einer Klasse auf genau eine Funktionalität führt auch zu Orthogonalität. Damit meint man, dass Funktionalitäten ohne (größere) Nebenwirkungen einfach miteinander kombiniert werden können. Das erleichtert es, größere Systeme wartbar aus kleineren Bestandteilen zusammenzubauen.

Woran kann man aber erkennen, dass das SRP verletzt wird? Ein Indiz dafür ist, dass es schwerfällt, prägnante Namen für eine Methode oder eine Klasse zu finden oder dass dieser Name mehrere Verben oder Nomen enthält. Methoden besitzen dann etwa Namen wie `collectAndFilter()`, `selectAndUpdate()` oder `sortAndPrint()`. Außerdem ist die schiere Methodenlänge ein recht guter Indikator: Je länger die Methode, desto größer ist die Wahrscheinlichkeit für einen Verstoß gegen das SRP.

Open Closed Principle

Das ***Open Closed Principle*** (OCP) zielt auf die leichte Erweiterbarkeit und korrekte Kapselung sowie Trennung von Zuständigkeiten. Wenn man es extrem sehen möchte, ***sollte sich eine Klasse nach ihrer Fertigstellung nur noch dann ändern müssen, wenn komplett neue Anforderungen oder Funktionalitäten zu integrieren sind oder aber Fehler korrigiert werden müssen***. Dahingegen sollten Änderungen an der eigenen Klasse nicht dadurch erforderlich sein, dass sich andere Klassen ändern. Hier besteht auch ein Zusammenhang zu dem zuvor beschriebenen Gesetz von Demeter.

Schauen wir uns ein Beispiel an, um das OCP ein wenig besser zu verstehen. Dazu betrachten wir eine Spieleapplikation, die verschiedene Bonuselemente, wie Extraleben oder Zusatzausrüstungen, als Anreiz anbietet. Ihre Aufgabe als Entwickler ist es nun, ein neues Level zu gestalten und dort neue Arten von Bonuselementen zu integrieren. Wünschenswert wäre es, wenn dies möglichst einfach realisierbar wäre, etwa wenn man lediglich neue Klassen für die neuen, speziellen Bonuselemente erstellen müsste. Das wäre beispielsweise dann möglich, wenn die Bonuselemente alle ein gemeinsames Interface `IBonusElement` erfüllen oder eine (abstrakte) Basisklasse implementieren würden. In diesem Fall sollte die restliche Applikation

kaum oder im besten Fall gar nicht von Änderungen bzw. Erweiterungen der Bonuselemente betroffen sein. ***Eine gemeinsame Basis aus Interface und/oder abstrakter Basisklasse und der Möglichkeit zur Definition von Spezialisierungen sind eine Variante, das OCP umzusetzen***. Ein Verstoß gegen OCP besteht dann, wenn die beschriebene Erweiterung an diversen Stellen zur Änderungen führen würde.

Auch das Entwurfsmuster SCHABLONENMETHODE (vgl. Abschnitt 18.3.3) kann dabei helfen, das OCP zu realisieren. Dort gibt man gewisse Rahmenbedingungen vor, erlaubt aber Modifikationen an verschiedenen Sollbruchstellen.

Info: OPEN CLOSED PRINCIPLE im realen Leben

Ein Paradebeispiel aus der realen Welt für das OCP sind Hifi-Verstärker. Diese können jede beliebige Quelle verstärken, solange diese über Cinch-Buchsen angeschlossen werden kann. Auf diese Weise kann man alte Kassettenrekorder, MiniDisc-Player, CD-Player, aber auch Blu-Ray-Player anschließen und deren Ton wiedergeben.

Im Zuge der Digitalisierung musste dann eine weitere Schnittstellenklasse in Form optischer bzw. koaxialer Anschlüsse geschaffen werden. Deutlich später stellte die Erweiterung auf Heimkino-Surround-Ton eine derart massive Änderung dar, dass wiederum eine neue Schnittstelle bzw. eine Erweiterung notwendig war und HDMI als neue Schnittstelle Einzug hielt. In allen Fällen wurde aber das OCP befolgt: Eine neue Schnittstelle entsprach einer neuen Anforderung und führte einmalig zu einer Änderung. Danach konnten damit alle kompatiblen Geräte angeschlossen werden. So ist es auch über 25 Jahre alten Verstärkern möglich, den Ton von DVD oder Blu Ray wiederzugeben, obwohl diese Technologien zum Herstellungszeitpunkt des Verstärkers noch lange nicht existierten.

Liskov Substitution Principle

Beim ***Liskov Substitution Principle*** (LSP) geht es um die Ersetzbarkeit und die Einhaltung der »is-a«-Beziehung. Damit ist gemeint, dass eine ***Instanz einer Subklasse überall dort problemlos genutzt werden können sollte, wo eine Instanz der Basisklasse zum Einsatz kommt***. Es geht darum, dass ein Nutzer ein erwartungskonformes Verhalten erhält, also dass ein von Subklassen in Methoden realisiertes Verhalten mit demjenigen der Basisklassen kompatibel sein muss.

Weil das kompliziert klingt (und es teilweise auch ist), betrachten wir ein Beispiel und nutzen dazu erneut grafische Figuren. Die Basisklasse `BaseFigure` bildet die Grundlage für Spezialisierungen wie Kreise, Rechtecke usw. Das Zeichnen sollte ohne Beachtung des konkreten Figurentyps in etwa wie folgt möglich sein:

```
figuresToDraw.add(new Circle(60, 70, 80));
figuresToDraw.add(new Rectangle(50, 50, 100, 200));

// …
```

```
for (final BaseFigure figure : figuresToDraw)
{
    figure.draw(graphicsContext);
}
```

Das gewünschte und erwartete Verhalten scheint recht natürlich. Ob dieses auch erreicht wird, hängt jedoch stark von den konkreten Subklassen ab – hier etwa, ob die Methode `draw()` auch das tut, was man anhand des Namens erwartet. Ein Verstoß bestünde etwa darin, einfach nichts zu zeichnen oder eine Datenbankabfrage auszuführen, also Dinge, die gegen die natürliche Erwartungshaltung bei Aufruf von `draw()` (massiv) verstoßen.

LSP für Eingabeparameter, Rückgabewerte und Exceptions

Das LSP gilt auch für die Typen von Rückgabewerten und Exceptions. Dazu nehmen wir folgende Hierarchie von Exceptions, Rückgabewerten und Klassen an:

```
class CalculationException extends Exception
{
    // …
}
class SpecialCalculationException extends CalculationException
{
    // …
}

class BaseFigure
{
    Number calcArea() throws CalculationException
    {
        return new Double(getWidth()* getHeight());
    }
}

class Polygon extends BaseFigure
{
    // Speziellere Rückgabe (Double extends Number) und speziellere Exception
    Double calcArea() throws SpecialCalculationException
    {
        // Bewusst, um gleich ein Problem zu zeigen
        return null;
    }
}
```

Wir erkennen Folgendes: Beim Überschreiben von Methoden dürfen Subklassen als Rückgabe einen spezielleren Typ zurückliefern als die Basisklasse. Auch dürfen spezifischere Exceptions ausgelöst werden (Details dazu finden Sie in Abschnitt 3.6 bei der Besprechung verschiedener Formen der Varianz). Weil dies unspektakulär klingt, betrachten wir nun einen möglichen Nutzer in Form der folgenden Klasse `Client`:

```
class Client
{
    void doSomethingWithFigure(final BaseFigure figure)
    {
```

```
        final Number result = figure.calcArea();
        // NullPointerException für Polygon
        final double area = result.doubleValue();
        // …
    }
}
```

Ebenso wie für das Zeichnen durch Aufruf der Methode `draw()` sollte auch für die Berechnung der Fläche durch die Methode `calcArea()` keine Unterscheidung der Figurentypen notwendig sein. Zur Demonstration der Verletzung des LSP gibt die Klasse `Polygon` jedoch statt einer Zahl den Wert `null` zurück, was beim Aufrufer zu einer `NullPointerException` führt. Ähnliche Probleme diskutieren wir später als BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.6.[32]

Besonderheiten in Klassenhierarchien

Im vorangegangenen Beispiel und bei Beachtung der Voraussetzungen einer »is-a«-Beziehung scheint das LSP nicht ganz so schwierig einzuhalten zu sein. Es gibt jedoch einige Beispiele aus der OO-Modellierung, wo gemäß der normalen Logik eine »is-a«-Beziehung gilt, bei deren Realisierung als Klassenhierarchie es aber Probleme gibt – insbesondere auch mit dem LSP. In Abschnitt 3.3.1 wurden am Beispiel einer Simulationssoftware für Lebensräume und verschiedene Tierarten einige Problemfälle der Vererbung aufgezeigt, etwa eine Basisklasse `Bird` mit der Methode `fly()` sowie den Subklassen `Ostrich` und `Penguin` für Sträuße bzw. Pinguine. Beides sind ganz offensichtlich Vogelarten und somit gilt die »is-a«-Beziehung, die eine Spezialisierung ausdrückt. Jedoch können diese speziellen Vogelarten nicht fliegen und somit lässt sich auch die Methode `fly()` nicht sinnvoll implementieren. Bleiben als möglichen Auswege, entweder die Methode leer zu implementieren oder aber eine Exception auszulösen.

Aber selbst bei (scheinbar) einfachen Vererbungsbeziehungen können sich Verstöße gegen das LSP ergeben, etwa bei der Abbildung der Beziehungen von Rechteck und Quadrat sowie von Ellipse und Kreis mithilfe von Vererbung. Mathematisch betrachtet sind ein Quadrat und ein Kreis jeweils eine Spezialform eines Rechtecks bzw. einer Ellipse. Somit könnte man beim objektorientierten Programmieren auf die (ansonsten meistens sinnvolle) Idee kommen, die reale Welt in Form einer analogen Klassenhierarchie abzubilden. Versuchen wir dies und schauen, warum es für OO leider so nicht funktioniert. Nehmen wir an, die Klasse `Square` wäre eine Subklasse von `Rectangle`, für die jeweils Breite und Höhe über Zugriffsmethoden veränderlich sind:

```
public class Rectangle
{
    private int width;
    private int height;

    public void setWidth(final int width)
    {
        this.width = width;
    }
```

```
    public void setHeight(final int height)
    {
        this.height = height;
    }

    public int calcArea()
    {
        return getWidth()* getHeight();
    }

    public int getHeight() { return this.height; }
    public int getWidth()  { return this.width;  }
}
```

Die Klasse `Square` erbt die Methoden zum Setzen der Breite und Höhe – allerdings besitzt ein Quadrat per Definition immer gleiche Werte für diese Seitenlängen. Damit müssen wir die `set()`-Methoden für Breite und Höhe anpassen, sodass sie immer beide Seitenlängen setzen. Dazu führen wir eine Methode `setSideLength(double)` ein, die zum Setzen der jeweiligen Werte die Methoden der Basisklasse aufruft. Die Implementierung geschieht folgendermaßen:

```
public class Square extends Rectangle
{
    public void setSideLength(final int sideLength)
    {
        // gleiche Seitenlänge sicherstellen
        super.setWidth(sideLength);
        super.setHeight(sideLength);
    }

    public void setWidth(final int width)
    {
        setSideLength(width);
    }

    public void setHeight(final int height)
    {
        setSideLength(height);
    }
}
```

Eine nutzende Klasse, etwa ein Unit Test, könnte nun prüfen, ob für Rechtecke nach dem Setzen unterschiedlicher Werte für Breite und Höhe z. B. den Werten 5 und 10 auch 50 als Fläche berechnet wird:

```
@Test
public void testAreaCalulaction()
{
   rectangle.setWidth(5);
   rectangle.setHeight(10);

   assertEquals(50, rectangle.calcArea());
}
```

Sofern die Variable vom Typ `Rectangle` ist, wird der Test bestanden. Würde man den Test allerdings mit einem `Square`-Objekt als Subtyp von `Rectangle` ausführen, bekäme man als Fläche entweder 25 oder 100, je nachdem, ob man Breite

oder Höhe als Zweites setzt.

Das Ergebnis der Flächenberechnung für eine Instanz von `Square` widerspricht aber der Erwartungshaltung an die Berechnung für eine Instanz vom Typ `Rectangle`. Somit verletzt die gezeigte Implementierung mit der Ableitung `Square extends Rectangle` das LSP.

Lösungsmöglichkeiten

Das klingt aber gar nicht gut und vor allem scheint die Lösung recht verzwickt. Tatsächlich kann man keine sinnvolle Lösung finden, sofern man eine Veränderlichkeit der Seitenlängen erlaubt. Wenn wir die Klassen jedoch in unveränderlichen Klassen umwandeln, was in diesem Fall lediglich das Entfernen der `set()`-Methoden und die Definition der Attribute als `final` erfordert, dann existiert auf einmal kein Problem mehr, weil bereits zum Konstruktionszeitpunkt für Quadrate gleiche Seitenlängen sichergestellt und nachträglich nicht geändert werden können. Das zeigt folgende Implementierung:

```
Square(final int seitenlaenge)
{
    super(seitenlaenge, seitenlaenge);
}

Rectangle(final int width, final int height)
{
    this.width = width;
    this.height = height;
}
```

Für diverse Anwendungsfälle ist der Verzicht auf Veränderlichkeit eine gute Lösung. Allerdings gibt es dabei noch etwas zu beachten.

Weitere Fallstricke

Nehmen wir an, wir würden mit den beiden Klassen folgenden Sourcecode schreiben:

```
final Square square = new Square(10);
final Rectangle rectangle = new Rectangle(10, 10);
```

Hier wird ein Quadrat vom Typ `Square` mit der Seitenlänge 10 und ein Rechteck vom Typ `Rectangle` mit der Breite 10 und der Höhe 10 erzeugt. Von seinen Abmessungen entspricht es einem Quadrat. Nun hat man semantisch zwei Quadrate der Seitenlänge 10, aber eins davon besitzt den Typ `Square` und das andere den Typ `Rectangle`. Puh! Was nun?

Zunächst einmal erkennt man, dass nicht alle vermeintlich trivialen Klassenhierarchien auch wirklich so einfach sind, wie sie scheinen, und dass deutlich mehr Fallstricke auf dem Weg zu einem gelungenen OO-Design lauern, als man sich gemeinhin so vorstellt.

Lösungsmöglichkeiten ohne Vererbung

Eine mögliche Lösung besteht darin, die Klasse `Square` zu entfernen und nur noch mit dem Typ `Rectangle` zu arbeiten. Um zu ermitteln, ob ein Rechteck quadratisch ist, fügen wir eine Methode `isSquare()` ein. Sofern die Klasse `Rectangle` als unveränderliche Klasse realisiert ist und ihre Breite sowie Höhe zum Konstruktionszeitpunkt erhält, kann man die Quadrateigenschaft auch schon dann bestimmen. Aber selbst wenn die Klasse `Rectangle` veränderlich gestaltet ist, lässt sich die Eigenschaft problemlos dynamisch über eine Abfrage `getWidth() == getHeight()` ermitteln. Bei dieser Umsetzung liegt keine Verletzung von LSP vor, weil erstens nicht mehr mit Vererbung gearbeitet und zweitens eine konsistente Realisierung der Quadrateigenschaft umgesetzt wird.

Interface Segregation Principle

Die Kernaussage des ***Interface Segregation Principle*** (ISP) ist, dass Benutzer einer Klasse eine ***möglichst spezifische, auf die jeweilige Aufgabe oder den Klienten zugeschnittene Schnittstelle bereitgestellt bekommen sollten***. Das impliziert, dass keine zu breite Schnittstelle (d. h. mit zu vielen oder auch semantisch nicht zusammengehörenden Methoden) angeboten wird, sondern möglichst eine solche, die genau den Anforderungen eines möglichen Benutzers entspricht.

Oftmals sieht man in der Praxis eher zu breite oder zu unspezifische Interfaces,[33] die folglich fast immer auch Funktionalität anbieten, die ein spezifischer Benutzer nicht benötigt, etwa wie folgt:

```
interface IUniversalFileCustumerAndPizzaService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;

    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);

    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

Als Abhilfe kann man die Schnittstelle einer Klasse über mehrere Interfaces beschreiben und so der Überfrachtung und einer schwierigeren Benutzbarkeit entgegenwirken. Für das vorherige Beispiel ließe sich das Interface etwa in folgende drei semantisch zusammengehörende Einzelinterfaces aufsplitten:

```
interface IFileService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;
}

interface ICustomerService
{
```

```
    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);
}

interface IPizzaService
{
    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

Bei einer Aufteilung sollte das Ziel jedoch nicht alleine darin bestehen, lediglich eine sehr feine Granularität der entstehenden Schnittstellen zu erreichen. Denn ansonsten bestünde eine Schnittstelle im Extremfall nur noch aus einer Methode. Das ist sogar möglicherweise in Ordnung, sofern diese eine Methode tatsächlich auch eine semantische Einheit für sich bildet. Im Beispiel ist das für die Methode `orderPizza()` und das Interface `IPizzaService` gegeben. Gleiches gilt etwa für diverse Interfaces aus dem JDK, die eine spezifische Funktionalität anbieten, etwa `Runnable` bei Multithreading oder `ActionListener` zur Ereignisbehandlung in GUIs. Normalerweise sollte man bei Interfaces mit nur einer Methode jedoch genauer hinschauen, ob nicht eine Kombination thematisch zusammengehörender Interfaces für mehr Zusammenhalt und Klarheit sorgen kann.

Schlussfolgernd kann man festhalten, dass der Entwurf einer gelungenen Schnittstelle gar nicht so leicht ist. Es gilt, die »richtige« Granularität zu finden. Dazu gehört etwas Erfahrung, Fingerspitzengefühl und auch ein wenig Ausprobieren – insbesondere auch eine Betrachtung aus Sicht möglicher Nutzer.

Dependency Inversion Principle

Das ***Dependency Inversion Principle*** (DIP) empfiehlt, dass Klassen möglichst unabhängig von anderen konkreten Klassen sein sollen, indem eine Abhängigkeit von einer konkreten Klasse durch eine Referenz auf deren Schnittstelle aufgelöst wird. Oder kurz: ***Verwende möglichst Interfaces (bzw. abstrakte Klassen), um (konkrete) Klassen voneinander zu entkoppeln***. Bei diesem Prinzip geht es also um die Reduktion der Kopplung.

Wenn Klassen lediglich über Interfaces miteinander interagieren, so kann man bei Bedarf die konkrete Implementierung auswechseln, im Idealfall sogar ohne dass dies Rückwirkungen auf den Aufrufer besitzt. Ein recht einfaches Beispiel wäre etwa, gegen das Interface `List<E>` statt gegen die konkreten Realisierungen `ArrayList<E>` oder `LinkedList<E>` zu entwickeln.

Das Programmieren gegen Interfaces kann die Wartbarkeit von Software erhöhen, weil es zu einer loseren Kopplung führt. Jedoch ist ein massiver Einsatz von Interfaces auch nicht sinnvoll. Generell dürfen sich Klassen innerhalb eines Packages durchaus direkt kennen.[34] Über Package-Grenzen hinweg oder wenn es mehrere Spezialisierungen gibt, sind Interfaces häufig ein Gewinn.

Beispiel

Schauen wir uns ein Beispiel an und bemühen dazu wieder den PizzaService. Bei der Ausführung des Bestellvorgangs werden verschiedene Daten das Kunden benötigt, dazu erfolgt ein Zugriff auf eine Datenbank, Zum Glück ist dies bereits durch ein sogenanntes Data Access Object (DAO) abstrahiert. Ebenso gibt es hin und wieder Rabattaktionen, die durch die Klasse `Discount` modelliert werden. Nachfolgend ist gezeigt, dass der `PizzaService` beide Abhängigkeiten durch Instanziieren der jeweiligen Klassen selbst auflöst (oder besser gesagt, selbst bereitstellt).

```
public class PizzaService implements IPizzaService
{
    private final Discount discount;
    private final CustomerDAO customerDAO;

    public PizzaService()
    {
        // Direkte Abhängigkeiten
        discount = new Discount();
        customerDAO = new CustomerDAO();
    }

    @Override
    public boolean orderPizza(final long customerId, final Pizza pizza)
    {
        final Customer customer = customerDAO.findById(customerId);
        final Receipt receipt = new Receipt(customer);

        final double price = discount.apply(pizza);
        receipt.addEntry(pizza, price);
        // …
    }
}
```

Durch die Konstruktion der Objekte im `PizzaService`-Konstruktor entstehen direkte Abhängigkeiten zu den beiden Klassen.[35] Was an diesen Abhängigkeiten ungünstig ist, wollen wir nachfolgend ein wenig genauer betrachten.

Analyse

Die eben vorgestellte Klasse `PizzaService` besitzt verschiedene Abhängigkeiten. Das ist im Besonderen ungünstig, wenn wir die Funktionalität der Klassen testen wollen. Das wäre aus folgenden zwei Gründen schwierig:

1. Da eine Fixierung auf die Datenquelle `CustomerDAO` besteht und diese Klasse die Kundendaten aus einer Datenbank liest, können wir keine Tests mit einem eng umrissenen und vordefinierten Testdatenbestand vornehmen. Außerdem sollten wir tunlichst keine Testdaten in diesen sensiblen Datenbestand einspielen, um Missverständnisse oder Fehlbestellungen oder sonstige Fallstricke zu vermeiden.
2. Die Auswirkungen verschiedener Rabattaktionen, etwa ein Einführungsangebot oder eine Happy Hour, sind schwierig zu testen, da momentan fix der durch die Klasse `Discount` realisierte Rabatt zur Preisreduktion genutzt wird.

Wie schon eingangs erwähnt, kann man die Abhängigkeiten durch Einführen von

Interfaces lösen. Das ist jedoch nur für zentrale Klassen und Bestandteile sinnvoll, nämlich genau für diejenigen, die Sollbruchstellen oder veränderliche, austauschbare Funktionalität anbieten. Im obigen Beispiel bildet die Rechnungserstellung mit der Klasse `Receipt` eine Ausnahme. Diese Funktionalität bleibt (vermutlich) stabil, daher fügen wir hier keine weitere Indirektion ein. Das könnte jedoch dann notwendig werden, wenn man verschiedene Zahlungs- und Rechnungsvorgänge unterstützen möchte.

Schritt 1: Interfaces einführen

Wir wollen hier zunächst die Rabattberechnung und die Zugriffe auf Kunden durch Interfaces unabhängig von konkreten Realisierungen halten. Wir führen dazu die Interfaces `IDiscountStrategy` und `ICustomerRepository` sowie konkrete Realisierungen davon ein.

```
interface IDiscountStrategy
{
   double apply(final Pizza pizza);
}

class XL_HalfPrice_Discount implements IDiscountStrategy
{
    // …
}

class HappyHourDiscount implements IDiscountStrategy
{
    // …
}

interface ICustomerRepository
{
    Customer findById(final long customerId);
}

class CustomerDAO implements ICustomerRepository
{
    // …
}
```

Mit diesen Interfaces und Klassen können wir in der Klasse `PizzaService` die Abhängigkeiten reduzieren, indem wir die Referenzen auf konkrete Klassen durch Interfaces ersetzen und so eine losere Kopplung erzielen.

Weitere Schritte

Insgesamt sind verschiedene weitere Transformationsschritte notwendig, die hier nicht gezeigt, aber nachfolgend kurz beschrieben werden. Als Erstes ersetzt man die direkten Abhängigkeiten durch Interfaces, konstruiert aber noch die Objekte innerhalb der Klasse. Damit ist noch nicht allzu viel erreicht, allerdings wird damit der Grundstein dafür gelegt, dass man Abhängigkeiten nun als Parameter »injizieren« kann. Als zweiten Schritt konstruiert der `PizzaService` die benötigten Objekte

nicht mehr selbst, sondern bekommt diese als Konstruktor- bzw. Methodenparameter übergeben. Man spricht in diesem Zusammenhang auch von ***Dependency Injection***, da Abhängigkeiten in die nutzende Klasse als Parameter übergeben werden. Abschließend kann man eine weitere Verbesserung vornehmen, indem die Rabattstrategie vom Typ `IDiscountStrategy` statt bereits an den Konstruktor erst später an die Methode übergeben wird, die diese Referenz benötigt und damit die Berechnung durchführt. Mithilfe des beschriebenen Vorgehens lässt sich die Methode und deren Funktionalität mit unterschiedlichen Strategien einfacher testen. Gemäß diesen Überlegungen ergibt sich folgende Implementierung der Klasse `PizzaService`, die nun dem DIP folgt. Zudem müssen wir die ursprüngliche `orderPizza()`-Methode so umgestalten, dass sie die im Listing gezeigt neue Variante nutzt:

```
public class PizzaService implements IPizzaService
{
    private final ICustomerRepository customerRepository;

    // Konstruktor-Injektion
    public PizzaService(final ICustomerRepository customerRepository)
    {
        this.customerRepository = customerRepository;
    }

    // Method-Injektion
    public boolean orderPizza(final long customerId, final Pizza pizza,
                              final IDiscountStrategy discountStrategy)
    {
        final Customer customer = customerRepository.findById(customerId);
        final Receipt receipt = new Receipt(customer);

        final double price = discountStrategy.apply(pizza);
        receipt.addEntry(pizza, price);
        // …
    }

   // …
}
```

Fazit

Neben den SOLID-Prinzipien haben wir auch noch das Gesetz von Demeter sowie das Geheimnisprinzip nach Parnas kennengelernt. Wenn wir uns beim Programmieren hin und wieder an diese erinnern und auch befolgen, werden wir in Zukunft bessere und leichter erweiter- und wartbare Designs produzieren.

Hinweis: Dependency Injection

Zur Erhöhung der Programmstabilität, Wartbarkeit und Austauschbarkeit einzelner Komponenten ist es ein erstrebenswertes Designziel, Software möglichst modular zu gestalten. Dazu sollten die einzelnen Komponenten lose miteinander gekoppelt sein und sich die konkreten Ausprägungen nur über Interfaces referenzieren.

Allerdings entsteht schnell das Problem, die Komponenten miteinander zu verbinden. Denn irgendeine Programmstelle muss die referenzierten Klassen ja erzeugen. Der Aufbau des Objektgraphs wird zunehmend komplexer, wenn viele Komponenten miteinander zu verbinden sind und dabei gewisse Reihenfolgen eingehalten werden müssen. Das ist immer dann notwendig, wenn eine Komponente auf anderen Komponenten basiert und folglich erst dann initialisiert werden kann, wenn die anderen Bestandteile bereits konstruiert sind.

Im folgenden Negativbeispiel verwendet die Klasse `OwnClass` zur Aufbereitung von HTML-Ausgaben die Klasse `HtmlOutputGenerator` aus einem anderen Package (hier `HtmlModule`) . Nur beim Ermitteln der Referenz auf diese Klasse benötigt man konkrete Informationen über die Klasse und das Package, im weiteren Sourcecode wird dann lediglich gegen das Interface `IHtmlOutputGenerator` gearbeitet:

```
public class OwnClass
{
   private IHtmlOutputGenerator htmlOutputGenerator =
                                  new HtmlModule.HtmlOutputGenerator();

   public void createHtmlOutput()
   {
      htmlOutputGenerator.createOutput(…);
   }

   // …
}
```

Im Listing erfolgen zum Aufbau des Objektgraphen der benötigten Klasse in der eigenen Klasse. Das führt zu einer engen Kopplung. Im Sinne der losen Kopplung und guter Trennung von Zuständigkeiten ist genau das aber problematisch. Eine Möglichkeit, die Abhängigkeiten aufzulösen und den direkten Konstruktoraufruf zu vermeiden, besteht darin, eine zentrale Registrierung einzuführen. Diese ist für die Konstruktion und Verwaltung allgemein bereitzustellender Klassen zuständig. Das Ganze geschieht für die nutzenden Klassen jedoch transparent. Diese ermitteln lediglich eine Referenz auf den gewünschten Typ in etwa wie folgt:

```
public class OwnClass
{
   private IHtmlOutputGenerator htmlOutputGenerator =
                                 Registry.lookUp(IHtmlOutputGenerator.class);

   // …
}
```

Diese Art der Realisierung verbessert die Situation. Allerdings entsteht immer noch einiges an Sourcecode, der nur dazu dient, Objekte miteinander zu verbinden. Das gilt insbesondere, wenn die Konstruktion komplexer ist.

Nehmen wir an, die gezeigte Klasse würde innerhalb einer Laufzeitumgebung ausgeführt, die Dependency Injection unterstützt. Mithilfe einer Annotation `@Inject` könnte man beispielsweise der Laufzeitumgebung mitteilen, dass in das derart annotierte Attribut eine Referenz auf die gewünschte Klasse zu injizieren ist:

```
public class OwnClass
{
   @Inject
   private IHtmlOutputGenerator htmlOutputGenerator;

   // …
}
```

Das Thema Dependency Injection wird im Artikel »Inversion of Control Containers and the Dependency Injection pattern« von Martin Fowler besprochen. Diesen findet man unter `http://www.martinfowler.com/articles/injection.html`.

3.6 Formen der Varianz

Zum Verständnis der mit JDK 5 eingeführten kovarianten Ergebnistypen und einiger Besonderheiten beim Einsatz generischer Typen stelle ich im folgenden Abschnitt 3.6.1 zunächst verschiedene Formen der Varianz vor. Anschließend geht Abschnitt 3.6.2 konkret auf kovariante Rückgabewerte ein.

3.6.1 Grundlagen der Varianz

Die »is-a«-Beziehung und das Substitutionsprinzip gemäß dem zuvor besprochenen LSP besagen, dass an allen Stellen in einem Programm, an denen ein Basistyp verwendet wird, auch jeder Subtyp verwendet werden können sollte. Zur Verdeutlichung nutze ich in den nachfolgenden Beispielen eine einfache Klassenhierarchie `Sub extends Base`. Allgemein gilt, dass aufgrund der Vererbungsbeziehung die technische Ersetzbarkeit von Objekten der Basisklasse `Base` durch Objekte der abgeleiteten Klasse `Sub` möglich ist. Eine semantische Ersetzbarkeit erhält man allerdings nur, wenn die »is-a«-Beziehung und das LSP eingehalten wird.

Vorbetrachtungen

Anhand einer einfachen Methodendefinition `method(Base)` und eines Aufrufs dieser Methode mit einer Eingabe vom Typ `Sub` möchte ich einige Vorbetrachtungen anstellen, die das Verständnis der Formen der Varianz erleichtern:

```
// Definition
Base method(final Base in)
{
    return new Sub(in);              // Speziellerer Rückgabetyp -> Kovarianz
}

// Aufruf
final Base out = method(new Sub()); // Speziellerer Eingabetyp ->
Kontravarianz
```

Für Methodenparameter gilt: Ein Eingabetyp darf immer spezieller sein als der von der Methode in der Signatur geforderte Eingabetyp, da in der Methode lediglich auf die Attribute und Methoden der Basisklasse zugegriffen werden kann und diese auch in der Subklasse vorhanden sind. Für Rückgabetypen gilt: Es darf ein speziellerer Typ als erwartet zurückgeliefert werden, da ein verwendender Aufrufer nur die Attribute und Methoden der Basisklasse kennt.

Mit dem Begriff ***Varianz*** werden unter anderem die beiden gezeigten Typabweichungen Ko- und Kontravarianz beschrieben. Für beide gilt in diesem Beispiel, dass der Eingabeparameter sowie der Rückgabewert zur Kompilierzeit jeweils vom Typ `Base` sind, aber zur Laufzeit der Typ `Sub` verwendet wird. Das ist durch gestrichelte Pfeile für die Typen des Eingabeparameters bzw. des Rückgabewerts in Abbildung 3-34 angedeutet.

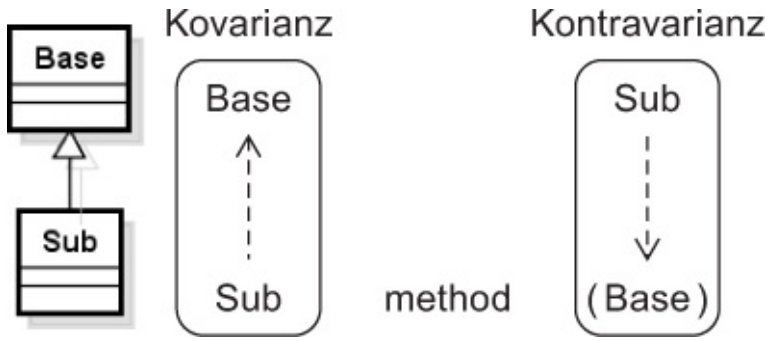


Abbildung 3-34 *Grundlagen der Varianzformen*

Beim Rückgabewert der obigen Methode `Base method(Base)` deutet der gestrichelte Pfeil die Typabweichung an. Hier wird eine Instanz vom Typ `Sub` an eine Referenz vom Basistyp `Base` übergeben. Damit folgt die Typabweichung in ***gleicher*** Richtung wie bei der Typhierarchie. Diesen Sachverhalt drückt die sogenannte ***Kovarianz*** aus. Beim Eingabeparameter wird mit `Sub` ein speziellerer Typ übergeben und der gestrichelte Pfeil deutet an, dass eine Übergabe ***entgegen*** der Typhierarchie erfolgt. Man spricht von ***Kontravarianz***. Ändert sich der Typ nicht, so spricht man von ***Invarianz***.

Kovarianz: Auswirkungen auf Arrays

Arrays besitzen in Java kovariantes Verhalten. Die Kovarianz besagt in diesem Fall Folgendes: Weil `String` ein Subtyp von `Object` ist, gilt das auch für Arrays dieser Typen: Ein `String[]` stellt in Java somit einen Subtyp von `Object[]` dar. Verdeutlichen wir uns dies anhand des UML-Diagramms in Abbildung 3-35.

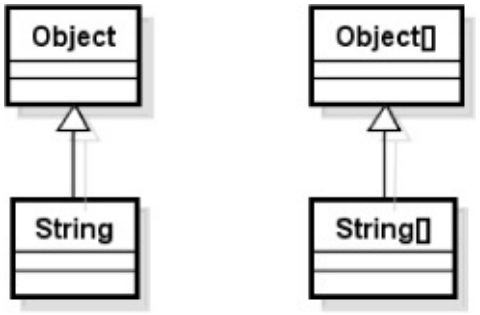


Abbildung 3-35 *Kovarianz für Array-Typen*

Aufgrund der Kovarianz ist eine Zuweisung eines `String[]` an ein `Object[]` möglich. Das scheint ganz selbstverständlich und bestimmt haben Sie derartige Zuweisungen von Arrays beliebiger Typen an ein `Object[]` schon des Öfteren in eigenen Programmen genutzt, etwa folgendermaßen:

```
final Object[] messages = new String[] { "Arrays", "sind", "kovariant" };
```

Obwohl diese Art der Zuweisung absolut praktisch ist und natürlich in der Handhabung erscheint, birgt das Ganze zur Laufzeit gewisse Fallstricke, die wir nun betrachten.

Fallstricke durch Kovarianz

Die Kovarianz ermöglicht, dass beliebige, nicht primitive Array-Typen an eine Referenz vom Typ `Object[]` zugewiesen werden können, etwa wie oben ein `String[]`. Danach könnte die Referenz auch auf ein `Integer[]` und später wieder auf ein `String[]` verweisen. Aufgrund des Basistyps `Object` für die einzelnen Elemente können im `Object[]` auch beliebige Typen gespeichert werden. Ohne einen Blick auf die Zuweisung könnte es etwa zu folgender (versehentlicher) Fehlverwendung kommen, wenn man nur die obige Variable `messages` sieht und dort ein `Message`-Objekt wie folgt speichern möchte:

```
// Typfehler, weil Message nicht kompatibel zu String ist
messages[1] = new Message("Typfehler durch Kovarianz!");
```

Derartige Typinkompatibilitäten lassen sich rein allein auf Basis von Typinformationen zur Kompilierzeit nicht unterbinden. ***Daher kann für Arrays (insbesondere vom Typ `Object[]`) Typsicherheit zur Kompilierzeit nicht sichergestellt werden und erfordert eine zusätzliche Prüfung zur Laufzeit***. Deshalb speichern Arrays auch zur Laufzeit noch Typinformationen über die in ihnen gespeicherten Elemente. Wird, wie im Listing gezeigt, eine Zuweisung inkompatibler Typen vorgenommen, hier statt eines Werts vom Typ `String` eine Zuweisung eines `Message`-Objekts, so werden `java.lang.ArrayStoreExceptions` ausgelöst.

Hintergundwissen: Automatische Typerzeugung bei Arrays

Bei der Angabe eines Arrays im Sourcecode wird durch den Compiler dynamisch ein neuer Typ erzeugt, falls es diesen noch nicht gibt. Beim Kompilieren der Anweisung

```
final Object[] infos = new String[] { "Arrays", "sind", "kovariant" };
```

entstehen zwei neue Typen: `Object[].class` und `String[].class`.

Einfluss von Varianz beim Überschreiben von Methoden

Nachdem wir bereits ein gewisses Verständnis für die verschiedenen Formen der Varianz aufgebaut haben, möchte ich darstellen, wie sich Varianz auf die Definition und das Überschreiben von Methoden in einer Klassenhierarchie auswirkt.

Zur Veranschaulichung verwende ich die bereits gezeigte Typhierarchie `Sub extends Base`, die Eingabe- bzw. Rückgabetyp von Methoden einer Klassenhierarchie definiert. Die Klassenhierarchie besteht aus den zwei Klassen `BaseClass` und `SubClass` sowie der Vererbungsbeziehung `SubClass extends BaseClass`. Jede dieser Klassen definiert eine Methode `method()`. Je nach Varianzform nutzt diese einen Eingabeparameter bzw. Rückgabewert vom Typ `Base` bzw. `Sub`. Das kann man sich am besten mithilfe eines kleinen Modells in UML-Schreibweise verdeutlichen. In Abbildung 3-36 ist dies dargestellt.

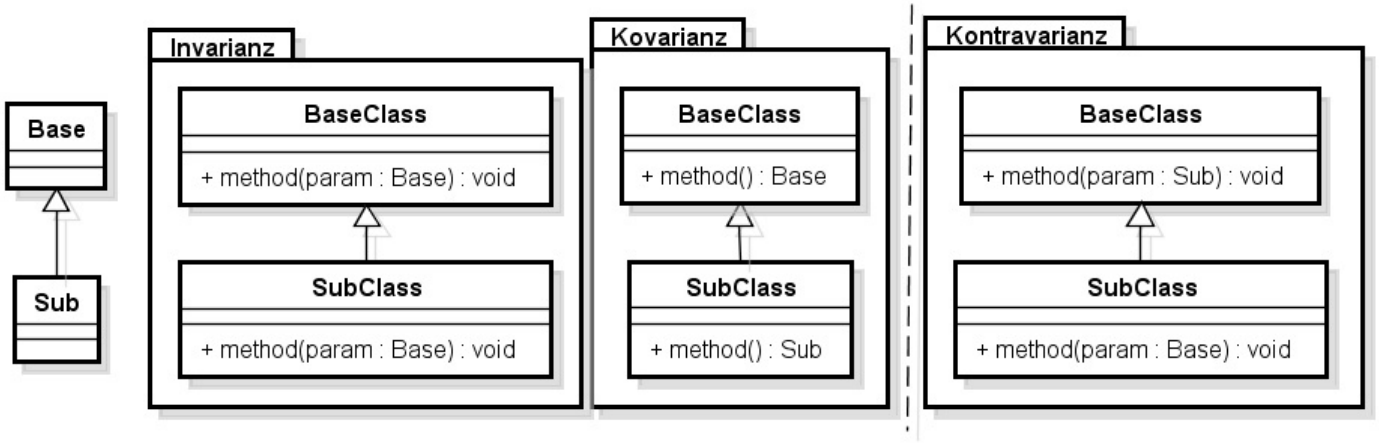


Abbildung 3-36 *Varianzformen und Vererbung*

Die gezeigten Varianzformen führen nicht immer zum Überschreiben von Methoden. In der Java Language Specification[36] (JLS) ist definiert, dass ein Überschreiben nur möglich ist, wenn eine Methode einer Subklasse dieselbe Signatur wie die Methode der Basisklasse besitzt. Wir erkennen daher Folgendes:

- Beim ***invarianten Überschreiben*** sind in einer Basisklasse und einer davon abgeleiteten Klasse der eingesetzte Typ eines Methodenparameters ***identisch***.
- Beim ***kovarianten Überschreiben*** folgt die Vererbungshierarchie der Rückgabetypen und die der einsetzenden Klassen der ***gleichen Richtung***. Ab JDK 5 wird dies in Form kovarianter Rückgabewerte unterstützt (vgl. Abschnitt 3.6.2).
- Wenn die Typhierarchie der jeweiligen Methodenparameter ***entgegengesetzt zur Vererbungshierarchie*** der verwendenden Klassen läuft, so spricht man von ***Kontravarianz***. In Java findet hierdurch *kein Überschreiben* statt: Stattdessen wird die Methode *überladen* und existiert zusätzlich zu der geerbten Methode.

3.6.2 Kovariante Rückgabewerte

Nach diesen Grundlagen stelle ich im Folgenden kovariante Rückgabewerte anhand eines Beispiels vor. Denken wir uns dazu wieder ein Programm, das verschiedene grafische Elemente der Typen `CircleFigure`, `LineFigure` und `RectFigure` verwaltet: Diese Klassen erweitern eine gemeinsame Basisklasse `BaseFigure`. Nehmen wir an, diese Basisklasse besäße eine Methode `copy()`, um Elemente zu

kopieren:

```
public abstract class BaseFigure
{
    public abstract BaseFigure copy();

    // …
```

Jede konkrete Subklasse muss diese abstrakte Methode realisieren. Im Folgenden betrachten wir dies mit und ohne den Einsatz kovarianter Rückgabewerte.

Realisierungen vor JDK 5

Vor JDK 5 entspricht für Subklassen der Typ des Rückgabewerts in überschriebenen Methoden demjenigen aus der Basisklasse, hier also `BaseFigure`. Das ist unpraktisch, wenn man eine Referenz eines konkreten Subtyps besitzt und davon eine Kopie erstellen möchte. Nach der obigen Definition erhält ein Aufrufer lediglich eine Referenz auf den Basistyp `BaseFigure`. Um bei Bedarf dennoch Zugriff auf durch Subklassen bereitgestellte Funktionalität zu erhalten, muss daher immer ein expliziter Cast erfolgen – natürlich bevorzugt abgesichert durch einen Aufruf von `instanceof`. Wollte man beispielsweise ein Objekt vom Typ `CircleFigure` duplizieren und anschließend auf dessen spezifische Methoden, etwa den Radius des Kreises, zugreifen, so kann das nur wie folgt realisiert werden:

```
final BaseFigure typeUnsafeCopyOfCircle = circleFigure.copy();

if (typeUnsafeCopyOfCircle instanceof CircleFigure)
{
    final CircleFigure circleTypeCopy = (CircleFigure)typeUnsafeCopyOfCircle;

    final double radius = circleTypeCopy.getRadius();

    // …
```

Realisierungen mit JDK 5 und höher

Seit JDK 5 kann man ***kovariante Rückgabewerte*** beim Überschreiben nutzen, um den Applikationscode einfacher und besser lesbar zu gestalten. Im Beispiel kann die abgeleitete Klasse `CircleFigure` die Methode `copy()` mit einem spezielleren Rückgabetyp überschreiben als in der Basisklasse:

```
public class CircleFigure
{
   public CircleFigure copy() // Kovariante Rückgabe
   {
       // …
```

Abbildung 3-37 zeigt kovariante Rückgabetypen als UML-Klassendiagramm für die Spezialisierungen `CircleFigure`, `LineFigure` und `RectFigure`.

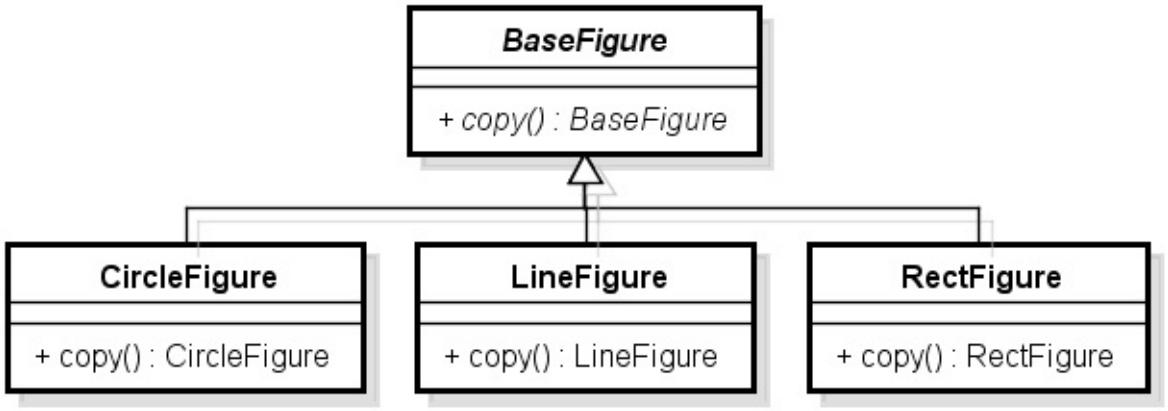


Abbildung 3-37 *Abstrakte Basisklasse und kovariante Rückgabewerte*

Betrachten wir nun den Einsatz für eine Referenzvariable vom Typ `CircleFigure`:

```
final BaseFigure baseFigure = new CircleFigure();
final CircleFigure circleFigure = new CircleFigure();

final CircleFigure circleCopy = circleFigure.copy(); //#1 Kovariante Rückgabe
final double radius = circleCopy.getRadius();

//final CircleFigure circleCopy2 = baseFigure.copy(); //#2 Compile-Error

final BaseFigure baseCopy = baseFigure.copy();
System.out.println(baseCopy.getClass().getSimpleName()); //#3 CircleFigure
```

Listing 3.4 *Ausführbar als* **'COVARIANTRETURNEXAMPLE'**

Dieser Sourcecode-Ausschnitt zeigt, dass Aufrufe mit kovarianter Rückgabe nur dann einen Subtyp zurückliefern, wenn die Variable auch zum Zeitpunkt des Kompilierens vom entsprechenden Typ ist – in diesem Beispiel, wenn das Original auch vom Typ `CircleFigure` ist (Variante #1).

Bei Variante #2 besitzt das Objekt `baseFigure` zwar den *Laufzeittyp* `CircleFigure`, aber den *Kompiliertyp* `BaseFigure`. Dafür würde zwar eine polymorphe Wahl der Methode erfolgen, es wird jedoch der kovariante Rückgabetyp zur Kompilierzeit nicht unterstützt. Dies führt zu folgender Fehlermeldung: »`Type mismatch: cannot convert from BaseFigure to CircleFigure`«.

Die Variante #3 mit der Anweisung `baseFigure.copy()` führt zu einer Kopie und einer Zuweisung an die Variable `baseCopy` vom Typ `BaseFigure`. Startet man das Programm COVARIANTRETURNEXAMPLE, wird der Klassenname "`CircleFigure`" ausgegeben. Daran erkennt man, dass tatsächlich ein `CircleFigure`-Objekt entsteht. Als Aufrufer kann man – sofern nicht ein expliziter Cast auf diesen Laufzeittyp erfolgt – aber nicht auf die spezifische Methode `getRadius()`, sondern nur auf die Methoden des Kompiliertyps `BaseFigure` zugreifen.

Fazit

Mit diesem Beispiel zu kovarianten Rückgabewerten beenden wir erst einmal die Vorstellung der Varianz und kommen nun zu generischen Typen. Beide Themen

spielen insbesondere im Zusammenhang mit den Containerklassen des Collections-Frameworks eine wichtige Rolle und werden dort in Abschnitt 5.4 nochmal aufgegriffen und ausführlicher erklärt.

3.7 Generische Typen (Generics)

Dieser Abschnitt stellt die Grundkonzepte der generischen Typen (kurz ***Generics***) vor. Dieses komplexe Thema wird hier nur so weit behandelt, wie dies hilfreich ist, um die Beispiele und Erklärungen in diesem Buch nachvollziehen zu können.

3.7.1 Einführung

Die Containerklassen wie Listen, Sets und Maps waren bis JDK 5 untypisiert. Somit können dort Objekte beliebigen Typs verwaltet werden. Jedoch sind derartige heterogene Container nur für wenige Anwendungsfälle wünschenswert. Viel öfter ist gewünscht, gleichartige Objekte, also diejenigen eines bestimmten Typs, zu speichern – man spricht von einer homogenen Zusammensetzung. Ohne das Sprachfeature Generics oder eine selbstgeschriebene Containerklasse kann man diese Forderung nur durch eine geeignete Namensgebung, etwa `personList`, ausdrücken, nicht aber vom Compiler sicherstellen. Seit JDK 5 ist die typsichere Definition von Containerklassen mithilfe von Generics ohne weiteren Implementierungsaufwand möglich. Es muss lediglich eine Typangabe bei der Definition einer Containerklasse erfolgen. Basierend darauf kann vom Compiler sichergestellt werden, dass nur gewünschte Typen gespeichert werden.

Herkömmliche, nicht generische Container

Um die Problematik nicht typsicherer Container zu rekapitulieren bzw. besser nachvollziehen zu können, betrachten als Beispiel die Datenspeicherung von `Person`-Objekten in einer `ArrayList` ohne Typangabe. Man spricht auch von einem sogenannten ***Raw Type***, weil dieser keine Typinformationen über den Inhalt besitzt. Somit sind die Zugriffsmethoden, etwa `Object get(int)` und `add(Object)`, alle mit Eingabeparametern oder Rückgabewerten des Typs `Object` definiert und es können Objekte beliebiger Typen verarbeitet werden. Zur Demonstration möglicher Auswirkungen wird in eine Liste von `Person`-Objekten bewusst auch ein `Dog`-Objekt eingefügt. Danach wird über die Liste iteriert, die Elemente werden ausgelesen und ausgegeben:

```
public static void main(final String[] args)
{
    // Achtung: Nur zur Demonstration auf Generics verzichtet
    final List personList = new ArrayList();
    personList.add(new Person("Max", new Date(), "Musterstadt"));
    personList.add(new Person("Moritz", new Date(), "Musterstadt"));
    personList.add(new Dog("Sarah vom Auetal"));
```

```
    for (int i = 0; i < personList.size(); i++)
    {
        // Explizite Typumwandlung notwendig zum Methodenaufruf
        final Person person = (Person) personList.get(i);
        System.out.println(person.getName() + " aus " + person.getCity());
    }
}
```

Listing 3.5 *Ausführbar als* **'OLDSTYLELIST'**

Der gezeigte Sourcecode kompiliert ohne Fehler, aber zur Laufzeit gibt es Probleme: Als Folge der expliziten Typumwandlung des Eintrags vom Typ `Dog` in eine Referenz vom Typ `Person` wird eine `ClassCastException` ausgelöst: Die Klasse `Dog` ist selbstverständlich keine Subklasse der Klasse `Person`.

Die Typumwandlung von `Object` auf `Person` ist erforderlich, damit ein Zugriff auf die Methoden `getName()` und `getCity()` der Klasse `Person` möglich wird, um den Namen einer Person und deren Wohnort ausgeben zu können. Aufgrund der Namensgebung `personList` erfolgt der Cast oder eine Typprüfung, da man davon ausgeht, dass die gespeicherten Elemente vom Typ `Person` sind. Wie das Beispiel zeigt, ist dies aber nicht garantiert und führt zu dem Fehler. Wie geht es also besser?

Info: Typsicherheit vor JDK 5

Zur Realisierung typsicherer Containerklassen musste man vor JDK 5 einiges an Aufwand treiben: Man konnte Vererbung oder aber Delegation nutzen. Beides besitzt gewisse Nachteile und ist fehleranfällig. Die Lösung mit Delegation ist zudem nicht kompatibel zu den Interfaces der Collections des JDKs. Nachfolgend deute ich die prinzipielle Umsetzung der jeweiligen Technik anhand der Realisierung der Methode `add(Object)` aus dem Interface `java.util.List<E>` an.

Realisierung mit Vererbung Zur Sicherstellung der Typkonformität müssen beim Einsatz von Vererbung in den überschriebenen Methoden jeweils Typprüfungen durchgeführt werden. Ist das übergebene Objekt vom erwarteten Typ, hier `Person`, so erfolgt ein Aufruf der geerbten Funktionalität. Ansonsten wird eine `IllegalArgumentException` ausgelöst, um den Typfehler zu signalisieren:

```
public class TypeSafePersonListDerived extends List
{
    public Object add(final Object obj)
    {
        if (!(obj instanceof Person))
            throw new IllegalArgumentException("unsupported type");

        super.add(obj);
    }
    // …
```

Realisierung mit Delegation Alternativ zur obigen Realisierung kann in eigenen Containerklassen eine `List`-Referenz als Attribut gehalten und per Delegation angesprochen werden. Man kann die Methoden mit gewünschter Signatur

definieren und insbesondere einen speziellen Typ nutzen, weil man eben das Interface `List` weder erfüllen muss noch kann. Letzteres gilt, weil dort keine Signaturen mit dem benötigten speziellen Typ, hier `Person`, existieren. In dieser fehlenden Interface-Kompatibilität besteht auch der große Nachteil der Variante mit Delegation: Zum einen fügt sich eine derart realisierte Klasse nicht in das Collections-Framework ein, und zum anderen wird häufig nur die gerade benötigte Teilmenge aus der im JDK-Container angebotenen Funktionalität realisiert. Nachfolgend zeige ich stellvertretend die Implementierung der Methode `add(Person)`:

```
public class TypeSafePersonListDelegation
{
    final List underlyingList = new ArrayList();

    public Person add(final Person obj)
    {
        underlyingList.add(obj);
    }
    // …
```

Typsichere, generische Container seit JDK 5

Wie im vorangegangene Infokasten »Typsicherheit vor JDK 5« beschrieben, waren für typsichere Container spezielle eigene Implementierungen nötig. Mit JDK 5 kann man praktischerweise Containerklassen durch die Angabe von Typparametern auf einen speziellen Typ festlegen. Dies geschieht mithilfe der Spitzen-Klammern-Notation, z. B. `ArrayList<Person>`. Basierend auf dieser Typangabe kann der Compiler bereits zur Kompilierzeit Typinkompatibilitäten aufdecken und dadurch sicherstellen, dass nur Objekte des gewünschten Typs (und auch Subtypen davon), hier `Person`, in den Container aufgenommen werden können.

Schauen wir nun auf die typsichere Definition einer Liste von `Person`-Objekten mithilfe von Generics:

```
public static void main(final String[] args)
{
    // Typsichere Definition mit Generics
    final List<Person> personList = new ArrayList<Person>();

    personList.add(new Person("Max", new Date(), "Musterstadt"));
    personList.add(new Person("Moritz", new Date(), "Musterstadt"));
    // personList.add(new Dog("Sarah vom Auetal")); // Compile-Error

    for (int i = 0; i < personList.size(); i++)
    {
        // Typsicheres Auslesen
        final Person person = personList.get(i);
        System.out.println(person.getName() + " aus " + person.getCity());
    }
}
```

Listing 3.6 *Ausführbar als* **'NEWSTYLELIST'**

Wie das Beispiel zeigt, lassen sich nunmehr wirklich nur `Person`-Referenzen[37] in die

so typisierte `ArrayList<Person>` aufnehmen und daraus auslesen. Bei Letzterem entfällt die zuvor notwendige explizite Typumwandlung und man erhält dann auch den gewünschten Typ, hier `Person`, statt wie früher den Typ `Object`.

Vereinfachungen bei Typangaben mit JDK 7

Der Einsatz von Generics erforderte vor JDK 7 etwas zusätzlichen Schreibaufwand durch die Angabe der Typinformation bei der Deklaration und deren Wiederholung bei der Definition etwa folgendermaßen:

```
final Map<String, Set<String≫ typeSafeMap = new HashMap<String,
Set<String≫();
```

Dies ist unleserlich und enthält Redundanzen, da die Typparameter bereits bekannt sind. Um diesem Manko zu begegnen, findet man in vielen Applikationen den Einsatz einer generischen Fabrikmethode (vgl. Abschnitt 18.1.2 zum Muster FABRIKMETHODE) wie dies in folgendem Listing mit der Methode `createTypeSafeHashMap()` gezeigt ist:

```
final Map<String, Set<String≫ typeSafeMap = createTypeSafeHashMap();

// Fabrikmethode
static <K,V> HashMap<K,V> createTypeSafeHashMap()
{
    return new HashMap<K,V>();
}
```

Statt der Definition einer Fabrikmethode kann man seit JDK 7 den sogenannten ***Diamond Operator*** '<>' zur Schreibweisenabkürzung wie folgt verwenden:

```
// Typ muss nicht wiederholt werden
final Map<String, Set<String≫ newJDK7StyleMap = new HashMap<>();
```

Definition eigener generischer Klassen

Wir wissen nun, wie man die Containerklassen des Collections-Frameworks typsicher gestaltet, doch oftmals ist es darüber hinaus wünschenswert, eigene typisierte Klassen erstellen zu können. Betrachten wir als einfaches Beispiel einen Datencontainer für Wertepaare beliebiger Typen.

Realisierung ohne Generics und sich daraus ergebende Probleme

Ohne Verwendung von Generics muss dieser Datencontainer zunächst ganz allgemein mit Referenzen vom Typ `Object` folgendermaßen realisiert werden:

```
public final class Pair
{
    private final Object first;
    private final Object second;
```

```
    public Pair(final Object first, final Object second)
    {
        this.first = first;
        this.second = second;
    }

    public final Object getFirst()        { return first; }
    public final Object getSecond()       { return second; }
}
```

Diese Realisierung bietet keine Einschränkung, welche Typen eingefügt werden können. Möchte man diese Klasse typsicher für diverse Typkombinationen ohne Einsatz von Generics bereitstellen, so müssten für jede gewünschte Typkombination jeweils eigene Klassen erstellt werden. Dies würde zu Sourcecode-Duplikation und zu einer Menge ähnlicher Klassen führen und zwar linear wachsend mit der Anzahl gewünschter Typkombinationen, wobei die Realisierung in den Typen der Attribute `first` und `second` entsprechend variiert. Daran erkennt man, dass es wenig sinnvoll ist, Datencontainer in dieser Weise auf spezielle Typen der Einzelelemente festzulegen, wenn man eine allgemeingültige Lösung sucht. Wie geht es also besser?

Realisierung mit Generics

Anstatt die Klasse `Pair` für jede benötigte Typkombination wieder erneut zu programmieren, wollen wir eine generische, für alle möglichen gewünschten Typkombinationen verwendbare Definition realisieren. Durch den Einsatz von Generics wird genau dies möglich. Wir definieren den Container `Pair` mit zwei formalen Typparametern `T1` und `T2`, die als Platzhalter für konkrete Typen beim Einsatz dienen, folgendermaßen:

```
public final class Pair<T1, T2>
{
    private final T1 first;
    private final T2 second;

    public Pair(final T1 first, final T2 second)
    {
        this.first = first;
        this.second = second;
    }

    public final T1 getFirst()        { return first; }
    public final T2 getSecond()       { return second; }
}
```

In der Regel werden als Typplatzhalter einzelne Zeichen verwendet, die bei Bedarf, wie hier, eine Nummerierung erhalten. Gebräuchlich sind die Kürzel `E` für Elemente von Containerklassen und `T` für beliebige Typen. Erst beim Einsatz dieser generischen Klasse werden die formalen Typparameter vom Compiler durch die im Sourcecode angegebenen konkreten Typen ersetzt.

Betrachten wir als Anwendungsfall die Speicherung und Verwaltung von Personen und deren Spitznamen. Mithilfe der vorgestellten Klasse `Pair` soll die Kombination aus Spitzname und Person modelliert werden. Dazu substituiert man

die beiden formalen Typparameter `T1` und `T2` durch die gewünschten Typen `String` und `Person`:

```
final Person wizard = new Person("Wizard", new Date(), "Kiel");
final Person mike = new Person("Mike", new Date(), "Bremen");

final Pair<String, Person> pair1 = new Pair<String, Person>("Dark", wizard);
final Pair<String, Person> pair2 = new Pair<String, Person>("Iron", mike);
```

Typeinschränkungen für Klassen

Teilweise möchte man bei der Definition einer generischen Klasse den erlaubten Typ eines zu ersetzenden Typparameters genauer spezifizieren. Das kann man durch folgende Notation erreichen:

```
public class GenericClass<T extends BaseType & Interface1 & Interface2>
```

Diese Angabe sichert folgende Eigenschaften für den Typ `T` ab:

- `T` muss die Basisklasse bzw. den Basistyp `BaseType` besitzen (oder selbst vom Typ `BaseType` sein).
- `T` implementiert außerdem die Interfaces `Interface1` und `Interface2`. Diese Bedingung ist auch erfüllt, wenn eine beliebige Basisklasse von `T` dies tut.

Vor dem ersten `'&'` können Typangaben verwendet werden, also sowohl der Name einer Klasse als auch der eines Interface. Nach dem `'&'` können lediglich Interfaces angegeben werden.

Typeinschränkungen für (statische) Methoden

Für Methoden kann auf ähnliche Weise eine Typeinschränkung erfolgen. Nehmen wir an, eine statische Hilfsmethode `doSomething()` soll Listen übergeben bekommen, die auf den Typ `BaseFigure` oder Subtypen davon festgelegt sind, etwa `List<RectFigure>`. Eine mögliche Schreibweise ist folgende:

```
public static void doSomething(final List<? extends BaseFigure> figures)
```

Alternativ kann man die Methode auch explizit als generisch definieren. Dazu wird der formale Typparameter vor dem Rückgabetyp in spitzen Klammern notiert:[38]

```
public static <T extends BaseFigure> void doSomething(final List<T> figures)
```

Analog kann man nicht statische Methoden mit einem Typparameter versehen, etwa wenn dieser abweichend vom Typparameter der Klassendefinition ist. Gebräuchlich ist aber vor allem die gezeigte Form für statische Methoden.

Die hier genutzte spezielle Notation `? extends` bzw. `T extends` mutet etwas merkwürdig an. Sie drückt Kovarianz aus, die vor allem für Containerklassen und Generics von Interesse ist und später im Detail in Abschnitt 5.4 besprochen wird.

3.7.2 Generics und Auswirkungen der Type Erasure

Als Generics mit JDK 5 eingeführt werden sollten, stand man vor der Herausforderung dies kompatibel zur JVM und den bisherigen Klassen des JDKs, insbesondere des Collections-Frameworks, realisieren zu müssen. Man entschied sich dazu, weder dynamisch neue Klassen zu erzeugen, wie dies für Arrays geschieht, noch den von der JVM generierten und ausgeführten Bytecode anzupassen.

Dies war nur durch einen Trick möglich: Generics sind mithilfe der sogenannten ***Type Erasure*** realisiert. Das bedeutet, dass die im Sourcecode typisierten Klassen durch den Compiler auf untypisierte Klassen bzw. die Parameter auf den allgemeinsten Typ `Object` abgebildet werden. Aus einer `ArrayList<Person>` wird dadurch beim Kompilieren eine `ArrayList` ohne Typinformationen. Als Folge der Type Erasure existieren also zur Laufzeit nur noch untypisierte Klassen.[39] Um das zu verstehen, schauen wir wieder auf eine Liste mit `Person`-Objekten:

```
final List<Person> personList = new ArrayList<>();

personList.add(new Person("Max", new Date(), "Musterstadt"));
final Person firstPerson = personList.get(0);
```

Dieser Sourcecode wird so kompiliert, als ob man Folgendes programmiert hätte:

```
final List personList = new ArrayList();

personList.add(new Person("Max", new Date(), "Musterstadt"));
final Person firstPerson = (Person)personList.get(0);
```

Das Entfernen des Typparameters und das Hinzufügen des Casts sind aber noch die am wenigsten weitreichenden Konsequenzen. Damit die Zuweisungen wie im Sourcecode angegeben funktionieren, muss der Compiler zum einen Casts passend in den Bytecode einfügen und zum anderen muss nun beim Kompilieren extrem streng geprüft werden, dass zur Laufzeit keine Typinkompatibilitäten auftreten können. Mögliche Probleme werden durch Warnungen signalisiert. Nur wenn keine Warnungen auftreten, sind keine Typinkompatibilitäten erkannt worden.

Insgesamt führt die Type Erasure zu einigen Einschränkungen und Merkwürdigkeiten beim Einsatz von Generics, etwa bei dynamischen Typabfragen zur Laufzeit, dem Erzeugen von Objekten mit generischem Typ und vor allem in Kombination mit Collections und Arrays. Im Folgenden stelle ich Auswirkungen sowie potenzielle Fallstricke vor und zeige mögliche Lösungen auf.

Auswirkung 1: Fehlende Typinformationen bei dynamischen Typabfragen

Wie schon in Abschnitt 3.6.1 beschrieben, erzeugt der Compiler bei der Definition eines Arrays dynamisch einen neuen Typ, falls es diesen noch nicht gab. Rekapitulieren wir dies am Beispiel von zwei Arrays für Spitznamen und Personen:

```
final String[] nicknames = { "Dragonman", "Iron Mike", "Lordmaster" };
final Person[] bowlingPeople = { new Person("Sven", new Date(), "Kiel"),
                                 new Person("Michael", new Date(), "Bremen"),
                                 new Person("Andreas", new Date(), "Kiel") };
```

Durch diese Anweisungen entstehen zwei neue Array-Typen: `String[].class` und `Person[].class`. Niemals würde man auf die Idee kommen, dass die Referenzen `nicknames` und `bowlingPeople` vom gleichen Typ wären. Anders ausgedrückt: Nur weil beide Arrays sind, gilt ***nicht*** `String[].class == Person[].class`. Schreiben wir etwas Sourcecode, um das zu überprüfen:

```
// Compile-Error: Incompatible operand types Class<String[]> and
Class<Person[]>
// final boolean sameType1 = (String[].class == Person[].class);
// final boolean sameType2 = (nicknames.getClass() ==
bowlingPeople.getClass());

final boolean sameType3 = (nicknames.getClass().equals(
                           bowlingPeople.getClass()));
System.out.println(sameType3); // false
```

Tatsächlich führen die gezeigten Vergleiche sogar zu Kompilierfehlern:[40] Ein Vergleich der Klassen mit `equals(Object)` liefert den Wert `false` und zeigt damit, dass es sich tatsächlich um unterschiedliche Klassen handelt.

Typabfragen bei Generics

Setzen wir nun Generics ein und verändern jetzt die speichernde Datenstruktur von einem Array zu einer `ArrayList<E>`, um zu schauen, welche Auswirkungen dies für Typabfragen besitzt. Wir verwenden hier die für unsere Zwecke praktische statische Hilfsmethode `Arrays.asList(T...)` aus dem Collections-Framework (vgl. Abschnitt 5.3.1). Diese erzeugt aus einem Array vom Typ `T` eine typisierte, unmodifizierbare Liste. Damit betrachten wir das Ganze erneut:

```
final List<String> nicknamesList = Arrays.asList(nicknames);
final List<Person> bowlingPeopleList = Arrays.asList(bowlingPeople);

// ACHTUNG: Compile-Error
// final boolean sameType1 = (List<String>.class == List<Person>.class);
final boolean sameType2 = (nicknamesList.getClass() ==
                           bowlingPeopleList.getClass());
System.out.println(sameType2); // true
```

Listing 3.7 *Ausführbar als* **'TYPEERASUREEXAMPLE'**

Ein direkter Vergleich der generischen Typen scheitert wiederum bereits an Kompilierfehlern. Um die Typinformationen zu den generischen Klassen zu erhalten, nutzen wir einen Aufruf von `getClass()`. Interessanterweise führt der Vergleich nun nicht mehr zu einem Kompilierfehler, sondern die ermittelten Typinformationen der Variablen `nicknamesList` und `bowlingPeopleList` sind ***gleich***. Daraus folgt: ***Durch die Type Erasure existiert tatsächlich nur genau ein Typ für jede mit verschiedenen Typparametern erzeugte Containerklasse***. Demnach gibt es also weder einen Typ `List<Person>.class` noch einen Typ `List<String>.class`, sondern lediglich den Typ `List.class`. Somit lässt sich also weder mit `instanceof` noch mit `getClass()` prüfen, ob eine Collection auf einen gewünschten Typ festgelegt ist. ***Es sind keine Typprüfungen generischer Containerklassen möglich***, um etwa vor dem Einfügen eines `Person`-Objekts in eine

Liste abzuprüfen, ob diese tatsächlich auf die Speicherung von Personen ausgelegt ist. Das ist eine recht bedeutende Einschränkung.

Auswirkung 2: Probleme beim Erzeugen von generischen Objekten

Weil beim Kompilieren ein generischer Typparameter `T` durch die Type Erasure in den Typ `Object` umgewandelt wird, stehen zur Laufzeit nicht die benötigten Informationen zum gewünschten Typ `T` bereit, um folgenden Konstruktoraufruf auszuführen:

```
// Achtung: Compile-Error
new T()
```

Wenn man aber Instanzen generischer Klassen dynamisch erzeugen möchte, wäre ein solcher Konstruktoraufruf eine wünschenswerte Funktionalität. Es gibt zwei mögliche Lösungen, die ausnutzen, dass generische Typen als Rückgabe erlaubt sind:

1. Man definiert eine Methode zur Konstruktion gemäß dem Muster ERZEUGUNGSMETHODE bzw. FABRIKMETHODE (vgl. Abschnitt 18.1.1 bzw. 18.1.2).
2. Man definiert ein Interface mit einer oder mehreren Methoden zur Konstruktion gemäß dem Muster ABSTRAKTE FABRIK (siehe Literatur in Abschnitt 18.4).

Lösung 1

Zum Erzeugen definiert man folgende Methode `createNewTypedObject(Class<T>)`, die mit Reflection (vgl. Abschnitt 8.1) arbeitet. Für Klassen eines beliebigen Typ `T` kann man wie folgt eine neue Instanz erzeugen:

```
@SuppressWarnings("unchecked")
public static <T> T createNewTypedObject(final Class<T> clazz)
{
    try
    {
        // Konstruktoraufruf per Reflection
        return (T) clazz.newInstance();
    }
    catch (final InstantiationException e)
    {
        // Keine Instanziierung möglich
        throw new IllegalStateException("Keine Instanziierung möglich", e);
    }
    catch (final IllegalAccessException e)
    {
        // Kein Zugriff möglich
        throw new IllegalStateException("Kein Zugriff möglich", e);
    }
}
```

In diesem Beispiel lernen wir die Annotation `@SuppressWarnings("unchecked")` kennen, die Typwarnungen unterdrückt.

Diese Warnungen sollten im Normalfall nicht ignoriert werden, da ansonsten Fehler zur Laufzeit drohen. Hier ist der Einsatz der Annotation sinnvoll, da durch die Anweisungen keine Typinkompatibilität entsteht.

Lösung 2

Die zweite Lösung ist etwas komplizierter. Zum Erzeugen von Objekten führt man eine generische, abstrakte Klasse `AbstractFactory<T>` ein, die ebenfalls eine Methode `createNewTypedObject()` wie folgt deklariert:

```
public abstract class AbstractFactory<T>
{
    abstract T createNewTypedObject();
}
```

Zur Konstruktion spezieller Typen werden dann konkrete Realisierungen dieser abstrakten Klasse mit spezifischen Realisierungen der Methode `createNewTypedObject()` bereitgestellt. Wollte man zum Beispiel auf diese Weise `Person`-Objekte konstruieren, so implementiert man dazu folgende Fabrikklasse `PersonFactory`:

```
public final class PersonFactory extends AbstractFactory<Person>
{
    public Person createNewTypedObject()
    {
        return new Person();
    }
}
```

Einsatz der Lösungen

Betrachten wir beide Lösungsmöglichkeiten (die Fabrikmethode bzw. die abstrakte Fabrik), um ein `Person`-Objekt zu konstruieren:

```
public static void main(final String[] args)
{
    // Variante 1 - Konstruktion
    final Person newPerson1 = createNewTypedObject(Person.class);

    // Variante 2 - Konstruktion
    final AbstractFactory<Person> factory = new PersonFactory();
    final Person newPerson2 = factory.createNewTypedObject();

    // Initialisierung für beide Varianten
    initPersonAttributes(newPerson1, "Mike1", "Aachen", new Date());
    System.out.println("Person1: " + newPerson1);

    initPersonAttributes(newPerson2, "Mike2", "Bremen", new Date());
    System.out.println("Person2: " + newPerson2);
}

private static void initAttributes(final Person newPerson, final String name,
                                   final String city, final Date birthday)
{
    newPerson.setName(name);
```

```
        newPerson.setCity(city);
        newPerson.setBirthday(birthday);
    }
```

Listing 3.8 *Ausführbar als* **'GENERICSCREATIONEXAMPLE'**

Anhand dieses Beispiels wird allerdings auch ein großer Nachteil beider Varianten der Konstruktion deutlich: Eine Erzeugung über eine Fabrikmethode bzw. eine abstrakte Fabrik erfordert die Definition eines Defaultkonstruktors in der Klasse. Weil ein solcher aber lediglich eine Defaultinitialisierung vornimmt und anschließend die so erzeugten Objekte mit der gewünschten Wertebelegung zu versehen sind, müssen Schreibzugriffe auf Attribute erfolgen können. Das erfordert die Bereitstellung und den Aufruf diverser (eventuell sogar öffentlicher) `set()`-Methoden. Warum dies für die Kontrolle des Objektzustands problematisch sein kann, wurde bereits in Abschnitt 3.1.5 besprochen.

Auswirkung 3: Keine generischen statischen Variablen möglich

Wir haben bereits einige durch die Type Erasure ausgelöste Fallstricke kennengelernt. Auch die Definition generischer statischer Attribute bereitet Probleme. Folgende Definition ist *nicht* möglich, sondern es kommt zu einem Kompilierfehler:

```
// Achtung: Diese Klasse lässt sich nicht kompilieren!
public class StaticGenericDataStore<T>
{
    // Compile-Error: Cannot make a static reference to the non-static type T
    private static T value;

    public static void setValue(final T newValue)
    {
        value = newValue;
    }

    public static T getValue()
    {
        return value;
    }
}
```

Klassen können demnach keine Attribute enthalten, die sowohl generisch als auch statisch sind. Der Grund ist folgender: Typparameter gehören, wie oben festgestellt, zu Instanzen und nicht zur Klasse an sich. Statische Attribute und Methoden gehören aber zur Klasse, woraus dann das Gesagte folgt.

Auswirkung 4: Probleme bei der Definition generischer Arrays

Beim Einsatz von Arrays in Kombination mit Generics kommt es aufgrund der Type Erasure mitunter zu Typwarnungen und Fehlern beim Kompilieren.

Schauen wir uns ein einfaches Beispiel an. Es sollen `Pair`-Objekte in einem Array

vom Typ `Pair<Integer, String>[]` gespeichert werden. Wie im Listing angedeutet, kommt es dabei jedoch zu folgendem Kompilierfehler:

```
final Pair<Integer, String> PAIR1 = new Pair<>(1, "One");
final Pair<Integer, String> PAIR2 = new Pair<>(2, "Two");

// Compile-Error: Cannot create a generic array of Pair<Integer, String>
// final Pair<Integer, String>[] PAIRS = { PAIR1, PAIR2 };
final Pair[] PAIRS = { PAIR1, PAIR2 };
```

Es ist nur die zweite gezeigte, nicht typisierte Definition `Pair[] PAIRS` erlaubt. Wie lässt sich das erklären? Bekanntermaßen sind Arrays kovariant ausgelegt und können Typsicherheit zur Kompilierzeit nicht 100% garantieren (vgl. Abschnitt 3.6.1). Um Typkompatibilität sicherzustellen, müssen Arrays auch zur Laufzeit Typinformationen besitzen und auswerten. Weil für Generics durch die Type Erasure die Typangaben aus dem Sourcecode entfernt werden, besäße ein wie oben gewünscht definiertes Array `Pair<Integer, String>[]` zur Laufzeit keine Typangaben über die generische Klasse `Pair` mehr, sondern wäre vom Typ `Pair<Object, Object>[]`. Damit fehlen die Typangaben zum Inhalt, die aber zur Laufzeitprüfung benötigt würden.

Nehmen wir kurz einmal an, das Ganze wäre doch erlaubt. Dann könnte man im Array auch `Pair`-Objekte speichern, die vom Typ her inkompatibel sind, etwa statt `Pair<Integer, String>` eine Kombination `Pair<String, Date>` wie folgt:

```
final Object[] objects = intStringPairs = new Pair<Integer, String>[7];

objects[0] = new Pair<String, Date>("Now", new Date()); // Typinkompatibilität
```

Während nicht generische Arrays bei Typinkompatibilitäten `ArrayStoreExceptions` auslösen können, wäre dies somit bei der Speicherung generischer Typen in Arrays nicht möglich, da Konflikte weder vom Compiler noch zur Laufzeit durch die JVM erkannt werden könnten. Um Typinkompatibilitäten also zur Laufzeit ausschließen zu können, ist die Definition generischer Arrays in Java nicht erlaubt.

Auswirkung 5: Kein Erzeugen generischer Arrays möglich

Auch das Erzeugen eines generischen Arrays ist aufgrund der Type Erasure nicht möglich, d. h., folgende Definition führt zu dem gezeigten Kompilierfehler:

```
public static <T> T[] createTypedArray(final int size)
{
    // Compile-Error: Cannot create a generic array of T
    final T[] typedArray = new T[size];
    return typedArray;
}
```

Da Arrays lediglich Referenzen auf Objekte verwalten, könnte man auf die Idee kommen, ein Array benötigter Größe des Typs `Object[]` zu erzeugen und dieses anschließend auf den Typ `T[]` zu casten. Dazu könnte man Folgendes implementieren:

```
public static <T> T[] createTypedArray(final int size)
{
    final T[] typedArray = (T[]) new Object[size];
    return typedArray;
}

public static void main(final String[] args)
{
    // java.lang.ClassCastException:
    // [Ljava.lang.Object; cannot be cast to [Ljava.lang.String;
    final String[] texts = createTypedArray(10);
    texts[0] = "Test";
}
```

Listing 3.9 *Ausführbar als* **'WRONGGENERICARRAYCREATION'**

Diese Umsetzung führt aber zu einer Warnung beim Kompilieren: »`Type safety: Unchecked cast from Object[] to T[]`«, da ein `Object[]` einem Subtyp `T[]` zugewiesen wird. Dies löst später zur Laufzeit eine `ClassCastException` aus.

Rekapitulieren wir nochmal: ***Arrays speichern und besitzen Typinformationen auch zur Laufzeit***. Aufgrund ihrer kovarianten Ausrichtung kann man zwar beispielsweise eine Referenz auf ein `String[]` einer Referenz an ein `Object[]` zuweisen. Umgekehrt führt das natürlich zu Problemen: Man kann einem `String[]` selbstverständlich kein `Object[]` zuweisen. Genau dies wird aber im obigen Beispiel ausgeführt: In der Hilfsmethode `createTypedArray(int)` wird ein Array vom Typ `Object[]` erzeugt und auf den Typ `T[]` gecastet. Durch Type Erasure wird daraus jedoch der Typ `Object[]`. Dieser wird dann in der `main()`-Methode dem Typ `String[]` zugewiesen. Als Folge kommt es zu einer `ClassCastException`, da logischerweise eine Typinkompatibilität vorliegt.

Hinweis: Typwarnungen bei Generics

Ganz allgemein gilt, dass man ***Typwarnungen*** im Zusammenhang mit Generics beim Kompilieren ***unbedingt Beachtung schenken*** sollte. Nur wenn man sich absolut sicher ist, dass wirklich Typkompatibilität gegeben ist, kann man eine Warnung mit der Annotation `@SuppressWarnings("unchecked")` unterdrücken. Im vorherigen Beispiel ist der Cast jedoch *nicht* sicher!

Lösung

Ich greife dem in Abschnitt 8.1 eingeführten Thema Reflection etwas vor, indem ich die Klasse `java.lang.reflect.Array` nutze. Ich verwende deren Methode `newInstance(Class<T>, int)`, um dynamisch zur Laufzeit ein Array vom übergebenen Typ und der angegebenen Größe mit der Hilfsmethode `createTypedArray(Class<T>, int)` wie folgt zu erzeugen:

```
@SuppressWarnings("unchecked")
public static <T> T[] createTypedArray(final Class<T> clazz, final int size)
{
    return (T[]) Array.newInstance(clazz, size);
```

```
}

public static void main(final String[] args)
{
    final String[] texts = createTypedArray(String.class, 2);

    System.out.println("Type: " + texts.getClass().getSimpleName());
    System.out.println("Size: " + texts.length);
    texts[0] = "Test1";
    texts[1] = "Test2";
    System.out.println("Content: " + Arrays.toString(texts));
}
```

Listing 3.10 *Ausführbar als* **'GENERICARRAYCREATION'**

Es kommt zu folgenden Ausgaben, die das bestätigen:

```
Type: String[]
Size: 2
Content: [Test1, Test2]
```

In diesem Fall ist Typsicherheit gegeben und man kann die Annotation `@SuppressWarnings("unchecked")` einsetzen, um eine Typwarnung zu verhindern. Wie schon zuvor erwähnt, sollte man mit der genannten Annotation sehr vorsichtig umgehen und alles penibel prüfen, bevor man sie nutzt.

3.8 Weiterführende Literatur

Dieses Kapitel hat einen fundierten Einstieg in das Thema Objektorientierung gegeben. Dabei wurde das Thema Generics so weit behandelt, dass ein Einsatz in der Praxis erleichtert wird. Wir haben auch ein erstes Verständnis für Varianzformen gewonnen und mögliche Fallstricke der Type Erasure kennengelernt. Ein paar hakelige Details von Generics in Kombination mit den Containerklassen des Collections-Frameworks werden später in Abschnitt 5.4 beleuchtet.

Wissenswertes zur objektorientierten Programmierung und zu Generics finden Sie in den folgenden Büchern und Onlinequellen:

- »**Object-Oriented Software Development Using Java**« von Xiaoping Jia [51] Dieses Buch beschäftigt sich intensiv mit den Grundlagen eines gelungenen objektorientierte Designs von Software. Dabei werden viele Sprachelemente sowie diverse weitere interessante Themen wie Collections, Multithreading sowie verteilte Programme besprochen.
- »**Core Java 2 – Grundlagen**« von Cay S. Horstmann und Gary Cornell [45] Der erste Band dieses Buchs bietet einen Einstieg in den objektorientierten Entwurf mit Java und stellt dabei unter anderem auch Generics vor.
- »**Design mit Java**« von Peter Coad und Mark Mayfield [15] Coad und Mayfield stellen anhand von Beispielen den objektorientierten Entwurf mit Java dar. Dabei gehen sie insbesondere auf Komposition und Vererbung sowie den Einsatz von Interfaces ein.

- »**Fortgeschrittene Programmierung mit Java 5**« von Johannes Nowak [70] Nowak beschäftigt sich intensiv mit den in JDK 5 hinzugekommenen Sprachfeatures Generics und den Concurrency Utilities.
- »**Java Generics and Collections**« von Maurice Naftalin und Philip Wadler [69] Dieses Buch ist eine gute Ergänzung des zuvor genannten Buchs, da es besonders auf die Neuerungen im JDK 5 eingeht – im Speziellen auf Generics und einige Erweiterungen des Collections-Frameworks.
- »**A Programmers's Guide to Java SCJP Certification**« von Khalid A. Mughal und Rolf W. Rasmussen [67]

  Dieses Buch bietet einen präzisen, etwas formalen Einstieg in die SCJP-Zertifizierung. Es geht dabei detailliert auf Generics ein.
- »**Generics in the Java Programming Language**« von Gilad Bracha `http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf`
- »**Java Generics FAQ**« von Angelika Langer `http://www.AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.`

4 Java-Grundlagen

Dieses Kapitel stellt einige grundlegende Java-Sprachelemente vor, die wir in den folgenden Kapiteln immer wieder nutzen. Nützliches und Wissenswertes rund um die Klasse `java.lang.Object` – die Basis aller anderen Objekte – sowie einige ihrer wichtigsten Methoden sind Thema von Abschnitt 4.1. Dann folgt in Abschnitt 4.2 eine Vorstellung der primitiven Datentypen und der korrespondierenden Wrapper-Klassen. Im Anschluss beschäftigen wir uns in Abschnitt 4.3 mit der Verarbeitung textueller Informationen mit Strings. Abschnitt 4.4 enthält eine Einführung in die Datumsarithmetik und nennt einige Fallstricke dabei. Anschließend stellt Abschnitt 4.5 Wissenswertes zu inneren Interfaces und inneren Klassen vor. Das Thema Ein- und Ausgabe und das Konzept von Ein- und Ausgabestreams wird in Abschnitt 4.6 besprochen. Schließlich beschreibt Abschnitt 4.7 verschiedene Varianten zur Behandlung von Fehlern. Dort werden sowohl die Stärken als auch die Tücken beim Einsatz von Exceptions und Assertions dargestellt.

Viele der behandelten Themen sind Bestandteil der Prüfung zum Oracle Certified Professional Java Programmer (OCPJP), ehemals Sun Certified Java Programmer (SCJP). Somit kann dieses Kapitel einen ersten Einstieg in dieses Thema geben. Ausführliche Informationen finden Sie in dem hervorragenden Buch »SCJP Sun Certified Programmer for Java 6 Study Guide« [83] von Kathy Sierra und Bert Bates.

4.1 Die Klasse `Object`

Die Klasse `java.lang.Object` ist die Basisklasse aller in Java existierenden Klassen. Jede Referenzvariable besitzt zumindest immer den Typ `Object`. Das ist praktisch, da diese Basisklasse bereits einige Methoden und damit elementare Verhaltensweisen bereitstellt. In der folgenden Aufzählung gebe ich einen kurzen einführenden Überblick über die wichtigsten Methoden und ihre Intentionen:

- `Class<?> getClass()` – Mithilfe dieser Methode kann der Laufzeittyp einer Klasse abgefragt werden. Diese Methode ist `final` und kann daher nicht überschrieben werden. Sie ist wichtig für den Einsatz von ***Reflection***. Damit wird es zur Laufzeit möglich, Klassenbestandteile zu ermitteln, etwa Methoden, Attribute, Oberklassen, implementierte Interfaces usw. Details beschreibt Abschnitt 8.1.
- `String toString()` – Diese Methode dient dazu, eine textuelle Repräsentation für ein Objekt zu erzeugen. Um aussagekräftige Informationen über ein Objekt zur Verfügung zu stellen, ist es ratsam, die Methode `toString()` zu überschreiben. In Abschnitt 4.1.1 werden wir dies genauer betrachten.
- `boolean equals(Object)` – Diese Methode realisiert einen Vergleich eines

Objekts mit einem anderen. Die Defaultimplementierung führt lediglich einen Referenzvergleich durch. ***Deshalb muss diese Methode in der Regel für eigene Klassen überschrieben werden, um einen sinnvollen Vergleich zweier Instanzen zu realisieren***. Abschnitt 4.1.2 beschreibt dies im Detail.

- `int hashCode()` – Die Methode `hashCode()` wird immer dann benötigt, wenn Objekte in hashbasierten Containern verarbeitet werden sollen. Sie bildet den Objektzustand (oder Teile davon) auf eine Zahl ab. ***Wenn man `equals(Object)` überschreibt, muss man auch immer `hashCode()` passend dazu überschreiben***.[1] Abschnitt 5.1.7 geht auf die Thematik genauer darauf ein.
- `void wait()`, `void notify()`, `void notifyAll()` – Diese Methoden dienen der Thread-Steuerung und sind alle `final` definiert, können also nicht überschrieben werden. Durch Aufruf der Methode `wait()` eines Objekts kann ein Thread darauf warten, dass eine Bedingung eintritt. Er wird inaktiv, bis diese Bedingung von anderen Threads hergestellt und dies durch Aufruf der Methode `notify()` bzw. `notifyAll()` des Objekts an den wartenden Thread kommuniziert wird. In Abschnitt 7.3.2 werden die genannten Methoden im Detail vorgestellt.
- `void finalize()` – Diese Methode ist für Aufräumarbeiten gedacht und wird vom Garbage Collector aufgerufen, bevor dieser den Speicher eines Objekts freigeben möchte, sobald es nicht mehr benötigt (d. h. durch andere Objekte referenziert) wird. Die Defaultimplementierung ist leer und kann überschrieben werden. Da nicht garantiert ist, dass ein Objekt tatsächlich auch von der Garbage Collection behandelt wird, stellt **`finalize ()`** ***daher nur eine letzte, wenn auch unsichere Lösung dar, Aufräumarbeiten auszuführen oder Ressourcen freizugeben.*** In Abschnitt 8.5 lernen wir die Methode `finalize()` genauer kennen.

Im Laufe dieses Kapitels werden Tipps und Realisierungshinweise zur Implementierung für die Methoden `toString()` und `equals(Object)` vorgestellt. Das geschieht exemplarisch anhand einer Klasse `Person`. Im Anschluss ist für diese eine Minimalimplementierung angegeben, die sukzessive ausgebaut wird. In Kapitel 5 ergänzen wir dann noch die Methode `hashCode()`.

Die in der Aufzählung letztgenannten Methoden sind bei der Anwendungsentwicklung in der Regel nicht ganz so häufig von Interesse – mit ihrer Bedeutung und groben Funktionsweise sollte aber jeder Java-Entwickler vertraut sein.

Beispielklasse `Person`

Die nachfolgend gezeigte Klasse `Person` besitzt die drei Attribute `name`, `birthday` und `city` und ist als unveränderliche Klasse folgendermaßen realisiert:

```
public final class  Person
{
   private final String name;
```

```
   private final Date   birthday;
   private final String city;

   public Person(final String name, final Date birthday, final String city)
   {
      // Utility-Klasse Objects neu in JDK 7
      Objects.requireNonNull(name, "parameter 'name' must not be null");
      Objects.requireNonNull(birthday, "parameter 'birthday' must not be
null");
      Objects.requireNonNull(city, "parameter 'city' must not be null");

      this.name = name;
      this.birthday = new Date(birthday.getTime());
      this.city = city;
   }

   public final String getName()          { return name; }
   public final Date   getBirthday()      { return new
Date(birthday.getTime()); }
   public final String getCity()          { return city; }
}
```

Bei dieser Realisierung sind mehrere Dinge erwähnenswert. Zum einen ist dies der Einsatz der Utility-Klasse `java.util.Objects`. Diese wird für die Prüfungen der Konstruktorparameter genutzt. Diese Utility-Klasse wurde mit JDK 7 eingeführt und bietet einige recht praktische Hilfsmethoden. Im Listing wird auf unerlaubte Eingaben von `null` mithilfe der generischen Methode `requireNonNull(T, String)` geprüft.

Zum anderen sollten wir uns kurz einige Gedanken zu der Unveränderlichkeit und deren Umsetzung machen. Eine unveränderliche Klasse darf lediglich Leseoperationen in Form von `get()`-Methoden in ihrer öffentlichen Schnittstelle anbieten. Das reicht jedoch aufgrund der Referenzsemantik von Java gewöhnlich nicht aus, um wirklich Unveränderlichkeit zu erzielen. Anhand des Attributs für den Geburtstag wird dies offensichtlich. Objekte vom Typ `java.util.Date` sind veränderlich. Deshalb muss man beim Erhalt und bei der Rückgabe jeweils Kopien erstellen, um ungewollte Modifikationen zu vermeiden. Für die anderen Attribute sind keine derartigen Kopieraktionen erforderlich, da die dort genutzte Klasse `java.lang.String` unveränderlich ist.

4.1.1 Die Methode `toString()`

Wie erwähnt, dient die Methode `toString()` dazu, eine möglichst aussagekräftige und lesbare textuelle Repräsentation eines Objekts zu erzeugen. Allerdings gibt die Defaultimplementierung lediglich den Klassennamen gefolgt von einer hexadezimalen Darstellung des über die Methode `hashCode()` berechneten Werts zurück (standardmäßig die Referenz des Objekts in einen `int` konvertiert). Das lernen wir an einem kleinen Beispielprogramm kennen, dass dabei hilft, die von der Defaultimplementierung erzeugten Ausgaben für Objekte der Klasse `Person` und für ein `Object`-Array zu analysieren:

```
public static void main(final String[] args)
{
    final Object obj = new Object();
    final Person tom = new Person("Tom", new Date(), "Hamburg");
    final Object[] objectArray = new Object[] { obj, tom };

    System.out.println("Object: " + obj);
    System.out.println("Person: " + tom);
    System.out.println("Object[]: " + objectArray);
}
```

***Listing 4.1** Ausführbar als* **'TOSTRINGEXAMPLE'**

Startet man das Programm TOSTRINGEXAMPLE, so wird etwa Folgendes ausgegeben:

```
Object: java.lang.Object@15db9742
Person: ch04_javagrundlagen.Person@6d06d69c
Object[]: [Ljava.lang.Object;@7852e922
```

Diese Form der Ausgabe ist wenig informativ. Wie bereits gesagt, wird lediglich der über `getClass()` ermittelte Klassenname und der mit der Methode `hashCode()` berechnete Hashwert als hexadezimale Zahl ausgegeben.

Ein Überschreiben der Methode `toString()` ist also immer dann sinnvoll, wenn man eine menschenlesbare Form der Ausgabe erreichen möchte. Wir sehen folgende Verbesserungsmöglichkeiten:

1. Die Repräsentation der Klasse `Person` ist nichtssagend und wenig hilfreich. Besser wäre beispielsweise die Ausgabe der Wertebelegungen der Attribute.
2. Auch die Ausgabe des Arrays entspricht nicht dem, was wir erwarten. Statt der kryptischen Notation `[Ljava.lang.Object;@7852e922` würde man sicher eine kommaseparierte Liste der Stringrepräsentationen enthaltener Objekte vorziehen.

Bevor wir die beiden Probleme behandeln, wollen wir zunächst einen Blick auf die Mechanismen bei der Ausgabe und Umwandlung von Werten mit `toString()` werfen.

Mechanismen bei Ausgaben über `System.out.println()`

In diesem Beispiel sehen wir keinen direkten Aufruf der Methode `toString()`. Tatsächlich wird diese Methode immer dann implizit, d. h. also »unsichtbar«, aufgerufen, wenn ein Objekt in einen String umgewandelt werden muss – hier zur Ausgabe über `System.out.println()`. Diese Umwandlung kann auch explizit durch einen direkten Aufruf von `toString()` für eine Objektreferenz durchgeführt werden. Dann muss allerdings für `null`-Referenzen eine Sonderbehandlung erfolgen, da es ansonsten zu `java.lang.NullPointerException`s kommt. Die Ausgabe von `null`-Werten mit `System.out.println()` wie in der folgenden Zeile

```
System.out.println("println is " + null + " safe");
```

führt dagegen nicht zu Exceptions, sondern zum Text "`null`". Wie kommt das? Zur

Umwandlung von Werten primitiver Datentypen und von Objektreferenzen in eine Stringrepräsentation sind in der Klasse `String` diverse überladene statische `valueOf()`-Methoden definiert. Diese werden implizit beim Aufruf von `System.out.println()` genutzt. Ein kurzer Blick auf die Definition der Methode `valueOf(Object)` erklärt das beschriebene Verhalten für Aufrufe mit `null`-Werten:

```
public static String valueOf(Object obj)
{
    return (obj == null) ? "null" : obj.toString();
}
```

Verbesserungen der `toString()`-Methode für die Klasse `Person`

Zur Verbesserung der Ausgabe überschreiben wir die `toString()`-Methode der Klasse `Person`. Die Realisierung sollte den Klassennamen, die Namen der Attribute sowie deren Werten enthalten. Im einfachsten Fall kann man sich eine Methode von Eclipse durch Aufruf des Menüs SOURCE –> GENERATE TOSTRING()… erzeugen lassen:

```
@Override
public String toString()
{
    return "Person [name=" + name + ", birthday=" + birthday +
                   ", city=" + city + "]";
}
```

Das ist eine gute Basis, die sich noch verbessern lässt, indem die Werte der Attribute durch Methodenaufrufe ermittelt werden. Zudem bietet es sich für textuelle Werte mitunter an, diese in Hochkommata (oder spitze Klammern bzw. Anführungszeichen) einzuschließen. Damit lassen sich Fehler erkennen, die durch führende oder nachfolgende Leerzeichen verursacht sind. Schauen wir uns folgende Implementierung an:

```
@Override
public String toString()
{
    return "Person: " +
           "Name='" + getName() + "' " +              // Variante 1
           "Birthday=<" + getBirthday() + ">  "  +  // Variante 2
           "City="" + getCity() + """;              // Variante 3
}
```

Listing 4.2 *Ausführbar als* **'PERSONTOSTRINGEXAMPLE'**

Der Aufruf des Programms PERSONTOSTRINGEXAMPLE zeigt alle drei Varianten der Stringausgabe und die Ausgabe auf der Konsole ist bereits recht gut lesbar:

```
Person: Name='Tom' Birthday=<Fri Dec 27 14:02:42 CET 2013> City="Hamburg"
```

Etwas ungewöhnlich ist noch die Form der Datumsausgabe. Wir akzeptieren dies für den Moment und diskutieren Lösungsmöglichkeiten später in Abschnitt 4.4.

Tipp: Die Annotation `@Override`

Die Annotation `@Override` dient dazu, Methoden zu kennzeichnen, die korrespondierende Methoden einer Basisklasse überschreiben oder eines Basisinterfaces implementieren (sollen). Dies wird dann durch den Compiler geprüft und sichergestellt. Die Angabe der Annotation hilft, sich vor Flüchtigkeitsfehlern zu schützen, etwa dem versehentlichen Überladen durch Unterschiede in der Methodensignatur. Auch Tippfehler im Methodennamen können so erkannt werden. Zudem kann durch Angabe der Annotation `@Override` vom Compiler eine Warnmeldung erzeugt werden, wenn in einer Basisklasse Methoden geändert werden. Abgeleitete Klassen würden diesen Sachverhalt ansonsten (ohne Annotation) nicht mitbekommen.

Anmerkungen zur Performance

Mitunter sieht man in Implementierungen von `toString()`-Methoden den Einsatz der Klasse `java.lang.StringBuffer` bzw. `java.lang.StringBuilder`. Der Einsatz erfolgt wohl mit dem Ziel, die Aufbereitung der Ausgabe bezüglich der Ausführungsgeschwindigkeit zu optimieren. Würde man die gerade gezeigte `toString()`-Methode der Klasse `Person` unter Nutzung der Methode `append(String)` der Klasse `StringBuilder` umbauen, so könnte Folgendes entstehen:

```
public String toString()
{
    final StringBuilder sb = new StringBuilder();
    sb.append(getClass().getSimpleName()).append(": ");
    sb.append("Name='").append(getName()).append("' ");
    sb.append("Birthday='").append(getBirthday()).append("' ");
    sb.append("City='").append(getCity()).append("' ");
    return sb.toString();
}
```

Anhand des Listings wird die schlechtere Lesbarkeit und umständlichere Schreibweise deutlich. Zudem lässt sich so nur dann ein geringer Performance-Gewinn erzielen, wenn obige Methode extrem häufig aufgerufen wird. ***Sofern also nicht triftige Gründe dagegen sprechen, sollte immer so lesbar wie möglich programmiert werden, um die Verständlichkeit und Wartbarkeit zu erleichtern***. Für das Beispiel heißt das, dass die Konkatenation einzelner Stringobjekte mit dem Operator '+' geschehen sollte.

Nur für den Fall, dass die Aufbereitung der Ausgaben sehr umfangreich ist und zudem innerhalb von Schleifen erfolgt, kann sich der Einsatz eines `StringBuilders` und dessen `append(String)`-Methode statt des Operators '+' lohnen. Abschnitt 4.3 und Kapitel 22 erklären die Gründe im Detail.

Verbesserungen der `toString()`-Ausgabe für Arrays

Eine menschenlesbare Ausgabe von Arrays ist relativ leicht zu erreichen, indem man statische Hilfsmethoden aus der Utility-Klasse `java.util.Arrays` nutzt. Mit der generischen Methode `asList(T...)` erhält man eine Referenz auf das Interface `java.util.List<E>`. In dessen Realisierungen liefert die `toString()`-Methode die gewünschte kommaseparierte Ausgabe. Als Alternative bietet die Klasse `Arrays` seit Java 5 diverse überladene `toString()`-Methoden, unter anderem die hier genutzte Methode `toString(Object[])`:

```
public static void main(final String[] args)
{
    final Person tom = new Person("Tom", new Date(), "Hamburg");
    final Person jerry = new Person("Jerry", new Date(), "Kiel");
    final Object[] persons = new Object[] { tom, jerry };

    System.out.println("Object[]: " + Arrays.asList(persons));    // JDK 1.4
    System.out.println("Object[]: " + Arrays.toString(persons)); // JDK 5.0
}
```

Listing 4.3 *Ausführbar als* **'ARRAYSTOSTRINGEXAMPLE'**

Startet man das Programm ARRAYSTOSTRINGEXAMPLE, so erhält man das gewünschte Resultat (hier gekürzt und leicht umformatiert):

```
Object[]:
[Person: Name='Tom' Birthday='Fri Dec 27 14:23:13 CET 2013' City='Hamburg',
Person: Name='Jerry' Birthday='Fri Dec 27 14:23:13 CET 2013' City='Kiel']
```

Die Vorstellung der Methode `toString()` ist damit abgeschlossen. Im Verlauf dieses Kapitels werden wir aber noch eine Verbesserung der Datumsausgabe vornehmen.

4.1.2 Die Methode `equals()`

Dieser Abschnitt stellt die Implementierung der Methode `equals(Object)` ausführlich dar, weil diese Methode eine zentrale Rolle bei der Verwaltung von Objekten in Containerklassen spielt. Wie bereits erwähnt, ist jedes Objekt durch seinen ***Zustand*** (Belegung der Attribute), sein ***Verhalten*** (Methoden) und seine ***Identität*** (Referenz) definiert. Zum Vergleich von Objekten gibt es unter anderem folgende Möglichkeiten:

1. **Operator '=='** – Mit dem Operator '==' werden Objektreferenzen verglichen. Somit wird überprüft, ob es sich um *dieselben* Objekte handelt.
2. **Aufruf der Methode `equals()`** – Man nutzt `equals(Object)`, wenn beim Vergleich von zwei Objekten nicht die Identität, sondern deren semantischer Zustand (d. h. die Wertebelegung der Attribute) von Interesse ist. Dazu muss die Methode `equals(Object)` in eigenen Klassen überschrieben werden, weil deren Defaultimplementierung aus der Klasse `Object` lediglich Referenzen mit dem Operator '==' vergleicht. In eigenen Realisierungen muss derjenige Teil des Objektzustands verglichen werden, der für die ***semantische Gleichheit***, also die inhaltliche Gleichheit, relevant ist.

Die Methode `equals(Object)` kann zwar explizit zum Vergleich aufgerufen werden, in der Regel geschieht dies aber implizit und automatisch bei verschiedenen Aktionen.

Die Rolle von `equals()` beim Suchen

Um erste Erkenntnisse zur Implementierung von `equals(Object)` zu gewinnen, betrachten wir ein einfaches Beispiel einer Klasse `Spielkarte`. Die Implementierung nutzt einen Aufzählungstyp `Farbe` und einen `int` als Wert der Karte. Im Konstruktor wird abgesichert, dass nur gültige Objekte vom Typ `Farbe` übergeben werden:

```
public final class Spielkarte
{
    enum Farbe { KARO, HERZ, PIK, KREUZ }

    private final Farbe farbe;
    private final int   wert;

    public Spielkarte(final Farbe farbe, final int wert)
    {
        this.farbe = Objects.requireNonNull(farbe, "farbe must not be null");
        this.wert = wert;
    }
}
```

Objekte vom Typ `Spielkarte` wollen wir in einer `java.util.ArrayList<E>` speichern, wobei hier als generischer Typ `Spielkarte` Verwendung findet. Das folgende Listing zeigt das Erzeugen einiger Objekte vom Typ `Spielkarte` und deren Speicherung in einer `ArrayList<Spielkarte>`.[2] Nach dem Hinzufügen einiger Objekte prüfen wir mit der Methode `contains(Object)`, ob die Karte »Pik 8« in der Datenstruktur enthalten ist, und speichern den Rückgabewert in der Variablen `gefunden`:

```
public static void main(final String[] args)
{
    // JDK 7: Diamond Operator: ArrayList<> statt ArrayList<Spielkarte>
    final ArrayList<Spielkarte> spielkarten = new ArrayList<>();

    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK,
8));
    System.out.println("gefunden=" + gefunden);
}
```

Listing 4.4 *Ausführbar als* **'SPIELKARTEEXAMPLE'**

Gefunden oder doch nicht? Auf den ersten Blick lautet die Antwort natürlich gefunden, weil ja eine »Pik 8« in die Datenstruktur eingefügt wurde. Starten wir das

angegebene Programm SPIELKARTEEXAMPLE und prüfen den Wert der Variablen `gefunden`, so erleben wir – je nach Java-Kenntnisstand – aber eine Überraschung, denn `gefunden` hat den Wert `false`! Die gesuchte »Pik 8« ist laut Rückgabe von `contains(Object)` nicht in der Datenstruktur enthalten. Wie kann das sein?

In der Implementierung von `contains(Object)` werden alle in der Collection gespeicherten Elemente durchlaufen und auf Gleichheit mit dem übergebenen Objekt durch Aufruf von `equals(Object)` geprüft. Die Klasse `Spielkarte` überschreibt diese Methode jedoch nicht. Es wird daher die Methode der Basisklasse `Object` aufgerufen, die nur auf Referenzgleichheit prüft und folgendermaßen implementiert ist:

```
public boolean equals(Object obj)
{
    return (this == obj);
}
```

Mit diesem Wissen erklärt sich das Ergebnis der oben gezeigten Suche. Dort wird die Methode `contains(Object)` mit einem neu erzeugten Objekt vom Typ `Spielkarte` aufgerufen. Dieses ist (natürlich) nicht in der Datenstruktur vorhanden.

Der `equals()`-Kontrakt

Vielfach benötigt man eine Prüfung auf inhaltliche Gleichheit. Dazu muss in eigenen Klassen die Methode `equals(Object)` passend überschrieben und dabei deren Kontrakt eingehalten werden. In der JLS [34] ist dazu folgende Signatur festgelegt:

```
public boolean equals(Object obj)
```

Diese Methode muss eine Äquivalenzrelation mit folgenden Eigenschaften realisieren:

- **Null-Akzeptanz** – Für jede Referenz `x` ungleich `null` liefert `x.equals(null)` den Wert `false`.
- **Reflexivität** – Für jede Referenz `x`, die nicht `null` ist, muss `x.equals(x)` den Wert `true` liefern.
- **Symmetrie** – Für alle Referenzen `x` und `y` darf `x.equals(y)` nur den Wert `true` ergeben, wenn `y.equals(x)` dies auch tut.
- **Transitivität** – Für alle Referenzen `x`, `y` und `z` gilt: Sofern `x.equals(y)` und `y.equals(z)` den Wert `true` ergeben, dann muss dies auch `x.equals(z)` tun.
- **Konsistenz** – Für alle Referenzen `x` und `y`, die nicht `null` sind, müssen mehrmalige Aufrufe von `x.equals(y)` konsistent den Wert `true` bzw. `false` liefern.

Mögen diese Forderungen ein wenig kompliziert klingen, so ist eine Umsetzung in der Praxis doch relativ einfach nach folgendem zweistufigen Vorgehen möglich:

1. **Prüfungen** – Vor dem inhaltlichen Vergleich sichern wir ab, dass

   a) keine `null`-Referenzen,

   b) keine identischen Objekte und

   c) nur Objekte des gewünschten Typs verglichen werden.

2. **Objektvergleich** – Anschließend werden diejenigen Attributwerte verglichen, die für die Aussage »Gleichheit« relevant sind.

Prüfungen

Die Implementierung der Methode `equals(Object)` in eigenen Klassen sollte mit einer Prüfung des Übergabeparameters auf `null` beginnen (Punkt a). Für eine bessere Performance wird danach auf Identität (Punkt b) geprüft, weil man sich bei Identität alle weiteren, gegebenenfalls aufwendigen Prüfungen sparen kann. Um nicht Äpfel mit Birnen zu vergleichen, erfolgt eine Typprüfung (Punkt c) vor dem eigentlichen Vergleich der Attribute.

Zu dieser Typprüfung findet man kontroverse Meinungen, ob diese mit `getClass()` oder `instanceof` erfolgen sollte. Eine Prüfung mit `instanceof` ist zwar in vielen Fällen ausreichend, beim Einsatz von Vererbung kann es jedoch zu Fehlern kommen: Wenn man sich unsicher ist, sollte man bevorzugt `getClass()` verwenden, da ansonsten schnell die im Kontrakt geforderte Symmetrie verletzt wird. Ausnahmen bilden die Fälle, in denen man explizit Instanzen von Subklassen und Basisklassen als »gleich« betrachten möchte, falls diese in der Wertebelegung ihrer Attribute übereinstimmen. Man sollte sich bewusst sein, dass es im Zusammenspiel mit Vererbung einiges bei der Typprüfung zu bedenken gibt. Das behandle ich später genauer.

Für die finale, nicht abgeleitete Klasse `Spielkarte` fällt die Wahl leicht: Wir verwenden hier `getClass()`. Aus den Punkten a) bis c) ergeben sich folgende erste Zeilen der Realisierung von `equals(Object)`:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                              // Null-Akzeptanz
        return false;
    if (this == other)                              // Reflexivität
        return true;
    if (this.getClass() != other.getClass())        // Typgleichheit sicherstellen
        return false;

    // …
```

Tipp: `equals()` mit `instanceof` vs. `getClass()`

Wie erwähnt, gibt es zur Typprüfung in `equals(Object)` kontroverse Meinungen. Eine Diskussion mit Joshua Bloch finden Sie online unter `http://www.artima.com/intv/bloch17.html`.

Eine Typprüfung mit `getClass()` testet, ob zwei Klassen exakt den gleichen Typ besitzen. Über `instanceof` werden auch alle Subklassen beim Vergleich auf eine Basisklasse akzeptiert. Es wird also eine Subtypbeziehung überprüft. Der Einsatz von `instanceof` ist allerdings nur dann erlaubt, wenn die geprüften Klassen eine gemeinsame Typhierarchie, d. h. eine gemeinsame Basisklasse ungleich `Object`, besitzen. Würde man eine beliebige Referenzvariable vom Typ `JButton` auf Typkonformität mit der zuvor vorgestellten Klasse `Person` prüfen wollen, so führt die Anweisung `okButton instanceof Person` zu einem Kompilierfehler. ***Unterschiedliche Vererbungshierarchien lassen sich mit `instanceof` nicht vergleichen. Praktischerweise ist `instanceof` aber `null`-sicher, d. h., `null instanceof XYZ` liefert immer `false` und keine `NullPointerException`.***

Objektvergleich

Wurden die initialen Prüfungen bestanden, so kann nun das übergebene Objekt sicher auf den entsprechenden Typ gecastet werden. Die einzelnen Attribute des Objekts werden (z. B. in der Reihenfolge ihrer Definition oder des ersten vermuteten Unterschieds) auf Gleichheit geprüft. Dabei gelten folgende Regeln:

1. Vergleiche alle primitiven Ganzzahltypen per Operator '==='. Für Gleitkommazahlen sind derartige Vergleiche fehleranfällig und fragil. Aufgrund der systemimmanenten Ungenauigkeit bei Berechnungen mit Gleitkommazahlen müssen wir bei deren Vergleich besondere Vorsicht walten lassen bzw. sie in `equals(Object)` möglichst vermeiden. Nur in Ausnahmefällen kann man eine spezielle Gleichheitsprüfung vornehmen. Beachten Sie dazu bitte die Ausführungen in Abschnitt 4.2 und den folgenden Hinweiskasten.
2. Vergleiche alle Objekttypen mit deren `equals(Object)`-Methode. Für optionale Attribute, bei denen `null` ein gültiger Wert ist, muss eine spezielle Behandlung erfolgen. In den folgenden Abschnitten wird dies genauer betrachtet und eine Hilfsmethode `nullSafeEquals(Object, Object)` entwickelt.

Die gewonnenen Erkenntnisse nutzen wir, um die `equals(Object)`-Methode für die Klasse `Spielkarte` wie folgt zu vervollständigen, wodurch dann auch die gesuchte »Pik 8« korrekt gefunden wird:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit sicherstellen
        return false;

    // Enum mit equals, int mit Wertevergleich
    final Spielkarte karte = (Spielkarte) other;
    return this.farbe.equals(karte.farbe) && this.wert == karte.wert;
}
```

Hinweis: Vergleich von Gleitkommatypen `float` und `double`

Zum Vergleich von Gleitkommazahlen kann man sich eine Hilfsmethode `isEqualWithinPrecision(double, double, double)` wie folgt definieren:

```
public static boolean isEqualWithinPrecision(final double value,
                      final double expected, final double epsilon)
{
    return (value > expected - epsilon && value < expected + epsilon);
}
```

Damit werden alle Werte für den erwarteten und den zu prüfenden Wert als »gleich« angesehen, falls deren Differenz kleiner als »Epsilon« ist. ***Selbst damit ist für `equals(Object)` oftmals immer noch keine sinnvolle Aussage möglich!***

Typische Fehler bei der Implementierung von `equals()`

Bis jetzt scheinen die Umsetzung des Kontrakts und die Realisierung der Methode `equals(Object)` relativ einfach zu sein. Bei der unbedachten Implementierung kann man jedoch schnell Fehler machen. Betrachten wir dies anhand einer ersten, naiven Umsetzung für die Klasse `Person`. Es wird hier momentan bewusst auf die bereits kennengelernte Annotation `@Override` verzichtet, um auf ein Problem aufmerksam machen zu können. Eine erste Realisierung, die die Nachnamen vergleicht, könnte wie folgt aussehen:

```
public boolean equals(final Person otherPerson)
{
    return this.getName().equals(otherPerson.getName());
}
```

Zunächst scheint so weit alles in Ordnung zu sein, doch tatsächlich enthält diese Lösung zwei Fehler. Prüfen wir die Signatur und den Kontrakt:

- **Signatur** ✗

  Die gezeigte Methode besitzt die Signatur `equals(Person)`. Bei eigenen Implementierung muss die Signatur korrekterweise den Eingabetyp `Object` verwenden, um beliebige Typen von Objekten vergleichen zu können.

- **Null-Akzeptanz** ✗

  Für jede Referenz `x` ungleich `null` sollte der Aufruf `x.equals(null)` den Wert `false` ergeben. Die obige Realisierung löst stattdessen eine `NullPointerException` durch den Aufruf von `getName()` auf einer `null`-Referenz aus.

- **Reflexivität** ✓

- **Symmetrie**[3] ✓

- **Transitivität ✓**
- **Konsistenz ✓**

Wir erkennen, dass die Methode eine falsche Signatur besitzt und darüber hinaus den Aspekt der Null-Akzeptanz des Kontrakts verletzt. Allerdings betreibt man hier ungewollt Overloading statt Overriding. Dieser Fehler fällt nicht auf, solange man nur Objekte gleichen Typs `T` explizit über `equals(T)` vergleicht. Zu Problemen kommt es erst, wenn `Person`-Objekte in Containerklassen des JDKs gespeichert werden oder Vergleiche mit `null` erfolgen. Die in der Klasse `Person` definierte Methode `equals(Person)` wird von den Containerklassen nicht aufgerufen, da sie *nicht* die benötigte Signatur besitzt. Stattdessen wird die Methode `equals(Object)` aus der Klasse `Object` aufgerufen, die einen Referenzvergleich von `Person`-Objekten statt des gewünschten Vergleichs von Attributwerten durchführt.

Neben diesen eher formalen Fehlern enthält diese Implementierung auch einen inhaltlichen Fehler. Zwei `Person`-Objekte werden bereits bei gleichem Namen als gleich angesehen. Herr Müller aus Kiel ist aber sicherlich nicht Herr Müller aus Hamburg.

Korrekturen

Zur Korrektur folgen wir den Forderungen des Kontrakts und definieren die Methode mit der Signatur `equals(Object)`. Eine Realisierung sichert zunächst die Forderungen Null-Akzeptanz, Reflexivität und Typgleichheit ab. Anschließend werden alle diejenigen Attribute verglichen, die notwendig sind, um `Person`-Objekte eindeutig voneinander zu unterscheiden bzw. als inhaltlich gleich zu erkennen. Dadurch ergibt sich folgende, den Kontrakt erfüllende Methode:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                                  // Null-Akzeptanz
        return false;
    if (this == other)                                  // Reflexivität
        return true;
    if (this.getClass() != other.getClass())            // Typgleichheit prüfen
        return false;

    final Person otherPerson = (Person) other;  // Attribute prüfen
    return  this.getName().equals(otherPerson.getName()) &&
            this.getCity().equals(otherPerson.getCity()) &&
            this.getBirthday().equals(otherPerson.getBirthday());
}
```

Das ist bereits ein guter Schritt in die richtige Richtung. Wir können eine weitere Verbesserung erzielen, indem wir den Sourcecode zum Prüfen der Attribute in eine typsichere Hilfsmethode `compareAttributes(Person)` auslagern:

```
private boolean compareAttributes(final Person otherPerson)
{
```

```
    return  this.getName().equals(otherPerson.getName()) &&
            this.getCity().equals(otherPerson.getCity()) &&
            this.getBirthday().equals(otherPerson.getBirthday());
}
```

Die Auslagerung der Attributprüfungen in eine Hilfsmethode erlaubt es, die Methode `equals(Object)` in eigenen Klassen uniform zu erstellen. Dort werden zunächst die formalen Kriterien abgesichert und dann jeweils die private, typspezifische und damit typsichere Hilfsmethode `compareAttributes()` aufgerufen:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                          // Null-Akzeptanz
        return false;
    if (this == other)                          // Reflexivität
        return true;
    if (this.getClass() != other.getClass())    // Typgleichheit prüfen
        return false;

    final Person otherPerson = (Person) other;  // Attribute prüfen
    return compareAttributes(otherPerson);
}
```

Behandlung optionaler Attribute in `equals()`

Manchmal sind einige Attribute eines Objekts optional und dürfen daher den Wert `null` besitzen. Nehmen wir an, die Klasse `Person` wäre um die optionalen Attribute `street` und `zipcode` erweitert worden:

```
public final class Person
{
   private final String name;
   private final Date   birthday;
   private final String city;
   private String       street;      // Optional => null-Wert erlaubt
   private String       zipcode;     // Optional => null-Wert erlaubt

   public Person(final String name, final Date birthday, final String city,
                 // Optionale Angaben
                 final String street, final String zipcode)
   {
      Objects.requireNonNull(name, "parameter 'name' must not be null");
      Objects.requireNonNull(birthday, "parameter 'birthday' must not be
null");
      Objects.requireNonNull(city, "parameter 'city' must not be null");

      this.name = name;
      this.birthday = new Date(birthday.getTime());
      this.city = city;
      this.street = street;
      this.zipcode = zipcode;
   }

   @Override
   public boolean equals(final Object other)
   {
      if (other == null)
```

```
            return false;
        if (this == other)
            return true;
        if (this.getClass() != other.getClass())
            return false;

        final Person otherPerson = (Person) other;
        return compareAttributes(otherPerson);
    }

    // …
```

Bis dato haben wir `null`-Werte für Attribute nicht betrachtet. Bei der Implementierung von `equals(Object)` ist dies aber speziell zu berücksichtigen. Man ist versucht, die Behandlung optionaler Attribute über eine `null`-Prüfung wie folgt zu realisieren:

```
private boolean compareAttributes(final Person otherPerson)
{
    return getName().equals(otherPerson.getName()) &&
           getCity().equals(otherPerson.getCity()) &&
           getBirthday().equals(otherPerson.getBirthday()) &&
       // ACHTUNG: falsche Behandlung optionaler Attribute
       (getStreet() != null && getStreet().equals(otherPerson.getStreet())) &&
       (getZipCode() != null &&
getZipCode().equals(otherPerson.getZipCode()));
}
```

Eine derartige Realisierung ist jedoch weder ausreichend noch korrekt: ***Zwei optionale Attribute, die jeweils den Wert `null` haben, werden nämlich fälschlicherweise nicht als gleich betrachtet!*** Wir lösen dies, indem wir zunächst einen Referenzvergleich durchführen, der sowohl `null`-Werte als auch eine mögliche Referenzgleichheit korrekt behandelt. Zur Überprüfung der inhaltlichen Gleichheit optionaler Attribute wird anschließend auf ungleich `null` verglichen und dann die Methode `equals(Object)` für das jeweilige Attribut aufgerufen:

```
private boolean compareAttributes(final Person otherPerson)
{
   return getName().equals(otherPerson.getName()) &&
          getCity().equals(otherPerson.getCity()) &&
          getBirthday().equals(otherPerson.getBirthday()) &&
       // korrigierte Behandlung optionaler Attribute
       (getStreet() == otherPerson.getStreet() ||
       (getStreet() != null && getStreet().equals(otherPerson.getStreet())))
&&
       (getZipCode() == otherPerson.getZipCode() ||
       (getZipCode() != null &&
getZipCode().equals(otherPerson.getZipCode())));
}
```

Abgesehen davon, dass die auf diese Weise implementierten Vergleiche der optionalen Attribute schlecht lesbar sind, lässt die Ähnlichkeit der jeweiligen Abfragen ahnen, dass es eine elegantere Umsetzung geben sollte. Tatsächlich kann man eine Hilfsmethode `nullSafeEquals(Object, Object)` in eine Utility-Klasse `EqualsUtils` wie folgt extrahieren:

```
public static boolean nullSafeEquals(final Object obj1, final Object obj2)
```

```
{
    return (obj1 == obj2) || (obj1 != null && obj1.equals(obj2));
}
```

Durch den Einsatz dieser Methode können wir dann in eigenen Realisierungen der Attributvergleiche für mehr Klarheit sorgen:

```
private boolean compareAttributes(final Person otherPerson)
{
    return getName().equals(otherPerson.getName()) &&
           getCity().equals(otherPerson.getCity()) &&
           getBirthday().equals(otherPerson.getBirthday()) &&
       // lesbare Behandlung optionaler Attribute
       EqualsUtils.nullSafeEquals(getStreet(), otherPerson.getStreet()) &&
       EqualsUtils.nullSafeEquals(getZipCode(), otherPerson.getZipCode());
}
```

Idiom: Null-sichere Vergleiche mit der Utility-Klasse `Objects`

Seit JDK 7 kann man die obigen Vergleiche durch den Einsatz der Utility-Klasse `Objects` noch kompakter schreiben. Diese Klasse haben wir bereits bei der Implementierung der Prüfung der Konstruktorargumente der Klasse `Person` kennengelernt. Zur Realisierung eigener `equals(Object)`-Methoden, die Attribute `null`-sicher vergleichen können sollen, ist in `Objects` eine statische Methode `equals(Object, Object)` definiert, die diese Funktionalität bereitstellt. Mit diesem Wissen vereinfachen wir die Implementierung. Wir nutzen dazu folgendes Vorgehen oder auch Idiom[4] genannt:

```
private boolean compareAttributes(final Person otherPerson)
{
    return getName().equals(otherPerson.getName()) &&
           getCity().equals(otherPerson.getCity()) &&
           getBirthday().equals(otherPerson.getBirthday()) &&
           // Einsatz der Utility-Klasse Objects
           Objects.equals(getStreet(), otherPerson.getStreet()) &&
           Objects.equals(getZipCode(), otherPerson.getZipCode());
}
```

Mit diesen letzten Schritten und Hinweisen haben wir das notwendige Rüstzeug für stabile Implementierungen von `equals(Object)` komplettiert – allerdings bisher nur für nicht abgeleitete Klassen. Um das Thema vollständig zu behandeln, fehlt noch die Betrachtung der Auswirkungen von Vererbung auf die Realisierung von `equals(Object)`. Das stellt der folgende Abschnitt vor.

Erweiterungen für `equals()` in Subklassen

Der Vergleich von Objekten nicht abgeleiteter Klassen war noch relativ einfach zu realisieren, da nach einer Typprüfung lediglich einige Attributwerte verglichen werden mussten. Das Ganze wird komplexer, wenn Objekte aus einer Klassenhierarchie zu vergleichen sind. Besonders die Prüfung auf Typgleichheit erfordert einige Überlegungen. Schauen wir uns dies schrittweise anhand einer Klassenhierarchie, bestehend aus einer Basisklasse `BaseClass` und einer Subklasse

SubClass, an.

Prüfung auf Typgleichheit

Beginnen wir mit einigen Vorüberlegungen zur Implementierung der Typprüfung, da diese den Knackpunkt darstellt und sich teilweise nicht mehr konform zum equals()-Kontrakt erstellen lässt, wenn Subklassen die Basisklasse um Attribute und eine eigene equals(Object)-Methode erweitern.

Zunächst einmal besteht die Forderung nach Symmetrie, d. h., für beliebige Objekte x und y muss Folgendes gelten: Wenn x.equals(y) gilt, so gilt auch y.equals(x). Nehmen wir dazu an, die Referenzvariable x wäre vom Typ BaseClass und y vom Typ SubClass:

- Ein Vergleich mit instanceof ist nicht symmetrisch, denn es gilt hierbei y instanceof BaseClass, aber nicht x instanceof SubClass.
- Ein Vergleich per getClass() liefert für beide Fälle false und ist symmetrisch.

Nutzt man getClass(), so ist man auf der sicheren Seite, wenn man die Methode equals(Object) konform zum Kontrakt implementieren möchte. Anhand des nachfolgenden Beispiels erkennt man, dass durch Einsatz von instanceof schnell die Gefahr besteht, die Symmetrie zu verletzen.

```
public static void main(final String[] args)
{
  final BaseClass x = new BaseClass();
  final SubClass y = new SubClass();

  // instanceof prüfen
  System.out.println("y is-a BaseClass: " + (y instanceof BaseClass)); // true
  System.out.println("x is-a SubClass: " + (x instanceof SubClass));   //
false

  // getClass() prüfen, Ergebnis: false
  System.out.println("getClass(): " + (x.getClass() == y.getClass()));
}
```

Listing 4.5 *Ausführbar als* **'EQUALSSYMMETRIE'**

Für den Fall, dass nur Objekte des *exakt* gleichen Typs in equals(Object) als gleich angesehen werden sollen, sollten die Typprüfungen über einen Aufruf der Methode getClass() erfolgen. Instanzen von BaseClass und SubClass können so niemals gleich sein. Für diverse Anwendungsfälle ist das genau so auch gewünscht.

Es gibt jedoch einige Ausnahmen, nämlich falls Objekte von Basis- und Subklassen als gleich betrachtet werden sollen, wenn deren Wertebelegung der Attribute bzw. ihres Inhalts (semantisch) übereinstimmen. Das ist im JDK beispielsweise für verschiedene Implementierungen von Listen der Fall: Von diesen erwartet man intuitiv, dass sie als gleich angesehen werden, solange sie den Typ java.util.List<E> erfüllen und den gleichen Inhalt besitzen. Damit die Typprüfung hier für unterschiedliche Implementierungen von Listen nicht false

liefert, nutzt man hier `instanceof`. Da dies, wie bereits bekannt, leicht die Forderung nach Symmetrie verletzt, darf ***lediglich die Basisklasse die Methode `equals(Object)` implementieren und darüber bestimmen, wann Subklassen als gleich angesehen werden sollen.*** Im JDK erfolgt daher für Listen die Implementierung in der abstrakten Basisklasse `java.util.AbstractList<E>`. Vereinfacht auf die Klassenhierarchie des Beispiels übertragen ergibt sich dann Folgendes:

```
class BaseClass
{
    private String attr1;
    private String attr2;

    @Override
    public boolean equals(final Object obj)
    {
        if (obj == this)
            return true;

        if (!(obj instanceof BaseClass))  // null-Prüfung + Typprüfung
            return false;

        final BaseClass otherBaseClass = (BaseClass) obj;
        return EqualsUtils.nullSafeEquals(this.attr1, otherBaseClass.attr1) &&
               EqualsUtils.nullSafeEquals(this.attr2, otherBaseClass.attr2);
    }
}
```

Die geführte Argumentation macht Folgendes klar: Wird die Subklasse `SubClass` um eine `equals(Object)`-Methode erweitert, so wird die Symmetrie verletzt.

Folglich lassen sich Objekte einer Klassenhierarchie entweder nur typrein vergleichen oder erfordern die ausschließliche Implementierung von `equals(Object)` in der Basisklasse mit `instanceof`. Letzteres wird bei mehrstufigen Ableitungshierarchien unpraktikabel, da lediglich die Attribute der untersten Basisklasse ausschlaggebend für den Vergleich sein können.

Tipp: Schwierigkeiten beim Implementieren von `equals(Object)`

Falls Sie sich zunächst etwas schwer mit der korrekten Implementierung der Typprüfung von `equals(Object)` tun, so sind Sie in bester Gesellschaft: Cay Horstmann und Gary Cornell weisen in ihrem Buch »Core Java – Band 1 Grundlagen« [45] darauf hin, dass man auch im JDK verschiedenste Realisierungen von `equals(Object)` findet. Zum Teil wird mit `getClass()` geprüft, teilweise mit `instanceof` und manchmal wird sogar ganz auf die Typprüfung verzichtet. Dadurch kommt es schnell mal zu Verletzungen des `equals()`-Kontrakts.

Prüfung auf semantische Gleichheit

Für Subklassen müssen in der Regel nicht nur die eigenen Attribute, sondern auch

alle relevanten Bestandteile der Klassenhierarchie geprüft werden. Das kann man wie folgt realisieren:

1. **Zugriff auf Attribute der Basisklasse** – Wenn eine Subklasse Zugriff auf die Attribute der Basisklasse (direkt oder indirekt über `get()`-Methoden) besitzt, so könnte man die bisher gezeigte `equals(Object)`-Implementierung für die Subklasse derart erweitern, dass sie alle Attribute (auch die der Basisklassen) vergleicht.

   Bewertung: Diese Art der Umsetzung ist nicht OO-gemäß. Sie verstößt gegen die Kapselung und die Trennung von Zuständigkeiten. Neben diesen Designaspekten ist diese Art der Realisierung fragil, denn Änderungen in der Basisklasse bedingen Änderungen in Subklassen. Wird die Basisklasse um Attribute ergänzt, so muss dies beim Vergleich in der Subklasse berücksichtigt werden. Die Auswirkungen steigen mit der Anzahl an Subklassen. Daraus ergibt sich folgende Regel: »***Nutze in `equals(Object)` niemals direkte Zugriffe auf Attribute der Basisklasse!***«

2. **Aufruf von `equals()` der Basisklasse** – Ein Vergleich der Basisklassenbestandteile lässt sich elegant durch einen Aufruf von `equals(Object)` der Basisklasse regeln, der gegebenenfalls wiederum Aufrufe von `equals(Object)` an weitere Basisklassen delegiert. Direkte Subklassen von `Object` dürfen dies jedoch nicht tun, da ansonsten ein Referenzvergleich ausgeführt würde. Dieser gibt nahezu immer `false` zurück – außer wenn beide Objekte identisch sind.

Zur Realisierung von `equals(Object)` in Subklassen wird die zweite Variante eingesetzt. Will man sich nicht darauf verlassen, dass die drei bereits bekannten Prüfungen auf Null-Akzeptanz, Reflexivität und korrekten Typ schon in der Basisklasse durchgeführt werden, ergänzt man diese Prüfungen vorsichtshalber vor dem Vergleich der Bestandteile der Basisklassen. Anschließend kann ein Aufruf an eine Methode `compareAttributes()` erfolgen, die den Vergleich der Attribute ausführt:

```
public final class SubClass extends BaseClass
{
    // …
    public boolean equals(final Object other)
    {
        // Drei Prüfungen nur vorsichtshalber
        if (other == null)                          // Null-Akzeptanz
            return false;
        if (this == other)                          // Reflexivität
            return true;
        if (this.getClass() != other.getClass()) // Typgleichheit prüfen
            return false;

         // Delegation an Basisklasse
         if (!super.equals(other))
             return false;

         final SubClass subClass = (SubClass)other;
         return compareAttributes(subClass);
```

```
    }
    // …
```

Fazit

Dieser Abschnitt ist ausführlich auf die Grundlagen und auf mögliche Fallstricke bei der Implementierung der Methode `equals(Object)` eingegangen, weil sich dabei recht schnell Fehler mit großen Nebenwirkungen einschleichen können. Daher ist es extrem wichtig, diese Methode konform zu ihrem Kontrakt zu implementieren: Insbesondere die korrekte Funktionsweise sehr vieler Containerklassen basiert darauf. Ergänzend sei auch nochmals darauf hingewiesen, dass wenn man `equals(Object)` überschreibt, man ebenfalls die Methode `hashCode()` konform dazu überschreiben sollte (bzw. muss). Grundlagen dazu werden später in Abschnitt 5.1.7 besprochen. Auf mögliche Fallstricke geht dann Abschnitt 5.1.9 genauer ein.

4.2 Primitive Typen und Wrapper-Klassen

In diesem Unterkapitel beschäftigen wir uns mit primitiven Datentypen sowie korrespondierenden Wrapper-Klassen. Einführend stelle ich dazu deren grundsätzliche Eigenschaften vor. Danach lernen wir die Konvertierung und Verarbeitung von Werten kennen. Insbesondere gehe ich dabei auf Besonderheiten des sogenannten Auto-Boxings und Auto-Unboxings ein, das eine automatische Konvertierung von einem primitiven Typ in den korrespondierenden Wrapper-Typ durchführt und andersherum. Abschließend schauen wir dann auf die Ausgabe und Verarbeitung von Zahlen und lernen neben der Dezimaldarstellung noch die Binär-, Oktal- und Hexadezimaldarstellung sowie eine Repräsentation von Zahlen als Bitdarstellung und Operationen dafür kennen.

4.2.1 Grundlagen

Fast alles ist in Java ein Objekt. Zusätzlich existieren verschiedene primitive Datentypen, die Werte ohne Verhalten repräsentieren. In Java sind das die Typen `byte`, `short`, `int`, `long`, `float` und `double` für Zahlen, `boolean` für boolesche Werte sowie der Typ `char` für einzelne Zeichen. Für die primitiven Datentypen gibt es ***Wrapper-Klassen***, die die primitiven Datentypen als unveränderliches Objekt darstellen und mit etwas Funktionalität »umhüllen«, z. B. das Parsen eines Wertes aus einem String.

Tabelle 4-1 listet primitive Typen und korrespondierende Wrapper-Klassen auf, nennt deren Anzahl an Bits sowie deren Wertebereich. Für die Wrapper-Klassen sind die Wertebereiche über jeweils eigene Konstanten `MIN_VALUE` und `MAX_VALUE` definiert. Darüber hinaus wissenswert ist, dass die Wrapper-Klassen `Short`, `Integer`, `Long`, `Float` und `Double` alle aus dem Package `java.lang` stammen

und die gemeinsame Basisklasse `java.lang.Number` besitzen.

Tabelle 4-1 *Primitive Datentypen und korrespondierende Wrapper-Klassen*

Primitiver Typ	Wrapper	Bits	Wertebereich
`byte`	`Byte`	8	-2^7 .. 2^7-1 (-128 .. 127)
`short`	`Short`	16	-2^{15} .. $2^{15}-1$ (-32.768 .. 32.767)
`int`	`Integer`	32	-2^{31} .. $2^{31}-1$ (-2.147.483.648 .. 2.147.483.647)
`long`	`Long`	64	-2^{63} .. $2^{63}-1$ (-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807)
`float`	`Float`	32	$\pm 1.4e-45 .. \pm 3.4e+38$
`double`	`Double`	64	$\pm 4.9e-324 .. \pm 1.8e+308$
`boolean`	`Boolean`	-	`false` und `true`
`char`	`Character`	16	0 .. $2^{16}-1$ (0 (`\u0000`) .. 65535 (`\uffff`))

Info: Motivation für die Existenz primitiver Datentypen

Die Gründe für die Existenz primitiver Datentypen in einer ansonsten objektorientierten Sprache sind Folgende: Nur durch primitive Datentypen ist eine Ansteuerung von Hardware und eine Kompatibilität zu C++-Aufrufen über JNI möglich. Weiterhin verbrauchen primitive Datentypen weniger Speicher und müssen nicht per `new` erzeugt werden. Dadurch wird eine Optimierung der Programmlaufzeit und des benötigten Speicherplatzes erreicht. Details beschreibt Kapitel 22 »Optimierungen«.

Interne Darstellung und das Zweierkomplement

Die Interpretation der Zahlen erfolgt gemäß dem sogenannten ***Zweierkomplement***, das beschreibt, wie Zahlen in ein Bitmuster umgerechnet werden. Mit n als Anzahl der Bits ergibt sich ein Wertebereich von -2^{n-1}, …, $+2^{n-1} - 1$. Für ein Byte entspricht dies dem Wertebereich -2^7, …, $+2^7 - 1 = -128 \ldots + 127$. Das Zweierkomplement

bildet vorzeichenbehaftete Zahlen wie folgt ab: Jede *positive* Zahl wird in ein Bitmuster mit einer führenden Null umgewandelt. Jede *negative* Zahl wird zunächst in das Bitmuster der entsprechenden positiven Zahl umgerechnet, danach wird dieses invertiert und der Wert 1 addiert. Damit ergeben sich folgende Bitmuster für den Typ `byte`:

```
+127 = 01111111
…
+  1 = 00000001
   0 = 00000000
-  1 = 11111111
…
-127 = 10000001
-128 = 10000000
```

Wissenswertes zu primitiven Datentypen

Wie aus der vorherigen Tabelle ersichtlich, besitzen die primitiven Datentypen `byte`, `short`, `int` und `long` unterschiedliche Wertebereiche und repräsentieren Ganzzahlen. Gleitkommazahlen werden durch die Typen `float` und `double` repräsentiert. Diese verschiedenen primitiven Datentypen sind in die Sprache Java integriert. Das bedeutet, dass es für diese spezielle Literale (Zeichenfolgen) gibt, die Repräsentanten oder Werte des primitiven Typs sind. Ganzzahlenliterale sind ohne weitere Angabe vom Typ `int`:

```
int nr = 4711;
int second = 2;

// Achtung: 'l' ist sehr leicht mit der Ziffer '1' zu verwechseln
long longNumber1 = 4711l;
long longNumber2 = 4711L;
```

Im Listing ist gezeigt, dass man durch ein nachgestelltes `l` bzw. ein großes `L` eine Zahl vom Typ `long` definieren kann. Wegen der besseren Lesbarkeit und der leichteren Unterscheidbarkeit von der Ziffer `1` empfiehlt sich das große `L` statt des kleinen `l`.

Bei der Verarbeitung von Gleitkommazahlen (auch Floating-Point genannt) sind die Werte im Gegensatz zu den Ganzzahlen defaultmäßig immer vom »größeren« Datentyp `double`. Optional können `doubles` durch Angabe von `d` bzw `D` explizit gekennzeichnet werden. Derartiges ist aber eigentlich nur für Werte vom Typ `float` notwendig: Ein nachgestelltes `f/F` »verkleinert« Werte auf den Typ `float`. Das Gesagte wird durch nachfolgendes Listing verdeutlicht:

```
double pi = 3.14159;
float  one_quarter = 1/4F;
double pi_2 = 3.14159D;
```

Merkwürdigkeiten bei Berechnungen

Um Sie für mögliche Probleme bei Berechnungen zu sensibilisieren, möchte ich ein

paar einfache Divisionen ausführen:

```
System.out.println(7 / 2);        // 3
System.out.println(7 / 2.0);      // 3.5
System.out.println(7.0 / 2);      // 3.5
System.out.println(7.0 / 2.0);    // 3.5
```

Ohne viel Nachdenken kann man sich bei der ersten Division fragen, wieso denn das Ergebnis 3 und nicht 3.5 ist, wie man es erwarten würde. Das liegt ganz einfach daran, dass die erste Division eine Operation auf zwei `int`-Werten ist und diese besitzen nun mal keine Nachkommastellen. Erst wenn einer der Operanden eine Gleitkommazahl ist, wird auch das Ergebnis zu einer Gleitkommazahl und man erhält das korrekte Ergebnis.

Selbst einfache Berechnungen mit den Gleitkommatypen `float` und `double` können bereits zu Ungenauigkeiten führen, was durch deren Repräsentation gemäß dem Format IEEE 754[5] bedingt ist. Folgendes Beispiel einer `for`-Schleife, die zehnmal den Wert `0.1` addiert und anschließend die Abweichung der berechneten Summe mit dem erwarteten Wert `1` ausgibt, macht mögliche Probleme eindrucksvoll deutlich. Außerdem werden merkwürdige Zwischenstände sowie eine Abweichung zwischen Literalen mit gleichem Zahlenwert für die Typen `float` und `double` gezeigt:

```
public static void main(final String[] args)
{
   float sum = 0.0f;
   for (int i = 0; i < 10; i++)
      sum += 0.1;

   System.out.println("sum:   1 != " + sum);

   System.out.println("3 * add  = " + (0.1 + 0.1 + 0.1));
   System.out.println("3 * 0.1  = " + 3 * 0.1);
   System.out.println("7 * 0.1  = " + 7 * 0.1);

   System.out.println("compare  = " + (0.3f == 0.3d));  // false, Float !=
Double
}
```

Listing 4.6 *Ausführbar als* **'FLOATUNGENAUIGKEIT'**

Das Programm FLOATUNGENAUIGKEIT produziert folgende Ausgaben:

```
sum:   1 != 1.0000001
3 * add  = 0.30000000000000004
3 * 0.1  = 0.30000000000000004
7 * 0.1  = 0.7000000000000001
compare  = false
```

Anhand der Ausgaben erkennt man mögliche Überraschungen durch Rundungsfehler ziemlich gut. Insbesondere erwähnenswert ist, dass man zunächst darüber verwundert sein könnte, dass gleiche Zahlenliterale für `float` und `double` verschiedene Werte besitzen. Nach kurzem Nachdenken wird dies durch die unterschiedliche Anzahl an Nachkommastellen und die dort auftretenden Rundungsfehler verständlich.

Wissenswertes zu Modulo und negativen Zahlen

Häufig nutzt man eine Modulo-Operation, um festzustellen, ob eine Zahl gerade ist. Dazu vergleicht man den Rest mit dem Wert 0. Wenn man nun feststellen möchte, ob eine Zahl ungerade ist, könnte man auf die Idee kommen, zu prüfen, ob der Rest 1 ist. Dabei kann man Überraschungen erleben. Folgendes Programm zeigt einige Beispiele:

```
public void evenAndOddChecks()
{
    // hier scheint noch alles okay zu sein …
    System.out.println("2 is even? " + (2 % 2 == 0));
    System.out.println("-2 is even? " + (-2 % 2 == 0));

    // hier kommen die Merkwürdigkeiten …
    System.out.println("3 is odd? " + (3 % 2 == 1));
    System.out.println("-3 is odd? " + (-3 % 2 == 1));
    System.out.println("-3 is odd? " + (-3 % 2 == -1));
}
```

Diese Methode gibt Folgendes aus:

```
2 is even? true
-2 is even? true
3 is odd? true
-3 is odd? false
-3 is odd? true
```

Wie man sieht, wird für ungerade negative Zahlen überraschenderweise das falsche Ergebnis ausgegeben. Die zweite Prüfung zeigt, dass die Modulo-Operation für negative Zahlen negative Werte liefert. Somit ist folgende Methode leider nicht 100% korrekt:

```
private static boolean isOdd(final int val)
{
    return (val % 2) == 1;
}
```

Diese Methode liefert für negative Eingaben falsche Ergebnisse. Eine Korrektur fällt nicht schwer, wenn man sich des Problems bewusst ist. Statt auf gleich 1 prüft man auf ungleich 0 und erhält damit folgende korrigierte und korrekte Realisierung:

```
private static boolean isOddCorrected(final int val)
{
    return (val % 2) != 0;
}
```

Tipp: Wahl von primitiven Datentypen

Bei Berechnungen mit Ganzzahlen bietet es sich an, bevorzugt mit `int` und `long` zu arbeiten. Nur selten ist der Einsatz von `byte` oder `short` sinnvoll. Bei Gleitkommazahlen sollte man möglichst `double` statt `float` nutzen, da Letzteres eine recht geringe Genauigkeit bei Berechnungen mit mehreren Nachkommastellen aufweist. Generell sollte man bei Gleitkommatypen immer Rundungsprobleme bedenken.

Typ- und Wertebereichserweiterungen sowie -verkleinerungen

Variablen können unterschiedliche Typen besitzen und müssen teilweise aufeinander abgebildet werden. Dabei sind verschiedene Dinge zu beachten.

Widening

Eine Wertebereichserweiterung ist in jedem Fall ungefährlich, weil etwa ein `long` einen größeren Wertebereich besitzt als ein `int` und somit alle Zahlen darstellen kann, die auch der kleinere Datentyp enthält. Dabei findet ein sogenanntes ***Widening*** statt, das folgende Kette durchläuft:

byte ⇒ short ⇒ int ⇒ long ⇒ float ⇒ double

Narrowing

Was passiert aber, wenn man den Wertebereich verkleinert? Das stellt potenziell eine gefährliche Operation dar, weil eventuell Informationen verloren gehen, etwa, wenn man den Wert 1.000.000 einer Variablen vom Typ `short` zuweist, die maximal Werte bis `32767` darstellen kann, z. B. folgendermaßen:

```
final short truncated = (short)1_000_000;
System.out.println("truncated: " + truncated);
```

Als Ausgabe erhält man Folgendes:

```
truncated: 16960
```

Derartige Typverkleinerungen (auch ***Narrowing*** genannt) bergen ganz offensichtlich die Gefahr für einen Informationsverlust und eine falsche Berechnung. Daher sind die Casts explizit zu notieren. Seit JDK 6 gibt es eine Ausnahme, die wir nun anschauen.

Besonderheit: Auto-Narrowing

So praktisch eine Fehlermeldung bei Narrowing in der Regel ist, so gilt dies nicht, wenn ein Zahlenliteral einen Wert darstellt, der kompatibel zu dem kleineren Typ ist. Folgendes Beispiel macht dies deutlich. Hier sehen wir die Zahl 4711, die offensichtlich zuweisungskompatibel zu einem `short` ist, weil sie in dessen Wertebereich liegt:

```
final short validShort = (short)4711;
```

Der hier gezeigte Cast war bis JDK 6 notwendig, weil, wie zuvor erwähnt, ja alle Ganzzahlenliterale – wenn nicht anders spezifiziert – vom Typ `int` sind. Der Compiler hat hier einen Verstoß gesehen, obwohl eigentlich kein Problem vorlag. Das war für uns als Entwickler unpraktisch. Seit JDK 6 werden die Wertebereiche der Literale auf Gültigkeit geprüft und es findet ein sogenanntes Auto-Narrowing statt. Diese automatische Typverkleinerung erlaubt es, auf den Cast zu verzichten.

Das verbessert die Lesbarkeit des Sourcecodes:

```
final short validShort = 4711;     // Auto-Narrowing seit JDK 6
// final short invalid = 65535;    // Compile-Error
```

Wertebereichsüberläufe und Silent Fail

Für Berechnungen mit primitiven Datentypen ist explizit zu beachten, dass sie »überlaufen« können! Was bedeutet das? Nehmen wir an, einer Variable ist ein Wert zugewiesen, der nahe des von diesem Datentyp erlaubten Maximalwerts ist, beispielsweise `MAX_VALUE - 7`. Addiert man dazu den Wert 10, dann führt dies nicht etwa zu einer Fehlermeldung, sondern zu einem Überlauf im Wertebereich. Sehen wir uns folgendes Programm an:

```
public static void main(final String[] args)
{
    final int value = Integer.MAX_VALUE - 7;
    System.out.println("Integer.MAX_VALUE - 7 + 10 = " + (value + 10));
}
```

Listing 4.7 *Ausführbar als* **'INTEGEROVERFLOWEXAMPLE'**

Das Programm INTEGEROVERFLOWEXAMPLE produziert folgende Ausgabe:

```
MAX_VALUE - 7 + 10 = -2147483646
```

Die merkwürdige Ausgabe liegt an der internen Repräsentation der Ganzzahlen durch das Zweierkomplement. Bei der oben erwähnten Addition bewegt man sich (vermutlich unerwartet) in den negativen Wertebereich, weil Folgendes gilt:

```
Integer.MAX_VALUE + 1 == Integer.MIN_VALUE
       2147483647 + 1 == -2147483648
```

Mit diesem Wissen und den Anmerkungen aus dem nachfolgenden Praxistipp sind Sie besser für Merkwürdigkeiten bei Berechnungen gewappnet und können mögliche Fehler bereits beim Programmieren vermeiden, wenn Sie an diese Dinge denken.

Verbesserungen in JDK 8

Wertebereichsüberläufe in Kombination mit Silent Fail und daraus resultierende Probleme haben in JDK 8 zu einer Erweiterung in der Klasse `java.lang.Math` geführt. Dort sind für gebräuchliche mathematischen Operationen wie +, -, * usw. korrespondierende Methoden wie `addExact()`, `subtractExact()` oder `multiplyExact()` definiert, die bei den entsprechenden Operationen auf Wertebereichsüberläufe prüfen und dafür Exceptions auslösen.

Hinweis: Anomalien bezüglich `MIN_VALUE`

Für die beiden Datentypen `int` und `long` gilt die Anomalie `MIN_VALUE == -MIN_VALUE`. Interessanterweise gilt dies nicht für die Typen `byte` und `short`:

```
public static void main(final String[] args)
{
    System.out.println(Byte.MIN_VALUE == -Byte.MIN_VALUE);          // false
```

```
    System.out.println(Short.MIN_VALUE == -Short.MIN_VALUE);        // false
    System.out.println(Integer.MIN_VALUE == -Integer.MIN_VALUE);    // true
    System.out.println(Long.MIN_VALUE == -Long.MIN_VALUE);          // true
}
```

Listing 4.8 *Ausführbar als* **'MINVALUETEST'**

Binär-, Oktal- und Hexadezimaldarstellung

Neben der gebräuchlichen Darstellung von Zahlen in der Dezimalschreibweise wird in der Informatik recht häufig auch eine Binär-, Oktal- und Hexadezimaldarstellung benötigt (also Zahlen mit der Basis 2, 8 bzw. 16). Zur Unterscheidung der Zugehörigkeit von Literalen zu Zahlensystemen gibt es Präfixe: '`0x`' bzw. '`0X`' steht für hexadezimale Zahlen, eine führende `0` für Oktalzahlen. Für Binärzahlen dient das Präfix '`0b`' bzw. '`0B`', gefolgt von einer Folge von Nullen und Einsen. Für Oktalzahlen dürfen analog nur die Ziffern 0–7 genutzt werden. Für hexadezimale Zahlen sind neben den Ziffern 0–9 auch die Buchstaben a–f bzw. A–F erlaubt, die die Werte 10–15 repräsentieren. Mit diesem Wissen betrachten wir Zahlenliterale in den verschiedenen Formaten:

```
public static void main(final String[] args)
{
    final int octalLiteral = 0567;
    final int hexLiteral = 0xABC;
    final int binaryLiteral = 0b01101001;   // Seit JDK 7

    System.out.println("octalLiteral " + octalLiteral);
    System.out.println("hexLiteral " + hexLiteral);
    System.out.println("binaryLiteral " + binaryLiteral);
}
```

Listing 4.9 *Ausführbar als* **'BINRARYOCTALHEXADECIMALEXAMPLE'**

Die Notation für Oktalliterale rein mit führender Null und ohne Buchstabenkürzel birgt den Fallstrick, dass jede Zahl mit einer führenden Null diese automatisch zu einer Zahl im Oktalsystem macht, auch wenn dies eventuell gar nicht beabsichtigt war.

Unterstriche in numerischen Literalen

Zur übersichtlicheren Notation von Konstanten ist es seit JDK 7 erlaubt, Ziffern in Ganzzahlenliteralen mit einem Unterstrich zu separieren. Dies kann man unter anderem zur Trennung von Nibbles (4 Bits) bei Bytewerten oder zur Simulation von Tausenderpunkten folgendermaßen einsetzen:

```
final byte binaryLiteral = 0b0110_1001;
final int oneMillion = 1_000_000;
```

Wie gezeigt, sind auch mehrere Unterstriche zur Separation erlaubt. Allerdings kann ein Zahlenliteral weder mit führendem noch mit abschließendem Unterstrich notiert werden.

Konstruktion von Wrapper-Klassen und weitere Konvertierungen

Wrapper-Objekte kann man durch einen Konstruktoraufruf mit einem String oder einem Wert eines primitiven Datentyps erzeugen:

```
final Integer valueByLiteral = new Integer(7);
final Integer valueByString = new Integer("14");
```

Natürlich entspricht nicht jeder beliebige String einer Zahl. Um einen Fehler beim Konvertieren auszudrücken, lösen die Konstruktoren mit Parametern vom Typ `String` eine `java.lang.NumberFormatException` aus. Auch die beiden Wrapper-Klassen `java.lang.Boolean` und `java.lang.Character` weisen einige Besonderheiten auf: Für `Booleans` wird nur die textuelle Eingabe `"true"` – unabhängig von Groß-und Kleinschreibung – in den Wert `Boolean.TRUE` gewandelt. Jeder andere Wert, auch `null`, wird akzeptiert und ergibt `Boolean.FALSE`. Zudem existiert für den `Character`-Wrapper kein Konstruktor, der einen String entgegennimmt, da ein `char` nur einzelnes Zeichen speichert.

Die Wrapper-Klassen ermöglichen, Werte primitiver Datentypen aus verschiedenen Repräsentationsformen (String, Typ-Literal) in Objekte zu verwandeln (***Boxing***). Aus einem solchen Wrapper-Objekt kann wieder ein primitiver Datentyp ermittelt werden (***Unboxing***). Ab Java 5 können primitive Datentypen und Wrapper-Klassen nahezu ohne Unterschied verwendet werden. Sofern benötigt, findet eine Umwandlung automatisch statt. Man spricht daher auch von ***Auto-Boxing*** und ***Auto-Unboxing***.

```
final Integer autoBoxing = 7;                // int -> Integer
final int autoUnboxing = new Integer(4711); // Integer -> int
```

Hierbei gibt es aber folgende Einschränkung: Es können keine Methoden auf primitiven Typen aufgerufen werden: Beispielsweise ist `5.intValue()` nicht möglich.

4.2.2 Konvertierung von Werten

Nachdem wir einen guten Fundus an Wissen über primitive Datentypen und mögliche Fallstricke bei der Angabe und Verarbeitung von Literalen gewonnen haben, wollen wir uns nun ein wenig genauer mit der Konvertierung von Werten beschäftigen.

Die Wrapper-Klassen bieten verschiedene Möglichkeiten, um Strings in Wrapper-Instanzen und Wrapper-Instanzen in Werte primitiver Typen sowie diese wieder in Wrapper-Instanzen umzuwandeln. Abbildung 4-1 veranschaulicht die Konvertierungen.

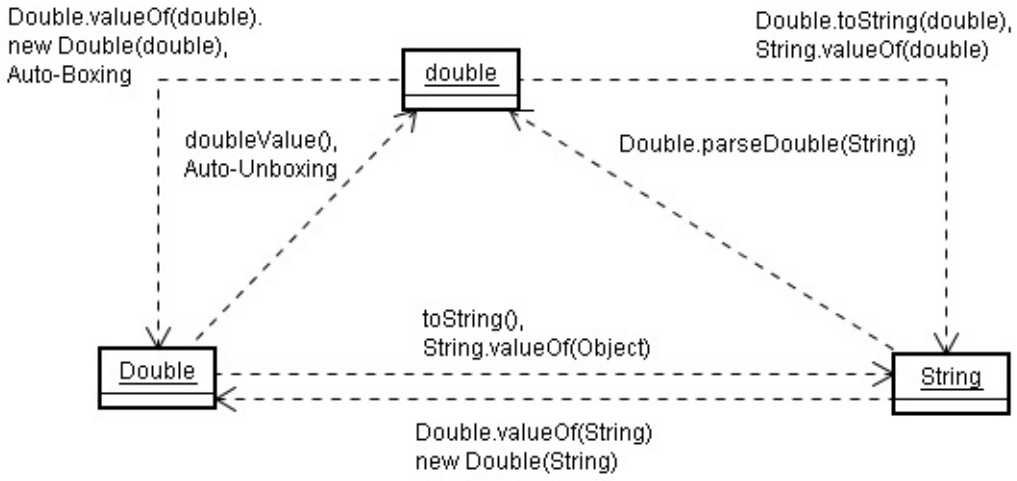


Abbildung 4-1 *Konvertierung von Wrappern, Strings und primitiven Werten*

- **String ⇒ Wrapper** – Die Umwandlung eines Strings in einen Wrapper erfolgt durch statische `valueOf()`-Methoden der jeweiligen Wrapper-Klasse bzw. durch Übergabe eines Strings an deren Konstruktor.
- **Wrapper ⇒ primitiver Typ** – Objekte der Wrapper-Klassen kann man über deren Objektmethoden `xyzValue()`, etwa `integerObj.intValue()`, in Werte eines primitiven Typs umwandeln. `xyz` steht dabei für alle möglichen primitiven Typen, also `byte`, `short`, `int`, `long`, `float` oder `double`. Zusätzlich kann immer noch in andere primitive Zahlentypen umgewandelt werden, etwa `byte`, `short` oder `int`.
- **String ⇒ primitiver Typ** – Mithilfe von statischen `parseXYZ()`-Methoden der jeweiligen Wrapper-Klasse kann man einen Zahlen repräsentierenden String direkt in einen Wert eines primitiven Typs umwandeln. `XYZ` steht dabei für alle möglichen primitiven Typen, also `byte`, `short`, `int`, `long`, `float` oder `double`, allerdings wird hier der erste Buchstabe groß geschrieben, also etwa `Integer.parseInt()`. Diese Methoden sind eine Abkürzung der Hintereinanderausführung der zuvor beschriebenen Methoden `valueOf()` und `xyzValue()`. Bei der Umwandlung von Strings in Werte eines primitiven Typs gibt es noch die Besonderheit zu bedenken, dass die Angabe im String nicht im Dezimalformat, sondern oktal oder hexadezimal vorliegt. Für diese Fälle kann man die Methode `decode()` der Klasse `Integer` bzw. `Long` nutzen.
- **Primitiver Typ ⇒ String** – Um einen primitiven Typ in einen String umzuwandeln, bietet die Klasse `String` überladene statische `valueOf()`-Methoden an. Diese werden u. a. zur Umwandlung bei Ausgaben mit `toString()` eingesetzt. Das haben wir bereits in Abschnitt 4.1.1 kennengelernt.

Fallstricke bei der Umwandlung mit `decode(String)`

Zwar kann man mit JDK 7 nun Binärdarstellungen und Unterstriche in Zahlenliteralen nutzen, jedoch wurden die statischen Methoden `decode(String)`

der Klasse `Integer` bzw. `Long` leider noch nicht derart erweitert, dass sie diese beiden Besonderheiten verarbeiten können. Folgendes Programm zeigt das. Führt man es aus, so kommt es zu einer `java.lang.NumberFormatException: For input string: "1_000_000"`:

```
public static void main(final String[] args)
{
    final String value1 = "4711";
    final String value2 = "0567";
    final String value3 = "0xABC";
    final String error1 = "1_000_000";
    final String error2 = "0b11110000";

    System.out.println("value1 " + Integer.decode(value1));
    System.out.println("value2 " + Integer.decode(value2));
    System.out.println("value2 " + Integer.decode(value3));
    System.out.println("error1 " + Integer.decode(error1));
    System.out.println("error2 " + Integer.decode(error2));
}
```

Listing 4.10 *Ausführbar als* **'INTEGERDECODINGPROBLEMSEXAMPLE'**

Tipp: Fallstricke im API

Ein API sollte möglichst so gestaltet sein, dass es Fehler ausschließt. Leider existieren in den Wrapper-Klassen drei Methoden, die Fehler provozieren. In den Klassen `Boolean`, `Integer` und `Long` sind jeweils Methoden mit den Namen `getBoolean(String)`, `getInteger(String)` bzw. `getLong(String)` definiert. Diese Methoden nehmen einen Parameter vom Typ `String` entgegen. Intuitiv würde man ein Parsing und eine Umwandlung in eine Zahl erwarten, wie dies beim Aufruf von `valueOf(String)` geschieht. Tatsächlich erfolgt durch die zuvor genannten Methoden jedoch ein Zugriff auf die System-Properties. Zu dem übergebenen String wird der Wert des Parameters gleichen Namens ermittelt. Anschließend erfolgt eine Konvertierung in ein Objekt der jeweiligen Wrapper-Klasse. ***Diese Funktionalität ist dort nicht nur überflüssig, sondern führt auch zu einer Kopplung an die Klasse `System`.*** Das ist unerwartet und unnatürlich.

Beispiel: Parsen von Zahlen

Möchte man prüfen, ob ein beliebiger Eingabestring eine gültige Ganzzahl darstellt, so kann man dafür die statische Methode `Integer.parseInt(String)` nutzen. Aufgrund des beschränkten Wertebereichs eines `Integers` ist diese jedoch nur für Eingabewerte bis etwas über 2 Milliarden geeignet. Für größere Wertebereiche kann die statische Methode `Long.parseLong(String)` zum Einsatz kommen. Für Gleitkommazahlen sollte das Parsing mithilfe der statischen Methoden `parseFloat(String)` bzw. `parseDouble(String)` der Klassen `Float` bzw. `Double` erfolgen.

Folgendes Beispiel zeigt eine Prüffunktionalität, um festzustellen, ob ein übergebener String `numberAsText` in eine Gleitkommazahl umgewandelt werden

kann:

```
private static boolean isFloatingNumber( final String numberAsText)
{
    try
    {
        Double.parseDouble( numberAsText);
        return true;
    }
    catch ( final NumberFormatException ex)
    {
        return false;
    }
}

public static void main( final String[] args)
{
    System.out.println( "isNumber(47)  " + isFloatingNumber( "47"));         //
true
    System.out.println( "isNumber(47a)  " + isFloatingNumber( "47a"));       //
false
    System.out.println( "isNumber(47.11)  " + isFloatingNumber( "47.11"));   //
true
    System.out.println( "isNumber(47,11)  " + isFloatingNumber( "47,11"));   //
false
}
```

Listing 4.11 *Ausführbar als* **'PARSENUMBER'**

Für Nachkommastellen wird nur die amerikanische Notation mit einem Punkt statt eines Kommas korrekt ausgewertet. Benötigt man mehr Flexibilität, so müssen landesspezifische Formatierungen genutzt werden. Das wird in Abschnitt 10.1.4 besprochen.

4.2.3 Wissenswertes zu Auto-Boxing und Auto-Unboxing

Wie bereits erwähnt, stellen Auto-Boxing und Auto-Unboxing eine automatische Konvertierung zwischen primitiven Datentypen und deren korrespondierenden Wrapper-Klassen dar. Dies ermöglicht das Mixen von Wrapper-Klassen und primitiven Datentypen und erleichtert dadurch zum Teil die Handhabung sowie die Lesbarkeit. Diese Vereinfachungen haben aber einige unangenehme Nebeneffekte, die man kennen sollte.

Vergleich von primitiven Werten und Wrappern

Bis einschließlich JDK 1.4 konnte man primitive Typen nicht direkt mit den korrespondierenden Wrapper-Klassen vergleichen. Durch die mit JDK 5 eingeführte automatische Konvertierung sind nun nachfolgend gezeigte Vergleiche syntaktisch möglich:

```
// Auto-Unboxing von Integer:  7 == new Integer(7)
```

```
System.out.println(7 == new Integer(7));                    // true
System.out.println(7777 == new Integer(7777));              // true

// Auto-Unboxing (ebenfalls)
System.out.println(new Integer(7) == 7);
System.out.println(new Integer(7777) == 7777);              // true
System.out.println(new Integer(7777) == new Integer(7777)); // false
```

Beim Vergleich von primitiven Werten mit deren Wrapper findet immer ein Auto-Unboxing statt, wodurch der zweitletzte Vergleich `true` liefert. Das mag überraschen, wenn man denkt, dass der rechte Operand durch Auto-Boxing in einen `Integer` gewandelt wird. Dass dem nicht so ist, zeigt der letzte Vergleich zweier `Integer`-Instanzen. Bevor wir uns gleich weitere Fallstricke für die Wrapper-Klassen anschauen, möchte ich kurz erwähnen, dass die Wrapper-Klassen die `equals(Object)`-Methode überschreiben und dort den Wert des ummantelten primitiven Typs vergleichen. Dadurch liefern auch die neu konstruierten Objekte für den Wert 7777 Gleichheit:

```
System.out.println(new Integer(7777).equals(7777));             // true
System.out.println(new Integer(7777).equals(new Integer(7777))); // true
```

Nebenwirkungen und Fallstricke beim Auto-Boxing

Auto-Boxing führt zu einem geringfügig höheren Speicherverbrauch (z. B. 16 Bytes für ein `Integer`-Objekt statt 4 Bytes für einen `int`) und geht mit einer minimal schlechteren Ausführungsgeschwindigkeit einher, wenn sehr viele solcher Konvertierungen erfolgen. Um dem entgegenzuwirken, wurden in den Klassen `Integer` und `Long` jeweils Caches für Werte im Bereich von `-128` bis `127`[6] eingeführt. Genau diese Optimierung durch das Caching führt aber zu Merkwürdigkeiten beim Vergleich von `Integer`- bzw. `Long`-Objekten, wenn Auto-Boxing erfolgt.

Die Umwandlung von Zahlenliteralen durch Auto-Boxing verwendet zum Ermitteln des korrespondierenden Wrapper-Objekts implizit die statische Methode `valueOf()`. Dort wird zunächst der Zahlenwert geprüft. Stammt der angeforderte Wert aus dem Wertebereich des Caches, so werden keine neuen Objekte erzeugt, sondern Referenzen auf im Cache gespeicherte Werte zurückgeliefert. Dies vermeidet zwar unter Umständen die Erzeugung vieler temporärer Objekte, löst aber Merkwürdigkeiten beim Referenzvergleich aus, wie dies nachfolgendes Beispiel zeigt:

```
public static void main(final String[] args)
{
    // Cache = Objekt aus dem Cache, New = Neues Objekt
    final Integer i1 = 7;                               // Auto-Boxing => Cache
    final Integer i2 = 4711;                            // Auto-Boxing => New

    System.out.println(i1 == new Integer(7));           // false, Cache != New
    System.out.println(i2 == new Integer(4711));        // false, New != New
    System.out.println(i1 == Integer.valueOf(7));       // TRUE !!!, Cache ==
Cache
    System.out.println(i2 == Integer.valueOf(4711)); // false,  New != New
```

```
}
```

Listing 4.12 *Ausführbar als* **'AUTOBOXINGCACHEEXAMPLE'**

Die dargestellten Vergleiche sind allesamt Referenzvergleiche auf `Integer`-Objekten. Die Umwandlung der Zahlenliterale `7` und `4711` erfolgt durch die statische Methode `Integer.valueOf(int)`. Der dritte Vergleich `i1 == Integer.valueOf(7)` liefert daher (und nur zunächst überraschend) den Wert `true`, da hier gleiche Referenzen aus dem Cache verglichen werden. Die restlichen Vergleiche liefern erwartungsgemäß den Wert `false`, da es sich jeweils um unterschiedliche Referenzen handelt.

Nebenwirkungen und Fallstricke beim Auto-Unboxing

Nicht nur beim Auto-Boxing, sondern auch beim Auto-Unboxing gibt es Merkwürdigkeiten. Im folgenden Beispiel werden zwei `Integer`-Objekte `i1` und `i2` per Konstruktor erzeugt. Danach werden scheinbar drei mathematische Vergleiche durchgeführt:

```
public static void main(final String[] args)
{
    final Integer i1 = new Integer(1);
    final Integer i2 = new Integer(1);

    System.out.println(i1 >= i2);    // true
    System.out.println(i1 <= i2);    // true
    System.out.println(i1 == i2);    // false
}
```

Listing 4.13 *Ausführbar als* **'AUTOBOXINGUNBOXINGPROBLEM'**

Führt man das Programm aus, so liefern die Vergleiche mit den Operatoren '>=' und '<=', wie erwartet, jeweils `true`. Der Vergleich mit dem Operator '==' liefert jedoch (zunächst überraschend) `false`. Wie ist das zu erklären?

Die Operatoren '>=' und '<=' sind nur für primitive Datentypen, nicht aber für `Integer`-Objekte definiert. Für die gezeigten Vergleiche findet zunächst ein Auto-Unboxing statt, um den Vergleich auszuführen. Der Operator '==' ist für Objekte definiert und benötigt kein Auto-Unboxing, sondern arbeitet auf `Integer`-Referenzen. Es findet also ein Referenzvergleich statt, und somit kommt es zur Rückgabe von `false`.

Tipp: Spezialitäten beim Auto-Boxing

Beim Auto-Boxing gibt es neben den bereits kennengelernten Besonderheiten noch zwei weitere Spezialfälle zu betrachten.

Auto-Boxing und Methodensignaturen Um bei einem Methodenaufruf eine passende Methode zu finden, führt die JVM folgende Schritte durch:

1. Test auf exakte Übereinstimmung der Parametertypen, dann auf Subtyp

2. Anpassung durch Widening: `byte` ⇒ `short` ⇒ `int` ⇒ `long` usw.

3. Anpassung durch Auto-Boxing und Auto-Unboxing

Dieses Wissen ist hilfreich, wenn man sich fragt, welche Methoden die JVM für folgenden Sourcecode ausführt:

```
public static void main(final String[] args)
{
    final List<Integer> indexList = new ArrayList<>();
    indexList.add(7);
    indexList.add(8);

        //  Auto-Boxing  =>  Integer.valueOf(0)  =>  remove(Object)  oder
remove(int)?
    final Integer removed = indexList.remove(0);
    // Gemäß Regel 1 => remove(int)
    System.out.println("removed " + removed);

    final Integer value = 8;
    // Auto-Unboxing => 8 => remove(int) ?
    // Subtypbeziehung Integer extends Object => remove(Object)
    System.out.println(indexList.remove(value));
}
```

Listing 4.14 *Ausführbar als* **'AUTOBOXINGANDMETHODSIGNATUREEXAMPLE'**

Für das `int`-Zahlenliteral `0` findet kein Auto-Boxing statt. Gemäß Regel 1 wird die passende Methode `remove(int)` des `List<Integer>`-Interface aufgerufen. Für den Aufruf von `remove(value)` mit dem `Integer`-Objekt `value` wird basierend auf Regel 1 und der Subtypbeziehung von `Integer` zur Klasse `Object` die Methode `remove(Object)` ausgeführt.

Auto-Boxing und Arrays Weil Arrays Objekte sind, erfolgt weder Auto-Boxing noch Auto-Unboxing: Beispielsweise wird ein `int[]` nicht automatisch in ein `Integer[]` umgewandelt. Im folgenden Beispiel kann somit die Methode `processValues(Integer[])` nicht mit einem `int[]` als Eingabe arbeiten:

```
private static void processValues(final Integer[] intArray)
{
    System.out.println(Arrays.toString(intArray));
}
```

Das Verhalten scheint zunächst unverständlich und wirkt unpraktisch. Nach einiger Überlegung kommt man aber zum Schluss, dass es doch ganz gut ist, um negative Auswirkungen auf den Speicherverbrauch und die Performance zu vermeiden.

Fazit

Wie vorgestellt gibt es einige Merkwürdigkeiten beim Einsatz von Auto-Boxing und Auto-Unboxing. Daher ist beides mit etwas Vorsicht zu genießen. Insbesondere kann es zu einer `NullPointerException` kommen, wenn ein Auto-Unboxing einer `null`-Referenz erfolgt. Um unerwünschte Konvertierungen zu finden, sollte man in Eclipse eine entsprechende Warnung beim Kompilieren aktivieren (vgl. Abschnitt 2.1).

4.2.4 Ausgabe und Verarbeitung von Zahlen

In diesem Abschnitt stelle ich einige nützliche Methoden der Wrapper-Klassen zur Ausgabe und zur Verarbeitung von Zahlen vor.

Ausgabe von Zahlen

Die Objektmethoden `toString()` der Klassen `Integer` und `Long` geben den Wert eines Wrapper-Objekts im Dezimalsystem aus. Ebenso kann man einen beliebigen Wert mithilfe der statischen Methoden `toString(int)` bzw. `toString(long)` dezimal ausgeben. Manchmal soll eine Ausgabe jedoch bezüglich einer anderen Basis erfolgen. Dazu sind in den Klassen `Integer` und `Long` überladene statische `toString()`-Methoden definiert, die die Angabe einer beliebigen Basis erlauben:

```
public static void main(final String[] args)
{
    // Zur Basis 10 - Dezimaldarstellung
    System.out.println("15=" + Integer.toString(15));

    // Zur Basis 2 - Binärdarstellung
    System.out.println("15=" + Integer.toString(15, 2));   // 15=1111

    // Zur Basis 4
    System.out.println("15=" + Integer.toString(15, 4));   // 15=33

    // Zur Basis 8 - Oktaldarstellung
    System.out.println("15=" + Integer.toString(15, 8));   // 15=17

    // Zur Basis 16 - Hexadezimaldarstellung
    System.out.println("15=" + Integer.toString(15, 16));  // 15=f
}
```

Listing 4.15 *Ausführbar als* **'NUMBEROUTPUTEXAMPLE'**

Die Ausgabe in einigen Zahlensysteme ist so gebräuchlich, dass dafür spezielle statische Methoden der Klassen `Integer` und `Long` definiert sind. Diese Methoden können eine binäre, hexadezimale oder oktale Darstellung erzeugen. Dabei gibt es Folgendes zu bedenken:

1. Führende Nullen werden grundsätzlich *nicht* mit ausgegeben. Für das Dezimal- und Oktalsystem ist dies nicht weiter störend. Im Binär- und Hexadezimalsystem wäre aufgrund des eher technischen Bezugs ein Auffüllen mit führenden Nullen zum Teil wünschenswert.
2. Es findet eine Vorzeichenerweiterung statt, wodurch es zu einer etwas überraschenden Ausgabe für den Wert `-1` im Hexadezimalsystem kommt. Hier wird ein Implementierungsdetail der Zahlen sichtbar, nämlich die schon besprochene interne Repräsentation der Zahlen als Zweierkomplement:

```
public static void main(final String[] args)
{
    System.out.println( "15=" + Integer.toBinaryString(15) ); // 15=1111
```

```
    System.out.println( "15=" + Integer.toOctalString(15) );   // 15=17
    System.out.println( "15=" + Integer.toHexString(15) );     // 15=f
    System.out.println( "255=" + Integer.toHexString(255) );   // 255=ff
    System.out.println( "-1=" + Integer.toHexString(-1) );     // -1=ffffffff
}
```

Listing 4.16 *Ausführbar als* **'NUMBEROUTPUTEXAMPLESPECIAL'**

Konvertierung von Gleitkommazahlen in eine Bitdarstellung

Manchmal kann es nützlich sein, Gleitkommazahlen in eine Bitdarstellung umzuwandeln. Mithilfe der Methode `floatToIntBits(float)` ist es möglich, einen `float`-Wert in einen `int`-Wert mit einer Bitrepräsentation gemäß dem IEEE-754-Format zu transformieren. Aus einer solchen Bitrepräsentation lässt sich mit der Methode `intBitsToFloat(int)` wieder der korrespondierende `float`-Wert ermitteln. Die Klasse `Double` bietet zur Konvertierung analog folgende Methoden:

```
public static long doubleToLongBits(double value)
public static double longBitsToDouble(long bits)
```

Aufgrund des größeren Wertebereichs eines `double` wird hier die Bitrepräsentation jedoch in einem `long`-Wert gespeichert.

Extraktion einzelner Bytes aus einem `int`

Java bietet zur Manipulation auf Bitebene diverse Operationen an: NOT, AND, OR, XOR und Bit-Shifts. Die meisten Operationen arbeiten so, wie man sich das intuitiv vorstellt. Ein bitweiser Links-Shift füllt mit Nullen auf. Im Gegensatz dazu erhält der bitweise Rechts-Shift das Vorzeichen und füllt dazu mit dem jeweiligen Vorzeichenbit auf. Der alternative bitweise Rechts-Shift II füllt mit Nullen auf. Man verliert dadurch die Vorzeicheninformation. Derart veränderte Zahlen sollten nur als Bitmuster, jedoch nicht als Zahl im Zweierkomplement ausgewertet werden. Für viele Bitoperationen ist diese Einschränkung aber durchaus in Ordnung. In Tabelle 4-2 sind die genannten Bitoperationen (auf 8 Bit gekürzt) dargestellt.

Tabelle 4-2 *Bitoperationen*

Bitmuster	Operation	Operand	Ergebnis	Beschreibung
11001100	~	-/-	00**11**00**11**	NOT
11001100	&	00001111	**00**00**11**00	AND
11001100	|	00001111	11001**11**	OR
11001100	^	00001111	1100**0011**	XOR
11001100	≪	4	1100**0000**	Links-Shift
11001100	≫	4	**1111**1100	Rechts-Shift
11001100	≫>	4	**0000**1100	Rechts-Shift II

Die in Tabelle 4-2 vorgenommene Darstellung ist vereinfacht. Tatsächlich arbeiten die Operatoren nicht auf dem Typ `byte`, sondern auf dem Typ `int`. Schauen wir uns ein Beispiel an:

```
public static void main(final String[] args)
{
    final int value = 1 | 8 | 256 | 32768;
    System.out.println("Value = " + value);
    System.out.println("Binary Value = " + Integer.toBinaryString(value));

    final byte byte0 = extractByteValue(value);
    System.out.println("byte0 = " + byte0);

    System.out.println("value>>8 = " + (value>>8));
    final byte byte1 = extractByteValue(value>>8);
    System.out.println("byte1 = " + byte1);
}

public static byte extractByteValue(final int value)
{
    return (byte) (value & 255);
}
```

Listing 4.17 *Ausführbar als* **'EXTRACTBYTEEXAMPLE'**

Man erhält folgende Ausgaben auf der Konsole:

```
Value = 33033
Binary Value = 1000000100001001
byte0 = 9
value≫8 = 129
byte1 = -127
```

Die Verarbeitung startet mit dem Wert 33033, binär '`1000000100001001`'. Schneidet man davon die letzten 8 Bit durch die Operation `& 255` ab, so erhält man den Wert '`00001001`', der der Zahl 9 entspricht. Ein Bit-Shift der binären

Darstellung des Werts 33033 um 8 Stellen nach rechts führt zu dem Bitmuster '10000001', das als int den Wert 129 ergibt. Dieser Wert übersteigt den Wertebereich eines byte und somit wird das Bitmuster als negative Zahl ausgewertet. Dabei gilt die Formel *byteValue* = –255 + *x* – 1. Damit ergibt sich in diesem Fall –255 + 129 – 1 = –127 als Wert.

4.3 Stringverarbeitung

Zur Speicherung und Verarbeitung von Zeichenketten existieren in Java die Klassen String, StringBuffer und StringBuilder. Alle drei erfüllen das Read-only-Interface java.lang.CharSequence.

Das Interface CharSequence

Das Interface CharSequence bietet hauptsächlich einen indizierten Zugriff auf einzelne Zeichen vom Typ char bzw. auf Zeichenfolgen, die wiederum vom Typ CharSequence sind. Dazu sind folgende Methoden deklariert:

```
public interface CharSequence
{
    public char charAt(int index);
    public int length();
    public CharSequence subSequence(int start, int end);
    public String toString();
}
```

Das Interface CharSequence ermöglicht es also, Zeichenketten indiziert zu verarbeiten, ohne konkretes Wissen über den speziellen Typ zu besitzen. Dadurch kann man Schnittstellen allgemeiner gestalten, sofern nicht spezifische Funktionalitäten der Klassen String bzw. StringBuffer/-Builder benötigt werden.

Wissenswertes zum Interface CharSequence

Beachten Sie unbedingt, dass das Interface CharSequence keine Aussagen zum Verhalten und den Kontrakten von equals(Object) und hashCode() macht. Was bedeutet das genauer? Wenn man zwei Instanzen per equals(Object) vergleicht, die beide den Typ CharSequence besitzen, so ist das Ergebnis des Vergleichs undefiniert. Sind etwa beide Instanzen vom Typ String, so ist das Ergebnis bei gleichen textuellen Inhalten true. Eine CharSequence-Instanz könnte aber auch vom Typ StringBuffer sein. Ein Vergleich der Typen String und StringBuffer liefert immer false. Nachfolgendes Programm demonstriert einige Vergleiche:

```
public static void main(final String[] args)
{
    final CharSequence cs1 = new StringBuilder("same");
    final CharSequence cs2 = new StringBuilder("same");
```

```
    final CharSequence cs3 = "same";
    final CharSequence cs4 = "same";
    final CharSequence cs5 = new String("same");

    System.out.println(cs1.equals(cs2));  // false
    System.out.println(cs1.equals(cs3));  // false
    System.out.println(cs3.equals(cs4));  // true
    System.out.println(cs3.equals(cs5));  // true
}
```

Listing 4.18 *Ausführbar als* **'CHARSEQUENCEEQUALSEXAMPLE'**

Achtung: `CharSequence` in Collections und Maps

Basierend auf dieser Argumentation sollte man Instanzen vom Typ `CharSequence` niemals als Elemente eines `Set<E>` oder als Schlüssel einer `Map<K, V>` nutzen.

4.3.1 Die Klasse `String`

Die Klasse `String` repräsentiert Zeichenfolgen, die aus Unicode-Zeichen bestehen und ihren Inhalt in Form eines Arrays von `char` speichern. Solche Zeichenfolgen kann man entweder durch einen Konstruktoraufruf der Klasse `String` erzeugen oder aber als Zeichenkette in Anführungszeichen, wie dies folgende zwei Zeilen zeigen:

```
final String stringObject = new String("New String Object");
final String stringLiteral = "Stringliteral";
```

Im ersten Fall wird ein ***Stringobjekt*** und im zweiten Fall ein ***Stringliteral*** erzeugt – tatsächlich ist das Argument des Konstruktors im ersten Fall auch ein Literal. Es ist wichtig, zwischen diesen beiden Formen zu unterscheiden. Stringobjekte erhalten durch den Konstruktoraufruf immer eine eigenständige Referenz. Nutzt man Stringliterale, so wird nur einmalig eine neue `String`-Instanz erzeugt und in einen speziellen Cache, dem sogenannten ***Stringliteral-Pool***, eingetragen. Existiert dort für ein Stringliteral bereits ein Eintrag, so wird keine neue `String`-Instanz erzeugt, sondern gleiche Literale »teilen« sich Referenzen auf Stringobjekte aus diesem Stringliteral-Pool. Konkret bedeutet dies: Wird ein Literal mit gleichem Inhalt wie ein bereits vorhandenes erzeugt, so verweist dieses auf dasselbe Objekt.

Gäbe es dieses Caching nicht, so müsste für jedes Stringliteral eine temporäre `String`-Instanz erzeugt und bald danach verworfen werden. Demnach erfolgt das Caching der Stringliterale, um möglichen Performance-Problemen entgegenzuwirken. Abschnitt 22.6.3 geht auf mögliche Performance-Auswirkungen verschiedener Stringoperationen ein.

Wissenswertes zu Stringliteralen und Stringvergleichen

Die Kenntnis des Unterschieds zwischen einem Referenzvergleich mit dem Operator '==' und einem Inhaltsvergleich per `equals(Object)` bildet die Grundlage zum Verständnis einiger Besonderheiten beim Vergleich von Strings. Durch das Caching

von Stringliteralen gibt es Unterschiede zwischen Stringobjekten und Stringliteralen.[7]

Betrachten wir, was passiert, wenn man verschiedene Stringliterale und Stringobjekte miteinander vergleicht. Dazu definieren wir uns die beiden Konstanten `LITERAL` und `STRINGOBJECT` jeweils mit dem Inhalt "`TEST`":

```
public static void main(final String[] args)
{
    final String LITERAL = "TEST";
    final String STRINGOBJECT = new String("TEST");

    if (LITERAL == "TEST")
    {
        System.out.println("LITERAL == TEST");
    }
    if (STRINGOBJECT == "TEST")
    {
        System.out.println("STRINGOBJECT == TEST");
    }
    if (STRINGOBJECT == LITERAL)
    {
        System.out.println("STRINGOBJECT == LITERAL");
    }
    if (STRINGOBJECT.equals(LITERAL))
    {
        System.out.println("STRINGOBJECT equals LITERAL");
    }
}
```

Listing 4.19 *Ausführbar als* **'STRINGCOMPARELITERALEXAMPLE'**

Es kommt zu folgender Ausgabe auf der Konsole:

```
LITERAL == TEST
STRINGOBJECT equals LITERAL
```

Aufgrund der Ausgabe erinnern wir uns daran, dass ein Caching von Stringliteralen erfolgt. Insbesondere auf den positiven Ausgang des ersten Vergleichs darf man sich (eigentlich) nicht verlassen, denn er basiert rein auf dem Caching. Verwenden wir explizit neue `String`-Instanzen, so ist nur noch der `equals(Object)`-Vergleich erfolgreich.

Überraschungen bei Berechnungen in Stringausgaben

Die Klasse `String` bietet zum einfachen Verknüpfen von Texten einen »überladenen« Operator '+' an. Das ist sehr praktisch und erhöht die Lesbarkeit bei der Aufbereitung von textuellen Ausgaben. Allerdings ist bei Berechnungen die Reihenfolge der Operationen entscheidend. Betrachten wir dies an einem Beispiel:

```
System.out.println("Test" + 1 + 2);     // 'Test12'
System.out.println(1 + 2 + "Test");     // '3Test'
```

Die Ausgabe ist sehr unterschiedlich. Wie kommt das? Weil die Auswertung immer von links nach rechts erfolgt, kommt es dabei auf den Typ des linken Operanden an.

Beginnen wir mit der einfachen Variante: Bei der zweiten Ausgabe findet zunächst eine Addition von `int`-Werten statt und erst anschließend erfolgt eine Umwandlung in ein Stringobjekt durch Aufruf der statischen Methode `valueOf()`. Wenn der linke Operand bereits vom Typ `String` ist, so findet in jedem Fall eine Stringkonkatenation statt. Dadurch werden scheinbare Additionen von Zahlen nicht als solche ausgeführt, sondern die Zahlen werden zunächst in einen String gewandelt und dann an den ursprünglichen Text angehängt. Folgende Aufschlüsselungen der Abläufe verdeutlichen das Gesagte:

```
System.out.println("Test" + 1 + 2);
                        |   |
                  String + int => String + String.valueOf(int)
                           |
                      String + int => String + String.valueOf(int)
=> "Test12"

System.out.println(1 + 2 + "Test");
                     |   |
                 int + int => int
                       |
                    int + String => String.valueOf(int) + String
=> "3Test"
```

Dieses Beispiel scheint konstruiert zu sein. Tatsächlich treten solche Anweisungen in der Praxis aber doch häufiger durch Flüchtigkeitsfehler auf. Das kann überraschende und irreführende Ausgaben zur Folge haben, wie wir es nachfolgend für die berechnete Länge einer Nachricht sehen:

```
log.error("info '" + msg + "' too long! LENGTH: " + tele.size() + msg.length);
=> "… LENGTH: 19240"

log.error("info '" + msg + "' too long! LENGTH: " + (tele.size() +
msg.length));
=> "… LENGTH: 259"
```

Ein weiteres Beispiel ist die Berechnung und Ausgabe der Differenz von zwei Zeitpunkten `time0` und `time1`:

```
// System.out.println("Copy took: " + time1-time0 + " ms"); // Compile-Error
System.out.println("Copy took: " + (time1-time0) + " ms");
```

Die obere Anweisung kompiliert nicht, da der Operator '–' auf Strings nicht definiert ist. An diesen Beispielen sieht man, dass fehlerhafte Berechnungen schnell auftreten können.

Besonderheiten beim Einsatz von `toLowerCase()` und `toUpperCase()`

Die Methoden `toLowerCase()` und `toUpperCase()` der Klasse `String` dienen dazu, einen gegebenen String in eine einheitliche Schreibweise in Klein- bzw. Großschreibung zu konvertieren. Das scheint trivial. Ist es aber nicht in jedem Fall. Mathematisch ausgedrückt wird keine bijektive Abbildung realisiert: Es gibt demnach nicht immer eine eineindeutige Hin- und Zurück-Transformation der Strings.

Intuitiv ist dies klar, weil etwa "TeST" und "TesT" jeweils in Kleinschreibung als "test" dargestellt werden. Es gibt aber subtilere Fallstricke beim Konvertieren von Groß- in Kleinschreibung und umgekehrt. Betrachten wir anhand des Worts »Fußball« einige Konvertierungsschritte:

```
Original: 'Fußball' toLower: 'fußball' toUpper: 'FUSSBALL' toLower: 'fussball'
```

Durch Hintereinanderausführung von Konvertierungen ist aus dem Text "Fußball" der Text "fussball" geworden, der nicht eindeutig auf "Fußball" rückabbildbar ist.

Unveränderlichkeit von Strings

Strings sind in Java als unveränderliche Objekte realisiert. Dadurch kommt es leicht zu Flüchtigkeitsfehlern in der Anwendung.[8] Betrachten wir dies an einem Beispiel:

```
public static void main(final String[] args)
{
    String test = "Heute";
    test.concat(" ist ein schöner Tag!");

    System.out.println(test);
}
```

Listing 4.20 *Ausführbar als* **'STRINGIMMUTABILTYEXAMPLE'**

Intuitiv würde man die Ausgabe "Heute ist ein schöner Tag! " erwarten. Die tatsächliche Ausgabe auf der Konsole lautet jedoch "Heute". Die Erklärung ist einfach: ***Stringverändernde Methoden liefern immer eine neue[9] Stringinstanz zurück, die die Änderungen enthält.*** Es muss demzufolge eine Zuweisung an eine Variable erfolgen, da ansonsten die neu erzeugte Stringinstanz und damit die Modifikation verloren geht. Folgende Operationen erzeugen neue Instanzen:

- +, +=
- concat(), replace(), subString()
- toUpperCase(), toLowerCase()

Verdeutlichen wir uns mögliche Folgen der temporären Instanzen aufgrund der Unveränderlichkeit an einem Beispiel:

```
public static void internalProcessingString()
{
    // Achtung: Schlechter Einsatz von Stringobjekten
    final String objectAndLiterals = new String("Dies ") + "ist " +
                                     new String("ein " + "String");
    System.out.println(objectAndLiterals);
}
```

Zunächst werden drei Stringliterale "Dies ", "ist " und "ein String" erzeugt. Wieso nur drei statt der vier im Sourcecode vorhandenen? Es erfolgt eine Optimierung durch den Compiler: Mit dem Operator '+' verknüpfte Stringliterale, hier "ein " und "String", werden zu einem Stringliteral zusammengefasst, bevor sie mit dem Stringliteral-Pool abgeglichen werden.

Nach den Andeutungen über ständig neu erzeugte Stringobjekte könnte man annehmen, intern würde etwa Folgendes ausgeführt:

```
public static void internalProcessingIdea()
{
    String temp  = new String( "Dies " );
    String temp2 = new String( "Dies ist " );
    final String objectAndLiterals = new String( "Dies ist ein String");
    System.out.println( objectAndLiterals);
}
```

Wäre dem so, dann würden zusätzlich zu den drei Stringliteralen also drei neue Stringobjekte erzeugt. Es bestünde dann die Gefahr, dass es bei Unachtsamkeit zu sehr vielen temporären Stringobjekten kommen könnte. Tatsächlich ist der Compiler schlauer und konvertiert eine '+'-Konkatenation von Stringobjekten in eine Konkatenation mithilfe der im folgenden Abschnitt vorgestellten Klasse `StringBuilder`.

4.3.2 Die Klassen `StringBuffer` und `StringBuilder`

Die Veränderlichkeit von textuellen Daten ist wünschenswert ist, aber für die Klasse `String` aufgrund deren Unveränderlichkeit nicht gegeben ist. Für Stringmanipulationen existieren die Klassen `StringBuffer` und `StringBuilder`. Beide besitzen ein identisches API mit dem Unterschied, dass die Methoden der Klasse `StringBuffer` synchronisiert sind. Das bedeutet, dass die Methoden jeweils nur von einem Thread zur Zeit aufgerufen werden können (vgl. Kapitel 7.2.1). Das gilt für die Methoden der Klasse `StringBuilder` nicht, was nicht stört, solange darauf nur ein Thread arbeitet.

Stringkonkatenation

Ein häufig anzutreffender Tipp ist, diese Klassen statt des Operators '+' zur Konstruktion umfangreicher Textausgaben einzusetzen. Tatsächlich muss man das Ganze etwas differenzierter betrachten. Einfache Stringkonkatenationen mit dem Operator '+', wie die aus dem obigen Beispiel der Methode `internalProcessingString()`, werden bei der Kompilierung bereits automatisch wie folgt durch den Einsatz eines `StringBuilder`-Objekts ersetzt:

```
public static void internalProcessingReal()
{
    final String objectAndLiterals = new StringBuilder().
                                         append( new String( "Dies " )).
                                         append( "ist " ).
                                         append( new String( "ein String" )).
                                         toString();
    System.out.println( objectAndLiterals);
}
```

Es wird bei Stringkonkatenationen immer zunächst ein `StringBuilder`-Objekt erzeugt und anschließend jedes Vorkommen eines Strings (Objekt oder Literal) mit der Methode `append()` hinzugefügt. Betrachten wir dies exemplarisch:

```
final String example = "text start" + "…" + "text end";
```

Der resultierende Bytecode ist identisch mit der folgenden Umsetzung, wie dies im folgenden Praxistipp anhand des erzeugten Bytecodes nachgewiesen wird.

```
final String example = new StringBuilder().append("text start").
                                           append("…").
                                           append("text end").toString();
```

Damit ist nachvollziehbar, warum es bei Stringverknüpfungen in der Regel keinen Sinn macht, auf die gut lesbare Schreibweise der Konkatenation durch den Operator '+' zu verzichten. Allerdings gibt es ein Einsatzgebiet, wo die explizite Verwendung eines `StringBuilders` sinnvoll sein kann. Dies ist immer dann der Fall, wenn sehr häufig mit dem Operator '+=' gearbeitet wird. Entwickeln wir ein Beispiel für die sinnvolle Nutzung eines `StringBuilders`:

```
String example = "text start";
example += "text end";
```

Für diese Anweisungen findet eine Transformation in folgende Aufrufe statt:

```
String example = "text start";
example = new StringBuilder().append(example).
                              append("text end").toString();
```

Es werden also für jeden Aufruf des Operators '+=' temporär eine `StringBuilder`-Instanz sowie ein Stringobjekt erzeugt. Geschieht dies innerhalb einer Schleife, so würde die Transformation Folgendes ergeben – unter der Annahme, die zu konkatenierenden Werte würden über eine Methode `getValue(int)` ermittelt:

```
String example = "text start"
for (int i = 0; i < n; i++)
{
    example = new StringBuilder().append(example).
                                  append(getValue(n)).toString();
}
example = new StringBuilder().append(example).
                              append("text end").toString();
```

IDIOM: PERFORMANTE STRINGKONKATENATION

Für Performance-kritische Abschnitte sollte man daher Stringkonkatenationen mithilfe eines `StringBuilder`-Objekts explizit nach folgendem Idiom realisieren:

```
String example = "text start";
final StringBuilder sb = new StringBuilder(example);
for (int i = 0; i < n; i++)
{
    sb.append(getValue(n));
}
example = sb.append("text end").toString();
```

Auf diese Weise vermeidet man sämtliche kurzlebigen temporären Objekte, die durch

eine automatische Transformation entstanden wären.

Tipp: Class-File-Disassembler-Werkzeug `javap`

Um die Details der Verarbeitung und die der Umsetzung in Bytecode sichtbar zu machen, kann man sich des Tools `javap` bedienen. Ein Aufruf von `javap <ClassFileName>` (Achtung: ohne Endung `.class`) zeigt unter anderem die erzeugten Konstanten sowie die Bytecode-Anweisungen der Methoden. Nutzt man `javap` zur Untersuchung der obigen Methode `internalProcessingString()`, so erhält man folgende Ausgabe:

```
public static void internalProcessingString();
  Code:
   Stack=4, Locals=1, Args_size=0
   0:   new     #5; //class java/lang/StringBuilder
   3:   dup
   4:   invokespecial   #6; //Method java/lang/StringBuilder."<init>":()V
   7:   new     #7; //class java/lang/String
   10:  dup
   11:  ldc     #8; //String Dies
   13:  invokespecial   #9; //Method java/lang/String."<init>":(Ljava/lang/
        String;)V
   16:  invokevirtual   #10; //Method java/lang/StringBuilder.append:(Ljava/
        lang/String;)Ljava/lang/StringBuilder;
   19:  ldc     #11;  //String ist
   21:  invokevirtual    #10; //Method java/lang/StringBuilder.append:(Ljava/
        lang/String;)Ljava/lang/StringBuilder;
   24:  new     #7; //class java/lang/String
   27:  dup
   28:  ldc     #12; //String ein String
   30:  invokespecial   #9; //Method java/lang/String."<init>":(Ljava/lang/
        String;)V
   33:  invokevirtual   #10; //Method java/lang/StringBuilder.append:(Ljava/
        lang/String;)Ljava/lang/StringBuilder;
   36:  invokevirtual   #13; //Method java/lang/StringBuilder.toString:()
        Ljava/lang/String;
   39:  astore_0
   40:  getstatic       #14; //Field java/lang/System.out:Ljava/io/
        PrintStream;
   43:  aload_0
   44:  invokevirtual   #15; //Method java/io/PrintStream.println:(Ljava/lang
        /String;)V
   47:  return
```

Fallstrick bei `equals()` von `StringBuffer/-Builder`

Bei der Aufbereitung von Stringrepräsentationen mithilfe der Klassen `StringBuffer` und `StringBuilder` benötigt man immer mal wieder eine Prüfung auf inhaltliche Gleichheit. Für Strings ist bekannt, dass man per == die Referenzen und mit `equals(Object)` deren Inhalt vergleicht. Übertragen wir diese Situation auf `StringBuffer` und `StringBuilder`. Intuitiv könnte man annehmen, dass deren `equals(Object)`-Methode auf inhaltliche Gleichheit prüft, wie dies für die Klasse `String` der Fall ist. Dann würde man etwa Folgendes schreiben:

```
final StringBuilder sb1 = new StringBuilder("Text");
final StringBuilder sb2 = new StringBuilder("Text");
// ACHTUNG: Falsche Lösung
final boolean contentIsEqual = sb1.equals(sb2)
```

Wenn man das Programm ausführt, liefert der Vergleich den Wert `false`, da hier (überraschenderweise) ein Referenzvergleich durchgeführt wird. Somit kann man `StringBuffer`- bzw. `StringBuilder`-Objekte nicht sinnvoll per `equals(Object)` vergleichen, sondern muss folgendes Idiom dafür nutzen.

IDIOM: PRÜFUNG AUF INHALTLICHE GLEICHHEIT

Wenn man den Inhalt zweier `StringBuffer`- bzw. `StringBuilder`-Objekte `sb1` und `sb2` miteinander vergleichen möchte, so muss man das wie folgt realisieren:

```
final boolean contentIsEqual = sb1.toString().equals(sb2.toString())
```

Dabei entstehen allerdings zwei Stringobjekte. Falls ein Objekt vom Typ `String` ist, kann man die Methode `contentEquals(CharSequence)` aus der Klasse `String` für die Prüfung auf inhaltliche Gleichheit nutzen:

```
final boolean stringContentIsEqual = string.contentEquals(charSequence);
```

Weitere Funktionalität und Vergleich mit der Klasse `String`

Praktisch ist, dass die beiden Klassen `StringBuffer` und `StringBuilder` Löschoperationen besitzen: Über die Methoden `deleteCharAt()` und `delete()` können Zeichen aus einer Stringrepräsentation entfernt werden. Analoge Methoden namens `insert()` erlauben es, Zeichen einzufügen. Das stellt einen Vorteil gegenüber Stringobjekten dar, wo dies nicht möglich ist. Allerdings haben sowohl der `StringBuilder` als auch der `StringBuffer` einen Nachteil, denn sie bieten (unverständlicherweise) leider diverse Methoden nicht, etwa `toLowerCase()` und `toUpperCase()` – dafür jedoch die (weniger häufig benötigte) Methode `reverse()`, die den Inhalt in umgekehrter Reihenfolge liefert. Tabelle 4-3 zeigt einige wichtige Methoden der Klasse `String` und eine Abbildung auf diejenigen des `StringBuffers` bzw. `StringBuilders`. Es existieren weitere Unterschiede und nicht dargestellte Methoden.

Tabelle 4-3 *Abbildung der Stringmethoden*

String	StringBuffer/-Builder
+, +=, concat()	append()
replace(), subString()	replace(), subString()
indexOf(),startsWith()	indexOf()
endsWith()	lastIndexOf()
- ***keine Methode vorhanden!*** -	reverse()
- ***keine Methode vorhanden!*** -	insert()
- ***keine Methode vorhanden!*** -	delete(), deleteCharAt()
toUpperCase()/toLowerCase() -	***keine Methode vorhanden!*** -

4.3.3 Ausgaben mit `format()` und `printf()`

Immer wieder müssen Zahlen, Datumsangaben und Texte formatiert werden. Die im Anschluss vorgestellten Methoden `String.format()` und `PrintStream.printf()` ermöglichen eine Ausgabe, wie sie unter C mit `printf()` erfolgt. Für viele Aufgaben reicht das bereits aus. Eine objektorientierte und flexiblere Realisierung bieten die `Format`-Klassen, die in Abschnitt 10.1.4 genauer vorgestellt werden.

Ausgaben mit `String.format()` aufbereiten

Mit der statischen Methode `format(String, Object[])` der Klasse `String` kann man Zeichenketten nach einem speziellen Muster mit Platzhaltern formatieren.

```
public static void main(final String[] args) throws IOException
{
    final Object[] sampleArgs = { "pi", 3.1415, 12345 };
    final String str = String.format("%S='%2.5f' Zahl='%,d'", sampleArgs);

    System.out.println(str); // PI='3,14150' Zahl='12.345'
}
```

Listing 4.21 *Ausführbar als* **'STRINGFORMATEXAMPLE'**

Die Platzhalter werden bei der Ausgabe durch übergebene Werte ersetzt. Nachfolgende Aufzählung nennt mögliche Platzhalter (vgl. die API-Dokumentation):

- '`%s`' steht für eine Ausgabe eines Strings,

'%S' wandelt den Wert in Großbuchstaben um.

- '%d' dient zur Ausgabe von dezimalen Zahlen,

  '%,d' gibt dezimale Zahlen mit Tausenderpunkte aus.
- '%f' bzw. '%m.nf' dient zur Ausgabe von Gleitkommazahlen, wobei m und n die Anzahl der Vor- bzw. Nachkommastellen festlegt. Ohne Angabe werden immer die benötigte Anzahl Vorkommastellen und sechs Nachkommastellen ausgegeben.
- '%x' bzw. '%X' gibt hexadezimale Zahlen aus.
- '%b' steht für eine Ausgabe von booleschen Werten.

Ausgaben mit `System.out.printf()`

Die Methode `printf(String, Object[])` gibt es in den Klassen `PrintWriter` und `PrintStream` und steht somit auch im Standardausgabestream `System.out` zur Verfügung. Dadurch wird eine formatierte Ausgabe direkt möglich, ohne zunächst ein Stringobjekt über `format(String, Object[])` aufzubereiten und anschließend über `System.out.println()` auszugeben. Folgendes Listing zeigt beide Varianten:

```
public static void main(final String[] args)
{
    final String str = String.format("Hi %s. Es ist %d Uhr.", "Mike", 12);
    System.out.println(str);   // Hi Mike. Es ist 12 Uhr.

    System.out.printf("Hi %s. Es ist %d Uhr.", "Mike", 12);
}
```

Listing 4.22 *Ausführbar als* **'STRINGFORMATVARARGSEXAMPLE'**

In beiden Fällen kommt es erwartungsgemäß zu der Ausgabe von »`Hi Mike. Es ist 12 Uhr.`«.

4.3.4 Die Klasse `StringTokenizer`

Die Klasse `StringTokenizer` erlaubt es, eine Zeichenkette in einzelne Bestandteile, sogenannte ***Tokens***, zu zerlegen, und implementiert zum Zugriff auf diese das Interface `java.util.Enumeration<Object>`. Dieses bietet die Methoden `boolean hasMoreElements()` und `Object nextElement()`. Die Klasse `StringTokenizer` besitzt zwei spezialisierte Methoden `boolean hasMoreTokens()` und `String nextToken()`. Damit lassen sich die Werte direkt als Strings verarbeiten, wodurch man beim Zugriff die ansonsten benötigten Casts vom Typ `Object` auf den Typ `String` vermeidet.

Die Klasse `StringTokenizer` nutzt zum Zerlegen lediglich einzelne Zeichen als Trennsymbole. Voreingestellt sind hier Tabulator, Leerzeichen und Zeilenumbruch. Trotz dieser Einschränkung auf einzelne Trennzeichen kann die

Klasse `StringTokenizer` beim Verarbeiten von Strings oder beim Lesen von Konfigurationsdateien nützlich sein.

Beispiel: Extraktion einer Versionsnummer

Betrachten wir ein konkretes Beispiel. Es sollen Versionsnummern aus einem String extrahiert werden, der diese im Format '`Major.Minor.Patchlevel`' enthält. Zusätzlich finden sich in der Eingabe weitere Informationen etwa zu Applikationsnamen und Build-Details, die wir ignorieren wollen. Nehmen wir außerdem an, der Aufbau der Eingabe sei aufgrund von Inkonsistenzen ähnlich, aber nicht identisch. Für die Ermittlung der Versionsinformationen sollen gewisse Unterschiede jedoch irrelevant sein. Das Parsing muss daher tolerant erfolgen, d. h., die Zeichen '. ' und '-' sowie '_' als Trennsymbole akzeptieren. Mögliche, gültige Eingaben sollen sein:

```
APPNAME 2.14.75 build b257
APPNAME 2-22_44 build b278 20.03.2009
```

Die folgende Methode `extractMajorVersion(String)` ermittelt die Major-Version, indem zunächst die nicht benötigten Bestandteile der Eingabe mithilfe einer Methode `cutOffAppnameAndBuild(String)` entfernt werden. Als Ergebnis befindet sich im String `versions` eine Versionsnummer, die von einem `StringTokenizer` in einzelne Bestandteile zerlegt wird.

```
public static int extractMajorVersion(final String strImplementationVersion)
{
    final String versions = cutOffAppnameAndBuild(strImplementationVersion);

    final StringTokenizer tokenizer = new StringTokenizer(versions, ". _-");
    if (tokenizer.hasMoreTokens())
    {
        final String versionValue = tokenizer.nextToken().trim();
        return Integer.parseInt(versionValue);
    }
    throw new IllegalArgumentException("passed version has no major version");
}
```

Zum Teil kann es nützlich sein, die Anzahl der vorhandenen Tokens zu erfragen. Dies geschieht über die Methode `int countTokens()`. Für die Extraktion der Minor-Version kann man durch Abfrage der Anzahl gefundener Tokens sicherstellen, dass mindestens zwei Tokens vorhanden sind. Das Ganze realisiert man wie folgt:

```
public static int extractMinorVersion(final String strImplementationVersion)
{
    final String versions = cutOffAppnameAndBuild(strImplementationVersion);
    final StringTokenizer tokenizer = new StringTokenizer(versions, ". _-");

    if (tokenizer.countTokens() > 1)
    {
        tokenizer.nextToken(); // skip major version

        final String versionValue = tokenizer.nextToken().trim();
        return Integer.parseInt(versionValue);
    }
```

```
    throw new IllegalArgumentException("passed version has no minor version");
}
```

4.3.5 Die Methode `split()` und das 1x1 der regulären Ausdrücke

Nachdem wir im vorherigen Abschnitt bereits einfache Zerlegungen von Zeichenketten mithilfe der Klasse `StringTokenizer` kennengelernt haben, wollen wir uns in diesem Abschnitt schrittweise komplexere Zerlegungen basierend auf regulären Ausdrücken und der Methode `split(String)` der Klasse `String` anschauen. Diese teilt einen gegebenen String in mehrere Teilstrings. Sie arbeitet dabei mit frei wählbaren Trennzeichen(-folgen), die als regulärer Ausdruck angegeben werden. Für das Verständnis regulärer Ausdrücke wollen wir folgende informelle Definition verwenden:

Definition 4.1 *Unter einem **regulären Ausdruck** versteht man eine Zeichenkette, die zur Beschreibung von anderen Zeichenketten dient. Im einfachsten Fall kann dies eine Folge von Zeichen sein. Die Darstellung gewisser Abfolgen oder Alternativen geschieht durch Spezial- oder Metazeichen, etwa [,], -, + usw. Seien a, b, c, d und e Zeichen, so gelten folgende einfache Regeln zur Konstruktion komplexerer regulärer Ausdrücke:*

Ausdruck	Bedeutung
a	**Einzelnes Zeichen**: Das Zeichen 'a' ist erlaubt.
a+	**Wiederholung**: Ein oder mehrere 'a' sind erlaubt.
[ace]	**Zeichenauswahl**: Eines der Zeichen 'a', 'c' oder 'e' ist erlaubt.
[b-d]	**Bereichsdefinition**: Eines der Zeichen aus dem Bereich von 'b' bis 'd' ist erlaubt.
a[c-e]	**Konkatenation**: Eine Zeichenkette, die mit einem 'a' beginnt und danach eines der Zeichen 'c', 'd' oder 'e' enthält, ist erlaubt.

Im Folgenden werde ich mit einem Beispiel einer einfachen Auswertung von Texten beginnen und dort schrittweise weitere Anforderungen integrieren.

Einfache Zeichenfolgen als regulärer Ausdruck

Nehmen wir an, in einer Textdatei wären verschiedene Werte durch die Zeichenfolge `"#-#"` voneinander getrennt. Wir nutzen einen einfachen regulären Ausdruck, der

exakt dieser Zeichenfolge entspricht. Im folgenden Beispiel ist dieser in der Konstanten `delimiter` definiert. Es wird die Eingabe `input` durch Aufruf der Methode `split(String)` aufgespalten:

```
public static void main(final String[] args)
{
    final String input = "#-# Wert1 #-# Wert2 #-# Wert3";
    final String delimiter = "#-#";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens); // Tokens = '[, Wert1 , Wert2 , Wert3]'
}

private static void printTokens(final String[] tokens)
{
    System.out.print("Tokens = '" + Arrays.asList(tokens) + "'");
}
```

Listing 4.23 *Ausführbar als* **'REGEXEXAMPLE'**

Als Ausgabe erhalten wir:

```
Tokens = '[, Wert1 , Wert2 , Wert3]'
```

Nur bei genauerer Betrachtung erkennt man, dass die Tokens teilweise Leerzeichen enthalten. Wir verbessern daher die Ausgaberoutine `printTokens(String[])` so, dass einzelne Tokens in Anführungszeichen eingeschlossen werden:

```
static void printTokens(final String[] tokens)
{
    System.out.print("Tokens = '[");
    for (int i = 0; i < tokens.length; i++)
    {
        System.out.print("'" + tokens[i] + "'");
        // Komma anfügen, wenn noch nicht letztes Element
        if (i != tokens.length - 1)
            System.out.print(", ");
    }
    System.out.println("]'");
}
```

Listing 4.24 *Ausführbar als* **'REGEXEXAMPLEIMPROVEDPRINT'**

Die Ausgabe in Hochkommata einzuschließen haben wir bereits für Stringausgaben kennengelernt. Auch hier werden Resultate klarer erkennbar:

```
Tokens = '[", ' Wert1 ', ' Wert2 ', ' Wert3']'
```

Nun fällt auf, dass nicht nur die Tokens Leerzeichen enthalten, sondern dass insbesondere auch die Ausgabe mit einem Leerstring beginnt. Dies ist dadurch begründet, dass das erste Vorkommen eines Trennzeichens an Position 0 liegt. Das Token davor hat demnach die Länge 0. Bei der Angabe der Trennzeichen muss man ebenfalls aufmerksam sein: Leerzeichen tragen dort semantische Bedeutung.

Varianten in regulären Ausdrücken

Ein Erweiterungswunsch könnte sein, Trennzeichen mit einer laufenden Nummer zu versehen. Die Trenner wären dann die Zeichenfolgen "`#1#`", "`#2#`" usw. Da wir

bisher lediglich auf Zeichenebene vergleichen, müssen wir alle erlaubten Muster in der Art "#1#", "#2#" usw. angeben und dazu eine ***ODER-Beziehung*** nutzen. Diese ***wird in regulären Ausdrücken über das Zeichen '|' dargestellt***. Der reguläre Ausdruck lautet damit:

```
final String delimiter = "#1#|#2#|#3#";
```

Je mehr der invariante Anteil des Musters dominiert, desto umfangreicher wird die Zeichenfolge zur Beschreibung der Trenner. Besser ist es, nur die Varianzen darzustellen. Schnell kommt der Wunsch auf, alle Ziffern von '0' bis '9' als Trennzeichen verwenden zu dürfen. Dies kann explizit durch Angabe aller Ziffern geschehen, etwa wie folgt:

```
public static void main(final String[] args)
{
    final String input = "#1# Wert1 #5# Wert5 #7# Wert7";
    final String delimiter = "#(0|1|2|3|4|5|6|7|8|9)#";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens);
}
```

Listing 4.25 *Ausführbar als* **'REGEXEXAMPLEVARIANTEN'**

Bereichsangaben und Wiederholungen in regulären Ausdrücken

Eleganter kann man das Ganze durch die Angabe von Wertebereichen schreiben, die die Notation `[Startwert-Endwert]` verwendet:

```
final String delimiter = "#[0-9]#";
```

Wahrscheinlich sollen die erlaubten Zahlen nicht nur einstellig bleiben. Eine Wiederholung von Zeichen wird über die Notation '*' (0 bis n Vorkommen) bzw. '+' (1 bis n Vorkommen) ausgedrückt. Die Angabe der Vorkommen (Multiplizität) lässt sich für beliebige Zeichen und Ausdrücke, hier im Speziellen für die '#'-Zeichen, anwenden. Wir können die Trenner nun flexibel gestalten und fordern lediglich, dass ein Trenner immer mit mindestens einem '#'-Zeichen beginnt und endet sowie mindestens eine Ziffer und optional weitere enthält. Damit ergibt sich der folgende reguläre Ausdruck:

```
public static void main(final String[] args)
{
    final String input = "#27# Wert27 ##228## Wert228 #3# Wert3";
    final String delimiter = "#+[0-9]*#+";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens);
}
```

Listing 4.26 *Ausführbar als* **'REGEXEXAMPLERANGES'**

Spezialzeichen in regulären Ausdrücken

Um Schreibarbeit in regulären Ausdrücken zu vermeiden, existieren einige vordefinierte Abkürzungen als Hilfe. Beispielsweise werden beliebige Ziffern über die Angabe von "\d" repräsentiert. Der zuvor hergeleitete Ausdruck lässt sich damit weiter verdichten, allerdings muss das Backslash-Zeichen in Strings speziell notiert werden (man spricht auch von Escapen), hier durch "\d":

```
public static void main(final String[] args)
{
    final String input = "#27# Wert27 ##228## Wert228 #3# Wert3";
    final String delimiter2 = "#+\d+#+";

    final String[] tokens2 = input.split(delimiter2);
    printTokens(tokens2);
}
```

Listing 4.27 *Ausführbar als* **'REGEXEXAMPLERANGES2'**

Einige gebräuchliche Spezialzeichen sind in Tabelle 4-4 aufgelistet.

Tabelle 4-4 *Spezialzeichen in regulären Ausdrücken*

Spezialzeichen	Bedeutung
.	Platzhalter für ein beliebiges Zeichen
\d	Platzhalter für eine beliebige Ziffer
\w	Platzhalter für einen Buchstaben oder eine Ziffer
\s	Platzhalter für Leerzeichen, Tabulatoren und Zeilenumbrüche
\n	Platzhalter für ein Zeilenumbruch
*	Angabe einer Wiederholung von 0 bis n Vorkommen
+	Angabe einer Wiederholung von 1 bis n Vorkommen
{n}	Wiederholung von genau n Vorkommen
?	Erlaubt 0 bis 1 Vorkommen

Man ahnt, dass das Muster des regulären Ausdrucks schnell unübersichtlich bzw. kryptisch wird. Dies gilt vor allem, wenn man nach diesen Spezialzeichen suchen möchte. Zur Angabe als String in Java ist dann ein weiteres Escapen notwendig.

Ein aus der Praxis stammendes Beispiel für den Einsatz spezieller Trennzeichen ist ein Newsticker, dessen Nachrichten durch die Zeichenfolgen "+++" voneinander separiert sind. Möchte man die einzelnen News-Texte ermitteln, so könnte man

intuitiv auf folgende Realisierung kommen:

```
final String input = "News: First News +++ Second News +++ End";
final String delimiter = "+++";
// Exception in thread »main« java.util.regex.PatternSyntaxException:
// Dangling meta character '+' near index 0
```

Diese Umsetzung führt jedoch zur Laufzeit zu einer `java.util.regex.PatternSyntaxException`, ausgelöst dadurch, dass hier ein Spezialzeichen mit Metainformation als Trennzeichen genutzt wird. Das löst man durch Escapen wie folgt:

```
public static void main(final String[] args)
{
    final String input = "News: First News +++ Second News +++ End";
    final String delimiter = "\+\+\+";

    final String[] tokens = input.split(delimiter);
    printTokens(tokens);
}
```

Listing 4.28 *Ausführbar als* **'REGEXEXAMPLENEWSTICKER'**

Fallstricke beim Ersetzen eines `StringTokenizers` durch `split()`

Kommen wir nochmal zum Beispiel der Extraktion von Versionsnummern zurück. Für den Fall, dass man statt der Klasse `StringTokenizer` einen Aufruf der Methode `split(String)` der Klasse `String` nutzt, muss man vorsichtig sein. Eine naive Umsetzung würde einfach die Trennzeichen aus dem `StringTokenizer` als Eingabe für die `split(String)`-Methode verwenden. Das ist im folgenden Listing als Versuch 1 mit einer korrespondierenden Methode `versuch1()` gezeigt:

```
public static void main(final String[] args)
{
    versuch1();
    versuch2();
    versuch3();
}

private static void versuch1()
{
    final String versions = "2.14-17";
    final String[] values = versions.split(". _-");
    printTokens(values);    // Tokens = '[ '2.14-17' ]'
}
```

Listing 4.29 *Ausführbar als* **'STRINGSPLITEXAMPLE'**

Schauen wir, was passiert: Die Versionsinformationen werden nicht korrekt geliefert, sondern es wird der komplette String zurückgegeben. Das liegt an der Formulierung der Trennzeichenkette **". _-"**, die als regulärer Ausdruck aber für ein beliebiges Zeichen gefolgt von den Zeichen '_' und '-' steht. Eine solche Zeichenfolge ist jedoch in dem Eingabewert `versions` nicht vorhanden. Der Grund ist, dass

`split(String)` nach einer Zeichenfolge und nicht nach jeweils einzelnen Trennzeichen sucht. Eine erste Idee könnte sein, eine ODER-Beziehung wie folgt zur Korrektur zu nutzen:

```
private static void versuch2()
{
    final String versions = "2.14-17";
    final String[] values2 = versions.split(".|_|-");
    printTokens(values2);   // Tokens = '[]'
}
```

Dadurch ist auf einmal die Treffermenge leer. Wieder muss man kurz nachdenken, dann ist die Erklärung einleuchtend: Durch das Spezialzeichen '. ' wird die gesamte Eingabe als Trennzeichen interpretiert und es bleibt kein Wert übrig. ***Die Verwendung von Spezialzeichen als Trennzeichen muss demnach besonders behandelt werden***. Dies lässt sich durch Escapen lösen, wodurch man ein zum `StringTokenizer` kompatibles Verhalten erhält:

```
private static void versuch3()
{
    final String versions = "2.14-17";
    final String[] values3 = versions.split("\.|_|-");
    printTokens(values3);   // Tokens = '['2', '14', '17']'
}
```

Tipp: Reguläre Ausdrücke mit Vorsicht einsetzen

Nachdem wir einige Grundlagen zu regulären Ausdrücken kennengelernt haben, möchte ich abschließend noch zeigen, wie man die zuvor für den `StringTokenizer` vorgestellte Extraktion der Versionsinformationen auch gestalten kann. Betrachten wir dazu folgendes Listing:

```
public static void main(final String[] args)
{
   final String[] inputs = { "prog 1.X", "prog-1.1", "prog 1.2 PATCH_15" };

   // Pattern zum Extrahieren von Versionsnummern mithilfe von Gruppen
   final Pattern pattern = Pattern.compile("[A-Za-z]+[- ]([0-9]+)" +
                           "\.([0-9]+)[-_]?([0-9A-Za-z._-]+)?");

   for (final String current : inputs)
   {
      final Matcher matcher = pattern.matcher(current);
      if (matcher.matches())
      {
         final String major = matcher.group(1);
         final String minor = matcher.group(2);
         final String patch = matcher.group(3);

         System.out.println(current + " => major=" + major +
                            ", minor=" + minor + ", patch=" + patch);
      }
   }
}
```

Das Programm produziert folgende Ausgaben:

```
prog-1.1 => major=1, minor=1, patch=null
```

```
prog 1.2 PATCH_15 => major=1, minor=2, patch=PATCH_15
```

Experimentieren Sie etwas mit diesem Beispiel, so wird Folgendes deutlich: Reguläre Ausdrücke können recht komplex und teilweise schwierig wartbar werden. Außerdem wird schnell die Intention unklar und selbst minimale Änderungen können leicht zu drastischen Änderungen oder gar zu Fehlern führen. Es empfiehlt sich daher ausdrücklich das Schreiben von Unit Tests, die wir in Kapitel 20 genauer kennenlernen werden. Zudem ist es sehr ratsam, die gewünschte Funktion zu kommentieren.

4.4 Datumsverarbeitung

Für die Verarbeitung von Datumswerten gibt es diverse APIs in Java und seit JDK 8 sogar noch ein umfangreiches weiteres, das man – sofern möglich – in neuen Projekten einsetzen sollte. Die Neuerungen in JDK 8 besprechen wir separat in Kapitel 13.

Nachfolgend wollen wir zunächst das `Date`- und das `Calendar`-API betrachten, um damit die eingangs dieses Kapitels vorgestellte Klasse `Person` um eine Methode `getAge()` zu erweitern sowie die Ausgabe des Geburtstags in `toString()` zu verbessern. Bevor wir damit starten, werde ich auf einige Fallstricke hinweisen, die in den Datums-APIs lauern.

4.4.1 Fallstricke der Datums-APIs

Die ursprünglich zur Datumsverarbeitung vorgesehene Klasse `java.util.Date` steckt voller Tücken und Überraschungen. Das lernen wir an ein paar Beispielen kennen. Anschließend schauen wir auf Probleme in der Klasse `java.util.Calendar`.

Konstruktion von `Date`-Instanzen

Zur Demonstration reicht bereits ein sehr einfaches Programm: Erzeugen wir ein neues `Date`-Objekt mit der Angabe der Werte Jahr, Monat, Tag wie folgt:

```
public static void main(final String[] args)
{
    // Mein Geburtstag: 7.2.1971
    final int year = 1971;
    final int month = 2;
    final int day = 7;

    System.out.println(new Date(year, month, day));
}
```

Listing 4.30 *Ausführbar als* **'DATEAPIPROBLEMS1'**

Zunächst ist die Parameterreihenfolge Jahr, Monat, Tag nicht besonders intuitiv. Gebräuchlich ist Tag, Monat, Jahr oder Monat, Tag, Jahr. Abgesehen davon erwartet

man als Ergebnis eine Ausgabe in der folgenden Art – zumindest in Deutschland:

```
7. 2. 1971
```

Aber stattdessen werden wir in den März des Jahres 3871 katapultiert:

```
Tue Mar 07 00:00:00 CET 3871
```

Wie kann das sein? Ein Blick in die Sourcen des JDKs verrät es. Auf den an den Konstruktor übergebenen Jahreswert wird immer der Wert 1900 addiert. Zudem beginnt die Zählung der Monate bei 0, die der Tage bei 1. Das ist alles andere als selbsterklärend. Mit diesem Wissen fällt die Korrektur jedoch nicht schwer:

```
System.out.println(new Date(year - 1900, month - 1, day));
```

Berechnungen mit `Date`

Die Klasse `Date` speichert intern das Datum in Form eines `long`-Werts in Millisekunden mit dem 1.1.1970, 00:00:00 Uhr als Referenzzeitpunkt. Mithilfe der Methode `getTime()` kann man diesen `long`-Wert ermitteln. Es ist ebenfalls möglich, ein neues Datum inklusive Uhrzeit durch Angabe von Millisekunden und den Aufruf von `setTime(long)` zu setzen. Alternativ kann die Konstruktion eines `Date`-Objekts mit einem `long`-Wert erfolgen. Ein Aufruf des Defaultkonstruktors `new Date()` verwendet den aktuellen Zeitpunkt als `long`-Wert. Ein negativer Wert geht vom oben angegebenen Referenzzeitpunkt in die Vergangenheit zurück. Folgendes Listing versetzt die Zeit auf den Silvesterabend 1969:

```
public static void main(final String[] args)
{
    System.out.println(new Date(-10_000_000));  // Wed Dec 31 22:13:20 CET
1969
}
```

Listing 4.31 *Ausführbar als* **'DATEAPIPROBLEMS2'**

Beim Einsatz des `Date`-APIs erkennen wir eine weitere Inkonsistenz: ***Die Verarbeitung über Millisekunden verwendet mit dem Jahr 1970 ein anderes Offset als die Angabe über die Werte für Jahr, Monat usw. im Konstruktor der Klasse `Date`. Dort ist die Basis das Jahr 1900.***

Merkwürdigkeiten in der Klasse `Calendar`

Mit JDK 1.2 wurde zur Datumsverarbeitung die Klasse `Calendar` eingeführt. Dieser zweite Versuch ist besser gelungen als die Klasse `Date`. Die Übergabe der Parameter und deren Interpretation sind nun relativ überraschungsfrei:

```
public static void main(final String[] args)
{
    final Calendar calendar = new GregorianCalendar(1971, 1, 7);
    System.out.println(calendar.getTime());     // Sun Feb 07 00:00:00 CET
1971

    final Calendar calendar2 = new GregorianCalendar(1971, 1, 7, 21, 22);
    System.out.println(calendar2.getTime());    // Sun Feb 07 21:22:00 CET
```

```
1971
}
```

Listing 4.32 *Ausführbar als* **'DATEAPIPROBLEMS3'**

Leider besitzt auch das `Calendar`-API einige kleinere Tücken: Erstens beginnt die Zählung der Monate hier wieder bei 0. Zweitens ist die Methode zum Ermitteln eines `Date`-Objekts unglücklich als `getTime()` benannt. Dadurch sieht man häufiger folgendes Konstrukt, um aus einem `Calendar`-Objekt einen `long`-Wert zu ermitteln:

```
final long time = calendar.getTime().getTime();
```

Der doppelte `getTime()`-Aufruf erscheint zunächst unsinnig oder zumindest merkwürdig. Tatsächlich liefert das erste `getTime()` ein `Date`-Objekt und das zweite einen `long`-Wert.

4.4.2 Das `Date`-API

Die vorherigen Beispiele verdeutlichen, dass es einige Merkwürdigkeiten und Probleme im `Date`-API gibt. Deswegen sind dort ca. 60 – 70% der öffentlichen Methoden und der Konstruktoren als veraltet gekennzeichnet und sollten in neu erstelltem Sourcecode möglichst nicht mehr verwendet werden. Vielmehr sollte man das `Date`-API höchstens noch zur Verarbeitung von Zeitangaben in Millisekunden nutzen.

Weil wir hier das `Date`-API kennenlernen wollen und weil es auch noch in vielen Projekten genutzt wird, werden wir es entgegen dem Hinweis trotzdem nutzen, um die schon erwähnte Erweiterung der Klasse `Person` um eine Methode `getAge()` zur Berechnung des Alters zu realisieren. Die Klasse `Date` bietet unter anderem folgende Methoden:

- `new Date()` – Erzeugt ein neues `Date`-Objekt, das die aktuelle Zeit und das aktuelle Datum widerspiegelt (`System.currentTimeMillis()`).
- `getYear()` – Liefert das Jahr.
- `getMonth()` – Liefert den Monat, 0-basiert! (Wertebereich 0 … 11).
- `getDate()` – Liefert entgegen der Bezeichnung der Methode nur den Tag im Monat, 1-basiert! (Wertebereich 1 … 31)
- `getDay()` – Liefert ***nicht(!)*** den Tag, sondern einen Aufzählungswert für den Wochentag. (Wertebereich 0 … 6 entspricht Sonntag … Samstag)

Abgesehen von `getDay()` werden wir diese Methoden nun für Berechnungen des Alters in Jahren einsetzen. Dieses ergibt sich vereinfacht aus der Differenz des aktuellen Jahrs und des Geburtsjahrs. Es muss gegebenenfalls noch eine Korrektur erfolgen, wenn der aktuelle Tag und Monat kleiner als der des Geburtstags sind, also man noch nicht in dem Jahr Geburtstag hatte: Die Jahresdifferenz muss dann um eins verringert werden. Mit diesen Informationen schreiben wir die Methode `getAge()` wie folgt:

```
public final int getAge()
{
    final Date now = new Date();

    int correction = 0;
    if (!birthdayWasAlreadyThisYear(now.getMonth(), birthday.getMonth(),
                                    now.getDate(), birthday.getDate()))
    {
        correction = -1;
    }

    return now.getYear() - birthday.getYear() + correction;
}

private boolean birthdayWasAlreadyThisYear(final int monthNow,
            final int monthBirthDay, final int dayNow, final int dayBirthDay)
```

```
{
    return monthNow > monthBirthDay ||
          (monthNow == monthBirthDay && dayNow >= dayBirthDay);
}
```

Vergleich von Datumswerten mit dem `Date`-API

Datumswerte lassen sich leicht und intuitiv mithilfe der Methoden `before(Date)`, `equals(Object)` und `after(Date)` der Klasse `Date` vergleichen. Damit kann man Vergleiche auf logischer Ebene formulieren. Das ist oftmals verständlicher, als wenn man die korrespondierenden `long`-Werte (Implementierungsdetails) der `Date`-Objekte vergleicht. Folgender Sourcecode-Ausschnitt zeigt beide Varianten:

```
public static void main(final String[] args)
{
    final long ONE_HOUR_MSEC = 60 * 60 * 1000;

    final Date now = new Date();
    final Date oneHourAgo = new Date(now.getTime() - ONE_HOUR_MSEC);

    // oneHourAgo < now
    System.out.println(oneHourAgo.before(now));
    System.out.println(oneHourAgo.getTime() < now.getTime());
}
```

Berechnungen von Datumswerten mit dem `Date`-API

Wie schon mit den vorherigen Beispielen angedeutet, finden Berechnungen im `Date`-API häufig auf Basis von `long`-Werten (Millisekunden-Repräsentationen) statt, wodurch sich die Berechnungen meistens nicht wirklich gut lesbar gestalten lassen. Um beispielsweise einen Bereich von einem Tag in die Vergangenheit bis zu einer Woche in die Zukunft abzudecken, müssen die korrespondierenden Werte von Hand berechnet werden. Das ist ziemlich unpraktisch. Daher bietet sich der Einsatz einiger Konstanten an. Nachfolgend wählen wir als Ausgangstag den 28. Februar 2000:

```
public static void main(final String[] args)
{
    final long ONE_SEC_MSEC = 1000;
    final long ONE_MIN_MSEC = 60 * ONE_SEC_MSEC;
    final long ONE_HOUR_MSEC = 60 * ONE_MIN_MSEC;
    final long ONE_DAY_MSEC = 24 * ONE_HOUR_MSEC;
    final long ONE_WEEK_MSEC = 7 * ONE_DAY_MSEC;

    // Schaltjahr 2000 => Offset Jahr: 1900
    final Date feb2000 = new Date(100,1,28);
    final Date oneDayAgo = new Date(feb2000.getTime() - ONE_DAY_MSEC);
    final Date oneWeekFromNow = new Date(feb2000.getTime() + ONE_WEEK_MSEC);

    System.out.println("oneDayAgo = " + oneDayAgo);
    System.out.println("feb2000 = " + feb2000);
    System.out.println("oneWeekFromNow = " + oneWeekFromNow);

    // oneDayAgo < feb2000 < oneWeekFromNow
```

```
    System.out.println(oneDayAgo.before(feb2000));
    System.out.println(feb2000.before(oneWeekFromNow));
}
```

Listing 4.33 *Ausführbar als* **'DATECALCULATIONEXAMPLE'**

Das obige Programm DATECALCULATIONEXAMPLE produziert folgende Ausgabe:

```
oneDayAgo = Sun Feb 27 00:00:00 CET 2000
now = Mon Feb 28 00:00:00 CET 2000
oneWeekFromNow = Mon Mar 06 00:00:00 CET 2000
true
true
```

Ausgehend vom Referenztag 28. Februar ist der Tag davor erwartungsgemäß der 27. Februar. Da das Jahr 2000 ein Schaltjahr ist, gibt es aber mit dem 29. Februar noch einen weiteren Tag in diesem Monat. Demnach entsprechen 7 Tage in der Zukunft dem 6. März und nicht dem 7. März, wie man zunächst vermuten könnte.

Diese Berechnungen waren noch relativ einfach. Komplizierter wird es, wenn man ein Datum um mehrere Monate verschieben möchte. Dann müsste man die unterschiedliche Anzahl von Tagen im Monat und möglicherweise auch Schaltjahre berücksichtigen. Für solche Berechnungen bietet sich der Einsatz der Klasse `Calendar` an.

Hinweis: Schwierigkeit nachträglicher API-Korrekturen

Das `Date`-API zeigt es sehr deutlich: Wurden beim Entwurf eines APIs Fehler gemacht und ist dieses erst einmal veröffentlicht und durch eine breite Nutzerbasis eingesetzt, so lässt sich das Ganze kaum mehr korrigieren, wenn man kompatibel zu bestehendem Sourcecode bleiben möchte. Stattdessen können Methoden lediglich als veraltet markiert werden. Dazu nutzt man den Javadoc-Kommentar `@deprecated`, die Annotation `@Deprecated` oder besser sogar beides. Idealerweise existiert ein Hinweis, welche Methode stattdessen genutzt werden sollte.

4.4.3 Das `Calendar`-API

Wie bereits erwähnt, wurde das `Calendar`-API eingeführt, um die Handhabung von Datumswerten zu erleichtern. Die abstrakte Klasse `Calendar` bietet einen objektorientierteren Zugang zur Verwaltung von Datumswerten als die Klasse `Date`. Wir setzen im Folgenden die konkrete Realisierung `GregorianCalendar` ein, um eine Datumsausgabe zu realisieren sowie Berechnungen von Datumswerten durchzuführen.

Aufbereitung für eine Ausgabe als String

Für Datumsberechnungen enthält die Klasse `Calendar` unter anderem eine spezielle

`get(int)`-Methode. Diese bietet Zugriff auf einzelne Datumswerte (Jahr, Monat, Tag usw.). Der Parameter referenziert über vordefinierte Schlüsselwerte den gewünschten Datumsteil:

- `DAY_OF_MONTH` – Tag im Monat (Wertebereich 1 … 31)
- `MONTH` – Monat, 0-basiert! (Wertebereich 0 … 11)
- `YEAR` – Jahr
- `HOUR_OF_DAY` – Stunde im 24 Stundensystem, 0-basiert! (Wertebereich 0 … 23)
- `MINUTE` – Minute

Nutzen wir nun das `Calendar`-API, um die Ausgabe des Geburtstags einer Person lesbarer und nach unseren Wünschen zu gestalten. Wie wir beim Implementieren der `toString()`-Methode in Abschnitt 4.1.1 gesehen haben, entspricht die Umwandlung eines `Date`-Objekts in einen String nicht unseren Erwartungen, denn in diesem Beispiel soll sie weder Zeitzonen noch sekundengenaue Zeitangaben enthalten, sondern lediglich im Format 'Tag.Monat.Jahr Stunde:Minute' erfolgen. Eine mögliche Realisierung, basierend auf der `get(int)`-Methode des `Calendar`, wird durch folgende `dateAsString(Date)`-Methode implementiert:

```
public static String dateAsString(final Date date)
{
    final Calendar cal = GregorianCalendar.getInstance();
    cal.setTime(date);

    final StringBuilder result = new StringBuilder();

    result.append(cal.get(Calendar.DAY_OF_MONTH));
    result.append('.');
    result.append(cal.get(Calendar.MONTH) + 1);
    result.append('.');
    result.append(cal.get(Calendar.YEAR));
    result.append(' ');
    result.append(cal.get(Calendar.HOUR_OF_DAY));
    result.append(':');
    result.append(cal.get(Calendar.MINUTE));

    return result.toString();
}
```

Richtig elegant und gut erweiterbar sind derartige »händische« Lösungen zur Aufbereitung von Ausgaben nicht. Einfacher und weniger fehleranfällig wird dies durch den Einsatz der in Abschnitt 10.1.4 vorgestellten `Format`-Klassen, die zudem eine flexible Anpassung der Ausgabe ermöglichen.

Berechnungen von Datumswerten mit dem `Calendar`-API

Berechnungen lassen sich im `Calendar`-API auf einer logischen Ebene ausführen und man muss nicht auf der Millisekunden-Repräsentation arbeiten. Als Konsequenz kann jedes beliebige Datumsattribut, etwa Jahr, Monat, Tag, Stunde, Minute, um

einen Wert erhöht oder erniedrigt werden. Dazu dient die Methode `add(int field, int value)`, die das korrespondierende Datumsattribut sowie die gewünschte Änderung übergeben bekommt.

Eine solche Änderung wirkt sich – falls erforderlich – auf ein »höherwertiges« Datumsattribut aus: Würde im Dezember zwei Monate in die Zukunft gesprungen, so ändert sich neben dem Monat auch das Jahr. Will man wirklich explizit nur auf dem jeweiligen Datumsattribut rechnen, so nutzt man dazu die Methode `roll(int field, int value)`. Des Weiteren kann man Werte über die Methode `set(int field, int value)` setzen. Für Wochentage und Monate existieren dazu vordefinierte Konstanten, beispielsweise `Calendar.SUNDAY` oder `Calendar.FEBRUARY`.

Wir wollen nun die zuvor mit dem `Date`-API implementierten Berechnungen mit dem `Calendar`-API ausführen. Praktischerweise müssen die korrespondierenden Zeitdifferenzen *nicht mehr* von Hand berechnet werden, sondern es kommt die Methode `add(int, int)` zum Einsatz, wodurch sich eine natürliche Datumsarithmetik realisieren lässt und der Sourcecode kürzer, klarer und verständlicher wird:

```
public static void main(final String[] args)
{
    // Schaltjahr 2000 => Berechnung bei Verschiebung um 1 Monat wichtig
    final Calendar now = new GregorianCalendar(2000, Calendar.FEBRUARY, 28);

    // Einen Tag in die Vergangenheit
    final Calendar oneDayAgo = (Calendar) now.clone();
    oneDayAgo.add(Calendar.DAY_OF_YEAR, -1);

    // Eine Woche in die Zukunft
    final Calendar oneWeekFromNow = (Calendar) now.clone();
    oneWeekFromNow.add(Calendar.WEEK_OF_YEAR, +1);

    // Ausgabe als Date-Objekt
    System.out.println("oneDayAgo = " + oneDayAgo.getTime());
    System.out.println("now = " + now.getTime());
    System.out.println("oneWeekFromNow = " + oneWeekFromNow.getTime());

    // Schaltjahr 2000 => oneDayAgo < now < oneWeekFromNow
    System.out.println(oneDayAgo.before(now));
    System.out.println(now.before(oneWeekFromNow));
}
```

Listing 4.34 *Ausführbar als* **'CALENDARCALCULATIONEXAMPLE'**

Das Listing zeigt, dass sich die Berechnungen lesbarer als mit der Klasse `Date` gestalten lassen. Führt man das Programm CALENDARCALCULATIONEXAMPLE aus, so kommt es zu folgender Ausgabe, die derjenigen der `Date`-Berechnungen entspricht:

```
oneDayAgo = Sun Feb 27 00:00:00 CET 2000
now = Mon Feb 28 00:00:00 CET 2000
oneWeekFromNow = Mon Mar 06 00:00:00 CET 2000
true
true
```

4.5 Innere Interfaces und innere Klassen

Nachfolgend stelle ich die Techniken innere Interfaces und innere Klassen vor. Beide helfen dabei, Informationen lokal zu halten und eine bessere Strukturierung zu erzielen.

Es sei daran erinnert, dass vor allem Packages zur Gruppierung und Strukturierung dienen. Innerhalb eines Packages können beliebig viele Klassen und Interfaces sowie weitere Subpackages definiert werden. Packages definieren einen Namensraum: Innerhalb von Packages müssen Namen von Klassen und Interfaces eindeutig sein, während diese Forderung über Package-Grenzen hinweg nicht existiert.

Hinweis: Bessere Designs mithilfe von Packages und Interfaces

Wenn Klassen aus mehreren Packages bzw. Komponenten genutzt werden sollen, trägt es zu einer besseren Trennung von Zuständigkeiten und zur loseren Kopplung bei, wenn deren Funktionalität über Interfaces angeboten wird. Dadurch vermeidet man Implementierungsabhängigkeiten zwischen Packages bzw. Komponenten. Das betrifft aber oftmals nur wenige zentrale von außen zugreifbare Funktionalitäten. Für jede Klasse ein Interface anzubieten, hilft nicht im Design und führt zur sogenannten »Interaceitis«.

4.5.1 Varianten innerer Klassen

In Java ist der Einsatz innerer Klassen ein verbreitetes Sprachmittel, mit dem sich einige Entwurfsprobleme elegant lösen lassen. Innere Klassen sind den gewöhnlichen Klassen ähnlich. Der Unterschied liegt darin, dass sie – wie es der Name bereits nahelegt – innerhalb von Klassen (oder sogar Methoden) definiert werden.

Innere Klassen stellen häufig Funktionalität bereit, die lediglich für die äußere Klasse von Interesse ist. Bei Bedarf können innere Klassen jedoch auch von anderen Klassen benutzt werden. Java bietet zwei verschiedene Varianten innerer Klassen: *normale* und *statische innere* Klassen. Beide Varianten sind im folgenden Listing gezeigt:

```
public class OuterClass
{
    public class InnerClass
    {
        // …
    }

    public static class StaticInnerClass
    {
        // …
    }
```

```
}
```

Die Sichtbarkeit einer äußeren Klasse wirkt sich auf dort definierte innere Klassen aus. Im Gegensatz zu äußeren Klassen, die nur die Sichtbarkeiten `public` und Package-private erlauben, sind für innere Klassen alle Sichtbarkeiten zulässig. Wenn die äußere Klasse nur Package-private definiert ist, sind innere Klassen für Package-fremde Klassen nicht sichtbar, selbst wenn die inneren Klassen `public` sind. Ist die äußere Klasse allerdings `public`, so gelten für die Sichtbarkeit die bekannten Regeln: `private` und Package-private definierte innere Klassen sind nur im Package selbst sichtbar. Öffentliche innere Klassen sind für alle anderen Klassen sichtbar. Die Sichtbarkeit `protected` ermöglicht den Zugriff für von der äußeren Klasse abgeleitete Klassen und für Klassen aus demselben Package.

»Normale« innere Klassen

Innere Klassen besitzen eine implizite Referenz auf eine Instanz der äußeren Klasse und können dadurch auf deren Elemente zugreifen – sogar auf die privaten. ***Allerdings folgt daraus auch, dass immer ein Objekt der umgebenden Klasse existieren muss, um ein Objekt einer inneren Klasse zu erzeugen***. Dadurch ergibt sich eine etwas merkwürdige Syntax zur Objekterzeugung:

```
// Variante 1
final OuterClass.InnerClass inner = new OuterClass().new InnerClass();

// Variante 2
final OuterClass outer = new OuterClass();
final OuterClass.InnerClass inner2 = outer.new InnerClass();
```

Statische innere Klassen

Wenn äußere Klassen lediglich erzeugt werden, um Zugriff auf innere Klassen zu bieten, so ist dies – wie eingangs dieses Unterkapitels gezeigt – eher unnatürlich. Das gilt insbesondere dann, wenn es sich bei den inneren Klassen um Hilfsklassen oder Datencontainer handelt. Diese dienen nur als semantische Strukturierung und sind in der Regel unabhängig von der äußeren Klasse. Dafür existiert das Sprachmittel der statischen inneren Klassen. Das illustriert die Klasse `TripleVO`. Diese realisiert einen Datencontainer, der die drei Attribute `value1` bis `value3` anbietet:

```
public final class StaticInnerClassExample
{
    public static final class TripleVO
    {
        private final int value1;
        private final int value2;
        private final int value3;

        private TripleVO(final int value1, final int value2, final int value3)
        {
            this.value1 = value1;
            this.value2 = value2;
            this.value3 = value3;
```

```
        }

        public final int getValue1() { return value1; }
        public final int getValue2() { return value2; }
        public final int getValue3() { return value3; }
    }

    // …
}
```

Statische innere Klassen besitzen keine implizite Referenz auf die äußere Klasse, wodurch auch keine Zugriffe auf nicht statische Attribute der äußeren Klassen möglich sind. Statische innere Klassen können mit folgender Syntax erzeugt werden:

```
final OuterClass.StaticInnerClass inner = new OuterClass.StaticInnerClass();
```

Spezielle Formen innerer Klassen

Neben den bisher vorgestellten Formen von inneren Klassen existieren noch die zwei im Folgenden vorgestellten Spezialformen: *methodenlokale* und *anonyme innere* Klassen.

Methodenlokale innere Klassen

Innere Klassen können sogar lokal innerhalb von Methoden definiert werden. Allerdings kann für diese speziellen Klassen keine Sichtbarkeit angegeben werden. Sie sind lediglich innerhalb der definierenden Methode sichtbar. Diesen lokalen inneren Klassen ist sowohl ein Zugriff auf Attribute der äußeren Klasse als auch (eingeschränkt) auf die in der Methode definierten Variablen und Methodenparameter möglich:

```
private void doSomething()
{
    int variable = 100;
    final int constant = 200;

    // Keine Sichtbarkeitsmodifier erlaubt, public usw. => Compile-Error
    /* public */ class MethodLocalInnerClass
    {
        public void printVar()
        {
            // System.out.println("variable = " + variable ); => Compile-Error
            System.out.println("constant = " + constant);
        }
    }

    new MethodLocalInnerClass().printVar();
}
```

Der Zugriff auf lokal in der Methode definierte Variablen ist bis einschließlich JDK 6 jedoch nur möglich, wenn diese Variablen `final` definiert sind – das gilt ebenfalls für den Zugriff auf Methodenparameter.[10] Dies ist dadurch bedingt, dass eine Stackvariable (etwa `int variable = 100`) beim Verlassen einer Methode

»abgeräumt« wird. Eine methodenlokale innere Klasse kann zu diesem Zeitpunkt aber weiterhin aktiv sein (weil sie einen Thread gestartet hat) und könnte später versuchen, auf eine solche Variable zuzugreifen, die es dann aber nicht mehr geben würde. Durch die Definition als `final` wird der Zugriff trotzdem möglich. Wie ist das zu erklären? Man kann sich das Vorgehen in etwa so vorstellen: Das Schlüsselwort `final` garantiert, dass sich der Wert einer Variablen im Verlauf der Methode nicht ändert. Dadurch ist es möglich, implizit eine Kopie des Werts der Variablen zu erstellen und diesen an die methodenlokale innere Klasse zu übergeben. Somit arbeitet diese mit einer Kopie statt mit dem Original aus der umgebenden Methode.

Interessanterweise kann man `enum`-Aufzählungen im Gegensatz zu Klassen nicht methodenlokal definieren.[11] Das dürfte allerdings auch nur in extrem seltenen Fällen notwendig sein. Für diese kann dann das ENUM-Muster aus Abschnitt 3.4.4 verwendet werden.

Anonyme innere Klassen

Manchmal sind innere Klassen so speziell und nur für eine einmalige Aufgabe verwendbar, dass man sie weder benennen noch mehrfach erzeugen möchte. Das lässt sich mithilfe anonymer innerer Klassen realisieren. Diese sind innerhalb von Klassen oder Methoden definiert und besitzen keinen Namen, sind also wirklich anonym. Ohne Klassennamen können sie allerdings keinen Konstruktor bereitstellen. Meistens bestehen diese Klassen nur aus wenigen Methoden – häufig sogar lediglich aus einer Methode. Man spricht von ***SAM-Typen*** (***Single Abstract Method***). Diese spielen für JDK 8 und ***Lambdas*** eine wichtige Rolle. Dieses Thema werden wir später in Kapitel 11 genauer betrachten. Anonyme innere Klassen sind in ihrer Definition dahingehend eingeschränkt, dass sie entweder auf einem Interface basieren oder eine Klasse erweitern müssen. Der ausschließliche Sinn besteht darin, die Methoden der Basisklasse zu überschreiben bzw. die Methoden eines Interface zu implementieren. Zwar kann man in einer anonymen inneren Klasse zusätzliche Methoden definieren. Diese können jedoch von außerhalb der Klasse niemals aufgerufen werden.

Die zur Definition verwendete Schreibweise ist zunächst vielleicht etwas gewöhnungsbedürftig, da weder das Schlüsselwort `extends` noch `implements` angegeben wird. Stattdessen folgt die Klassendefinition syntaktisch direkt der Instanziierung der Basisklasse bzw. des Interface:[12]

```
final Runnable newRunnable = new Runnable()
{
    public void run()
    {
        // …
    }
}; // ACHTUNG DAS SEMIKOLON IST WICHTIG => SONST COMPILE-ERROR => OCPJP/SCJP
```

4.5.2 Innere Interfaces

Bevor wir gleich auf innere Interfaces eingehen, rekapitulieren wir kurz einige Grundlagen zu Klassen und Interfaces: In Java kann eine Klasse immer nur von genau einer Basisklasse erben. Vererbungshierarchien, in denen man sich aus verschiedenen Basisklassen »bedient«, sind dadurch nicht möglich. Das verhindert automatisch eine Implementierungswiederverwendung durch Vererbung ohne Beachtung der Substituierbarkeit (»is-a«-Beziehung). Allerdings ist es seit Java 8 möglich, sogenannte ***Defaultmethoden*** in Interfaces zu definieren und dort Implementierungen vorzugeben. Da eine Klasse durchaus mehrere Interfaces erfüllen kann, sind nun Situationen möglich, wo es zu Konflikten durch mehrere Implementierungen von Methoden kommt. Details beschreibt Abschnitt 11.2.

Um das Konzept verschiedener Verhaltensweisen oder Rollen ausdrücken zu können, ist es wünschenswert, mehrere Typen realisieren zu können. Daher ist eine Implementierung mehrerer Interfaces möglich (vgl. Abschnitt 3.3.2).

Im Speziellen kann man Interfaces innerhalb von Klassen definieren. Diese inneren Interfaces sind implizit statisch, d. h., auf sie kann auch ohne eine Instanz der äußeren Klasse über die Zugriffssyntax statischer innerer Klassen zugegriffen werden. Dabei sind sämtliche Sichtbarkeiten von `private` bis `public` möglich. In folgendem Beispiel sind die Varianten Package-private und `public` gezeigt:

```
public class InnerInterfaceExample
{
    interface Calculator
    {
        int calc(int value1, int value2);
    }

    public interface Calculator2
    {
        int calc2(int value1, int value2);
    }
}
```

Hinweis: Auffindbarkeit von inneren Klassen und inneren Interfaces

Als weitere Strukturierungsstufe zu Packages existieren die zuvor besprochenen inneren Klassen und inneren Interfaces. Allerdings lassen sich diese nur über Tools (IDE) finden, da es keine Dateien mit korrespondierenden Namen gibt. Gleiches gilt für die in Abschnitt 4.5.4 besprochenen lokal definierten Klassen und Interfaces.

4.5.3 Designbeispiel mit inneren Klassen und Interfaces

Nachdem wir in den vorangegangenen Abschnitten einige Details zu inneren Klassen und inneren Interfaces kennengelernt haben, betrachten wir nun ein etwas komplexeres Beispiel. Zur Demonstration der technischen Möglichkeiten von

inneren Klassen und inneren Interfaces wird hier bewusst nicht die einfachste Lösung umgesetzt.

Ein inneres Interface `Calculation` definiert eine Schnittstelle mit zwei Methoden `calc()` und `getName()`. Die abstrakte Klasse `AbstractCalculation` realisiert dieses Interface und implementiert die Methode `getName()`. Spezifische Berechnungen werden in den konkreten Berechnungsklassen `Plus`, `Minus` und `Modulo` durchgeführt. Die unterschiedlichen Zugriffsmodifier bestimmen die Sichtbarkeit für andere Klassen. Da eine öffentliche Schnittstelle angeboten wird, können auch in anderen Packages konkrete Realisierungen des Interface `Calculation` erfolgen:

```
public final class CalculationInnerClassWithIFExample
{
    // Basisinterface
    public interface Calculation
    {
        public int calc(final int value1, final int value2);
        public String getName();
    }

    // Nur zur Demonstration von Vererbung
    private static abstract class AbstractCalculation implements Calculation
    {
        public String getName()
        {
            return getClass().getSimpleName();
        }
    }

    // Für alle Klassen zugreifbar
    public static final class Plus extends AbstractCalculation
    {
        public int calc(final int value1, final int value2)
        {
            return value1 + value2;
        }
    }

    // Nur innerhalb des Packages und abgeleiteter Klassen zugreifbar
    protected static final class Minus extends AbstractCalculation
    {
        public int calc(final int value1, final int value2)
        {
            return value1 - value2;
        }
    }

    // Extrem selten sinnvoll, nur innerhalb dieser Klasse zugreifbar
    private static class Modulo extends AbstractCalculation
    {
        public int calc(final int value1, final int value2)
        {
            return value1 % value2;
        }
    }

    public static void main(final String[] args)
```

```
    {
        final Calculation[] calculations = { new Plus(), new Minus(),
                                             new Modulo() };

        for (final Calculation calculation : calculations)
        {
            System.out.println("7 " + calculation.getName() + " 2 = " +
                               calculation.calc(7, 2));
        }
    }
}
```

Listing 4.35 *Ausführbar als* **'CALCULATIONINNERCLASSWITHINTERFACEXAMPLE'**

Die `main()`-Methode erzeugt ein `Calculation[]`, das mit den konkreten Realisierungen `Plus`, `Minus` und `Modulo` initialisiert wird. Anschließend wird für jede Berechnungsklasse die `calc()`-Methode mit den Werten 7 und 2 aufgerufen. Das Programm CALCULATIONINNERCLASSWITHINTERFACEEXAMPLE gibt Folgendes aus:

```
7 Plus 2 = 9
7 Minus 2 = 5
7 Modulo 2 = 1
```

4.5.4 Lokal definierte Klassen und Interfaces

Man kann zusätzlich zu einer `public` definierten Klasse einer Java-Datei lokal innerhalb dieser Datei weitere Klassen und Interfaces definieren. Dabei werden die Definitionen zwar innerhalb der Datei der öffentlichen Klasse angegeben, aber im Unterschied zu inneren Klassen nicht innerhalb der Klassendefinition. Folgendes Listing verdeutlicht dies:

```
// Lokal definiertes Interface
interface LocalInterface
{
    int calc(int value);
}

// Lokal definierte Klasse
class LocalClass
{
    // …
}

public class LocalClassAndInterfaceExample
{
    // …
}
```

Derart definierte Klassen und Interfaces besitzen die Sichtbarkeit Package-private. Sie können Funktionalität im Package »verstecken«, da sie von außerhalb weder sichtbar noch zugreifbar sind. Innerhalb dieser Klassen definierte Methoden können eine beliebige Sichtbarkeit besitzen und diese demnach auch beliebig einschränken. Für Interfaces gilt dies nicht. Laut JLS sind in Interfaces immer alle Methoden `public`. Somit sind leider auch in Package-privaten Interfaces definierte Methoden

öffentlich. Wünschenswert für eine bessere Lesbarkeit und zur sicheren Vermeidung von externen Aufrufen wäre es vielmehr, wenn man in einem solchen Interface Methoden nicht `public`, sondern Package-private definieren könnte. Damit würde man klarer ausdrücken können, dass diese Methoden lediglich aus Klassen des Packages aufgerufen werden sollen. Das ist aber eher ein theoretisches Problem. Externe Klassen haben keinen Zugriff auf das Package-private Interface und somit sind selbst die öffentlichen Methoden verborgen, solange keine Referenzen auf die das Package-private Interface implementierende Klasse selbst nach außen gegeben werden.

Hinweis: Abstrakte Klassen zur Definition von Schnittstellen

Als Alternative kann man zur Vorgabe einer Schnittstelle statt eines Interface auch eine abstrakte Klasse nutzen, die dann wie auch ihre Methoden Package-private sein kann.

Beispiel

In folgendem Beispiel setzen wir lokal definierte Klassen und Interfaces ein, um die enge Kopplung in eine lose zu wandeln und eine bessere Strukturierung zu erzielen. Betrachten wir dazu ein Package `msghandling`, das verschiedene Klassen zum Transfer von Daten anbietet. Abbildung 4-2 zeigt die relevanten Klassen und deren Assoziation.

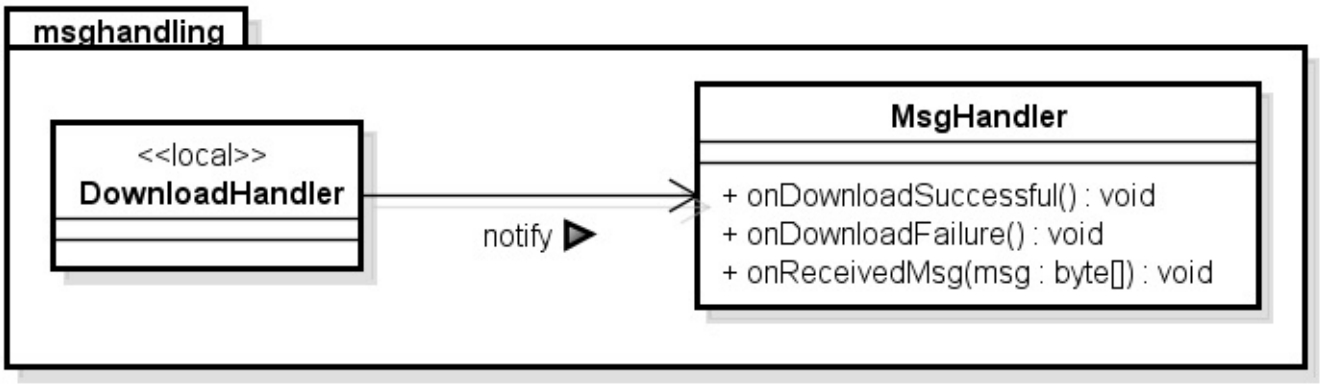


Abbildung 4-2 *Klassendiagramm des Packages* `msghandling`

Die öffentliche Klasse `MsgHandler` soll Daten verarbeiten, die von einer lokalen Hilfsklasse `DownloadHandler` über einen Datentransfer eingelesen werden. Je nach Verlauf eines Datentransfers werden verschiedene sogenannte ***Callback-Methoden*** der Klasse `MsgHandler` aufgerufen, hier entweder die Kombination von `onDownloadSuccessful()` gefolgt von `onReceivedMsg(byte[])` oder einzeln `onDownloadFailure()`. Abbildung 4-3 zeigt ein zugehöriges Sequenzdiagramm.

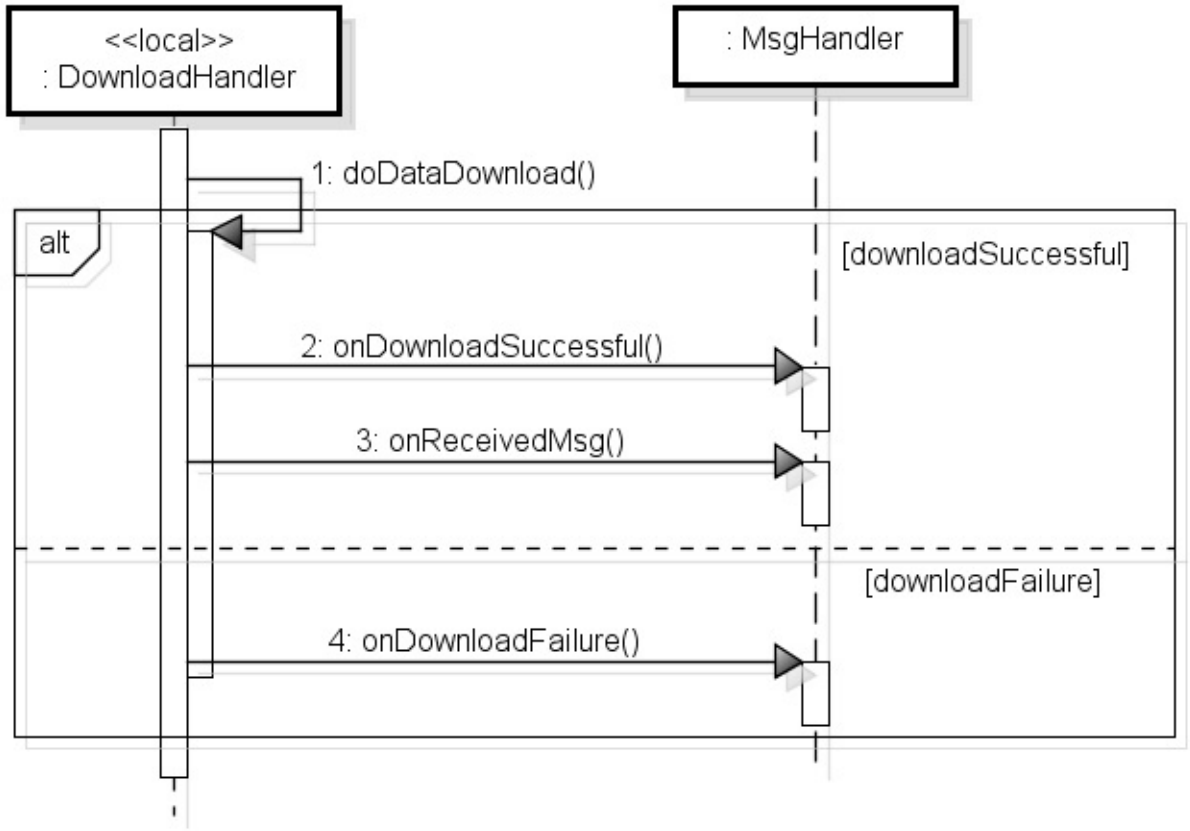


Abbildung 4-3 *Ablauf einer Datenübertragung als Sequenzdiagramm*

Um die genannten Methoden aufrufen zu können, hält die Hilfsklasse `DownloadHandler` eine Referenz auf die Klasse `MsgHandler`. Diese Art der Realisierung koppelt die beiden Klassen eng miteinander und ermöglicht der Klasse `DownloadHandler` den Zugriff auf alle öffentlichen Methoden der äußeren Klasse `MsgHandler`. Für die zu realisierende Funktionalität ist eine solche Verbindung unnötig, denn die Klasse `DownloadHandler` soll lediglich die drei Callback-Methoden der Klasse `MsgHandler` aufrufen (können), um über den Verlauf von Datenübertragungen zu informieren.

Zur Entkopplung der beiden Klassen wird ein Callback-Interface `IMsgHandlerCallBack` lokal innerhalb der Klasse `MsgHandler` wie folgt definiert:

```
/* private */ interface IMsgHandlerCallBack
{
    public void onDownloadSuccessful();
    public void onDownloadFailure();
    public void onReceivedMsg(final byte[] msg);
}
```

Die öffentliche Klasse `MsgHandler` implementiert dieses Interface. Der eigentliche Download wird weiterhin in der Hilfsklasse `DownloadHandler` gekapselt. Um eine möglichst klare Struktur und lose Kopplung zu erzielen, verwendet diese Hilfsklasse nun aber lediglich das Interface `IMsgHandlerCallBack`, um mit der Klasse `MsgHandler` zu kommunizieren. Damit ergibt sich folgende Realisierung:

```
/**
* Kapselt den FTP-Transfer in einer eigenen Klasse, überlässt die
* Auswertung aber der eigentlichen Klasse
```

```
*/
/* private */ final class DownloadHandler
{
    private final IMsgHandlerCallBack msgHandlerCallBack;

    private DownloadHandler(final IMsgHandlerCallBack msgHandlerCallBack)
    {
        this.msgHandlerCallBack = msgHandlerCallBack;
    }

    public void doDataDownload()
    {
        final byte[] msg = new byte[] { 0, 1, 2, 3, 4, 5 /* Beispieldaten */};

        if (getDownloadState() == SUCCESS)
        {
            msgHandlerCallBack.onDownloadSuccessful();
            msgHandlerCallBack.onReceivedMsg(msg);
        }
        else
        {
            msgHandlerCallBack.onDownloadFailure();
        }
    }

    // …
}
```

Die durch diese Implementierung erzielte lose Kopplung an die öffentliche Verarbeitungsklasse `MsgHandler` ermöglicht es, die lokale Klasse `DownloadHandler` bei Bedarf in eine eigene Datei auszulagern, etwa wenn der Umfang dieser Hilfsklasse wächst. Es ergibt sich das in Abbildung 4-4 dargestellte Klassendiagramm. Es ist offensichtlich, dass es sogar problemlos möglich wäre, die Klasse `DownloadHandler` in ein anderes Package zu verschieben.

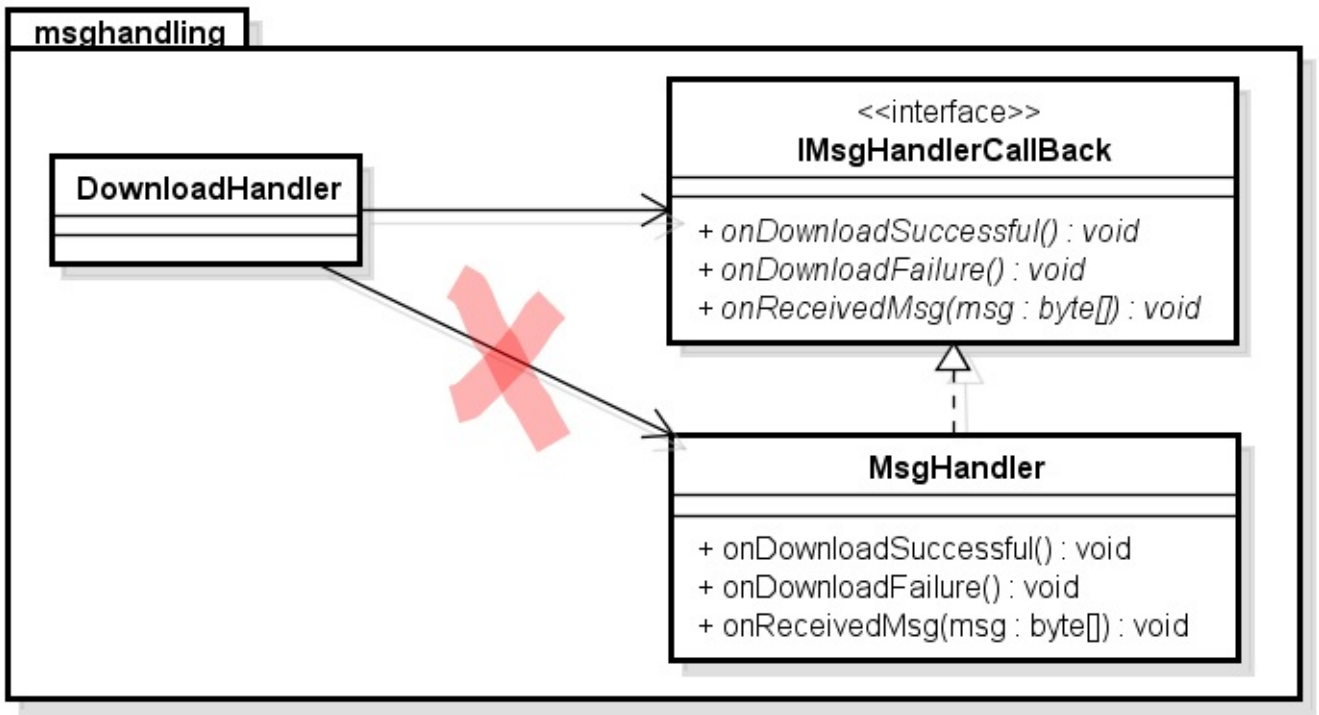


Abbildung 4-4 *Erweiterungen im Package* `msghandling`

4.6 Ein- und Ausgabe (I/O)

Die meisten Programme müssen mit anderen Applikationen interagieren oder Benutzereingaben entgegennehmen, um Berechnungen auszuführen. Java bietet zur Ein- und Ausgabe in den Packages `java.io` und `java.nio` einen objektorientierten Zugang.

Bitte beachten Sie, dass es bei der Kommunikation und Ein- und Ausgabe immer auch zu Fehlern oder zumindest Problemen kommen kann, etwa dass eine Datei nicht vorhanden ist oder nicht in sie geschrieben werden kann. Aus Gründen der Übersichtlichkeit und weil wir das Thema Fehlerbehandlung erst später in Abschnitt 4.7 vertiefen, wird in den nachfolgenden Beispielen zur API-Demonstration der Ein- und Ausgabe auf eine ansonsten notwendige Fehlerbehandlung weitestgehend verzichtet.

4.6.1 Dateibehandlung und die Klasse `File`

Eine Instanz der Klasse `java.io.File` kann entweder eine Datei oder ein Verzeichnis im Dateisystem repräsentieren. Dazu speichert ein `File`-Objekt den Namen und den Pfad lediglich textuell, aber nicht das tatsächliche Dokument oder dessen Inhalt. Es existieren daher in der Klasse `File` keine Methoden, um Daten in eine Datei zu schreiben oder daraus zu lesen. Dazu lernen wir im Verlauf dieses Unterkapitels in Abschnitt 4.6.2 die Ein- und Ausgabestreams kennen.

Pfad-Aktionen

Ein `File`-Objekt konstruiert man aus der Angabe eines Pfadnamens. Dieser kann aus dem Datei- oder Verzeichnisnamen und optional der Angabe des übergeordneten Verzeichnisses bestehen:[13]

- `new File(String pathname)`
- `new File(String parentDirName, String filename)`
- `new File(File parentDir, String filename)`

In der Pfadangabe werden sowohl einfache Slashes '/' als auch Backslashes '\' als Pfadtrenner akzeptiert. Auch die Metaverzeichnisse '.' (aktuelles Verzeichnis) und '..' (übergeordnetes Verzeichnis) können dort enthalten sein. Wie auch bei regulären Ausdrücken ist für das Backslash-Zeichen allerdings ein Escapen notwendig. Das folgende Beispiel verdeutlicht verschiedene Varianten der Pfadangabe für Verzeichnisse:

```
public static void main(final String[] args)
{
    printFileInfo(new File("C:\toBeChecked"));
    printFileInfo(new File("C:\toBeChecked\./../toBeChecked/"));
```

```
}

private static void printFileInfo(final File file)
{
    System.out.println("Name='" + file.getName() + "' / " +
                       "Path='" + file.getAbsolutePath() + "'");
}
```

Listing 4.36 *Ausführbar als* **'PATHEXAMPLE'**

Nachdem ein `File`-Objekt erzeugt wurde, kann man verschiedene Aktionen ausführen. Die Methode `getName()` liefert den Dateinamen ohne Pfadinformationen. Diese erhält man über `getAbsolutePath()`. Dabei werden auch alle enthaltenen Metainformationen (beispielsweise '. ' und '. . ') ausgegeben. Allerdings werden Slashes '/' und Backslashes '\' in das jeweilige Format des Betriebssystems vereinheitlicht und es kommt (unter Windows) zu folgenden Ausgaben:

```
Name='toBeChecked' / Path='C:\toBeChecked'
Name='toBeChecked' / Path='C:\toBeChecked...\toBeChecked'
```

Eine Auflösung einer beliebigen Pfadangabe in Form eines Strings in einen realen Pfad im Dateisystem erfolgt über die Methode `getCanonicalPath()`. Der Pfad "`C:\ toBeChecked. /../toBeChecked/`" wird beispielsweise in folgendes reale Verzeichnis übersetzt: "`C:\toBeChecked`". Da die Methode `getCanonicalPath()` zum Auflösen der Pfadinformationen nicht nur Strings verkettet, sondern zum Teil auf dem Dateisystem operiert, können `java.io.IOExceptions` ausgelöst werden.

File-Aktionen

Nachdem wir zuvor einige wichtige Details zur Verarbeitung von Pfaden kennengelernt haben, betrachten wir nun einige elementare Methoden der Klasse `File`:

- `isFile()` und `isDirectory()` – Diese Methoden prüfen, ob ein `File`-Objekt eine Datei bzw. ein Verzeichnis repräsentiert.
- `exists()` – Prüft, ob ein `File`-Objekt tatsächlich im Dateisystem existiert. Wir nutzen dies, um folgende Methode `directoryExists(File)` zu schreiben, die die bereits genannten Methoden zu einem neuen Baustein kombiniert:

  ```
  public static boolean directoryExists(final File directoryToCheck)
  {
      if (directoryToCheck == null)
          return false;

      return directoryToCheck.exists() && directoryToCheck.isDirectory();
  }
  ```

- `createNewFile()` – Erzeugt eine neue Datei, die durch das `File`-Objekt repräsentiert wird.

- `delete()` – Löscht die Datei, die durch das `File`-Objekt repräsentiert wird. Gleiches gilt für Verzeichnisse, allerdings nur, wenn diese leer sind.
- `mkdir()` und `mkdirs()` – Ein Aufruf der Methode `mkdir()` erzeugt das durch das `File`-Objekt angegebene Verzeichnis. Mit `mkdirs()` werden bei Bedarf auch alle dazu benötigten übergeordneten Verzeichnisse angelegt.
- `list()` – Ermittelt den Inhalt eines Verzeichnisses als `String[]`. Die `list()`-Methode liefert in Fehlersituationen den Wert `null` statt ein `String`-Array der Länge 0. Details dazu beschreibt der folgende Hinweis »Rückgabewerte der `list()`- bzw. `listFiles()`-Methoden«. Aufrufer müssen daher Spezialfälle unterscheiden, was einerseits zusätzliche Schreibarbeit bedeutet und andererseits potenziell fehlerträchtig ist oder vergessen wird. Zur leichteren Handhabung realisieren wir folgende Methode `getContents(File)`, die das Problem behebt:

```
public static String[] getContents(final File directoryToCheck)
{
    if (directoryExists(directoryToCheck))
    {
        final String[] contents = directoryToCheck.list();
        if (contents != null)
        {
            return contents;
        }
    }

    return new String[0];
}
```

- `listFiles()` – Ermittelt, ähnlich zu `list()`, den Inhalt eines Verzeichnisses. Die Rückgabe erfolgt hier als `File[]`.

Die in der Aufzählung implementierten einfachen Hilfsmethoden lassen bereits die Idee erkennen, orthogonal zu programmieren, d. h. viele kleine funktional unabhängige Bausteine in einer Utility-Klasse (hier `FileUtils`), zu erstellen und diese zu neuen Bausteinen zu kombinieren.

Hinweis: Rückgabewerte der `list()`- bzw. `listFiles()`-Methoden

Die beiden Methoden `list()` und `listFiles()` provozieren `NullPointerExceptions` bei Aufrufern, da sie `null` zurückgeben, wenn kein Verzeichnisinhalt ermittelt werden kann. Dies ist immer dann der Fall, wenn der Pfad im `File`-Objekt kein Verzeichnis darstellt oder aber ein I/O-Fehler auftritt. Besser wäre für den ersten Fall die Rückgabe eines Arrays der Länge 0 gewesen. Ein I/O-Fehler sollte meiner Meinung nach durch eine `IOException` und nicht durch einen `null`-Wert kommuniziert werden. Genauer diskutiere ich die Problematik in Abschnitt 16.3.5 als BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL. Für die hier genannten Methoden müssen alle Anwender jeweils Spezialbehandlungen einbauen. Um dies zu vermeiden, wurde die Hilfsmethode

getContents() geschrieben, die durch Aufruf der null-sicheren eigenen Methode directoryExists(File) ebenfalls null-sicher ist und bei solcher Eingabe ein leeres String[] liefert.

Die Interfaces FileFilter und FilenameFilter

Realisierungen der Interfaces java.io.FileFilter und java.io.FilenameFilter erlauben, die Ergebnismenge der gerade vorgestellten list()- bzw. listFiles()-Methoden einzuschränken, etwa um in einem Verzeichnis alle pdf-Dateien zu ermitteln. In beiden Interfaces ist eine Methode accept() definiert. Diese ist so zu implementieren, dass sie für gewünschte Objekte den Wert true zurückgibt.

Im Falle des FileFilter erhält die Methode accept() ein File-Objekt als Parameter. Dadurch besteht innerhalb der Implementierung der Methode Zugriff auf den Dateinamen und verschiedene weitere Attribute, insbesondere auch den Pfad.

```
public interface FileFilter
{
    boolean accept(File pathname);
}
```

Das Interface FilenameFilter abstrahiert weiter. Statt eines File-Objekts dient nur noch der Dateiname sowie das übergeordnete Verzeichnis als Eingabe:

```
public interface FilenameFilter
{
    boolean accept(File dir, String name);
}
```

Diese Angabe ist hilfreich, weil man nur mit dem Dateinamen keinen Zugriff auf die Attribute der Datei besitzt. Benötigt man diese und weitere Informationen, so kann man den zusätzlichen Parameter, d. h. das übergeordnete Verzeichnis, auswerten.

Filterung am Beispiel

Der folgende Sourcecode-Ausschnitt zeigt, wie man nach .xml- bzw. .pdf-Dateien filtern kann. Ersteres wird in diesem Beispiel durch einen FilenameFilter realisiert. Letzteres nutzt einen FileFilter als Basis. Beide Klassen implementieren wir als statische innere Klassen einer Utility-Klasse FileFilterUtils und wandeln die jeweiligen Dateinamen mit toLowerCase() in Kleinbuchstaben um, wodurch der Vergleich unabhängig von der Schreibweise bleibt.

```
public static class XmlFilenameFilter implements FilenameFilter
{
    @Override
    public boolean accept(final File dir, final String fileName)
    {
        return fileName.toLowerCase().endsWith(".xml");
    }
}
```

```
public static class PdfFileFilter implements FileFilter
{
    @Override
    public boolean accept(final File pathname)
    {
        return pathname.getName().toLowerCase().endsWith(".pdf");
    }
}
```

So definierte Filter kann man wie folgt nutzen:

```
final String[] xmlContents = dir.list(new XmlFilenameFilter());
final File[] pdfContents = dir.listFiles(new PdfFileFilter());
```

Parametrisierte Filter

Wollte man einen etwas komplexeren Filter realisieren, der exakt auf ein als Parameter übergebenes Präfix und Postfix prüft, so könnte man Folgendes implementieren:

```
public class PreAndPostFixFilenameFilter implements FilenameFilter
{
   private final String prefix;
   private final String postfix;

   public PreAndPostFixFilenameFilter(final String prefix, final String
postfix)
   {
      this.prefix = prefix;
      this.postfix = postfix;
   }

   @Override
   public boolean accept(final File dir, final String name)
   {
      return name.startsWith(prefix) && name.endsWith(postfix);
   }
}
```

Vereinfachungen mit JDK 8

Mit dem neuen Sprachfeature der Lambda-Ausdrücke (vgl. Kapitel 11) lässt sich die Implementierung der beiden Interfaces deutlich kürzer wie folgt schreiben:

```
final FileFilter pdfFilter = (final File file) ->
{
    return file.getName().toLowerCase().endsWith(".pdf");
}

final FilenameFilter xmlFilter = (final File dir, final String filename) ->
{
    return filename.toLowerCase().endsWith(".xml");
}
```

Auch der speziellere Prä- und Postfix-Filter lässt sich recht einfach implementieren. Wir müssen hier einen kleinen Trick nutzen: Weil wir Parameter an den Filter

übergeben wollen, erzeugen wir den Filter mithilfe einer Methode, die einen Lambda zurückliefert:

```
public static FilenameFilter createPreAndPostFixFilter(final String prefix,
                                                        final String postfix)
{
    return (dir, fileame) -> { return filename.startsWith(prefix) &&
                                      filename.endsWith(postfix); };
}
```

Diese Beispiele zeigen, wie hilfreich die mit JDK 8 eingeführten Lambdas sind, um kurze und prägnante Realisierungen zu erstellen.

Beispiel: Verzeichnisüberwachung

Wir haben nun alle benötigten kleinen Bausteine des `File`-APIs kennengelernt, um eine einfache Variante einer Verzeichnisüberwachung zu implementieren. Häufig spricht man von einem Hot Directory, wenn Änderungen in einem Verzeichnis überwacht und gemeldet werden. Diese Funktionalität realisiert die folgende Klasse `DirectoryObserver`, die wir im Verlauf dieses Buchs weiter ausbauen werden.

Die Klasse `DirectoryObserver` besitzt zwei Attribute: zum einen das zu überwachende Verzeichnis als `File`-Objekt und zum anderen ein Zeitintervall. Dieses Intervall legt fest, in welchen Zeitabständen das zu überwachende Verzeichnis auf Veränderungen geprüft werden soll.

Die eigentliche Überwachung wird von der Methode `checkDirectory()` vorgenommen, die zyklisch durch Aufruf der bereits vorgestellten Methode `directoryExists(File)` prüft, ob es das gewünschte Verzeichnis gibt, und im Fehlerfall die Callback-Methode `onDirectoryNotExisting()` ausführt. Im Normalfall wird über die Methode `checkContentsChanged()` eine Änderung am Verzeichnisinhalt überprüft. Hier nutzen wir die zuvor vorgestellte Methode `getContents(File)`, um den Verzeichnisinhalt zu ermitteln. In diesem einführenden Beispiel bestimmen wir lediglich die Anzahl der vorhandenen Dateien und schauen, ob sich diese gegenüber dem vorherigen Durchlauf verändert hat, um gegebenenfalls eine Änderungsnachricht mithilfe der Callback-Methode `onContentsChanged(int, int)` auszuführen:

```
public class DirectoryObserver
{
   protected static final int DEFAULT_CHECK_INTERVAL_IN_SEC = 5;

   private final int          checkIntervalInSec;
   private final File         directoryToCheck;

   public DirectoryObserver(final String nameOfDirToCheck)
   {
      this(nameOfDirToCheck, DEFAULT_CHECK_INTERVAL_IN_SEC);
   }

   public DirectoryObserver(final String nameOfDirToCheck,
                            final int checkIntervalInSec)
```

```
    {
        this.checkIntervalInSec = checkIntervalInSec;
        this.directoryToCheck = new File(nameOfDirToCheck);
    }

    public void checkDirectory()
    {
        System.out.println("starting directory check");

        int numOfFiles = FileUtils.getContents(directoryToCheck).length;

        while (!Thread.currentThread().isInterrupted())
        {
            System.out.println("checkDirectory… '" + directoryToCheck + "'");
            if (FileUtils.directoryExists(directoryToCheck))
            {
                numOfFiles = checkForContentsChanged(numOfFiles);
            }
            else
            {
                onDirectoryNotExisting();
                numOfFiles = 0;
            }
            SleepUtils.safeSleep(TimeUnit.SECONDS, checkIntervalInSec);
            System.out.println("…checkDirectory");
        }
    }

    protected int checkForContentsChanged(final int numOfExpectedFiles)
    {
        final String[] currentContents = FileUtils.getContents(directoryToCheck);
        if (currentContents.length != numOfExpectedFiles)
        {
          onContentsChanged(currentContents.length, numOfExpectedFiles);
        }
        return currentContents.length;
    }

    protected void onContentsChanged(final int newFileCount,
                                     final int oldFileCount)
    {
        System.out.println("new FileCount=" + newFileCount + " / " +
                           "old FileCount=" + oldFileCount);
    }

    protected void onDirectoryNotExisting()
    {
        System.out.println("missing directory='" + directoryToCheck + "'");
    }
    public static void main(final String[] args) throws IOException
    {
        // Zugriff auf das systemspezifische tmp-Directory
        final String tmpDir = System.getProperty("java.io.tmpdir");
        new DirectoryObserver(tmpDir).checkDirectory();
    }
    //…
}
```

Listing 4.37 *Ausführbar als* **'DIRECTORYOBSERVER'**

Erwähnenswert ist, dass Änderungen nicht direkt verarbeitet, sondern über CallbackMethoden propagiert werden. Dadurch ist es beispielsweise möglich, andere

Klassen über Änderungen zu informieren oder beliebige Aktionen durchzuführen, ohne Aufwand für Modifikation und ohne Abhängigkeiten in den Algorithmus der Überprüfung einzuführen. In diesem simplen Beispiel werden diese Methoden lediglich dazu genutzt, eine Meldung auf der Konsole auszugeben.

Zum Start der Überwachung wird ein `DirectoryObserver`-Objekt mit der Angabe des zu überwachenden Verzeichnisses erzeugt und anschließend dessen Methode `checkDirectory()` ausgeführt. Im Beispiel verwenden wir dazu das vom Betriebssystem genutzte temporäre Verzeichnis, auf das wir über die System-Property `java.io.tmpdir` Zugriff haben.

4.6.2 Ein- und Ausgabestreams im Überblick

Wenden wir uns nach der Vorstellung der Dateiverarbeitung nun der Modifikation von Dateiinhalten zu. Man unterscheidet zwischen byteorientierter und zeichenbasierter Ein- und Ausgabe. Erstere stellen die Klassen `java.io.InputStream` und `java.io.OutputStream` bereit. Diese beiden bilden die Schnittstelle zu Datenquellen und -senken, deren Daten als Bytefolgen repräsentiert sind. Die zeichenbasierte Einbzw. Ausgabe erfolgt in Form von `chars` mithilfe der Basisklassen `java.io.Reader` bzw. `java.io.Writer`. Diese Klassen können problemlos Umlaute und Sonderzeichen verarbeiten, sofern die gewählte Zeichencodierung (Encoding) stimmt – Details erläutere ich später in Abschnitt 4.6.3. Konzentrieren wir uns hier zunächst auf Streams.

Definition 4.2 *Unter einem **Stream** (oder Strom) versteht man eine geordnete Abfolge oder Sequenz von Bytes oder Zeichen beliebiger Länge. Der Zugriff erfolgt sequenziell. **Eingabestreams** importieren Daten in ein Java-Programm. **Ausgabestreams** exportieren Daten aus einem Java-Programm.*

Für `Input-` und `OutputStreams` und `Reader` und `Writer` existieren zwei analoge Vererbungshierarchien. Die Konvertierung eines `InputStreams` in einen `Reader` wird durch die Klasse `java.io.InputStreamReader` ermöglicht. Das entsprechende Gegenstück für die Ausgabe stellt die Klasse `java.io.OutputStreamWriter` dar, die einen `OutputStream` in einen `Writer` umwandelt.

Abbildung 4-5 zeigt wichtige Subklassen der abstrakten Basisklasse `InputStream`, beispielsweise um die Eingabe aus einer Datei zu lesen (`java.io.FileInputStream`). Spezielle Klassen erlauben, dynamisch weitere Funktionalität, etwa eine Pufferung (`java.io.BufferedInputStream`), hinzuzufügen. Zwei besondere Streams zur Interaktion mit der Konsole werden durch die statischen Attribute `System.in` und `System.out` als gepufferte Streams bereitgestellt. ***Aufgrund der Pufferung sind dort manche Ausgaben nicht direkt sichtbar.***

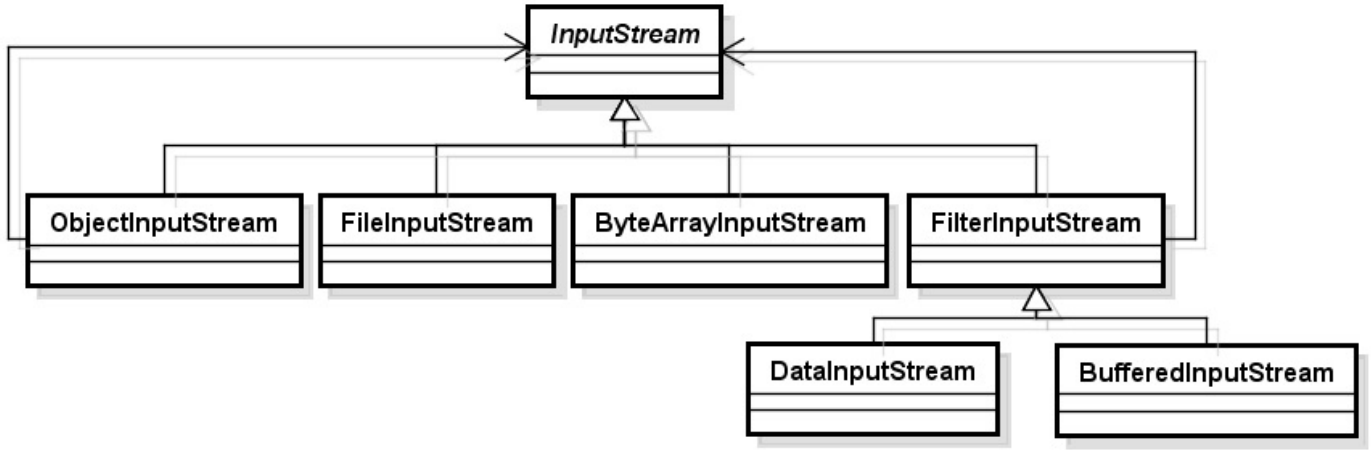


Abbildung 4-5 *Ableitungen der Klasse* `InputStream`

Zur Verarbeitung des Inhalts einer Datei kann man die Klassen `FileInputStream` und `java.io.FileOutputStream` einsetzen. Um die vom Betriebssystem bereitgestellten Ressourcen zu allozieren bzw. wieder freizugeben, bedarf es der Operationen Öffnen und Schließen. Das Öffnen erfolgt meistens implizit (z. B. bei der Konstruktion eines Streams). Für das Schließen ist die Applikation dagegen selbst verantwortlich. Das sollte immer in einem `finally`-Block erfolgen oder durch Einsatz des mit JDK 7 eingeführten Automatic Resource Management (ARM) (vgl. Abschnitt 4.7).

Die Klassen `InputStream` und `Reader`

Alle `InputStreams` bieten Methoden zum Lesen von Daten und zum Prüfen, ob Daten zum Einlesen vorliegen.

Die Klasse `InputStream`

In der abstrakten Basisklasse `InputStream` sind bereits einige überladene `read()`-Methoden implementiert. Viele rufen die abstrakte Methode `read()` auf, die von konkreten Realisierungen der Java-I/O-Bibliotheken abgestimmt auf das jeweilige Eingabemedium (Datei, Socket usw.) implementiert wird. Folgende Aufzählung stellt die wichtigsten Methoden und deren Intention vor:

- `abstract int read()` – Liest das nächste Byte aus dem Eingabestrom. Dieses wird jedoch als `int` codiert. Ein Rückgabewert von -1 signalisiert das Ende des Streams. Ansonsten liegen die eingelesenen Werte im Bereich von 0 – 255 und müssen zur Weiterverarbeitung gegebenenfalls auf ein `byte` gecastet werden. Werte über 127 werden dadurch in negative Zahlen umgewandelt.
- `int read(byte[] buffer)` – Liest die nächsten Bytes aus dem Eingabestrom ein, maximal aber so viele wie das übergebene `byte[]` groß ist. Sind im Eingabestrom keine Daten vorhanden, so signalisiert ein Rückgabewert von -1 das Ende des Streams. Ansonsten werden die Werte im übergebenen `byte`-Array `buffer` gespeichert, und es wird die Anzahl der tatsächlich

gelesenen Bytes zurückgeliefert. Sollen mehr Daten gelesen werden, als derzeit im Eingabestrom vorliegen, so blockiert der Aufruf die weitere Ausführung.[14]

- `int read(byte[] buffer, int pos, int len)` – Sofern verfügbar, werden die nächsten `len` Bytes aus dem Eingabestrom eingelesen und ab Position `pos` im übergebenen `byte`-Array `buffer` gespeichert.
- `int available()` – Liefert die Anzahl der im Eingabestrom zur Verfügung stehenden Bytes. Diese können anschließend über eine der `read()`-Methoden ausgelesen werden, ohne eine Blockierung auszulösen.
- `void close()` – Der Eingabestrom wird geschlossen, und es werden alle dadurch belegten Betriebssystemressourcen wieder freigegeben.

Die Klasse `Reader`

Die Klasse `Reader` besitzt sehr ähnliche Methoden wie die Klasse `InputStream`. Allerdings wird beim `Reader` nicht auf Bytes, sondern auf Zeichen gearbeitet. Statt wie beim `InputStream` das nächste einzelne Byte zu lesen, stützen sich alle Einlesemethoden auf eine abstrakte Methode `read(char[], int, int)` ab, die einen Teilbereich einlesen kann. Nachfolgend werden einige Methoden der Klasse `Reader` kurz aufgelistet:

- `abstract int read(char[] buffer, int pos, int len)` – Sofern verfügbar, werden die nächsten `len` Zeichen aus dem Eingabestrom eingelesen und ab Position `pos` im übergebenen `char`-Array `buffer` gespeichert.
- `int read()` – Liest das nächste Zeichen aus dem Eingabestrom als `int` und liefert es in der Unicode-Codierung zurück, d. h., es werden gültige Werte im Bereich von 0 – 65535 geliefert. Ein Wert von -1 signalisiert das Ende des Streams.
- `int read(char[] buffer)` – Liest die nächsten Zeichen aus dem Eingabestrom, maximal aber `buffer.length` Zeichen. Die Werte werden im übergebenen `char`-Array `buffer` gespeichert.
- `boolean ready()` – Prüft, ob im Eingabestrom Daten zur Verfügung stehen. Im Unterschied zur Klasse `InputStream` werden keine Angaben über die Anzahl der verfügbaren Daten gemacht.

Die Klassen `OutputStream` und `Writer`

Alle Ausgabestreams bieten Methoden zum Schreiben von Daten in einen Stream.

Die Klasse `OutputStream`

Ähnlich zum `InputStream` existieren hier einige überladene `write()`-Methoden, die sich auf eine abstrakte `write()`-Methode stützen. Diese wird von den konkreten

Klassen der Java-I/O-Bibliotheken, spezifisch für das jeweilige Ausgabemedium, implementiert. Folgende Aufzählung stellt die wichtigsten Methoden und ihre Aufgaben vor:

- `abstract void write (int byte)` – Schreibt das übergebene Byte `byte` in den Ausgabestrom. Die höherwertigen Bytes des übergebenen `int`-Werts werden dabei ignoriert.
- `void write(byte[] buffer)` – Schreibt alle Bytes aus dem übergebenen Byte-Array `buffer` in den Ausgabestrom.
- `void flush()` – Entleert den Ausgabestrom und erzwingt ein physikalisches Schreiben möglicherweise zwischengespeicherter Daten.
- `void close()` – Der Ausgabestrom wird geschlossen und es werden alle dadurch belegten Betriebssystemressourcen wieder freigegeben.

Die Klasse `Writer`

Die Klasse `Writer` besitzt sehr ähnliche Methoden wie die Klasse `OutputStream`, die allerdings nicht auf Bytes, sondern auf Zeichen arbeiten. Neben den Methoden `flush()` und `close()` finden sich einige überladene `write()`-Methoden, die sowohl `char`-Arrays oder Bereiche als auch Strings und Teilstrings in den Stream schreiben können.

4.6.3 Zeichencodierungen bei der Ein- und Ausgabe

Für textuelle Daten gibt es insbesondere bei deren Verarbeitung mit byteorientierten Streams einige Dinge zu bedenken: Das Ganze setzt eine Umwandlung in und aus einer Folge von Bytes voraus. Beim Austausch bytecodierter Daten ist allerdings nicht immer sichergestellt, dass die Gegenseite die gleiche Interpretation der Folge von Bytes bei der Rücktransformation in einen String vornimmt. Betrachten wir dies genauer.

Die Klasse `String` speichert Texte als ein Array von `char`, das eine Unicode-Codierung verwendet. Um eine Abbildung zwischen einem Stringobjekt und einem `byte[]` zu ermöglichen, existieren spezielle ***Charsets*** oder Zeichencodierungen, die Zeichen auf Zahlenwerte abbilden. Mithilfe verschiedener Charsets kann man die interne Unicode-Repräsentation in einen beliebigen anderen Zeichensatz umwandeln: Es existieren sogenannte ***Encoder*** und ***Decoder***, die dazu dienen, eine Sequenz von Bytes in einer beliebigen Codierung mit einem `CharsetDecoder` in Unicode-Zeichen umzuwandeln bzw. mit einem `CharsetEncoder` eine Transformation von Unicode in eine Folge von Bytes zu vollziehen.

Die Umwandlung eines Strings in ein `byte[]` kann durch Aufruf der Methoden `getBytes()`, `getBytes(Charset)` und `getBytes(String)` erfolgen, die einen String in ein Byte-Array gemäß dem Defaultcharset bzw. dem angegebenen

Charset konvertieren. Dabei kann das Charset entweder als `Charset`-Objekt oder per Name übergeben werden. Leider sind nicht alle der erlaubten Namen als Konstanten definiert und zudem manchmal recht kryptisch.[15] So nutzt die DOS-Konsole unter Windows den Zeichensatz »Cp850« (Cp steht für Codepage). Windows selbst verwendet die Codierung »Cp1252« (Windows-1252 bzw. Latin 1), die eine Anpassung der Codierung »ISO-8859-1« ist. Weitere bekannte Codierungen sind »UTF-8« und »UTF-16«.

Einfluss des gewählten `Charset`-Objekts beim Aufruf von `getBytes()`

Während die »normalen« Buchstaben in den meisten gebräuchlichen Charsets den gleichen Zahlenwert (ASCII-kompatibel) besitzen, lassen die sich unterschiedlichen Umsetzungen durch Umlaute klar erkennen. Anhand des Strings `"ÄÖÜ To"` zeige ich den Einfluss verschiedener Charsets. Umlaute werden einerseits durch verschiedene Zahlenwerte dargestellt und andererseits werden sie in UTF-8 mit zwei Bytes statt in einem Byte codiert. Tabelle 4-5 zeigt die Werte für die Umwandlungen des Strings `"ÄÖÜ To"`.

Tabelle 4-5 *Einfluss verschiedener Encodings*

Charset-Encoding	Ä	Ö	Ü	␣	T	o
Default (für Windows)	-60	-42	-36	32	84	111
Cp850	-114	-103	-102	32	84	111
ISO-8859-1	-60	-42	-36	32	84	111
UTF-8	-61, -124	-61, -106	-61, -100	32	84	111

Zum Nachvollziehen starten Sie das folgende Programm StringEncodings – möglicherweise gibt es Abweichungen in den Werten, wenn man Macintosh und Linux nutzt.

```
public static void main(final String[] args) throws
UnsupportedEncodingException
{
    final byte[] def = "ÄÖÜœTo".getBytes();
    final byte[] dos = "ÄÖÜ To".getBytes("Cp850");
    final byte[] iso = "ÄÖÜ To".getBytes("ISO-8859-1");
    final byte[] utf8 = "ÄÖÜ To".getBytes("UTF-8");

    printBytes(def);  // "Default:" [-60, -42, -36, 32, 84, 111]
    printBytes(dos);  // "Cp850": [-114, -103, -102, 32, 84, 111]
    printBytes(iso);  // "ISO-8859-1": [-60, -42, -36, 32, 84, 111]
```

```
    printBytes(utf8); // "UTF-8": [-61, -124, -61, -106, -61, -100, 32, 84,
111]
}
```

Listing 4.38 *Ausführbar als* **'STRINGENCODINGS'**

Ermitteln verfügbarer `Charset`-Objekte

Wie schon erwähnt, gibt es keine Sammlung von Konstanten aller gültiger `Charset`-Objekte oder ihrer Namen.[16] Die verfügbaren `Charsets` können über die statische Methode `Charset.availableCharsets()` ermittelt werden. Man erhält eine `Map<String, Charset>` mit etwa 150 Abbildungen von Namen auf die zugehörigen `Charset`-Objekte. Zugriff auf ein konkretes `Charset` bietet die statische Methode `forName(String)`.

Auswahl des passenden `Charset`-Objekts

Für einen sicheren Datenaustausch sollte man sich nicht darauf verlassen, dass das Defaultcharset passt. Vielmehr führt diese Annahme immer wieder zu Problemen. Besser ist es, dass gewünschte Charset explizit zu spezifizieren. Die Wahl hängt davon ab, welche Komponenten auf die Daten zugreifen können. Greifen auf eine Datei ausschließlich Java-Komponenten, sowohl lesend als auch schreibend, zu, wird man in der Regel den Standard (UTF-16 für die Klasse `String`) wählen. Sind dagegen Komponenten als Erzeuger oder Konsument beteiligt, die dieses Charset nicht unterstützen, etwa ältere Programme, die z. B. mit ASCII-Darstellungen arbeiten, wird man ein passendes `Charset`-Objekt nutzen, mit dessen Codierung diese Komponenten als Leser umgehen können bzw. die sie als Erzeuger vorgeben. Darüber hinaus gibt es Dateiformate, bei denen das Encoding festgelegt ist und die damit die Auswahlmöglichkeiten einschränken.

4.6.4 Speichern und Laden von Daten und Objekten

Im Folgenden werde ich anhand der Klasse `Person` die Speicherung eines Objekts in einem Stream sowie das Lesen und Rückgewinnen eines Objekts aus einem Stream vorstellen. In diesem einführenden Beispiel werden in der Realität auftretenden Probleme, etwa verschiedene Zeichensätze, umfangreiche Datenmengen, verschiedene Versionen der gespeicherten Daten usw. nicht betrachtet.

Wir beginnen mit einer Verarbeitung von einfachen Bytestreams, um danach den Einsatz der komfortableren `DataOutputStreams` bzw. `PrintStreams` zu zeigen. Die nachfolgenden Beispiele verzichten der Übersichtlichkeit halber auf eine Fehlerbehandlung. Dieses wichtige Thema behandle ich dann in Abschnitt 4.7.

Einsatz der Klassen `FileInputStream` und `FileOutputStream`

Es sollen Objekte vom Typ `Person` gespeichert und geladen werden. Folgendes Listing zeigt zur Erinnerung die zu verarbeitenden Attribute:

```
public class Person
{
    private final String name;
    private final String city;
    private final Date birthday;
    // …
```

Ein erster, naiver Ansatz besteht darin, direkt die Methoden `read()` und `write()` aus den Klassen `InputStream` und `OutputStream` zu verwenden, um die drei Attribute zu verarbeiten. Damit die Applikation nicht mit Details überfrachtet wird, erstellen wir zwei Hilfsmethoden `readPersonFromStream(InputStream)` und `writePersonToStream(Person, OutputStream)`, die diese Aufgaben unter Verwendung von `read()` und `write()` erledigen und kapseln. Folgendes Programm erzeugt ein `Person`-Objekt und ruft anschließend die beiden Hilfsmethoden auf, um das Objekt vom Typ `Person` zu speichern und anschließend wieder zu laden:

```
// Achtung: Kein korrektes Resource Handling … später dazu mehr
public static void main(final String[] args) throws Exception
{
    final File file = new File("Person.ser1");
    final Person max = new Person("Max", new Date(), "Hamburg");

    // Speichere Person 'Max' in einer Datei und lies die Daten wieder ein
    writePersonToStream(max, new FileOutputStream(file));
    final Person newPerson = readPersonFromStream(new FileInputStream(file));

    // Prüfe auf inhaltliche Gleichheit
    System.out.println("Gleich? " + max.equals(newPerson));
}
```

Listing 4.39 *Ausführbar als* **'PERSONSTREAMFILESTREAM'**

Schauen wir uns nun einige Details sowie die Realisierung der beiden Methoden an. Beginnen wir dabei mit dem Speichern von Daten in einem Stream.

Schreiben

Das Schreiben von Daten in einen Stream ist komplizierter, als es zunächst wirkt. Durch die Forderung, die Daten später wieder korrekt einlesen zu können, muss man sich eine spezielle Form der Speicherung überlegen, die beim Einlesen eine korrekte Rekonstruktion des Objekts erlaubt. Das erfordert für jedes Attribut eine Repräsentation und Konvertierung in eine Folge von Bytes. Alle Daten ohne Trennzeichen oder Längeninformation hintereinander zu schreiben, würde das Einlesen nahezu unmöglich machen: In dem Bytestrom wäre keine Struktur erkennbar und die Daten der Attribute wären nicht voneinander abzugrenzen.[17] Zur Unterscheidung könnte man spezielle Trennzeichen einfügen, die in den Nutzdaten natürlich nicht enthalten sein dürfen (bzw. ein gesondertes Escapen erfordern). Alternativ kann man für jede Byterepräsentation eines Attributs die Länge der

Konvertierung mit in den Datenstrom schreiben. Letzteres wollen wir nutzen. Dazu bietet es sich an, eine Hilfsmethode zu implementieren, die die Daten in eine Codierung aus Längenangabe, Leerzeichen und anschließend der Nutzbytes überführt und schreibt. Für dieses Beispiel wird vereinfachend davon ausgegangen, dass die zu verarbeitenden Daten lediglich aus Folgen von Bytes bestehen, die nicht länger als 255 Zeichen sind. Diese Vereinfachung erlaubt es, die Methode übersichtlich zu halten und nicht mit Logik zur Längenverarbeitung zu überfrachten. Dadurch lässt sich eine Hilfsmethode `writeByteArray(OutputStream, byte[])` wie folgt realisieren:

```
public static void writeByteArray(final OutputStream os,
                                   final byte[] bytesToWrite) throws
IOException
{
    // Die beiden folgenden Aufrufe nutzen die Methode write(int)
    // Tatsächlich werden die Werte automatisch auf den Typ byte beschränkt
    os.write(bytesToWrite.length);
    os.write(' ');
    // Aufruf der Methode write(byte[])
    os.write(bytesToWrite);
}
```

Hinweis: Resource Handling

Die in den Listings gezeigten Realisierungen sind vereinfacht (aber funktionstüchtig). Für die Praxistauglichkeit fehlen zwei wichtige Dinge: Erstens müssen Streams durch Aufruf von `close()` geschlossen werden, um Ressourcen freizugeben. Zweitens sollte ein Aufruf der Methode `flush()` nach dem Schreiben erfolgen, um in den Stream geschriebene Daten tatsächlich auch physikalisch in das durch den Stream gekapselte Medium zu übertragen, etwa bei gepufferter Verarbeitung einer Datei oder über das Netzwerk.

Konvertierung in ein `byte[]`

Wir können nun ein `byte[]` in einen Stream schreiben. Das ist schon ein guter Schritt, aber die Daten der Attribute liegen selten in Form eines `byte[]` vor. Also stellt sich die Frage: Wie konvertieren wir die Daten in ein `byte[]`? Für Strings ist diese Umwandlung noch relativ einfach, da die Klasse `String` bekanntermaßen eine `getBytes()`-Methode anbietet. Aber bereits beim `Date`-Attribut für das Geburtsdatum müssen wir ein wenig tricksen: Würde man eine simple textuelle Repräsentation verwenden, so käme es schnell zu Problemen durch verschiedene Interpretationen beim Parsing. Wir wandeln daher das Datum zunächst in einen `long`-Wert. Mithilfe von `Long.toString()` erhalten wir dann ein Stringobjekt, das wir anschließend wieder über `getBytes()` in eine Folge von Bytes umwandeln können. Zum Schreiben des `Person`-Objekts in einen `OutputStream` erstellen wir die Hilfsmethode `writePersonToStream()` wie folgt:

```
public static void writePersonToStream(final Person person,
                                       final OutputStream os) throws
IOException
```

```
{
    // String => getBytes()
    final byte[] nameBytes = person.getName().getBytes();
    final byte[] cityBytes = person.getCity().getBytes();
    // Date => long => String => getBytes()
    final long time = person.getBirthday().getTime();
    final String timeString = Long.toString(time);
    final byte[] timeBytes = timeString.getBytes();

    // Schreibe Attribute in den Stream
    writeByteArray(os, nameBytes);
    writeByteArray(os, cityBytes);
    writeByteArray(os, timeBytes);
}
```

Beim Betrachten des Sourcecodes erahnen wir, dass eine solche Art der Konvertierung relativ aufwendig und fehleranfällig werden kann. Doch machen wir erst einmal weiter, bevor wir nach Alternativen schauen.

Lesen

Bei der Rückgewinnung von Werten sind einige Dinge zu beachten, damit wir beim Einlesen aus einer Menge von Bytes wieder die korrekten Informationen ermitteln können. Dabei ist es wichtig, die Details und die Codierung des Schreibens genau zu beachten. Basierend auf der Methode `writeByteArray(OutputStream, byte[])` zum Schreiben erstellen wir die korrespondierende Hilfsmethode `readByteArray()` zum Einlesen wie folgt:

```
public static byte[] readByteArray(final InputStream is) throws IOException
{
    final int bytesLength = is.read();
    is.read();    // Überspringe Leerfeld
    final byte[] bytes = new byte[bytesLength];
    is.read(bytes);
    return bytes;
}
```

Sichere Objektrekonstruktion aus Streams

Wir haben nun die Grundbausteine zum Lesen und Schreiben erstellt, aber wie wandeln wir die Daten aus einem `InputStream` in ein `Person`-Objekt? Voraussetzung dafür ist, dass beim Einlesen die Implementierungsdetails des Schreibens bekannt sind und das Einlesen der Werte in der gleichen Reihenfolge wie beim Schreiben erfolgt. Auch die Umwandlung in die gewünschten Typen der Attribute setzt eine genaue Kenntnis des zu rekonstruierenden Objekts voraus, weil wir ja keine Typinformationen in den Stream geschrieben haben, da nur einfache Hilfsmethoden erstellt werden sollen.[18]

Um inkonsistente Objektzustände und andere Probleme zu vermeiden, ist es empfehlenswert, erst alle für die Objektkonstruktion nötigen Informationen aus dem Stream zu sammeln und das Objekt nur dann zu erzeugen, wenn die Daten vollständig sind:

```
private static Person readPersonFromStream(final InputStream is) throws
      FileNotFoundException, IOException
{
    final byte[] nameBytes = readByteArray(is);
    final String name = new String(nameBytes);

    final byte[] cityBytes = readByteArray(is);
    final String city = new String(cityBytes);

    final byte[] birthdayBytes = readByteArray(is);
    final long time = Long.parseLong(new String(birthdayBytes));
    final Date birthday = new Date(time);

    // Trick: immer erst alle Daten vor der Konstruktion einlesen
    return new Person(name, birthday, city);
}
```

Mögliche Fallstricke bei der Objektrekonstruktion aus Streams

Auf ein spezielles Problem beim Einlesen und der Objektrekonstruktion möchte ich an dieser Stelle hinweisen. Häufig sieht man initial die Konstruktion eines Objekts ohne Zustandsinformationen durch Aufruf eines Defaultkonstruktors. Der Grund ist simpel: Es fehlen noch einige Informationen, weil diese noch nicht eingelesen wurden. Stück für Stück werden diese Informationen dann nach erfolgreichem Einlesen per `set()`-Methoden den entsprechenden Attributen zugewiesen. Der Sourcecode sieht dann, vereinfacht und auf die Klasse `Person` übertragen, wie folgt aus:

```
final Person newPerson = new Person();
// read name
newPerson.setName(name);
// read birthday
newPerson.setBirthday(birthday);
// read city
newPerson.setCity(city);
```

Diese Technik der sukzessiven Initialisierung über `set()`-Methoden ist zu vermeiden, da sie fehleranfällig ist. Bei dieser Vorgehensweise drohen inkonsistente Objektzustände, falls nicht alle Daten ermittelt werden können, das Objekt aber schon angelegt wurde. Eine ausführliche Erörterung dieses Problems finden Sie in Abschnitt 3.1.5.

Einsatz der Klassen `DataInputStream` und `DataOutputStream`

Das Speichern und Lesen von `Person`-Objekten in bzw. aus Streams lässt sich einfacher als bisher lösen, wenn man dazu die Klassen `java.io.DataInputStream` und `java.io.DataOutputStream` nutzt. Beide Klassen bieten spezielle Methoden an, um Daten in einen Stream zu speichern bzw. daraus zu lesen. Dadurch muss man sich für primitive Typen und Strings keine Gedanken um die Konvertierung in Bytefolgen machen. Nützlich sind für dieses

Beispiel unter anderem folgende Methoden:

- `writeUTF()` und `readUTF()` – Verarbeitung von Strings
- `writeLong()` und `readLong()` – Verarbeitung von primitiven Typen, hier `long`

Im folgenden Listing ist eine Umsetzung der Verarbeitung von `Person`-Objekten in Streams mithilfe dieser Methoden gezeigt. Das Schreiben von Personendaten wird damit sehr einfach wie folgt möglich:

```
public static void writePersonToStream(final Person person,
                                       final DataOutputStream dataOutStream)
                                              throws IOException
{
    dataOutStream.writeUTF(person.getName());
    dataOutStream.writeUTF(person.getCity());
    dataOutStream.writeLong(person.getBirthday().getTime());
}
```

Ebenfalls wird das Einlesen erleichtert und lässt sich folgendermaßen realisieren – wobei hier der obige Tipp bezüglich des sukzessiven Einlesens berücksichtigt ist:

```
public static Person readPersonFromStream(final DataInputStream dataInStream)
      throws IOException
{
    final String name = dataInStream.readUTF();
    final String city = dataInStream.readUTF();
    final Date birthday = new Date(dataInStream.readLong());

    return new Person(name, birthday, city);
}
```

Offensichtlich sind diese Realisierungen kürzer und übersichtlicher. Sie besitzen zudem folgende Vorteile:

1. Es werden gut getestete Standardmethoden aus dem JDK eingesetzt. Die Realisierungen sind dadurch weniger fehleranfällig, besser lesbar und auch für Entwickler verständlich, die die zuvor selbst geschriebenen Methoden nicht kennen.
2. Die Umwandlung verschiedener Typen (`int`, `long` usw.) in eine Folge von Bytes und zurück geschieht durch die spezifischen, überladenen Methoden der Streams, etwa `readLong()`, für jede nutzende Applikation transparent.
3. Der Einsatz der Methoden `readUTF()` und `writeUTF()` erleichtert die Verarbeitung von Strings, z. B. müssen keine Längeninformationen der zu verarbeitenden Strings in den Stream geschrieben bzw. daraus gelesen werden. Dies war bei den gezeigten Eigenimplementierungen erforderlich, bei der eine Codierung der Länge als Byte stattfand. Dadurch kam es zu Längenbeschränkungen für die zu verarbeitenden Strings. Selbst wenn ein primitiver Datentyp mit größerem Wertebereich zum Einsatz kommt, besteht das Problem weiterhin.

Allerdings gibt es nach wie vor das Problem, die Reihenfolge bei der Speicherung und

bei einem späteren Einlesen exakt einzuhalten – ansonsten drohen Inkonsistenzen, Zugriffsfehler oder gar Exceptions. Im ungünstigsten Fall merkt man es noch nicht einmal!

Ausgaben mit der Klasse `PrintStream`

Die Klasse `java.io.PrintStream` stellt die von der Konsolenausgabe bekannten, für verschiedene Typen überladenen `print()`- und `println()`-Methoden zur Verfügung. Verwendet man die Klasse `PrintStream` zur Aufbereitung der Ausgabe, so kann man von den einzelnen spezifischen `write()`-Methoden des `DataOutputStreams` weiter abstrahieren. Man nutzt stattdessen die überladenen `println()`-Methoden:

```
public static void writePersonToPrintStream(final Person person,
                                            final PrintStream printStream)
                                                  throws IOException
{
    printStream.println(person.getName());
    printStream.println(person.getCity());
    printStream.println(person.getBirthday().getTime());
}
```

Es existiert allerdings kein passendes Stream-Gegenstück zum `PrintStream`, das dessen Ausgaben wieder einliest. Das Einlesen kann man in diesem Fall durch die Klasse `Scanner` realisieren, die nun besprochen wird.

Die Klasse `Scanner`

Die Klasse `java.util.Scanner` kann Eingabestrings aus verschiedenen Datenquellen in einzelne Bestandteile, sogenannte ***Tokens***, zerlegen und dabei primitive Typen direkt aus der Eingabe parsen. Bei der Konstruktion eines `Scanner`-Objekts ist die gewünschte Eingabequelle zu übergeben, die vom Typ `String`, `File`, `InputStream` oder `java.lang.Readable` ist. Letzteres wird von den beiden abstrakten Klassen `CharBuffer` und `Reader` implementiert und ermöglicht so eine Verarbeitung der I/O-Hierarchie der zeichenbasierten `Reader`. Nach der Konstruktion kann über die Methode `useDelimiter(String)` spezifiziert werden, welcher reguläre Ausdruck die Trennzeichenfolge beschreibt. Sofern dies nicht geschieht, wird als Trennzeichenfolge jede Folge von Whitespace-Charactern (Spaces, Tabs und Zeilenumbruch) verwendet.

Die Klasse `Scanner` implementiert das Interface `Iterator<String>`, wobei jedoch die Methode `remove()` eine `java.lang.UnsupportedOperationException` auslöst. Die Methode `next()` gibt das nächste Token als `String` zurück. Dies ist jedoch nur sicher möglich, wenn die Methode `hasNext()` die Existenz eines nachfolgenden Elements bestätigt hat. Ansonsten kann es zu einer `java.util.NoSuchElementException` kommen, wenn keine weiteren Daten

mehr folgen. Das Einlesen von Werten primitiver Datentypen erledigen überladene `next<Typ>()`-Methoden, etwa `nextLong()`. Analog zur `hasNext()`-Methode existieren `hasNext<Typ>()`-Methoden, etwa `hasNextLong()`. Durch deren Aufruf kann festgestellt werden, ob ein weiteres Token vom gewünschten Typ in der Eingabe vorhanden ist. Liest man jedoch ungeprüft beispielsweise einen `long`-Wert aus einer beliebigen Eingabequelle, und entspricht der gelesene Wert nicht dem erwarteten Typ, so reagiert der `Scanner` mit einer `java.util.InputMismatchException`.

Beispiel: Texte mit Scanner verarbeiten

Wir nutzen nun die bereits im Beispiel zur Klasse `StringTokenizer` verwendete Trennzeichenfolge aus den Zeichen '.', '_' und '-', um einen Text in Einzelbestandteile aufzusplitten.

```
public static void main(final String[] args)
{
    // ARM: Spezielle try-Syntax seit JDK 7
    try (final Scanner scanner = new Scanner("Version-2.17_45"))
    {
        scanner.useDelimiter("\.|_|-");
        while (scanner.hasNext())
        {
            System.out.print(scanner.next() + " ");
        }
    }
}
```

Listing 4.40 *Ausführbar als* **'SCANNEREXAMPLE'**

Startet man das Programm SCANNEREXAMPLE, so erhält man folgende Ausgabe:

```
Version 2 17 45
```

Abschließend sei darauf hingewiesen, dass man am Ende einer Verarbeitung mit einem `Scanner` diesen durch Aufruf der Methode `close()` wieder schließen sollte, um eventuell belegte Systemressourcen freizugeben. Im obigen Listing sehen wir den Einsatz von Automatic Resource Management (ARM) (vgl. Abschnitt 4.7.4), wodurch automatisch eine Ressourcenfreigabe erfolgt und kein expliziter Aufruf von `close()` erforderlich ist. Das macht den Sourcecode kürzer und übersichtlicher.

Ausgaben einlesen

Kommen wir nun zu unserem Beispiel zurück: Mithilfe der Klasse `PrintStream` wurden die einzelnen Attribute eines `Person`-Objekts mit einem Zeilenendezeichen als Trenner in einen Stream geschrieben. Diese Daten sollen mit einem `Scanner` wieder eingelesen werden. Dieser nutzt Whitespace und insbesondere ein Zeilenende als Begrenzer, weil nicht explizit Trennzeichen über `useDelimiter()` bekanntgegeben worden sind. Das Einlesen kann man wie folgt realisieren:

```
private static Person readPersonWithScanner(final Scanner scanner) throws
      IOException
```

```
{
    final String name = scanner.next();
    final String city = scanner.next();
    final long time = scanner.nextLong();
    final Date birthday = new Date(time);

    return new Person(name, birthday, city);
}
```

Die hier gezeigte Umsetzung verzichtet bewusst auf die normalerweise sinnvollen `hasNext()`-Prüfungen. Wir erwarten ein ganz spezielles Datenformat, da wir dieses zuvor selbst geschrieben haben. Fehler in den Eingabedaten werden durch diese Verarbeitung sofort sichtbar und führen zu Exceptions.

Tipp: Fehler während des Parsens

Bei der Verarbeitung von Eingaben durch einen `Scanner` kann es zu Fehlern in Form von `IOExceptions` kommen. Diese werden nicht weiter propagiert, sondern abgefangen und intern gespeichert. Die zuletzt aufgetretene Exception liefert die Methode `ioException()`, die bei Fehlerfreiheit `null` zurückgibt.

Vorteile gegenüber `StringTokenizer` und `String.split()`

Die Klasse `Scanner` verwendet zum Zerlegen ihrer Eingabe einen regulären Ausdruck, wodurch ein `Scanner` flexibler als ein `StringTokenizer` einzusetzen ist, da letzterer nur einzelne Zeichen als Trennzeichen zulässt. Auch bezüglich der `split()`-Methode der Klasse `String` stellt der `Scanner` eine Erweiterung dar, weil Daten aus verschiedenen Eingabequellen zerlegt werden können.

Fazit

Anhand der vorangegangenen Beispiele der manuellen Verarbeitung von `Person`-Objekten mit Streams sieht man, dass das Speichern und Einlesen von Objekten recht kompliziert werden kann. Dies gilt insbesondere, da in der Praxis Objekte selten so einfach aufgebaut sind wie in diesem Beispiel. Vielmehr werden häufig Referenzen auf andere Objekte gehalten oder diverse andere Objekte aggregiert. Auch diese gehören zum Zustand eines Objekts und müssen beim Speichern und Einlesen berücksichtigt werden. Die Komplexität steigt, wenn man Objekte mit allen referenzierten Objekten, den sogenannten ***Objektgraph***, speichern möchte. Es bietet sich in diesem Fall eher der Einsatz des in Java integrierten Serialisierungsmechanismus an, der in Abschnitt 8.3 genauer beschrieben wird.

Info: Das New I/O im Überblick

Mit JDK 1.4 wurde das New I/O (kurz: NIO) eingeführt. Dessen Klassen und Interfaces befinden sich im Package `java.nio`. Jedoch ist der Name NIO etwas

missverständlich, denn das herkömmliche, streambasierte I/O wird nicht ersetzt, sondern erweitert. Statt einzelner Bytes werden beim NIO Blöcke von Bytes verarbeitet. Diese werden durch Subklassen der Basisklasse `java.nio.Buffer` modelliert. Auf diesen Puffern arbeiten sogenannte Channels. Das sind Realisierungen des Interface `java.nio.channels.Channel`. Diese bieten eine neue Abstraktion für Verbindungen zu Datenquellen und -senken:

- **Buffer** – Die abstrakte Klasse `Buffer` definiert den Zugriff auf ein Array eines primitiven Typs. Entsprechende Subklassen realisieren die Funktionalität für die jeweiligen primitiven Typen, etwa `java.nio.ByteBuffer`, `java.nio.CharBuffer`, `java.nio.DoubleBuffer` usw.
- **Channels** – Die Implementierungen des Interface `Channel` definieren ähnlich wie die Streams im herkömmlichen `java.io` einen Kommunikationskanal zu I/O-»Geräten« wie Dateien und Sockets. Konkrete Realisierungen (`java.nio.channels.FileChannel`, `java.nio.channels.DatagramChannel`, `java.nio.channels.SocketChannel` usw.) implementieren die jeweilige Kommunikation. Das Besondere daran ist, dass nicht blockierende Lesezugriffe möglich sind und parallel sowohl Lese- als auch Schreibzugriffe erfolgen können. Für Streams ist dies nicht möglich: Ein Stream repräsentiert eine Subklasse von entweder `InputStream` oder aber `OutputStream`.

Der Einsatz des NIO sollte gut abgewogen werden, da die Benutzung nicht immer intuitiv ist und auf einer deutlich niedrigeren Abstraktion als über Streams stattfindet. Für viele Anwendungen sollte man mehr Wert auf Verständlichkeit als auf maximale Performance usw. legen. Hier bieten sich dafür weiterhin die vorgestellten Stream-und Reader-/Writer-Klassen an.

Ausführliche Informationen zum NIO finden Sie im Buch »Java NIO« von Ron Hitchens [40].

4.6.5 Dateiverarbeitung in JDK 7

Die Verarbeitung von Dateien war vor JDK 7 nicht immer komfortabel, etwa das Kopieren und Verschieben sowie das Resource Handling konnten mühsam sein. Schauen wir uns in diesem Abschnitt einige der Neuerungen zur Dateiverarbeitung in JDK 7 an, die diese Probleme adressieren. Wir betrachten unter anderem die Klassen und Interfaces `DirectoryStream`, `Files`, `FileSystem`, `Path`, `Paths`, `FileVisitor` und `WatchService`. Alle entstammen dem Package `java.nio.file`.

Das Interface `Path`

Einen wichtigen Bestandteil zur Arbeit mit Pfaden bildet das Interface `Path`. Zugriff

auf ein Objekt vom Typ `Path` erhält man durch Aufruf der überladenen Methoden `get(String)` bzw. `get(java.net.URI)` aus der Hilfsklasse `Paths`. Alternativ kann man die Methode `getPath(String, String...)` der Klasse `FileSystem` nutzen. Über das Interface `Path` wird ein Pfad oder im Speziellen eine Datei repräsentiert, die verarbeitet werden soll. Für die eigentlichen Aktionen und Zugriffe ist aber die Hilfsklasse `Files` zuständig. Dort finden wir etwa die Methoden `copy(Path, Path, CopyOption...)` und `move(Path, Path, CopyOption...)` zum Kopieren und Verschieben von Dateien. Beides wird im folgenden Listing genutzt:

```
public static void main(final String[] args) throws IOException
{
    // Zugriff auf das systemspezifische tmp-Directory
    final String tmpDir = System.getProperty("java.io.tmpdir");

    // Konstanten für die betroffenen Dateien
    final String SOURCE_FILE_PATH = "src/ch04_javagrundlagen/fileio/" +
                                    "FileIOExample.java";
    final String DESTINATION_FILE_PATH1 = tmpDir + "CopiedFile.java";
    final String DESTINATION_FILE_PATH2 = tmpDir + "CopyOfFileIOExample.java";

    // Erzeugen eines Path-Objekts mit der Klasse FileSystem
    final FileSystem local = FileSystems.getDefault();
    final Path fromPath = local.getPath(SOURCE_FILE_PATH);

    // Erzeugen von Path-Objekten mit der Utility-Klasse Paths
    final Path toPath1 = Paths.get(DESTINATION_FILE_PATH1);
    final Path toPath2 = Paths.get(DESTINATION_FILE_PATH2);

    // Dateien vorsorglich löschen
    Files.deleteIfExists(toPath1);
    Files.deleteIfExists(toPath2);
    // Kopieren und verschieben
    Files.copy(fromPath, toPath1);
    Files.move(toPath1, toPath2);

    final File dir = new File(tmpDir);
    final String[] content =
                   dir.list(new PreAndPostFixFilenameFilter("Copy", ".java"));
    System.out.println("Content of tmpDir '" + tmpDir + "': " +
                       Arrays.toString(content));
}
```

Listing 4.41 *Ausführbar als* **'FILEIOEXAMPLE'**

Das Interface `Path` stellt diverse nützliche Methoden zum Bearbeiten von `Path`-Objekten bereit. Es kann beispielsweise die Anzahl der Pfadbestandteile ermittelt oder ein Teilpfad mit `subpath(int, int)` extrahiert werden. Ein Aufruf der Methode `relativize(Path)` wandelt einen Pfad in einen Pfad relativ zu einem bestehenden Pfad um. Die neue Funktionalität kann auch ohne eine vollständige Umstellung auf das neue API in älterem Sourcecode verwendet werden. Dazu besteht die Möglichkeit, `File`-Objekte durch Aufruf der Methode `toPath()` in `Path`-Objekte zu konvertieren, wie dies hier für das aktuelle Verzeichnis gezeigt ist:

```
public static void main(final String[] args)
{
```

```
    // Konvertierung java.io.File -> java.nio.file.Path
    final Path currentDirOldStyle = new File(".").toPath();

    final FileSystem fileSystem = FileSystems.getDefault();
    final Path currentDir = fileSystem.getPath(".");

    // Als absoluten Pfad ausgeben
    final Path absolutePath = currentDir.toAbsolutePath();
    System.out.println(absolutePath);

    // Pfadbestandteile ermitteln
    System.out.println(absolutePath.getNameCount());
    for (int i = 0; i < absolutePath.getNameCount(); i++)
    {
        System.out.println(absolutePath.getName(i));
    }
}
```

Listing 4.42 *Ausführbar als* **'PATHOPERATIONSEXAMPLE'**

Das Interface `DirectoryStream`

Das Interface `DirectoryStream` implementiert das Interface `Iterable<Path>` und ermöglicht dadurch, über den Inhalt eines Verzeichnisses zu iterieren. Auf diese Weise kann ein sukzessives Einlesen der darin enthaltenen Dateien und Unterverzeichnisse erfolgen. Durch Aufruf der Methode `newDirectoryStream(Path)` der Hilfsklasse `Files` erhält man den beschriebenen Zugriff:

```
public static void main(final String[] args) throws IOException
{
    // Konvertierung java.io.File -> java.nio.file.Path
    final Path currentDir = new File(".").toPath();
    // ARM: Spezielle try-Syntax seit JDK 7
    try (final DirectoryStream<Path> dirStream =
                                    Files.newDirectoryStream(currentDir))
    {
        for (final Path entry : dirStream)
        {
            System.out.println(entry);
        }
    }
}
```

Listing 4.43 *Ausführbar als* **'DIRECTORYSTREAMEXAMPLE'**

Zunächst mag man sich fragen, was der Vorteil daran sein soll. Tatsächlich ist bei Verzeichnissen auf dem lokalen Rechner mit einer üblichen Anzahl an Dateien (10 – 100) der Vorteil nicht sofort ersichtlich. Es wird aber eine effiziente Verarbeitung auch für solche Verzeichnisse möglich, die sehr viele Einträge enthalten. Bei dieser Form der Verarbeitung werden zudem weniger Ressourcen benötigt, als wenn man den Verzeichnisinhalt vollständig »in einem Rutsch« einlesen würde. Letzteres führt außerdem häufig zu (längeren) Wartezeiten. Durch den Einsatz eines `DirectoryStreams` können selbst bei einer großen Anzahl enthaltener Dateien und Verzeichnisse gute Antwortzeiten sichergestellt werden. Diesen Vorteil erkennt

man besonders, wenn man Verzeichnisse auf anderen Computern über ein Netzwerk mithilfe eines `DirectoryStream`s untersucht.

Ähnlich zu den Methoden `list()` und `listFiles()` der Klasse `File` ist es durch die Angabe eines Filters möglich, die Treffermenge einzuschränken.

Das Interface `FileVisitor`

Manchmal sind Aktionen auf allen Elementen eines Verzeichnisses (auch allen rekursiv enthaltenen) auszuführen. Dies kann man etwa für die rekursive Berechnung der Gesamtgröße eines Verzeichnisses nutzen. Im folgenden Listing wird die beschriebene Funktionalität eingesetzt, um alle Dateien im Verzeichnis 'config/tiles' zu ermitteln, deren Dateiname die übergebene Dateiendung besitzt, hier für "JPG" gezeigt:

```
public static void main(final String[] args) throws IOException
{
    final ExtensionFilterVisitor jpgVisitor = new
ExtensionFilterVisitor("JPG");
    Files.walkFileTree(Paths.get("config/tiles"), jpgVisitor);

    final List<Path> results = jpgVisitor.getResults();
    System.out.println("JPG-Files: " + results);
}
```

Listing 4.44 *Ausführbar als* **'FILEVISITOREXAMPLE'**

Die statische Methode `walkFileTree(Path, FileVisitor<? super Path>)` der Utility-Klasse `Files` erlaubt es, einen Verzeichnisbaum zu durchlaufen. Diese Methode erledigt die Ermittlung enthaltener Elemente sowie, falls nötig, den rekursiven Durchlauf aller enthaltenen (Unter-)Verzeichnisse.

Die eigentliche Verarbeitung der Dateien wird durch Realisierungen des Interface `FileVisitor<Path>` erledigt. Dort ist eine Methode `visitFile(Path, BasicFileAttributes)` definiert. Diese Methode wird als Callback-Methode für jede besuchte Datei aufgerufen. Realisierungen des Interface können in der Implementierung der Methode beliebige Aktionen für die einzelnen Elemente definieren.

Zur Demonstration der Möglichkeiten des Interface `FileVisitor<Path>` implementiere ich eine statische innere Klasse `ExtensionFilterVisitor`. In der Realisierung der Methode `visitFile(Path, BasicFileAttributes)` erfolgt eine Filterung auf Dateien, die der übergebenen Dateiendung entsprechen. Dazu werden die einzelnen Elemente betrachtet. Diese werden als `Path`-Objekte an die Methode übergeben. Die Methoden der Klasse `Path`, etwa `getFileName()` oder `endsWith(Path)`, liefern bzw. prüfen wiederum `Path`-Objekte. Daher wird erst eine Konvertierung in ein Stringobjekt und danach ein Vergleich über dessen Methode `endsWith(String)` durchgeführt:

```
public static class ExtensionFilterVisitor extends SimpleFileVisitor<Path>
{
    private final String extension;
```

```
    private final List<Path> results = new ArrayList<>();

    public ExtensionFilterVisitor(final String extension)
    {
        this.extension = extension.toLowerCase();
    }

    public FileVisitResult visitFile(final Path file,
                                     final BasicFileAttributes attrs)
    {
        final Path filePath = file.getFileName();
        System.out.println("Visiting file " + filePath);

        // stringObj.endsWith(String) statt file.getFileName().endsWith(other)
        if (filePath.toString().toLowerCase().endsWith(extension))
        {
            results.add(file);
        }

        // Verarbeitung fortsetzen
        return FileVisitResult.CONTINUE;
    }

    private List<Path> getResults()
    {
        return new ArrayList<>(results);
    }
}
```

Durch verschiedene Rückgabewerte vom Typ `java.nio.file.FileVisitResult` kann gesteuert werden, ob die Abarbeitung fortgesetzt, beendet oder einzelne Elemente übersprungen werden. Zur Vereinfachung der Programmierung eigener Realisierungen existiert eine Defaultimplementierung aller Methoden des `FileVisitor`-Interface in der Klasse `java.nio.file.SimpleFileVisitor`. Dadurch müssen in eigenen Spezialisierungen von `SimpleFileVisitor` nur noch die Methoden überschrieben werden, die tatsächlich spezifisches Verhalten definieren sollen.

`DirectoryObserver` basierend auf JDK 7

Das Interface `Path` erweitert das Interface `java.nio.file.Watchable` und ermöglicht es, einen Beobachter für Veränderungen am Verzeichnisinhalt zu registrieren. Ein solcher Beobachter wird durch die Fabrikmethode `newWatchService()` erzeugt und kann dann bei einem `Path`-Objekt angemeldet werden. Dadurch kann man bei Änderungen am Verzeichnisinhalt informiert werden, ohne selbst auf Änderungen zu pollen. Durch Einsatz der neuen Klassen des JDK 7 können wir die Klasse `DirectoryObserver` aus Abschnitt 4.6.1 nun ohne Polling wie folgt realisieren:

```
public class DirectoryCheckerWithWatchService extends DirectoryObserver
{
    private final Path          pathToCheck;
    private final WatchService  watchService;
    private WatchKey            registerKey;
```

```
    public DirectoryCheckerWithWatchService(final String
nameOfDirectoryToCheck)
            throws IOException
    {
        super(nameOfDirectoryToCheck);

        this.pathToCheck = Paths.get(nameOfDirectoryToCheck);
        this.watchService = FileSystems.getDefault().newWatchService();

        this.registerKey = pathToCheck.register(watchService, ENTRY_CREATE,
                                                ENTRY_DELETE, ENTRY_MODIFY);
    }

    @Override
    public void checkDirectory()
    {
        System.out.println("starting directory check for directory='" +
                           getDirectoryToCheck() + "'");

        int numOfFiles = FileUtils.getContents(getDirectoryToCheck()).length;

        while (!Thread.currentThread().isInterrupted())
        {
            System.out.println("checkDirectory…");
            if (FileUtils.directoryExists(getDirectoryToCheck()))
            {
                numOfFiles = checkForContentsChanged(numOfFiles);
            }
            else
            {
                onDirectoryNotExisting();
                SleepUtils.safeSleep(getCheckIntervalInSec(),
TimeUnit.SECONDS);
                numOfFiles = 0;
            }
            System.out.println("…checkDirectory");
        }
    }

    @Override
    protected int checkForContentsChanged(final int numOfFiles)
    {
        try
        {
            // Blockierend auf Änderungen warten
            final WatchKey key = watchService.take();

            for (final WatchEvent<?> event : key.pollEvents())
            {
                reportModification(event);
            }
            key.reset();
         }
         catch (final InterruptedException e)
         {
             Thread.currentThread().interrupt();
         }
         return getContents(getDirectoryToCheck()).length;
    }
```

```
    private void reportModification(final WatchEvent<?> event)
    {
        final Path file = (Path) event.context();
        System.out.println(event.kind() + ": " + file);
    }

    public static void main(final String[] args) throws IOException
    {
        final String tmpDir = System.getProperty("java.io.tmpdir");
        new DirectoryCheckerWithWatchService(tmpDir).checkDirectory();
    }
}
```

Listing 4.45 *Ausführbar als* **'DIRECTORYCHECKERWITHWATCHSERVICE'**

4.7 Fehlerbehandlung

Jedes Programm muss mit gewissen Fehlersituationen umgehen können. Es erfordert immer etwas an Mehrarbeit und zusätzliche Zeilen Sourcecode, auf unerwartete Situationen zu reagieren oder Fehler abzufangen. Dieser Aufwand ist allerdings erforderlich, um für stabile Software zu sorgen.

Eine fehlende oder inkorrekte Fehlerbehandlung macht sich im User Interface z. B. dadurch bemerkbar, dass die Sanduhr als Wartecursor niemals wieder auf den Pfeil (Standardcursor) gesetzt wird. In der Business-Logik oder der Datenzugriffsschicht kommt es u. a. zu nicht geschlossenen Dateien, Datenbankverbindungen oder Sockets. Solche Probleme in tieferen Ebenen der Software bleiben dem Anwender häufig zunächst verborgen und machen sich möglicherweise erst viel später bemerkbar.

4.7.1 Einstieg in die Fehlerbehandlung

Java bietet zur Reaktion auf Fehlersituationen mehrere Alternativen. Durch den Einsatz von Ausnahmen, sogenannten ***Exceptions***, kann man auf ungewöhnliche Situationen reagieren. Mithilfe von Zusicherungen oder ***Assertions*** ist es möglich, gewisse Zustände im Programm zu prüfen. Auch durch spezielle Rückgabewerte können Fehlersituationen beschrieben werden.

Eine Faustregel ist, Exceptions für solche Situationen zu verwenden, die außerhalb des erwarteten Verhaltens liegen. Kann eine aufgerufene Methode einen Rückgabewert liefern, der es erlaubt, eine Fehlersituation sinnvoll auszudrücken, so sind Rückgabewerte zum Teil handlicher. Lässt sich eine Fehlersituation auf diese Weise allerdings nicht darstellen oder kann bei einem fehlerhaften Eingabewert nicht sinnvoll weitergearbeitet werden, so sollte man eine Exception auslösen. Diese Fälle werden im nachfolgenden Listing anhand einer Methode beispielhaft illustriert:

```
public Object findById(final Long objectId)
{
    // Ungültiger Parameter => Exception
    if (objectId == null)
```

```
            throw new NullPointerException("objectId must not be null");

        // Ungültiger Parameterwert => Exception
        if (objectId < 0)
            throw new IllegalArgumentException("objectId must not be negative");

        if (DbHelper.isValidId(objectId))
        {
            final Object obj = DbHelper.findById(objectId);

            // Zustandsprüfung mit Assertion
            assert (obj != null);

            return obj;
        }

        // Spezieller Rückgabewert für keinen Treffer
        return null;
    }
```

Bereits anhand dieses Beispiels wird Folgendes erkennbar: Die Behandlung von Fehlern besitzt zahlreiche Facetten und ein allgemein richtiges Verhalten gibt es nicht. Vielmehr sollte man eine für die jeweilige Situation adäquate Reaktion auf einen Fehler vornehmen. Beispielsweise sollte immer ein passender Exception-Typ gewählt werden. Vier recht gebräuchliche finden sich im Package `java.lang`:

- `IllegalArgumentException` – Mit einer `IllegalArgumentException` können falsche Belegungen von Parametern ausgedrückt werden.
- `NullPointerException` – Sind Eingabewerte `null`, so kann man darauf mit einer `NullPointerException` reagieren.
- `IllegalStateException` – Sind benötigte Daten nicht korrekt initialisiert, so kann dies über eine `IllegalStateException` kommuniziert werden.
- `UnsupportedOperationException` – Auf eine fehlende Implementierung kann mittels einer `UnsupportedOperationException` hingewiesen werden.

Hinweis: IllegalArgumentException vs. NullPointerException

Ich persönlich bevorzuge auch bei einem Übergabewert von `null` eine `IllegalArgumentException` auszulösen, da sie klarer ausdrückt, dass ein Parameter einen falschen Wert besitzt. Im obigen Beispiel habe ich zu Demonstrationszwecken ausnahmsweise eine `NullPointerException` genutzt. Eine solche ist für mich eher eine Exception, die auf unerwartete Initialisierungsprobleme hindeutet. Bei der Nutzung der JDK-Hilfsmethode `Objects.requireNonNull()` weiche ich von dem genannten Ratschlag ab, da diese eine `NullPointerException` auslöst.

Behandlung von Exceptions

Exceptions machen auf Fehlersituationen aufmerksam, die behandelt werden sollten bzw. müssen. Dazu dient in Java ein `catch`-Block. In diesem notiert man den Sourcecode, der zur Behandlung der Fehlersituation ausgeführt werden soll.

Leider sieht man in einigen Büchern und auch in realem Anwendungscode `catch`-Blöcke, die leer sind oder gerade noch die Exception per `ex.printStackTrace()` auf der Konsole ausgeben, etwa folgendermaßen:

```
// Achtung: Keine geeignete Reaktion auf Fehler
catch (final IllegalArgumentException ex)
{
    ex.printStackTrace();
}
```

Dies ist natürlich keine angemessene Reaktion auf eine Fehlersituation – sondern Verhalten von unzuverlässiger Schönwettersoftware. Unser Ziel sollte es sein, dass man sich als Anwender des Programm selbst in Fehlersituationen darauf verlassen kann. Für unsere Kunden ist etwa wichtig, dass keine Daten zerstört werden oder verloren gehen. Für uns als Entwickler ist wichtig, dass wir möglichst genaue Informationen zu den Ursachen und auslösenden Programmzeilen sowie aktuellen Wertebelegungen relevanter Variablen erhalten. Daher gilt: ***Behandle auftretende Exceptions, wenn dies sinnvoll möglich ist. Im Zweifelsfall sollten Exceptions (ggf. mit Wrapping) an höhere Aufrufebenen weiter propagiert werden***.

Am Beispiel der zuvor gezeigten Methode `findById(Long)` schauen wir uns nun die Behandlung von Exceptions an: Dazu muss jeder Abschnitt des Sourcecodes, der potenziell eine Exception auslösen kann, entweder in einem `try`-Block ausgeführt werden, wie dies nachfolgend gezeigt ist,[19] oder aber die möglicherweise auftretende Exception in der Methodensignatur angeben:

```
try
{
    // Potenziell Exception auslösend
    final Object obj = findById(objectId);

    // …
}
catch (final IllegalArgumentException ex)
{
    // Fehlerbehandlung
    errorCounter++;

    JOptionPane.showMessageBox("Ungültige objectId – darf nicht leer sein");

    // …
}
```

Spezifische Fehlerbehandlung und abgeleitete Exceptions

Oftmals können durch die Anweisungen in einem `try`-Block verschiedene Exceptions ausgelöst werden. Für diesen Fall kann man zur Behandlung mehrere `catch`-Blöcke wie folgt dafür definieren:

```
try
{
    // …
    final File file = new File(…);
    // …
}
catch (final IOException ioe)
{
    // Fehlerbehandlung
    handleIOException(ioe);
}
catch (final ParseException pex)
{
    // Fehlerbehandlung
     handleParseException(pex);
}
```

Dabei gibt es einen Spezialfall zu bedenken: Stehen die Exceptions in einer Vererbungsbeziehung, etwa `FileNotFoundException` als Spezialisierung von `IOException`, so muss in den `catch`-Blöcken immer die ***spezialisierteste Exception*** zuerst behandelt werden: Die Reihenfolge der in den `catch`-Blöcken gefangenen Exceptions muss ***entgegengesetzt zur Ableitungshierarchie*** erfolgen:

```
try
{
    // …
    final File file = new File(…);
    // …
}
// Fehlerbehandlung entgegengesetzt zur Ableitungshierarchie der Exceptions
catch (final FileNotFoundException fnfe)
{
    handleFileNotFoundException(fnfe);
}
catch (final IOException ioe)
{
    handleIOException(ioe);
}
```

Um weniger Schreibaufwand zu haben, sieht man teilweise Folgendes:

```
catch (final Exception ex)
{
    handleException(ex);
}
```

Das ist zwar syntaktisch möglich, jedoch nur extrem selten adäquat, da man so nicht mehr spezifisch auf unterschiedliche Probleme reagieren kann. Das behandeln wir später als BAD SMELL: FANGEN DER ALLGEMEINSTEN EXCEPTION in Abschnitt 16.3.4.

Rückgabewerte vs. Exceptions

Nach dieser Einführung in die Fehlerbehandlung mithilfe von Exceptions möchte ich auf einen weiteren wichtigen Aspekt dabei eingehen: Die Fehlerbehandlung sollte möglichst sauber vom »Nutzcode« trennbar sein. Meistens lässt sich dies durch die Nutzung von Exceptions einfacher als durch den Einsatz von Rückgabewerten

erreichen. Letzteres verursacht recht schnell ein Chaos und sorgt für eine Vermischung von Anwendung und Fehlerbehandlung, was zu einer schlechten Trennung von Zuständigkeiten führt. Exceptions bieten mit ihrer Verarbeitung in `catch`-Blöcken eine bessere Trennung. Allerdings kann es durch mehrere zu behandelnde Exceptions schnell zu vielen korrespondierenden `catch`-Blöcken kommen.

Ein Vorteil von Exceptions ist, dass sich diese an Aufrufer propagieren lassen, wenn man meint, an anderer Stelle besser und angemessener auf Fehlersituationen reagieren zu können. Ein Nachteil ist, dass durch Exceptions ein Overhead zur Laufzeit für das Erzeugen (Aufbereitung des Stacktrace), Durchreichen usw. entsteht.[20] Für Rückgabewerte gilt das nicht – allerdings lassen sich diese auch nicht so gut weiter nach oben durchreichen.

Fallstricke bei der Fehlerbehandlung

Erfahrenen Entwicklern ist bewusst, dass zum Teil mehr Mühe und Gehirnschmalz in die Behandlung möglicher Fehlersituationen einfließen muss als in die normale Programmlogik. Dies gilt vor allem dann, wenn man mit anderen Systemen, Komponenten, Ein- und Ausgabegeräten und im Besonderen mit dem Benutzer interagiert. Für unerfahrene Programmierer ist Fehlerbehandlung oftmals eine der Sachen, um die man sich eher ungern oder halbherzig kümmert, meistens gerade so weit, dass das Programm nicht abstürzt. Zwei Beispiele für misslungene Fehlerbehandlungen sind folgende:

- **»Anstands«-Null-Prüfungen** – Solche Prüfungen dienen lediglich zur Vermeidung von `NullPointerExceptions`:

```
public static void updateSystemState()
{
    final SystemState state = getLifeCheckSystemState();
    if (state != null)
    {
        systemStateMap.put(KEY_SYSTEM, state);
    }
}
```

  Im Falle eines `null`-Werts für die Variable `state` geschieht einfach nichts. Problematisch daran ist, dass auf diese Weise ein spezieller Zustand (`state == null`), der möglicherweise gar ein Fehlerfall ist, kaschiert wird und zudem keine Behandlung oder Warnmeldung erfolgt. Abschnitt 16.3.7 diskutiert dies als BAD SMELL: SONDERBEHANDLUNG VON RANDFÄLLEN.

- **»Delayed Exception«** – Ein Beispiel für diese »Fehlerbehandlungsstrategie« sind öffentliche Methoden, die keine Konsistenzprüfung ihrer Parameter durchführen und so die Eingabe ungültiger Daten ermöglichen. Aufgrund der fehlenden Prüfung führen derartige Eingaben später potenziell zu fehlerhaften Berechnungen oder gar zu (unerwarteten) Exceptions. Im nachfolgenden Beispiel können `null`-Werte in einer `systemStateMap` gespeichert werden,

obwohl diese keine gültige Eingabe darstellen. Allerdings erfolgt die Konsistenzprüfung erst bei Lesezugriffen (und damit zu spät) in der `getSystemState()`-Methode:

```
public static void setSystemState(final SystemState state)
{
    systemStateMap.put(KEY_SYSTEM, state);
}

public static SystemState getSystemState()
{
    final SystemState state = systemStateMap.get(KEY_SYSTEM);
    if (state == null) // Beispiel für schlechte Fehlerbehandlung
        throw new IllegalStateException("No entry for system state");

    return state;
}
```

Das Problem fällt zunächst nicht auf. Erst wenn später ein Aufruf von `getSystemState()` erfolgt, wird die ungültige Eingabe sichtbar und löst eine Exception aus: Eine mögliche Fehleingabe bleibt dadurch unter Umständen sehr lange unerkannt und ein solcher Fehler wird viel zu spät, vielleicht sogar nie erkannt.

4.7.2 Checked Exceptions und Unchecked Exceptions

In Java unterscheidet man zur Behandlung von Fehlersituationen zwischen Checked Exceptions und Unchecked Exceptions. Nachfolgende Abbildung 4-6 zeigt die Ableitungshierarchien der beiden Arten von Exceptions.

Checked Exceptions sind Bestandteil des »Vertrags« zwischen Aufrufer und Bereitsteller einer Methode und zeigen mögliche, durch Aufrufer zu erwartende Fehlersituationen an. Sie müssen daher in der Methodensignatur mit dem Schlüsselwort `throws` angegeben werden. Dies sichert ab, dass Aufrufer aktiv darauf reagieren (Behandlung mit einem `catch`-Block oder durch die Propagation an weitere aufrufende Methoden). Eine Behandlung ist für ***Unchecked Exceptions*** nicht zwingend erforderlich, da diese normalerweise schwerwiegende Programmierprobleme oder unerwartete Situationen ausdrücken, auf die ein Aufrufer nicht sinnvoll reagieren kann.[21] Man kann Unchecked Exceptions etwa zur Signalisierung ungültiger Parameterwerte in Form einer `IllegalArgumentException` nutzen.

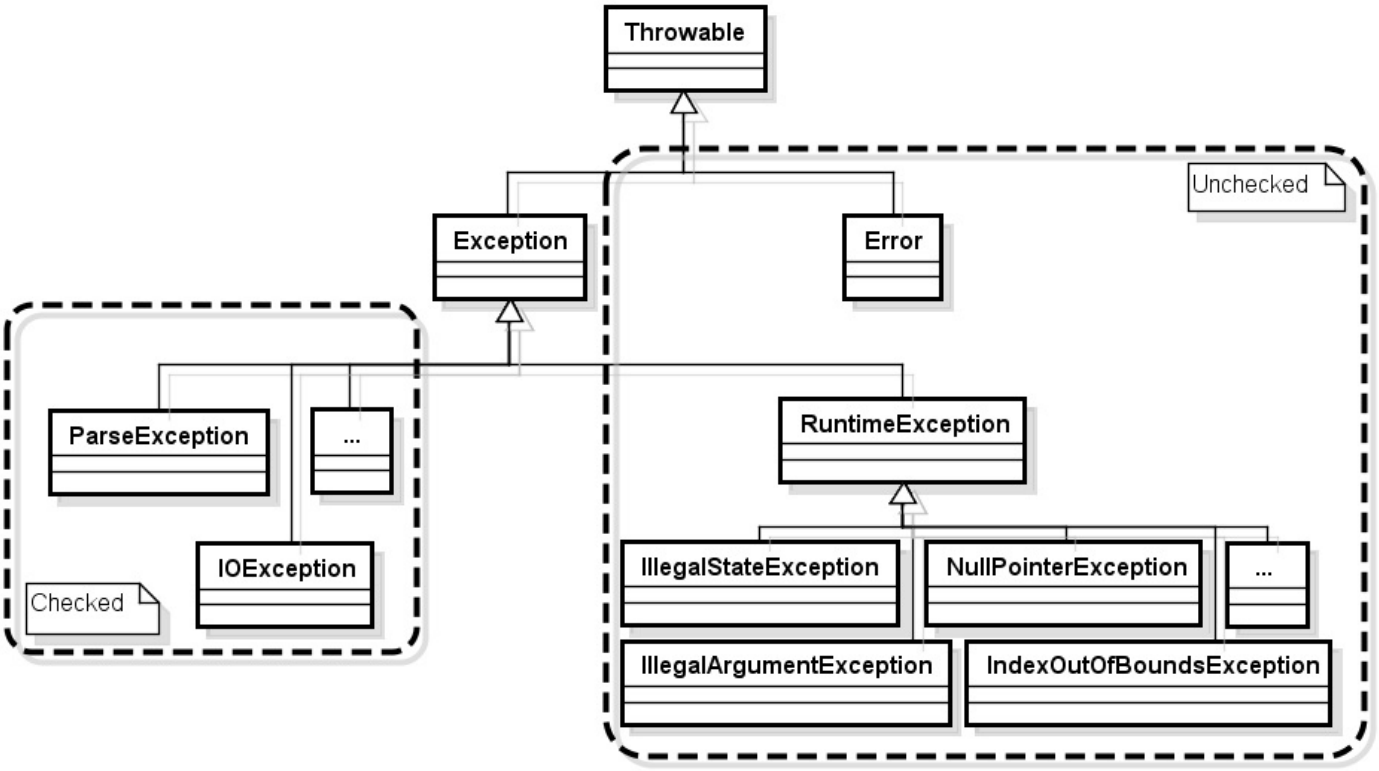


Abbildung 4-6 *Exception-Hierarchie*

Einige Programmierer geraten in die Versuchung, nur noch Unchecked Exceptions einzusetzen und eigene Exception-Klassen von `java.lang.RuntimeException` abzuleiten. Obwohl dies vom »lästigen« Bearbeiten einer möglicherweise auftretenden Exception befreit und auch keine Fehler beim Kompilieren provoziert, erschwert ein derartiges Vorgehen es manchmal doch, angemessen auf Fehlersituationen reagieren zu können: Einerseits sieht man in der Methodensignatur nicht unbedingt, dass eine Exception geworfen wird, weshalb dies in der Methodendokumentation aufgeführt werden sollte. Andererseits kann die Exception einfach ignoriert werden. Sie führt dann zur Laufzeit zu einem unerwarteten Programmfehler. Im Extremfall wird eine solche Exception unbehandelt bis zur `main()`-Methode bzw. `run()`-Methode des gerade aktiven Threads und damit zur JVM propagiert, wodurch dieser Thread terminiert.

Aus dieser kurzen Diskussion leiten wir folgende Regel ab: ***Wenn ein Aufrufer eine außergewöhnliche Situation behandeln kann, so sollte in der Regel eine Checked Exception geworfen werden. Ist ein Aufrufer höchstwahrscheinlich nicht in der Lage, eine Fehlersituation zu korrigieren, so verwendet man eine Unchecked Exception***.

4.7.3 Exception Handling und Ressourcenfreigabe

Das Thema Exceptions und Ressourcenfreigabe möchte ich nun ausführlicher behandeln. Zur Demonstration wähle ich eine Netzwerkkommunikation über Sockets, wobei Sie zur Einführung und bei Bedarf einige Informationen zu Sockets im nachfolgenden Hinweis »Netzwerkkommunikation mit Sockets im

Kurzüberblick« finden. Das Beispiel zeigt eine unzureichende Fehlerbehandlung. Das Ganze wird schrittweise verbessert. In der letzten Ausbaustufe ist dann sichergestellt, dass belegte Systemressourcen auch wieder freigegeben werden.

Hinweis: Netzwerkkommunikation mit Sockets im Kurzüberblick

Sockets bilden logische Endpunkte einer Verbindung zwischen Computern und erlauben das Senden und Empfangen von Nachrichten. Es existieren verschiedene Arten von Sockets für unterschiedliche Verbindungsarten. Eine Kommunikation kann z. B. über TCP erfolgen. ***TCP*** steht für ***Transmission Control Protocol*** und stellt eine sichere und fehlerfreie Verbindung zwischen zwei Kommunikationsendpunkten im Netz dar. Ähnlich wie in einem Telefonnetz einer Firma können Kommunikationspartner über Durchwahlnummern, sogenannte ***Ports***, erreicht werden.

Sockets bieten eine streambasierte Schnittstelle zur Kommunikation. Mithilfe von Sockets können Daten bidirektional übertragen werden. Soll eine Kommunikation zwischen zwei Kommunikationspartnern erfolgen, so geschieht dies in der Regel, indem einer der beiden Kontakt zu dem anderen aufnimmt (***Client-Server-Kommunikation***). Bei dieser Kommunikation stellt ein Client den aktiven Part der Kommunikation dar: Er schickt Anfragen an einen Server, der wiederum Antworten an den Client sendet. Dazu wartet ein Server mithilfe eines `ServerSocket` und dessen `accept()`-Methode auf eingehende Verbindungen. Die Kommunikation zwischen Client und Server wird dann mit der Klasse `Socket` realisiert, die Zugriff auf jeweils eine `InputStream`- und eine `OutputStream`-Instanz bietet. Ergänzende Informationen liefert das Buch »Java Network Programming« von Elliotte Rusty Harold [38].

Ausgangsbeispiel

Im nachfolgenden Listing wird ein `ServerSocket` erzeugt und dann durch Aufruf einer Methode `handleIncomingConnections()` auf eingehende Daten gewartet.

```
// Beispiel für schlechtes EXCEPTION HANDLING
public static void openAndProcess()
{
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        final ServerSocket serverSocket = new
ServerSocket( AUTO_ALLOCATE_PORT);

        // Weitere Verarbeitung - hier jedoch uninteressant
        handleIncomingConnections( serverSocket);

        serverSocket.close();
    }
```

```
    catch ( final IOException ex)
    {
        ex. printStackTrace();
    }
}
```

Hier wird jedoch eine problematische Fehlerbehandlung implementiert: Solange alle Aufrufe erfolgreich sind bzw. ohne Exceptions ablaufen, wird nach Abarbeitung der durch einen Kommentar angedeuteten Aktionen das Socket durch einen Aufruf der Methode `close()` wieder geschlossen. Wird jedoch eine Exception ausgelöst, so wird die Ausführung innerhalb des `try`-Blocks unterbrochen und die Anweisungen des `catch`-Blocks ausgeführt. Insbesondere kommt es *nicht* zum Aufruf von `close()`. Bei derartigen Fehlersituationen bleibt demnach die vom Betriebssystem bereitgestellte Netzwerkressource alloziert (zumindest eine gewisse Zeit).

Schritt 1: Ressourcenfreigabe im Fehlerfall

Ein erster Reparaturschritt besteht darin, im `catch`-Block einen Aufruf der Methode `close()` zu ergänzen. Dies erfordert aber einige Umbauarbeiten: Zunächst benötigen wir Zugriff auf die Variable `serverSocket` im `catch`-Block. Dazu muss diese vor dem `try-catch`-Block definiert werden. Zudem erfordert der Aufruf der Methode `close()` im `catch`-Block einen weiteren `try-catch`-Block, da in der Signatur der `close()`-Methode eine `IOException` definiert ist. Diese Exception kann man sehr selten sinnvoll und oftmals nur durch einen leeren `catch`-Block behandeln.

```
// Beispiel für schlechtes EXCEPTION HANDLING
public static void openAndProcess()
{
    ServerSocket serverSocket = null;
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        serverSocket = new ServerSocket( AUTO_ALLOCATE_PORT);

        handleIncomingConnections( serverSocket);

        serverSocket. close();
    }
    catch ( final IOException ex)
    {
        if ( serverSocket != null)
        {
            try
            {
                serverSocket. close();
            }
            catch ( final IOException e)
            {
                // Sollte close() eine IOException werfen? Ja und Nein!
                // Häufig ist das eher unpraktisch. Es ist aber hilfreich,
                // wenn sich das Programm merken will, dass eine Ressource
                // nicht korrekt geschlossen werden konnte und der weiter-
                // gehende Zugriff erst einmal unterbunden werden soll.
```

```
            }
        }
    }
}
```

Der Konstruktor der Klasse `ServerSocket` kann eine `IOException` werfen. Daher darf man im `catch`-Block nicht davon ausgehen, dass der Variablen `serverSocket` ein Wert zugewiesen wurde und diese ungleich `null` ist. Vor dem Aufruf von `close()` muss daher eine Prüfung auf `null` in den `catch`-Block eingebaut werden. Solche Spezialbehandlungen machen den Sourcecode unübersichtlich. Sowohl die bedingte Anweisung als auch die Behandlung der `IOException` erzeugen weitere Komplexität und reduzieren die Lesbarkeit. Diese »hässliche« Form der Fehlerbehandlung sieht man – bedingt durch das Java-API bei Sockets und dem Stream-I/O – leider häufiger. Darüber hinaus verstoßen die mehrfachen Aufrufe von `close()` gegen das sogenannte DRY-Prinzip. Diese Abkürzung bedeutet »Don't Repeat Yourself« und steht dafür, Duplikation von Sourcecode möglichst zu vermeiden (vgl. »Der Pragmatische Programmierer« von Andrew Hunt und David Thomas [48]).

Schritt 2: Duplikation entfernen

Die Sourcecode-Duplikation lässt sich leicht korrigieren. Im Sinne der Wiederverwendbarkeit faktoriert man eine Methode `closeServerSocket()` heraus, die aus den Anweisungen des `catch`-Blocks besteht. Die Methode ist tolerant gegenüber der Übergabe von `null`-Werten und zudem wird die in der Signatur der Methode `close()` angegebene `IOException` ignoriert, da man diese nicht sinnvoll behandeln kann.

```
private static void closeServerSocket(final ServerSocket serverSocket)
{
    if (serverSocket != null)
    {
        try
        {
            serverSocket.close();
        }
        catch (final IOException e)
        {
            // Designfehler? close() sollte keine IOException werfen! (s.o.)
        }
    }
}
```

Hilfsmethoden wie diese sind häufig nützlich und deren Einsatz macht den Sourcecode oftmals deutlich übersichtlicher, weil der Nutzcode viel kürzer und besser lesbar geschrieben werden kann. Dadurch erhalten wir folgende Methode, die den gemeinsamen Aufruf von `closeServerSocket()` hinter den `catch`-Block verlagert:

```
public static void openAndProcess()
{
    ServerSocket serverSocket = null;
```

```
    try
    {
        final int AUTO_ALLOCATE_PORT = 0;
        serverSocket = new ServerSocket( AUTO_ALLOCATE_PORT);

        handleIncomingConnections( serverSocket);

        // nach untern verlagert: closeServerSocket( serverSocket);
    }
    catch ( final IOException ex)
    {
        // nach untern verlagert: closeServerSocket( serverSocket);
    }
    // DRY: keine Duplikation
    closeServerSocket( serverSocket);
}
```

Das war bereits ein guter Schritt in die richtige Richtung: Die Methode ist nun deutlich besser lesbar und behandelt Fehler relativ gut. Wieso nur relativ gut? Wir haben noch ein »Schlupfloch« vergessen. Es existiert ein Problem, wenn `return`-Anweisungen im `try`- oder im `catch`-Block verwendet werden oder dort Exceptions ausgelöst oder weiter propagiert werden. Wird ein `try`- oder `catch`-Block auf diese Weise verlassen, so wird der dem `catch`-Block nachfolgende Sourcecode, in diesem Fall die `closeServerSocket()`-Methode, nicht ausgeführt. Dadurch kann sich das Programmverhalten in Fehlersituationen auf unbestimmte Weise verändern. Möglicherweise werden belegte Systemressourcen nicht wieder freigegeben. Betrachten wir mögliche Abhilfen.

Tipp: Apache Commons IOUtils und Google Guava

Für viele elementare, aber auch komplexere Funktionalitäten existieren Fremdbibliotheken. Google Guava und Apache Commons sind besonders erwähnenswert. In Letzterer findet sich eine große Menge nützlicher Funktionalität rund um I/O, etwa in der Klasse `IOUtils` eine für Streams, Reader, Writer und Sockets überladene Methode `closeQuietly()`, die analog zu der gezeigten Methode `closeServerSocket()` arbeitet und wie folgt genutzt werden kann:

```
import org.apache.commons.io.IOUtils;

// …

IOUtils.closeQuietly(socket);
```

Die Version 2.4 dieser Bibliothek binden wir in unser Projekt ein, indem wir deren Abhängigkeit in der Datei `build.gradle` angeben. Alternativ gilt dies analog für Google Guava in der derzeit aktuellen Version 18.0:

```
dependencies
{
    compile 'commons-io:commons-io:2.4'
    compile 'com.google.guava:guava:18.0'
}
```

Das Thema Fremdbibliotheken und Applikationsbausteine behandle ich in Kapitel 6 in größerer Tiefe und beschreibe dort insbesondere verschiedene Funktionalitäten

aus Google Guava – dabei verweise ich immer auf Ähnliches in Apache Commons.

Schritt 3: Variante a) Nutzung von `finally`

Mithilfe des Schlüsselworts `finally` kann das Exception Handling elegant und robust gestaltet werden: Laut JLS wird der `finally`-Block nämlich immer ausgeführt, und zwar unabhängig davon, ob

- der `try`-Block normal beendet,
- eine Exception im `try`-Block geworfen oder
- ein `return` im `try`- bzw. `catch`-Block ausgeführt wird.

Durch dieses Verhalten lassen sich alle angeforderten Systemressourcen freigeben – unabhängig davon, wie eine Methode beendet wird. Betrachten wir die folgende Realisierung, die abhängig vom Ergebnis des Aufrufs einer Methode `errorDetected()` per `return` die Abarbeitung der Methode beendet:

```
public static void openAndProcess()
{
   ServerSocket serverSocket = null;
   try
   {
      final int AUTO_ALLOCATE_PORT = 0;
      serverSocket = new ServerSocket( AUTO_ALLOCATE_PORT);

      handleIncomingConnections( serverSocket);

      if ( errorDetected())
         return;
   }
   catch ( final IOException e)
   {
      // Die Freigabe der Ressource erfolgt hier in finally.  Exception darf
      // nicht propagiert werden,  weil sie nicht in der Signatur definiert
ist.
   }
   finally
   {
      closeServerSocket( serverSocket);
   }
}
```

Es fällt auf, dass wir trotz der ganzen Korrekturen immer noch gezwungen sind, eine `IOException` abzufangen und einen `catch`-Block zu schreiben. Das liegt daran, dass es sich bei der `IOException` um einen speziellen Typ von Exception handelt, nämlich eine sogenannte Checked Exception: Diese muss entweder per `throw` in der Methodensignatur definiert sein oder aber durch einen `catch`-Block behandelt werden (vgl. Abschnitt 4.7.2). Des Weiteren definiert die Methode `openAndProcess()` in ihrer Signatur keine Exception, sodass wir dies aus Kompatibilitätsgründen für diese öffentliche Methode auch nicht ändern dürfen. Natürlich sollten wir aber belegte Systemressourcen freigeben. In unserem Beispiel geschieht dies durch den Methodenaufruf von `closeServerSocket()` im

finally-Block in jedem Fall.

Schritt 3: Variante b) Nutzung von ARM

Mithilfe des mit JDK 7 eingeführten ***Automatic Resource Management*** (***ARM***) können wir das Ganze eleganter und kürzer schreiben. Das ARM lässt sich für alle Typen nutzen, die das mit JDK 7 eingeführte Interface java.lang.AutoClosable implementieren und damit signalisieren, dass sie ARM-kompatibel sind. Praktischerweise müssen wir uns dann nicht mehr selbst um die Freigabe von Ressourcen kümmern. Wir nutzen hier die neue Syntax des ARM (Details finden Sie in Abschnitt 4.7.4) und schreiben die Ressourcenallokation direkt in einen try-Block wie folgt:

```
public static void openAndProcess()
{
   final int AUTO_ALLOCATE_PORT = 0;
   try (final ServerSocket serverSocket = new
ServerSocket(AUTO_ALLOCATE_PORT))
   {
      // Weitere Verarbeitung
      handleIncomingConnections(serverSocket);

      if (errorDetected())
         return;
   }
   catch (final IOException ex)
   {
      // Die Freigabe der Ressource erfolgt hier durch ARM. Die Exception darf
      // nicht propagiert werden, weil sie nicht in der Signatur definiert
ist.
   }
}
```

4.7.4 Besonderheiten beim Exception Handling mit JDK 7

Nachfolgend beschäftigen wir uns mit einigen Neuerungen aus JDK 7, die das Exception Handling vereinfachen und insbesondere auch das Resource Handling adressieren.

Multi Catch

Manchmal ist es wünschenswert, mehrere Exception-Typen gleichartig zu behandeln. Idealerweise möchte man nur einen catch-Block definieren und nicht für jeden zu verarbeitenden Exception-Typ einen.

Betrachten wir zunächst die Situation vor JDK 7 und hier zur Demonstration eine main()-Methode, die eine weitere Methode exceptionThrowingMethod() aufruft, die wiederum zwei verschiedene Exceptions in ihrer Signatur definiert. Die

klassische Umsetzung einer Behandlung sieht folgendermaßen aus:

```
public static void main(final String[] args)
{
    try
    {
        exceptionThrowingMethod();
    }
    // Gleichartige Behandlung durch zwei catch-Blöcke
    catch (final RemoteException ex)
    {
        reportException(ex);
    }
    catch (final FileNotFoundException ex)
    {
        reportException(ex);
    }
}

private static void exceptionThrowingMethod() throws RemoteException,
                                                     FileNotFoundException
{
    // …
}
```

Mehrmals einen identischen `catch`-Block anzugeben stellt eine Form der Sourcecode-Duplikation dar und führt zu einer Verletzung des DRY-Prinzips. Für dieses Beispiel sollte man die identische Behandlung nur einmalig durchführen. Mit JDK 7 ist das Fangen mehrerer Exceptions mit nur einem `catch`-Block – ***Multi Catch*** genannt – wie folgt möglich:

```
public static void main(final String[] args)
{
    try
    {
        exceptionThrowingMethod();
    }
    // Mehrfachangabe unterschiedlicher Exceptions
    catch (final RemoteException | FileNotFoundException ex)
    {
        reportException(ex);
    }
}
```

Das Verhalten von Multi Catch ist dem mehrerer hintereinander liegender `catch`-Blöcke ähnlich. Allerdings gibt es Unterschiede. Dies gilt insbesondere für die Abarbeitung bzw. die Spezialisierung und Überdeckung von Exceptions. Bei mehreren `catch`-Blöcken kann man eine explizite Sonderbehandlung für spezialisierte Typen realisieren, indem der speziellere Typ von Exception im Sourcecode zuerst wie folgt angegeben wird:

```
catch (final FileNotFoundException ex)
{
    reportFileException(ex);
}
catch (final IOException ex)
{
    reportIOException(ex);
}
```

Das ist mithilfe von Multi Catch so nicht möglich, denn es gilt, dass im Multi Catch nur Exceptions mit unterschiedlichem Basistyp erlaubt sind. Schreiben wir trotzdem einmal Folgendes:

```
// Führt zu einem Compile-Error
catch (final FileNotFoundException | IOException ex)
{
    // …
}
```

Für die gefangenen `IOException` sowie deren Subtyp `FileNotFoundException` kommt es beim Kompilieren zu folgender Fehlermeldung – egal in welcher Reihenfolge Sie die beiden Exceptions angeben: »`The exception FileNotFoundException is already caught by the alternative IOException`«. Demnach ist ein einfacher `catch (IOException ex)` -Block zu nutzen.

Automatic Resource Management (ARM)

Bis hierher haben wir gesehen, dass Aufräumarbeiten bei I/O-Operationen durch die dazu notwendigen `try-catch`-Blöcke recht umfangreich und unleserlich werden können. Mit JDK 7 wurde genau für diese Aufgaben ein automatisches Ressourcenmanagement in Java integriert. Dieses entlastet den Entwickler von manuellen Aufräumarbeiten und es hilft dabei, weniger Fehler zu machen. Betrachten wir zunächst den Aufwand der manuellen Aufräumarbeiten ohne ARM beim Einsatz eines `BufferedReader`s:

```
// I/O ohne ARM
public static void main(final String[] args)
{
    BufferedReader br = null;
    try
    {
        br = new BufferedReader(new FileReader(path));
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
    finally
    {
        // Diese manuellen Aufräumarbeiten werden durch ARM überflüssig
        try
        {
            if (br != null)
            {
                br.close();
            }
        }
        catch(final IOException ioe)
        {
            // ignore
        }
    }
```

```
}
```

Zur Aktivierung des ARM muss die Verarbeitung in einem speziellen `try`-Block und der Angabe der später freizugebenden Ressourcenvariablen erfolgen:

```
public static void main(final String[] args)
{
    // Spezielle Angabe der Ressourcenvariablen
    try (final BufferedReader br = new BufferedReader(new FileReader(path)))
    {
        return br.readLine();
    }
    catch (final IOException ex)
    {
        // handle or rethrow
    }
}
```

Für jede in den runden Klammern des `try`-Blocks angegebene Variable wird beim Verlassen des `try`-Blocks *automatisch* die Methode `close()` aufgerufen. Voraussetzung dafür ist, dass die dort definierten Referenzvariablen das Interface `java.lang.AutoCloseable` erfüllen. Die Aufrufe werden vom Compiler an den korrespondierenden Stellen in den Bytecode generiert.

Final Rethrow

Das Sprachfeature Final Rethrow ermöglicht es, mehrere Exception-Typen aus einem einzigen `catch`-Block weiter zu propagieren. Somit ist es möglich, dass verschiedene Typen von Exceptions per `catch (Exception ex)` gefangen werden, diese dann aber spezifisch weiter propagiert werden können, ohne dies explizit ausprogrammieren zu müssen. Dadurch kann der `catch`-Block übersichtlich gehalten und um gewünschte Funktionalität ergänzt werden.

Betrachten wir zum Verständnis folgende Methode `finalRethrowV1(String)`, die laut Signatur zwei verschiedene Typen von Exceptions werfen kann:

```
private void finalRethrowV1(final String fileName) throws IOException,
                                                          RemoteException
```

Nehmen wir an, wir wollten diese Methode derart erweitern, dass zur besseren Wartbarkeit eine gegebenenfalls auftretende Exception in eine Log-Datei geschrieben und anschließend weiter propagiert wird. Folgende Realisierungsidee (ohne den Einsatz von Multi Catch) wäre denkbar: Durch die Notation `catch (Exception)` werden einfach alle Typen von Exceptions gefangen, dann erfolgt ein Logging der gefangenen Exception und über `throw ex` deren Propagierung:

```
private void finalRethrowV2(final String fileName) throws IOException,
                                                          RemoteException
{
    try
    {
        finalRethrowV1(fileName);
    }
```

```
    catch (final Exception ex)
    {
        log.error("exception occurred", ex);
        throw ex; // Compile-Error vor JDK 7
    }
}
```

Tatsächlich kompiliert diese Umsetzung vor JDK 7 nicht, da durch die Anweisung `throw ex` lediglich der allgemeine Typ `Exception`, nicht aber die in der Signatur definierten Exceptions propagiert werden. Zudem ist unklar, welche Typen von Exceptions propagiert werden können. Eine (wenn auch schlechte) Idee könnte darin bestehen, den Typ `Exception` in die Signatur aufzunehmen:

```
// ACHTUNG, nicht machen, ganz schlecht
private void finalRethrowV3(final String fileName) throws IOException,
                                                          RemoteException,
                                                          Exception
```

Mit einer solchen Angabe verliert die Signatur allerdings sämtliche Aussagekraft bezüglich der von der Methode geworfenen Exceptions. Er wird lediglich deutlich, dass überhaupt Exceptions auftreten können.

Mit JDK 7 wird es möglich, mehrere Typen von Exceptions allgemein zu fangen und diese spezialisiert weiterzuleiten. Während es in früheren Versionen von JDK 7 noch notwendig war, die im `catch` angegebene Exception `final` zu definieren – woher auch der Name Final Rethrow herrührt –, kann in den aktuellen JDK-7-Versionen auf die Angabe von `final` verzichtet werden, weil der Compiler im Hinblick auf JDK 8 erweitert wurde. Die moderneren Compiler können die Finalität einer Variablen selbst erkennen und erfordern keine explizite Kennzeichnung mit `final` mehr. Man spricht in diesem Zusammenhang auch von »effectively final«.

```
private void finalRethrow(final String fileName) throws IOException,
                                                         RemoteException
{
    try
    {
        // …
    }
    // Die Angabe von final verhindert für frühe JDK 7 einen Compile-Error
    catch (final Exception ex)
    {
        log.error("exception occurred", ex);
        throw ex;
    }
}
```

4.7.5 Assertions

Nachdem wir uns nun recht intensiv mit Exceptions, vor allem im Zusammenhang mit I/O und Ressourcenmanagement, beschäftigt haben, möchte ich nachfolgend auf Assertions (Zusicherungen) eingehen. Diese dienen dazu, erwartete Zustände abzusichern. Zur Formulierung einer solchen Zusicherung wird das Schlüsselwort `assert` sowie eine boolesche Bedingung angegeben. Wird diese zu `false` evaluiert,

so wird ein `java.lang.AssertionError` ausgelöst. Das ist ein Subtyp von `java.lang.Throwable`. Demzufolge können `AssertionErrors` zwar durch einen `try-catch`-Block bearbeitet werden, allerdings ist dieses Vorgehen selten sinnvoll, da man einen Programmfehler aufdecken und nicht verschweigen möchte. Dabei sollte man zusätzlich wissen, dass die Verarbeitung von Assertions für die JVM standardmäßig deaktiviert ist und explizit aktiviert werden muss, damit die Zusicherungen tatsächlich ausgewertet werden.

Assertions am Beispiel

Nachfolgend sollen aus einem Eingabestring `versions` die zwei Versionsinformationen `majorVersion` und `minorVersion` extrahiert werden. Jeder Bestandteil der Versionsinformation muss mindestens aus einem Zeichen bestehen. Um dies zu gewährleisten, wird die Länge mithilfe von Assertions überprüft. Die aus der Eingabe korrespondierenden Teilstrings ermitteln wir mithilfe eines `StringTokenizers` und dessen Methode `nextToken()`. Um die spätere Verarbeitung und gegebenenfalls eine Umwandlung in Zahlen zu vereinfachen, werden Leerzeichen durch einen Aufruf von `trim()` abgeschnitten.

```
public static void main(final String[] args)
{
    // ACHTUNG: fehlendes Token Minor-Version
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");

    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1)
    {
        / Versionen auslesen
        final String majorVersion = tokenizer.nextToken().trim();
        final String minorVersion = tokenizer.nextToken().trim();

        // Sicherstellen, dass Tokens einen Wert enthalten
        assert !majorVersion.isEmpty();
        assert !minorVersion.isEmpty();

        System.out.println("Major: '" + majorVersion + "'");
        System.out.println("Minor: '" + minorVersion + "'");
        System.out.println("#Tokens: '" + tokenCount + "'");
    }
}
```

Listing 4.46 *Ausführbar als* **'ASSERTEXAMPLE'**

Im Beispiel enthält der zu verarbeitende Eingabewert bewusst einen Leerstring als zweite Versionsinformation, um die Funktionsweise von Assertions zu demonstrieren. Demnach sollte der String `minorVersion` in diesem speziellen Fall eine Länge von 0 besitzen und damit die Zusicherung `!minorVersion.isEmpty()` verletzen. Zwar erwartet man bei der Eingabe von "`12. `" ein Fehlschlagen der Assertion und als Folge einen Programmabbruch mit einem `AssertionError`, doch das Programm ASSERTEXAMPLE läuft ohne Fehler. Zunächst ist das möglicherweise verwunderlich, doch wenn wir auf die Konsolenausgabe schauen, wird der Grund

nachvollziehbar:

```
Major: '12'
Minor: ''
#Tokens: '2'
```

Mit den Aussagen der Einleitung im Hinterkopf erinnern wir uns daran, dass ***Assertions standardmäßig deaktiviert sind, aber explizit angeschaltet werden können.*** Demnach wurden die zwei Assertions nicht ausgeführt! Um Assertions zu aktivieren, existieren die JVM-Parameter `-ea` bzw. `-enableassertions`. Mit `-da` bzw. `-disableassertions` deaktiviert man Assertions. Starten wir das Programm mit dem Parameter `-ea`, so schlägt die Zusicherung fehl und es kommt zu folgender Konsolenausgabe:

```
Exception in thread "main" java.lang.AssertionError
```

Ein solcher `AssertionError` ist nicht aussagekräftig. Es ist sinnvoll, einen Hinweistext zur Fehlerursache anzugeben. Dies kann textuell oder durch Aufruf einer Methode mit Rückgabewert erfolgen. Beide Varianten werden im folgenden Listing gezeigt:

```
public static void main(final String[] args)
{
    // ACHTUNG: fehlendes Token Minor-Version
    final String versions = "12. ";
    final StringTokenizer tokenizer = new StringTokenizer(versions, ".");

    final int tokenCount = tokenizer.countTokens();
    if (tokenCount > 1)
    {
        final String majorVersion = tokenizer.nextToken().trim();
        final String minorVersion = tokenizer.nextToken().trim();

        // Sicherstellen, dass Tokens einen Wert enthalten
        assert !majorVersion.isEmpty() : "Major-Version must not be empty";
        assert !minorVersion.isEmpty() : buildWarnMessage("Minor-Version");

        System.out.println("Major: '" + majorVersion + "'");
        System.out.println("Minor: '" + minorVersion + "'");
        System.out.println("#Tokens: '" + tokenCount + "'");
    }
}

private static String buildWarnMessage(final String versionName)
{
    return versionName + " must not be empty!";
}
```

Listing 4.47 *Ausführbar als* **'ASSERTEXAMPLEWITHMETHODS'**

Beim Ausführen des Programms AssertExampleWithMethods erhalten wir folgende Ausgabe auf der Konsole:

```
Exception in thread "main" java.lang.AssertionError: Minor-Version must not be
      empty!
```

Tipps zum Einsatz von Assertions

Weil es möglich ist, Assertions zur Laufzeit an- und abzuschalten, kann man sie als ein »löchriges« Sicherheitsnetz auffassen. ***Dies kann problematisch sein, wenn Assertions zur Absicherung der falschen Dinge eingesetzt werden, etwa zur Sicherstellung der Programmkonsistenz bei ungültigen Eingabewerten öffentlicher Methoden.*** Aus diesen Vorbemerkungen ergeben sich folgende Hinweise zum Einsatz von Assertions.

Assertions und Eingabeparameter

Da Assertions jederzeit ohne unsere Kontrolle ein- bzw. ausgeschaltet werden können, stellen sie kein geeignetes Mittel dar, Eingabeparameter öffentlicher Methoden zu prüfen. Dies gilt im Besonderen für Eingabeparameter aus einem GUI oder der Kommandozeile. Zur Prüfung von Werten innerhalb privater Methoden ist der Einsatz von Assertions vertretbar, da hier der eigene Objektzustand eigentlich immer gesichert sein sollte, indem bereits zuvor durch Parameterprüfungen in öffentlichen Methoden Fehleingaben verhindert wurden. Nichtsdestotrotz bevorzuge ich selbst für private Methoden, mit Exceptions statt Assertions auf Fehleingaben zu reagieren, sofern dort eine Prüfung angebracht ist.

Assertions für Tests

Die JLS schlägt vor, Assertions für Situationen einzusetzen, die »niemals« auftreten können bzw. sollen, etwa in einem `switch` für den »unmöglichen« `default`-Fall, wenn alle gültigen Eingabewerte durch ein `case` abgedeckt sind. Auch hier bevorzuge ich das Auslösen einer Exception. Der Grund ist einfach: Für `switch` wird durch ein fehlendes `break` schnell ein Fall-Through und ein Fehler provoziert. Dies fällt bei deaktivierten Assertions nicht sofort als Programmfehler auf.

Assertions als semantischer Kommentar

Mit Assertions lassen sich Bedingungen in den Sourcecode einbringen, die wie ein Kommentar zu lesen sind, aber zusätzlich die Validierung gewisser Zusicherungen erlauben. Dadurch können Annahmen klarer als lediglich mit einem reinen Kommentar formuliert werden.

Vorsicht vor Seiteneffekten

Die Ausführung von Assertions sollte keinen Einfluss auf den Kontrollfluss haben. Es ist daher empfehlenswert, dass die in Assertions aufgerufenen Methoden keine ***Seiteneffekte***[22] verursachen. Nehmen wir zur Verdeutlichung an, die Variable `tokenCount` wäre nicht mehr lokal, sondern als Attribut definiert. Nachfolgend wird gezeigt, wie der Aufruf von `checkLength()` zu einem unerwarteten Seiteneffekt führt:

```
private static boolean checkLength(final String version)
```

```
{
    tokenCount = 7;   // Seiteneffekt: Attribut wird geändert
    return version.length() > 0;
}
```

4.8 Weiterführende Literatur

Es gibt viele Bücher zum Einstieg in Java. Viele beschreiben lediglich das API anhand einfacher Beispiele, ohne Fallstricke zu nennen. Für alle, die mehr wollen und Hintergründe kennenlernen möchten, empfehle ich folgende Bücher ausdrücklich:

- »**SCJP Sun Certified Programmer for Java 6 Study Guide**« von Kathy Sierra und Bert Bates [83]

  Ein sehr gelungenes Buch zum Thema SCJP-Zertifizierung zu Java 6. Auf der beiliegenden CD ist Wissenswertes zur Zertifizierung zum SCJP sowie zum SCJD vorhanden.

- »**A Programmers's Guide to Java SCJP Certification**« von Khalid A. Mughal und Rolf W. Rasmussen [67]

  Dieses Buch bietet einen etwas formaleren und noch präziseren Einstieg in die SCJP-Zertifizierung als das Buch von Sierra und Bates.

- »**The Java Programming Language**« von Ken Arnold, James Gosling und David Holmes [2]

  Ein unglaublich gutes Buch über die Sprache Java, das es schafft, die Informationen detailreich und extrem präzise darzustellen. Es geht in vielen Punkten noch weiter auf die Feinheiten von Java ein als die zuvor genannten SCJP-Bücher.

Weiterführende Informationen zum Thema NIO finden Sie in folgenden Büchern:

- »**Java NIO**« von Ron Hitchens [40]

  Dieses Buch beschreibt die Klassen des NIO-Frameworks und geht zudem genauer auf reguläre Ausdrücke sowie Character Sets ein.

- »**TCP/IP Sockets in Java**« von Kenneth L. Calvert und Michael J. Donahoo [12]

  Dieses Buch stellt die Klassen des NIO-Frameworks im Zusammenhang mit TCP/IP-Sockets zur Netzwerkprogrammierung vor.

Teil II

Bausteine stabiler Java-Applikationen

5 Das Collections-Framework

In diesem Kapitel stelle ich Ihnen das Collections-Framework vor, das wichtige Datenstrukturen wie Listen, Mengen und Schlüssel-Wert-Abbildungen zur Verfügung stellt. Die Lektüre dieses Kapitels soll Ihnen helfen, ein gutes Verständnis für die Arbeitsweise der genannten Datenstrukturen und mögliche Besonderheiten oder Nebenwirkungen ihres Einsatzes zu entwickeln.

Abschnitt 5.1 beschreibt in der Praxis relevante Datenstrukturen und zeigt kurze Nutzungsbeispiele. Auf gebräuchliche Anwendungsfälle wie Suchen, Sortieren und Filtern gehe ich in Abschnitt 5.2 ein. Diverse weitere nützliche Funktionalitäten werden durch die zwei Utility-Klassen `Collections` und `Arrays` im Package `java.util` bereitgestellt und in Abschnitt 5.3 beschrieben. Abschnitt 5.4 beschäftigt sich mit dem Zusammenspiel von Generics und Collections und zeigt, welche Dinge vor allem in Kombination mit Vererbung beachtet werden sollten. Abschließend werden in Abschnitt 5.5 einige Besonderheiten und Fallstricke in den Realisierungen des Collections-Frameworks dargestellt, deren Kenntnis dabei hilft, Fehler zu vermeiden.

Das Thema Datenstrukturen und Multithreading wird in diesem Kapitel nicht vertieft. Das gilt ebenso für Hinweise zur Optimierung durch die Wahl geeigneter Datenstrukturen für gewisse Einsatzkontexte. Auf Ersteres geht Abschnitt 7.6.1 ein. Letzteres wird in Kapitel 22 behandelt.

5.1 Datenstrukturen und Containerklasse

Häufig genutzte Datenstrukturen sind die eingangs erwähnten Listen, Mengen und Schlüssel-Wert-Abbildungen. Deren Realisierung erfolgt durch sogenannte ***Containerklassen***. Sie heißen so, weil sie dazu dienen, andere Klassen zu verwalten. Im Collections-Framework sind Containerklassen durch die Interfaces `List<E>`, `Set<E>` bzw. `Map<K, V>` aus dem Package `java.util` repräsentiert.

Bevor wir uns konkret mit Datenstrukturen beschäftigen, möchte ich explizit auf eine Besonderheit hinweisen. Nur Arrays können Elemente eines beliebigen Typs speichern – insbesondere können nur sie direkt primitive Typen wie `byte`, `int` oder `double` enthalten. Alle im Folgenden vorgestellten Containerklassen können lediglich Objektreferenzen speichern. Die Verwaltung primitiver Typen ist dort nur möglich, wenn diese in ein Wrapper-Objekt umgewandelt werden.

5.1.1 Wahl einer geeigneten Datenstruktur

Um Daten sinnvoll zu speichern und performant darauf zugreifen zu können, ist der Einsatz geeigneter Datenstrukturen wichtig. Das Collections-Framework stellt

bereits eine qualitativ und funktional hochwertige Sammlung von Containerklassen bereit. Diese lassen sich grob in zwei Ableitungshierarchien mit den Interfaces `Collection<E>` und `Map<K, V>` als Basis unterteilen. Muss für eine gegebene Aufgabenstellung eine geeignete Datenstruktur gefunden werden, so ist zunächst basierend auf den Anforderungen und dem zu lösenden Problem die grundsätzliche Entscheidung zwischen Listen und Mengen mit den Basisinterfaces `Collection<E>` sowie Schlüssel-Wert-Abbildungen mit dem Basisinterface `Map<K, V>` zu treffen. Anschließend gilt es, eine geeignete konkrete Realisierung zu finden. Im Folgenden gebe ich einige Hinweise, wo man in der Ableitungshierarchie des Collections-Frameworks »abbiegen« sollte, wenn man auf der Suche nach einer passenden Datenstruktur ist.

Wahl einer Datenstruktur basierend auf dem Interface `Collection`

Für Sammlungen von Elementen mit dem Basistyp `E` wählt man eine Implementierung des Interface `Collection<E>` und muss sich dabei zwischen Listen und Mengen entscheiden. Für Daten, die eine Reihenfolge der Speicherung erfordern und auch (mehrfach) gleiche Einträge enthalten dürfen, setzen wir Realisierungen des Interface `List<E>` ein. Möchte man dagegen doppelte Einträge automatisch verhindern, so stellt eine Realisierung des Interface `Set<E>` die geeignete Wahl dar. Abbildung 5-1 zeigt die Typhierarchie von Listen und Mengen, wobei aus Gründen der Übersichtlichkeit die generische Definition nicht gezeigt wird.

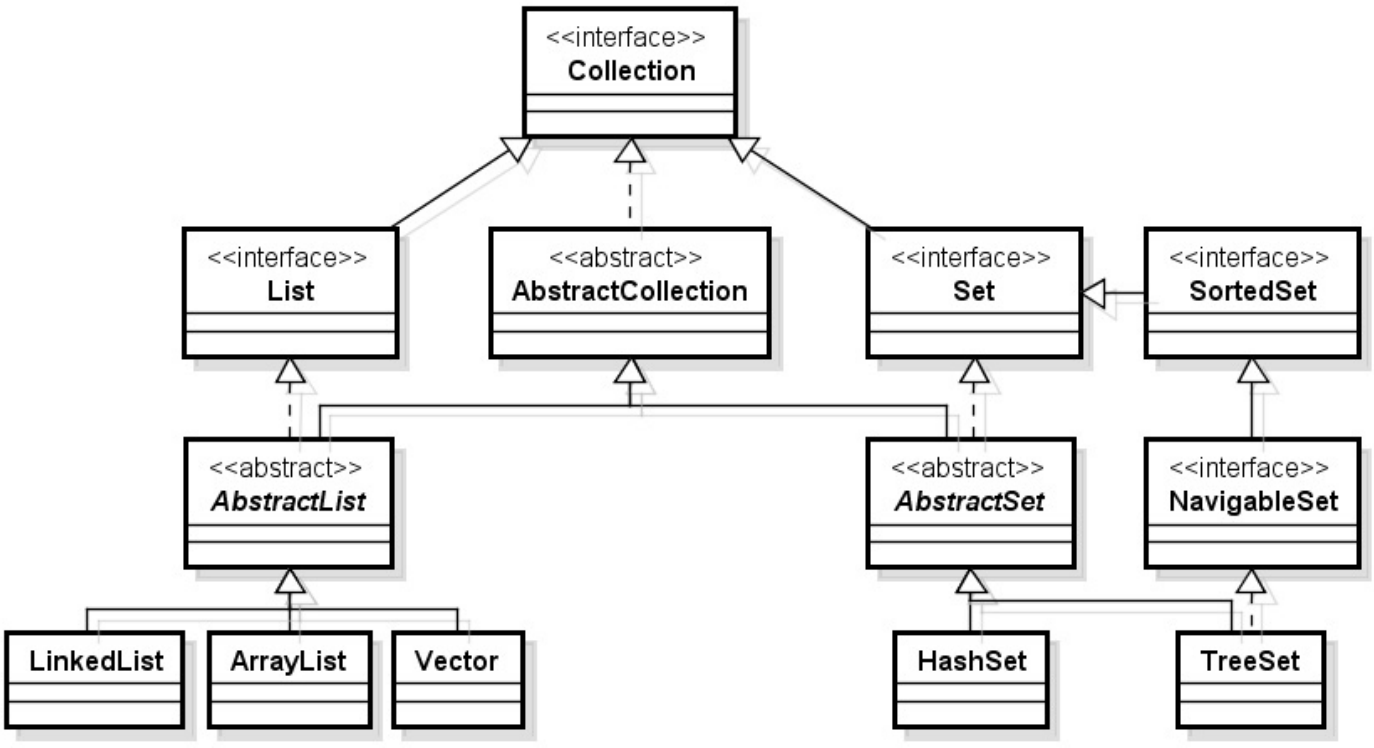


Abbildung 5-1 `Collection`*-Hierarchie*

Wahl einer Datenstruktur basierend auf dem Interface `Map`

In der Praxis findet man diverse Anwendungsfälle, in denen man Abbildungen von Objekten auf andere Objekte realisieren muss. Man spricht hier von einer Abbildung von Schlüsseln auf Werte. Dazu nutzt man sinnvollerweise das Interface `Map<K, V>`, wobei `K` dem Typ der Schlüssel und `V` demjenigen der Werte entspricht. Verschiedene Ausprägungen von Maps mit ihrer Typhierarchie, bestehend sowohl aus Klassen als auch aus weiteren, von `Map<K, V>` abgeleiteten Interfaces, zeigt Abbildung 5-2 (auch hier wieder ohne generische Typinformation).

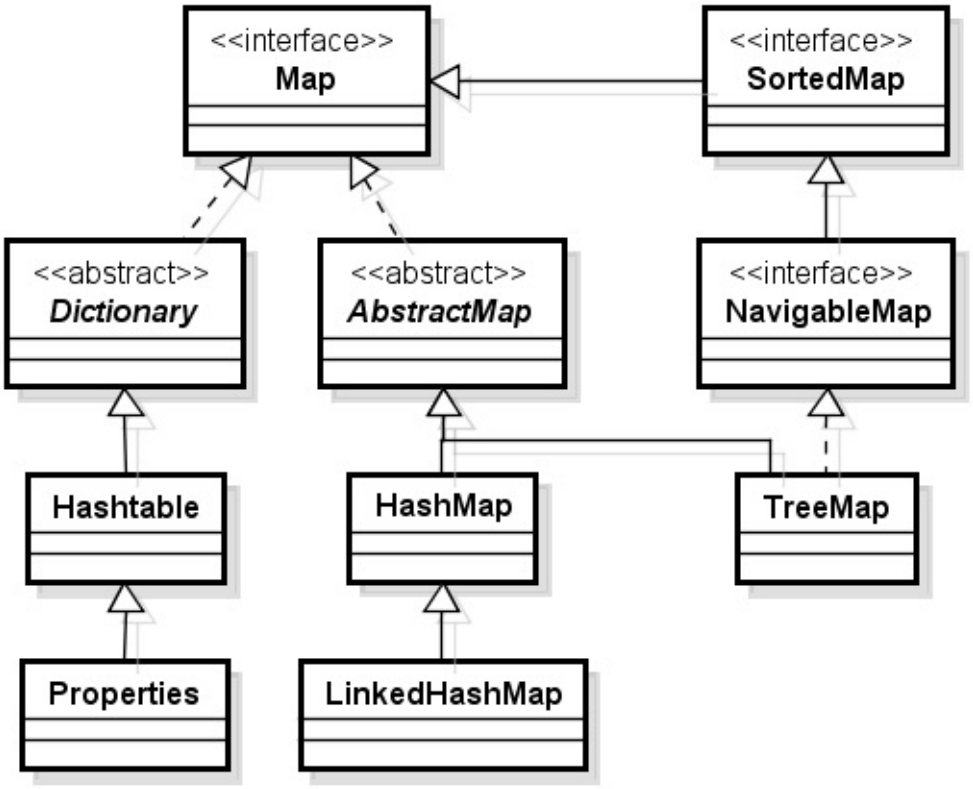


Abbildung 5-2 *`Map`-Hierarchie*

Erweiterungen in JDK 5 und 6

Mit JDK 5 und 6 wurde das Collections-Framework erweitert. Unter anderem wurden folgende Interfaces neu eingefügt:

- `Queue<E>` – Durch das Interface `Queue<E>` wird eine sogenannte Warteschlange modelliert. Diese Datenstruktur ermöglicht das Einfügen von Datensätzen am Ende und die Entnahme vom Anfang – nach dem FIFO-Prinzip (First-In-First-Out).
- `Deque<E>` – Dieses Interface definiert die Funktionalität einer doppelseitigen Queue, die Einfüge- und Löschoperationen an beiden Enden erlaubt und zudem das `Queue<E>`-Interface erfüllt.
- `NavigableSet<E>` – Dieses Interface erweitert das Interface `SortedSet<E>` um die Möglichkeit, Elemente bezüglich einer Reihenfolge zu finden. Damit ist

gemeint, dass nach Elementen gesucht werden kann, die – gemäß einem Sortierkriterium – kleiner, kleiner gleich, gleich, größer gleich oder größer als übergebene Kandidaten sind. Im Speziellen können damit für bestimmte Suchbegriffe passende Elemente gefunden werden, etwa bei Eingaben in einer Combobox zur Vervollständigung.

- `NavigableMap<K, V>` – Dieses Interface erweitert die `SortedMap<K, V>`. Es gelten die für das `NavigableSet<E>` gemachten Aussagen, wobei sich die Sortierung innerhalb der Map auf die Schlüssel und nicht auf die Werte bezieht.

Mit JDK 6 wurden spezielle Implementierungen der Interfaces `SortedSet<E>` und `SortedMap<K, V>` eingeführt, die eine Sortierung mit einem hohen Grad an Parallelität kombinieren. Dies sind die Klassen `ConcurrentSkipListSet<K, V>` und `ConcurrentSkipListMap<K, V>` aus dem Package `java.util.concurrent`.

5.1.2 Arrays

Arrays sind Datenstrukturen, die in einem zusammenhängenden Speicherbereich entweder Werte eines primitiven Datentyps oder Objektreferenzen verwalten können, nachfolgend für 100 Zahlen vom Typ `int` und zwei Namen vom Typ `String` gezeigt – im letzteren Fall wird die Kurzschreibweise und syntaktische Besonderheit der direkten Initialisierung verwendet, bei der die Größe des Arrays automatisch vom Compiler durch die Anzahl der angegebenen Elemente bestimmt wird:

```
final int[] numbers = new int[100];
final String[] names1 = new String[] { "Tim", "Micha" }; // Normalschreibweise
final String[] names2 = { "Tim", "Micha" };              // Kurzschreibweise
```

Ein Array stellt lediglich einen einfachen Datenbehälter bereit, dessen Größe zur Kompilierzeit festgelegt ist und der keinerlei Containerfunktionalität bietet, d. h., es werden weder Zugriffsmethoden angeboten noch findet eine Datenkapselung statt. Diese Funktionalität muss bei Bedarf in einer nutzenden Applikation selbst programmiert werden, die wir uns an folgendem Beispiel des Einlesens von Personendaten aus einer Datenbank verdeutlichen, wobei initial Platz für 1000 Elemente bereitgestellt wird.

```
// Initiale Größenvorgabe
Person[] persons = new Person[1000];

int index = 0;
while (morePersonsAvailableInDb())
{
    if (index == persons.length)
    {
        // Größenanpassung, siehe nachfolgenden Praxistipp
        // »Anpassungen der Größe in einer Methode kapseln«
        persons = Arrays.copyOf(persons, persons.length * 2);
    }
    person[index] = readPersonFromDb();
```

```
    index++;
}
```

Eigenschaften von Arrays

Betrachten wir mögliche Auswirkungen beim Einsatz von Arrays: Vorteilhaft ist, dass indizierte Zugriffe typsicher und maximal schnell möglich sind. Auch entsteht kein Overhead wie bei Listen, die gewisse Statusinformationen verwalten, Prüfungen vornehmen und Zugriffsmethoden auf Elemente bieten, wie es in der nachfolgenden Aufzählung angedeutet ist. Arrays eignen sich damit insbesondere dann, wenn kaum oder sogar keine Containerfunktionalität, sondern hauptsächlich ein indizierter Zugriff benötigt wird. Auch ist es nur in Arrays möglich, Werte primitiver Typen direkt zu verwalten. Auf der anderen Seite besitzen Arrays gegenüber Listen folgende Einschränkungen:

- Bei der Konstruktion eines Arrays wird zur Kompilierzeit eine fixe Größe festgelegt, die das Fassungsvermögen (auch ***Kapazität*** genannt) bestimmt. Eine sinnvolle Angabe der Kapazität ist jedoch nur dann möglich, wenn die Anzahl zu speichernder Datensätze bei der Konstruktion annähernd bekannt ist. Problematisch wird der Einsatz eines Arrays, wenn die Anzahl der zu speichernden Daten im Voraus schlecht schätzbar ist oder sich dynamisch ändern können muss, etwa bei Suchen.
- Anhand der Größe eines Arrays kann man zudem keine Aussage darüber treffen, wie viele Elemente tatsächlich gespeichert sind. Die Metainformation über den ***Füllgrad*** des Arrays, d. h. die Anzahl der dort gespeicherten Elemente, lässt sich nur aufwendig durch Iterieren über alle Einträge und durch Vergleich des gespeicherten Werts mit einem speziellen Wert, der als Indikator »kein Eintrag« dient, ermitteln. Allerdings muss auch ein solcher spezieller Wert existieren (und darf nicht Bestandteil der erlaubten Werte sein). Häufig eignet sich dazu der Wert `-1`, `0` oder `null`. Eine solche Codierung ist jedoch nicht immer möglich.
- Das Vorhalten ungenutzter Kapazität führt zu einer Verschwendung von Speicherplatz. Dies ist in der Regel für kleinere Arrays (< 1.000 Elemente) vernachlässigbar. Für große Datenstrukturen (einige 100.000 Elemente) kann sich dies aber negativ auf den belegten sowie den restlichen verfügbaren Speicher auswirken.
- Ist die gewählte Größe zu gering, so lassen sich nicht alle gewünschten Daten speichern, da keine automatische Anpassung der Größe stattfindet. Dies muss selbst programmiert werden: Im vorherigen Beispiel haben wir dazu die Methode `Arrays.copyOf()` genutzt, wodurch ein neues, größeres Array angelegt und anschließend alle Elemente des ursprünglichen Arrays in das neue kopiert werden. Dieses Vorgehen ist recht umständlich – insbesondere wenn die Größenanpassung an mehreren Stellen im Sourcecode erforderlich wird. Es bietet sich dann an, diese Funktionalität in einer Methode zu realisieren, wie dies der folgende Praxistipp »Anpassungen der Größe in einer Methode kapseln«

vorstellt.

- Ein Nachbau spezifischer Containerfunktionalität ist wenig sinnvoll und erhöht die Gefahr für Probleme, etwa durch veraltete Referenzen: Das gilt, wenn einige Programmteile Referenzen auf ein Array halten und nach einer Größenänderung und einem Kopiervorgang weiterhin mit diesen arbeiten, anstatt die neue Referenz zu verwenden. Eine Lösung ist, sämtliche Zugriffe auf das Array zu kapseln. Dann beginnt man aber mit dem Nachbau einer Datenstruktur ähnlich zu der bereits existierenden `ArrayList<E>`, was aber wenig sinnvoll ist.

Wir haben nun einige Beschränkungen von Arrays kennengelernt, die besonders dann zum Tragen kommen, wenn die Zusammensetzung der Elemente einer größeren Dynamik unterliegt. Häufig lässt sich für diese Fälle durch den Einsatz von Listen oder Mengen mit dem Basisinterface `Collection<E>`, das ich im folgenden Abschnitt vorstelle, eine vereinfachte Handhabung erreichen.

Tipp: Anpassungen der Größe in einer Methode kapseln

Nehmen wir an, wir würden die Attribute `byte[] buffer` sowie `int writePos` zur Speicherung von Eingabewerten nutzen und diese über folgende Methode `storeValue(byte)` einlesen:

```
private static void storeValue(final byte byteValue)
{
    buffer[writePos] = byteValue;
    writePos++;
}
```

Werden viele Daten gespeichert, so kann die anfangs gewählte Größe nicht ausreichend sein. Es kommt dann zu einer `java.lang.ArrayIndexOutOfBoundsException`. Diese Fehlersituation lässt sich dadurch vermeiden, dass die Größe des Arrays bei Bedarf angepasst wird. Das erfordert vor dem eigentlichen Speichern eines neuen Eingabewerts eine Prüfung, ob das Ende des Arrays erreicht ist. Wird das Ende des Arrays erkannt, so muss ein größeres Array erzeugt und die zuvor gespeicherten Werte in das neue Array kopiert werden. Dazu musste man bis einschließlich JDK 5 `System.arraycopy()` wie folgt nutzen:

```
final byte[] tmp = new byte[buffer.length + GROW_SIZE];
System.arraycopy(buffer, 0, tmp, 0, buffer.length);
buffer = tmp;
```

Glücklicherweise lässt sich dies seit JDK 6 durch den Einsatz von statischen Hilfsmethoden aus der Utility-Klasse `Arrays` einfacher implementieren. Arrays und Teilbereiche können mit den für diverse Typen überladenen, statischen Hilfsmethoden `Arrays.copyOf(T[] original, int newLength)` bzw. `Arrays.copyOfRange(T[] original, int from, int to)` kopiert werden.

In der folgenden Methode `storeValueImproved(byte)` wird zur Größenanpassung die Methode `Arrays.copyOf(byte[], int)` wie folgt

verwendet:

```
private static void storeValueImproved(final byte byteValue)
{
    if (writePos == buffer.length)
    {
        buffer = Arrays.copyOf(buffer, buffer.length + GROW_SIZE);
    }
    buffer[writePos] = byteValue;
    writePos++;
}
```

Die neuen Methoden in der Utility-Klasse `Arrays` sind praktisch: Sie erlauben, zu realisierende Aufgaben auf einer höheren Abstraktionsebene als mit `System.arraycopy()` zu beschreiben.

5.1.3 Das Interface `Collection`

Das Interface `Collection<E>` definiert die Basis für diverse Containerklassen, die das Interface `List<E>` bzw. `Set<E>` erfüllen und somit Listen bzw. Mengen repräsentieren. Wie bereits erwähnt, dienen Containerklassen dazu, mehrere Elemente zu speichern, auf diese zuzugreifen und gewisse Metainformationen (z. B. Anzahl gespeicherter Elemente) ermitteln zu können. Das Interface `Collection<E>` bietet *keinen* indizierten Zugriff, aber folgende Methoden:

- `int size()` – Ermittelt die Anzahl der in der Collection gespeicherten Elemente.
- `boolean isEmpty()` – Prüft, ob Elemente vorhanden sind.
- `boolean add(E element)` – Fügt ein Element zur Collection hinzu.
- `boolean addAll(Collection<? extends E> collection)` – Ist eine auf eine Menge bezogene, sogenannte Bulk-Operation (Massenoperation) und fügt alle übergebenen Elemente zur Collection hinzu.

Im Interface `Collection<E>` nutzen einige Methoden den Typparameter `Object` oder '?' und sind daher nicht typsicher[1]:

- `boolean remove(Object object)` – Entfernt ein Element aus der Collection.
- `boolean removeAll(Collection<?> collection)` – Entfernt mehrere Elemente aus der Collection.
- `boolean contains(Object object)` – Prüft, ob das Element in der Collection enthalten ist.
- `boolean containsAll(Collection<?> collection)` – Prüft, ob mehrere Elemente in der Collection enthalten sind.
- `boolean retainAll(Collection<?> collection)` – Behält alle Elemente einer Collection bei, die in der übergebenen Collection auch enthalten

sind.

Hinweis: Typkürzel beim Einsatz von Generics

In den obigen Methodensignaturen haben wir folgende Typkürzel verwendet:

- `E` – Steht für den Typ der Elemente des Containers.
- `?` – Steht für einen unbekannten Typ.
- `? extends E` – Steht für einen unbekannten Typ, der ein Subtyp von `E` ist.

Die Buchstaben stellen lediglich Vereinbarungen dar und können beliebig gewählt werden. Ein konsistenter Einsatz erleichtert jedoch das Verständnis. Weitere gebräuchliche und in den folgenden Abschnitten genutzte Typkürzel sind:

- `T` – Steht für einen bestimmten Typ.
- `K` bzw. `V` – Steht bei Maps für den Typ des Schlüssels (`K` => key) bzw. des Werts (`V` => value).

Mengenoperationen auf Collections

Dieser Abschnitt verdeutlicht die Arbeitsweise einiger der zuvor genannten Methoden. Mit `contains(Object)` bzw. `containsAll(Collection<?>)` kann geprüft werden, ob ein oder mehrere gewünschte Elemente in einer Collection vorhanden sind. Man kann über `containsAll(Collection<?>)` bestimmen, ob eine Collection *C* eine Teilmenge einer Collection *A* ist. Mit `removeAll(Collection<?>)` lässt sich die Differenzmenge zweier Collections berechnen, indem z. B. aus einer Collection *A* alle Elemente einer anderen Collection *B* gelöscht werden. Mit `retainAll(Collection<?>)` berechnet man die Schnittmenge: Man behält in einer Collection *A* alle Elemente einer anderen Collection *B*. Zum leichten Verständnis ist die Arbeitsweise in Abbildung 5-3 für die Collections *A*, *B* und *C* visualisiert.

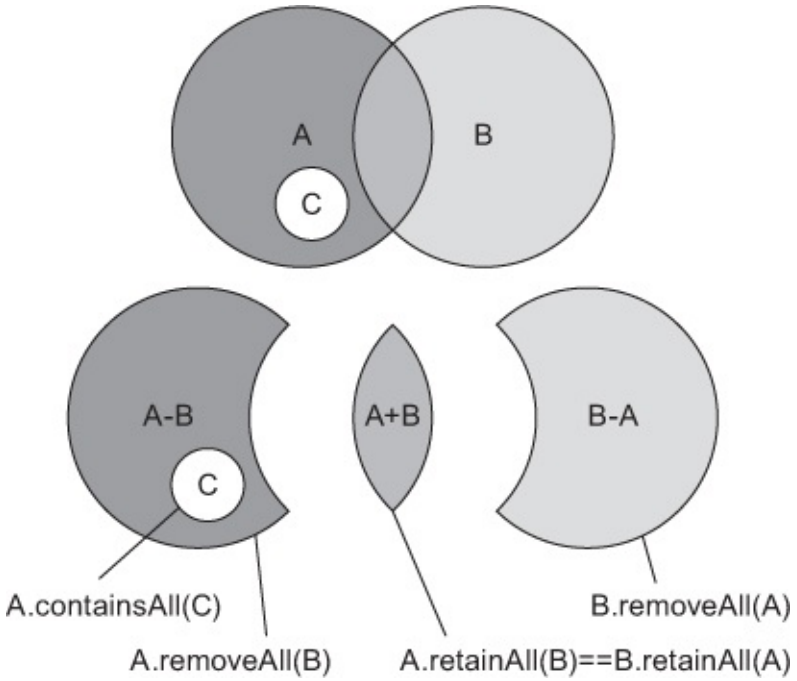


Abbildung 5-3 *Mengenoperationen auf Collections*

5.1.4 Das Interface `Iterator`

Alle Datenstrukturen, die das Interface `Collection<E>` erfüllen, bieten über die Methode `iterator()` Zugriff auf das Interface `java.util.Iterator<E>`, das einen sogenannten ***Iterator*** modelliert. Damit ist das Durchlaufen der Inhalte möglich, die in den Instanzen der Containerklassen des Collections-Frameworks gespeichert sind.

IDIOM: TRAVERSIERUNG VON COLLECTIONS MIT DEM INTERFACE `ITERATOR`

Zum Durchlaufen einer Collection mit Iteratoren definieren wir zunächst einen Datenbestand. Dabei nutzen wir den Trick, ein Array in eine Liste durch Aufruf der statischen Hilfsmethode `Arrays.asList(T...)` (vgl. Abschnitt 5.3.1) wandeln zu können. Das Ergebnis ist eine typisierte, aber insbesondere auch unmodifizierbare `List<T>`. Von dieser erhalten wir durch Aufruf von `iterator()` einen Iterator. Mit dessen Methode `hasNext()` kann man ermitteln, ob noch weitere Elemente zum Durchlaufen vorhanden sind. Ist dies der Fall, kann auf das nächste Element über die Methode `next()` zugegriffen werden. Damit ergibt sich folgendes Idiom zum Iterieren:

```
public static void main(final String[] args)
{
    final String[] textArray = { "Durchlauf", "mit", "Iterator" };
    final Collection<String> infoTexts = Arrays.asList(textArray);

    final Iterator<String> it = infoTexts.iterator();
    while (it.hasNext())
    {
```

```
        System.out.print(it.next());
        if (it.hasNext()) // Sonderbehandlung letzter Eintrag
            System.out.print(", ");
    }
}
```

Listing 5.1 *Ausführbar als* **'ITERATIONEXAMPLE'**

Wie erwartet, werden alle Elemente kommasepariert nacheinander ausgegeben:

```
Durchlauf, mit, Iterator
```

Löschfunktionalität im Interface `Iterator`

Im Interface `Iterator<E>` ist auch die parameterlose Methode `remove()` definiert, die es erlaubt, das aktuelle, d. h. das zuvor über die Methode `next()` ermittelte Element zu löschen. Allerdings muss nicht jede Realisierung eines Iterators auch tatsächlich diese Löschfunktionalität unterstützen. In diesem Fall sollte ein Aufruf von `remove()` laut JDK-Kontrakt eine `UnsupportedOperationException` auslösen. Dies gilt etwa dann, wenn es sich um eine unveränderliche Datenstruktur handelt.

Die Definition der Methode `remove()` im Interface `Iterator<E>` wirkt überflüssig, weil doch bereits im Interface `Collection<E>` eine Methode `remove(Object)` existiert. Warum diese scheinbar doppelte Definition notwendig ist, zeige ich an einem Beispiel. Nehmen wir dazu an, aus einer Liste von Stringobjekten sollen diejenigen herausgefiltert werden, die mit einer speziellen Zeichenkette beginnen. Eine intuitive Realisierung mit den zuvor vorgestellten Methoden der Interfaces `Collection<E>` und `Iterator<E>` sieht etwa folgendermaßen aus:

```
private static void removeEntriesWithPrefix(final List<String> entries,
                                            final String prefix)
{
   final Iterator<String> it = entries.iterator();
   while (it.hasNext())
   {
      final String name = it.next();
      if (name.startsWith(prefix))
      {
          // ACHTUNG: remove() der Collection ist intuitiv aber falsch
          entries.remove(name);
      }
   }
}
```

Schreiben wir ein Testprogramm, um die Funktionalität zu überprüfen. Wir definieren dazu einige Namen in einer Liste von Strings. Aus dieser sollen alle mit 'M' beginnenden Namen mit der zuvor definierten Methode gelöscht werden:

```
public static void main(final String[] args)
{
    final String[] names = { "Andreas", "Carsten", "Clemens",
                             "Merten", "Michael", "Peter" };
```

```
    final List<String> namesList = new ArrayList<>();
    namesList.addAll(Arrays.asList(names));

    removeEntriesWithPrefix(namesList, "M");
    System.out.println(namesList);
}
```

***Listing 5.2** Ausführbar als* **'ITERATORCOLLECTIONREMOVEEXAMPLE'**

Man würde erwarten, dass als Ergebnis die Namen "Andreas", "Carsten", "Clemens" und "Peter" in der Liste verbleiben und ausgegeben werden. Stattdessen kommt es zu einer `java.util.ConcurrentModificationException`:

```
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(ArrayList.java:781)
    at java.util.ArrayList$Itr.next(ArrayList.java:753)
   [...]
```

Eine derartige Exception deutet normalerweise auf Probleme und Veränderungen einer Datenstruktur bei parallelen Zugriffen durch mehrere Threads hin. Etwas merkwürdig ist das schon, weil hier lediglich ein Thread läuft. ***Es ist demnach möglich, eine auf Multithreading-Probleme hindeutende Exception selbst in einfachen Programmen ohne Nebenläufigkeit zu provozieren.*** Gehen wir der Sache auf den Grund. Verursacht wird die Exception dadurch, dass in jeder Collection ein Modifikationszähler zum Schutz vor konkurrierenden Zugriffen genutzt wird. Jeder Iterator ermittelt dessen Wert zu Beginn seiner Iteration und vergleicht diesen bei jedem Aufruf von `next()` mit dem Startwert. Weicht dieser Wert ab, so wird eine `ConcurrentModificationException` ausgelöst. Dieses Verhalten der Iteratoren nennt man ***fail-fast***.

Mit diesem Hintergrundwissen wird die Fehlerursache klar: Der Aufruf der Methode `remove(Object)` auf der Datenstruktur `entries` führt zu einer Änderung des Modifikationszählers! Die einzig sichere Art, während einer Iteration Elemente aus einer Collection zu löschen, ist durch Aufruf der Methode `remove()` aus dem Interface `Iterator<E>`. Wir korrigieren unsere Methode wie folgt:

```
private static void removeEntriesWithPrefix(final List<String> entries,
                                            final String prefix)
{
    final Iterator<String> it = entries.iterator();
    while (it.hasNext())
    {
        if (it.next().startsWith(prefix))
            it.remove();    // KORREKT: Zugriff über remove() des Iterators
    }
}
```

Man erkennt, dass die Methode `remove()` aus dem Interface `Iterator<E>` keinen Parameter benötigt. Der Grund ist einfach: Sie löscht immer das zuvor über die Methode `next()` ermittelte Element.

Anhand der geführten Diskussion ist verständlich, warum die Methode `remove()` nicht nur im Interface `Collection<E>`, sondern auch im Interface `Iterator<E>` definiert ist. Bei dessen Nutzung gibt es noch einen kleinen

Fallstrick: Vorsicht ist geboten, wenn man beispielsweise zwei aufeinander folgende Elemente löschen möchte. Intuitiv könnte man dies folgendermaßen umsetzen:

```
it.remove();
// FALSCH: it.next()-Aufruf fehlt
it.remove();
```

Das führt zu einer `IllegalStateException`, da, wie zuvor beschrieben, einem Aufruf von `remove()` immer ein Aufruf von `next()` vorangehen muss.

Achtung: Die Methode `remove()` im Interface `Iterator`

Mein Verständnis von einem Iterator basiert auf dem Entwurfsmuster ITERATOR, das ich in Abschnitt 18.3.1 beschreibe. Laut dessen Definition sollte ein Iterator (lediglich) zum Durchlaufen einer Datenstruktur genutzt werden. Modifikationen der Datenstruktur waren dabei ursprünglich nicht vorgesehen. Man könnte daher die Existenz der Methode `remove()` im Interface `Iterator<E>` kritisieren. Aufgrund der Implementierungsentscheidung für die Fail-fast-Iteratoren wurde die Aufnahme dieser Methode in das Interface `Iterator<E>` allerdings notwendig, um Löschoperationen während einer Iteration zu erlauben.

Keine Methode `add()` Mit derselben Begründung könnte man für eine Methode `add(E)` im Interface `Iterator<E>` plädieren. Diese existiert aber aus gutem Grund nicht. Sie kann nicht angeboten werden, da das Einfügen eines Elements im Gegensatz zu dessen Entfernen nicht allgemeingültig auf Basis der aktuellen Position möglich ist: Für automatisch sortierende Container entspricht beispielsweise die momentane Position des Iterators in der Regel nicht der korrekten Einfügeposition in der Datenstruktur.

5.1.5 Listen und das Interface `List`

Unter einer Liste versteht man eine geordnete Folge von Elementen. Das Collections-Framework definiert zur Beschreibung von Listen das Interface `List<E>`. Bekannte Implementierungen sind die Klassen `ArrayList<E>` und `LinkedList<E>` sowie `Vector<E>`. Das Interface `List<E>` ermöglicht einen indizierten Zugriff und erlaubt das Hinzufügen und Entfernen von Elementen – wobei es vereinzelte Ausnahmen gibt.[2]

Das Interface `List`

Das Interface `List<E>` bildet die Basis für alle Listen und bietet *zusätzlich* zu den Methoden des Interface `Collection<E>` folgende indizierte, 0-basierte Zugriffe:

- `E get(int index)` – Ermittelt das Element der Liste an der Position `index`.
- `void add(int index, E element)` – Fügt das Element `element` an der Position `index` der Liste ein.

- `E set(int index, E element)` – Ersetzt das Element an der Position `index` der Liste durch das übergebene Element `element`. Liefert das zuvor an dieser Position gespeicherte Element zurück.[3]
- `E remove(int index)` – Entfernt das Element an der Position `index` der Liste. Liefert das gelöschte Element zurück.
- `int indexOf(Object object)` und
- `int lastIndexOf(Object object)` – Mit diesen Methoden wird die Position eines gesuchten Elements zurückgeliefert. Gleichheit wird mit der Methode `equals(Object)` überprüft. Die Suche startet dabei entweder am Anfang (`indexOf(Object)`) oder am Ende der Liste (`lastIndexOf(Object)`).

Folgendes Listing zeigt einige der obigen Methoden im Einsatz. Zunächst werden einer `ArrayList<E>` verschiedene Elemente am Ende und per Positionsangabe hinzugefügt, danach wird indiziert zugegriffen. Schlussendlich werden Löschoperationen per Index ausgeführt, wobei im letzteren Fall zuvor eine Suche mit `indexOf(Object)` erfolgt:

```
public static void main(final String[] args)
{
    // Erzeugen und Hinzufügen von Elementen
    final List<String> list = new ArrayList<>();
    list.add("First");
    list.add("Last");
    list.add(1, "Middle");
    System.out.println("List:  " + list);

    // Indizierter Zugriff
    System.out.println("3rd:   " + list.get(2));

    // Vorderstes Element löschen, "Last" mit indexOf() suchen und löschen
    list.remove(0);
    list.remove(list.indexOf("Last"));
    System.out.println("List:  " + list);
}
```

Listing 5.3 *Ausführbar als* **'FIRSTLISTEXAMPLE'**

Startet man das Programm, so kommt es zu folgender Ausgabe:

```
List: [First, Middle, Last]
3rd:  Last
List: [Middle]
```

Sublisten

Die Methode `List<E> subList(int, int)` liefert einen Ausschnitt aus der Liste von Position `fromIndex` (einschließlich) bis `toIndex` (ausschließlich) und ermöglicht verschiedene Operationen auf Teillisten: Da die Rückgabe vom Typ `List<E>` ist, können tatsächlich alle Methoden des Interface `List<E>` aufgerufen werden. Man kann beispielsweise innerhalb eines gewissen Bereichs eine Suche durchführen oder diesen Bereich löschen. ***Dabei sollte man allerdings beachten, dass***

lediglich eine Sicht auf die ursprüngliche Liste geliefert wird. Dadurch kommt es bei Veränderungen an der Teilliste auch zu Änderungen in der ursprünglichen Liste.

Um das zu verdeutlichen, zeige ich die Realisierung einer Hilfsmethode `truncateListToMaxSize()`, die eine übergebene Liste auf eine gewünschte Länge zurechtstutzt, sofern diese länger ist. Als Eingabe dient hier eine Liste mit fünf Fehlertexten als Strings, die auf die Länge drei gekürzt wird:

```
public static void main(final String[] args)
{
    // Merkwürdige Aufrufkombination
    final List<String> errors = new ArrayList<>(Arrays.asList(
                                        "Error1", "Error2", "Error3",
                                        "Critical Error", "Fatal Error"));

    truncateListToMaxSize(errors, 3);


    System.out.println(errors);  // [Error1, Error2, Error3]
}

// ACHTUNG: Hier Seiteneffekt: Übergebene Liste wird gegebenenfalls
verkleinert
private static void truncateListToMaxSize(final List<?> listToTruncate,
                                          final int maxSize)
{
    if (listToTruncate.size() > maxSize)
    {
        final List<?> entriesAfterMaxSize = listToTruncate.subList(maxSize,
                                                    listToTruncate.size());
        // ACHTUNG: clear() wirkt sich auch in der Originalliste aus
        entriesAfterMaxSize.clear();
    }
}
```

Listing 5.4 *Ausführbar als* **'SUBLISTEXAMPLE'**

Hier kommt die im Listing fett markierte, etwas merkwürdig anmutende Kombination aus Konstruktoraufruf und Hilfsmethode zum Einsatz. Das ist notwendig, weil `Arrays.asList(...)` eine unmodifizierbare Liste zurückgibt, die dann durch den Konstruktoraufruf in eine neue veränderliche Liste überführt wird. Eleganter lässt sich das mithilfe einer Hilfsmethode `asModifiableList(T...)` wie folgt schreiben:

```
final List<String> errors = asModifiableList("Error1", "Error2", "Error3",
                                              "Critical Error", "Fatal Error");

private static <T> List<T> asModifiableList(T... items)
{
    return new ArrayList<>(Arrays.asList(items));
}
```

Anstatt derartige Hilfsmethoden selbst zu schreiben, sollte man besser einen Blick in existierende Bibliotheken wie Apache Commons oder Google Guava werfen. Insbesondere Letztere besprechen wir etwas genauer in Kapitel 6.

Das Interface `ListIterator`

Alle Datenstrukturen, die das Interface `List<E>` erfüllen, bieten über die Methode `listIterator()` Zugriff auf einen speziellen Iterator vom Typ `ListIterator<E>`, der auf die Besonderheiten des indizierten Zugriffs angepasst wurde. Mit einem solchen Iterator ist das Durchlaufen einer Liste zusätzlich zu einem normalen Iterator auch in Rückwärtsrichtung möglich. Dazu dienen die beiden Methoden `hasPrevious()` sowie `previous()`. Außerdem kann man den Index des nächsten bzw. des vorherigen Elements über die Methoden `nextIndex()` bzw. `previousIndex()` abfragen.

Achtung: Die Methoden `set()` und `add()` im Interface `ListIterator`

Zusätzlich zur Methode `remove()` wurden in das Interface `ListIterator<E>` mit den Methoden `set(E)` und `add(E)` zwei weitere Daten verändernde Methoden eingeführt. Gemäß der Argumentation aus dem vorherigen Praxistipp kann man deren Existenz kritisieren. Wenn man Listen während einer Iteration manipulieren will, muss man diese modifizierenden Methoden im `ListIterator<E>` allerdings anbieten. Ohne sie würde aufgrund der Fail-fast-Eigenschaft ein Aufruf etwa der Methode `add(E)` aus dem Interface `Collection<E>` eine Exception auslösen.

Arbeitsweise der Klassen `ArrayList` und `Vector`

Die Klassen `ArrayList<E>` und `Vector<E>` nutzen zur Datenspeicherung Arrays und erweitern diese um Containerfunktionalität: Wächst die Anzahl zu speichernder Elemente, so wird automatisch dafür gesorgt, dass ausreichend Speicher zur Verfügung steht. Bei Überschreiten der Kapazität des verwendeten Arrays wird automatisch ein neues, größeres Array angelegt und die Elemente des alten Arrays werden in das neue kopiert. Fortan wird das neue Array zur Speicherung der Elemente verwendet. Das alte Array ist daraufhin obsolet. Die Tatsache, dass intern mit Arrays gearbeitet wird, ist für den Benutzer transparent. Neben dieser Kapselung und der automatischen Größenanpassung, die für viele Anwendungen entscheidende Vorteile gegenüber dem Einsatz von Arrays darstellen, bieten die Klassen `ArrayList<E>` und `Vector<E>` einige weitere Annehmlichkeiten einer Containerklasse: Ein Beispiel dafür ist die Unterscheidung zwischen der Füllstandsabfrage (`size()`), also der Anzahl der momentan gespeicherten Elemente, und den zur Verfügung stehenden Speicherplätzen (Kapazität), die die tatsächliche Größe des Arrays, also das momentane Fassungsvermögen, bestimmt. Der Aufbau einer Array-basierten Liste wird in Abbildung 5-4 skizziert, wobei die Texte `Obj 1`, `Obj 2` usw. nicht das Objekt selbst, sondern die Referenz darauf repräsentieren.

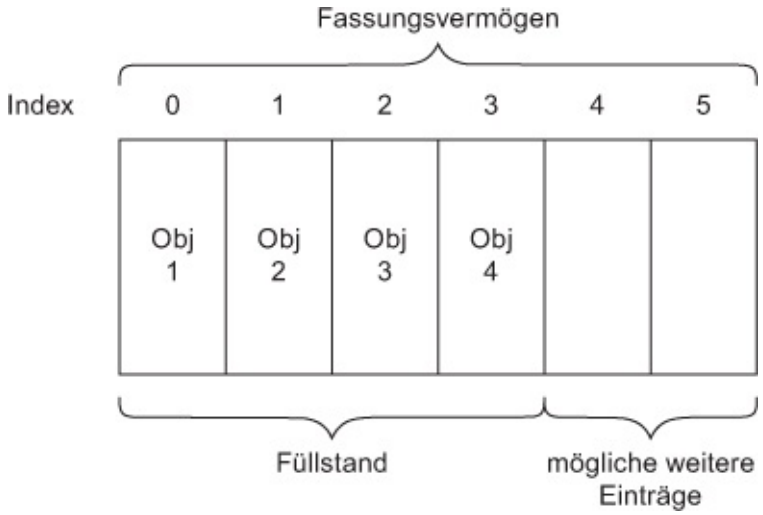


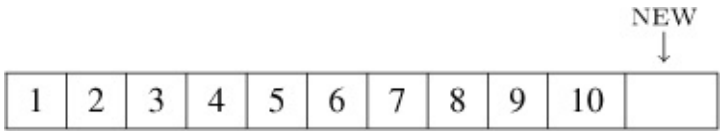
Abbildung 5-4 *Array der Klasse* `ArrayList`

Zugriff auf ein Element an Position *index* – `get( index)`

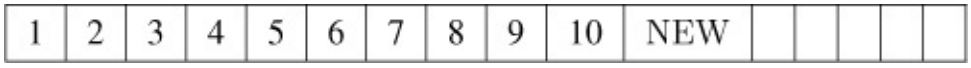
Der Zugriff auf beliebige Elemente wird durch einen Array-Zugriff realisiert und ist sehr performant.

Element an letzter Position hinzufügen – `add( element)`

Nehmen wir an, es ist (gerade) noch ausreichend Kapazität im Array vorhanden und es soll ein Element als letztes Element in die Datenstruktur eingefügt werden. In diesem Fall muss nur die Referenz in dem Array gespeichert werden:

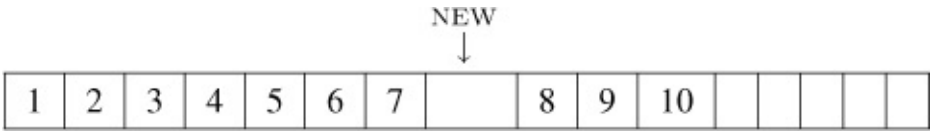


Das zugrunde liegende Array ist jetzt komplett belegt und es ist kein Platz für ein weiteres Element vorhanden. Soll erneut ein Element hinzugefügt werden, muss als Folge das Array in seiner Größe angepasst werden:



Element an Position *index* einfügen – `add( index, element)`

Wird an einer Position *index* ein Element eingefügt, so müssen als Folge alle Elemente mit einer Position >= *index* nach hinten verschoben werden, um Platz für das neue Element zu schaffen. Mit *index* = 0 muss der Inhalt des gesamten Arrays verschoben werden. Reicht die Kapazität nicht mehr aus, so wird Speicher für ein neues Array alloziert und mit dem Inhalt aus dem alten Array gefüllt. Im Folgenden ist dies für das Einfügen an Position 8 und das Element NEW gezeigt:



Element an Position *index* entfernen – `remove( index)`

Wird an einer Position *index* ein Element gelöscht, so müssen als Folge alle Elemente des Arrays mit einer Position > *index* nach vorne verschoben werden. Im Extremfall mit *index* = 0 geschieht dies für das gesamte Array.

Größenanpassungen und Speicherverbrauch

Beim Einfügen sorgen die Klassen `ArrayList<E>` und `Vector<E>` automatisch dafür, dass bei Bedarf die Größe schrittweise angepasst wird. Die Kopiervorgänge kosten mit zunehmender Größe immer mehr Zeit – aber kaum relevant. Zudem muss der Garbage Collector (vgl. Abschnitt 8.5) die obsoleten Arrays wieder wegräumen: Das ist Arbeit, die sich häufig verringern oder vermeiden lässt, indem man ***die erwartete Maximalgröße als Konstruktorparameter übergibt***. Allerdings ist die Wahl einer geeigneten Größe, wie bereits in Abschnitt 5.1.2 für Arrays erwähnt, nicht immer einfach oder gar möglich.

Bei zunehmendem Datenvolumen erhöht sich zudem die Wahrscheinlichkeit für Probleme durch die Speicherung als Array, weil immer ein zusammenhängender Speicherbereich benötigt wird. Je größer die Anzahl der Elemente wird, desto stärker wirkt sich dies aus. Zwei Probleme treten auf: Erstens kann im Extremfall eine Out-of-Memory-Situation eintreten, obwohl im Grunde noch genug Speicher vorhanden ist, aber kein zusammenhängender Speicherbereich der erforderlichen Größe bereitgestellt werden kann. Zweitens werden bei einem Vergrößerungsschritt beim Kopieren in ein neues, größeres Array temporär zwei Arrays gebraucht: einmal das alte und dann noch das neue Array. Wenn das alte Array z. B. 500 MB groß ist, dann benötigt man beim Kopieren vorübergehend ungefähr 1,25 GB.[4] Das kann ebenfalls zu einer Out-of-Memory-Situation führen, obwohl eigentlich noch genug Speicher vorhanden ist – allerdings nur für eine Version des Arrays. Besonders verwirrend ist dies, wenn man als Programmierer nicht weiß, dass im Hintergrund eine Kopie angelegt wird.

Achtung: Versteckte Memory Leaks und Abhilfemaßnahmen

Die Größe des datenspeichernden Arrays wird bei Einfügeoperationen bei Bedarf automatisch angepasst. Für das Entfernen von Elementen gilt dies allerdings nicht: Eine einmal bereitgestellte Kapazität wird dabei nicht wieder reduziert.

Wurde einmalig viel Speicher alloziert, so erzeugt man ein verstecktes ***Memory Leak***, dadurch, dass die `ArrayList<E>` bzw. der `Vector<E>` immer noch den gesamten Speicher belegt, obwohl durch Löschoperationen mittlerweile viel weniger Elemente zu speichern sind.

Mithilfe der Methode `trimToSize()` kann man in einem solchen Fall dafür sorgen, dass das Array auf die benötigte Größe verkleinert wird. Der zuvor belegte Speicher ist anschließend unreferenziert und kann vom Garbage Collector freigeräumt werden. Der Applikation steht dieser Speicher daraufhin wieder zur Verfügung.

`ArrayList` oder `Vector`?

Die Klassen `ArrayList<E>` und `Vector<E>` unterscheiden sich in ihrer Arbeitsweise lediglich in einem Detail: In der Klasse `Vector<E>` sind die Methoden `synchronized` definiert, um bei konkurrierenden Zugriffen für Konsistenz der gespeicherten Daten zu sorgen. Häufig möchte man in einer Anwendung eine Kombination mehrerer Aufrufe schützen, sodass diese feingranulare Art der Synchronisierung nicht ausreichend für die benötigte Art von Thread-Sicherheit ist (vgl. Kapitel 7). Somit gibt es eher selten Anwendungsfälle für einen `Vector<E>` und in der Regel sollte man die ***`ArrayList<E>` bevorzugen***.

Arbeitsweise der Klasse `LinkedList`

Die Klasse `LinkedList<E>` verwendet zur Speicherung von Elementen miteinander verbundene kleine Einheiten, sogenannte ***Knoten*** oder ***Nodes***. Jeder Knoten speichert sowohl eine Referenz auf die Daten als auch jeweils eine Referenz auf Vorgänger und Nachfolger. Dadurch wird eine Navigation vorwärts und rückwärts möglich. Diese Art der Speicherung führt dazu, dass die `LinkedList<E>` nur eine Größe (Anzahl der Knoten), aber keine Kapazität besitzt. Den schematischen Aufbau zeigt Abbildung 5-5, wobei `Obj 1`, `Obj 2` usw. Referenzen auf Objekte repräsentieren.

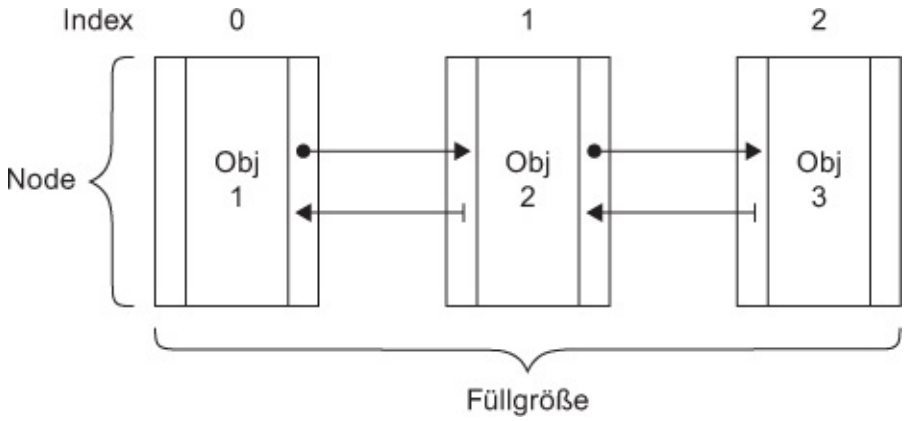


Abbildung 5-5 *Aufbau eines Objekts der Klasse* `LinkedList`

Zugriff auf ein Element an Position *index* – `get(index)`

Durch die Organisation als verkettete Liste erfordert ein indizierter Zugriff auf ein Element an einer Position *index* immer einen Durchlauf bis zu der gewünschten Stelle. Als Optimierung werden Zugriffe entweder vom Anfang oder Ende gestartet,

abhängig davon, was davon näher bei dem Zielindex liegt. Im Gegensatz zur `ArrayList<E>` mit konstanter Zugriffszeit wächst daher die Zugriffszeit bei der `LinkedList<E>` linear mit der Anzahl der gespeicherten Elemente. Dies kann bei relativ großen Datenmengen (genaue Angaben sind schwierig, in der Regel aber mehr als 500.000 Einträge) negative Auswirkungen auf die Laufzeit haben. Das zeigt sich deutlich bei der Optimierung der Darstellung umfangreicher Datenmengen mit einer `JTable` in Abschnitt 22.4.

Element an letzter Position hinzufügen – `add( element)`

Wenn ein Element hinten an letzter Position angefügt werden soll, so wird zunächst ein neuer Knoten erzeugt und danach mit dem bisher letzten Knoten verbunden.

Element an Position *index* einfügen – `add( index, element)`

Um ein Element an einer beliebiger Position einzufügen, wird ein neuer Knoten erzeugt. Zudem muss die Einfügeposition bestimmt werden, was linearen Aufwand durch eine Iteration durch die Liste bis zu der gewünschten Stelle erfordert. Schließlich sind nur einige Referenzen umzusetzen, um den neuen Knoten in die Liste einzufügen.

Element an Position *index* entfernen – `remove( index)`

Für eine Löschoperation sind lediglich Referenzanpassungen nötig, um den zu löschenden Knoten aus der Liste auszuschließen. Auch hier muss zunächst mit linearem Aufwand zur Löschposition *index* iteriert werden.

Größenanpassungen und Speicherverbrauch

Die `LinkedList<E>` besitzt bezüglich des Speicherverbrauchs einige Vorteile gegenüber der `ArrayList<E>`. Erstens wird, abgesehen von einem gewissen Overhead, immer nur genau so viel Speicher für Elemente belegt, wie tatsächlich benötigt wird. Zweitens kann durch den Garbage Collector eine automatische Freigabe des Speichers an das System erfolgen, wenn Elemente gelöscht werden. Insbesondere bei großer Dynamik und temporär hohen Datenvolumina ist dies von Vorteil, da Speicherbereiche nicht unbenutzt belegt bleiben, wie dies beim Einsatz der `ArrayList<E>` der Fall sein kann. Allerdings wird das durch einen Nachteil erkauft: Während eine `ArrayList<E>` für jedes gespeicherte Element lediglich eine Objektreferenz hält, speichert jeder Knoten einer `LinkedList<E>` zusätzlich je eine Referenz auf den Vorgänger und auf den Nachfolger. Somit benötigt eine `LinkedList<E>` zur Verwaltung insgesamt mehr Speicher (in etwa Faktor drei) als eine `ArrayList<E>` mit gleicher Anzahl gespeicherter Elemente. Beachten Sie unbedingt, dass sich dieser Speicherbedarf nur auf den Verbrauch durch die

Datenstruktur selbst bezieht und nicht auf den Speicherplatzbedarf der dort referenzierten Objekte, der in der Regel um Größenordnungen höher sein wird.

5.1.6 Mengen und das Interface `Set`

Das mathematische Konzept der Mengen besagt, dass diese keine Duplikate enthalten. In Java werden Mengen durch des Interface `Set<E>` beschrieben, das auf dem Interface `Collection<E>` basiert. Im Gegensatz zum Interface `List<E>` sind im Interface `Set<E>` keine Methoden zusätzlich zu denen des Interface `Collection<E>` vorhanden – allerdings wird ein anderes Verhalten für die Methoden `add(E)` und `addAll(Collection<? extends E>)` vorgeschrieben. Dieser Unterschied zwischen `Set<E>` und dem zugrunde liegende Interface `Collection<E>` ist nötig, um Duplikatfreiheit zu garantieren, selbst dann, wenn der Menge das gleiche Objekt mehrfach hinzugefügt wird.

Beispiel: Realisierungen von Mengen und ihre Besonderheiten

Um ein wenig vertraut mit Mengen zu werden, erzeugen wir mit einem `HashSet<E>` und einem `TreeSet<E>` zwei verschiedene Typen von Mengen und füllen jeweils eigene Instanzen davon mit Werten vom Typ `String` und auch `StringBuilder`:

```
public static void main(final String[] args)
{
    fillAndExploreHashSet();
    fillAndExploreTreeSet();
}

private static void fillAndExploreHashSet()
{
    // String definiert hashCode() und equals()
    final Set<String> hashSet = new HashSet<String>();
    addStringDemoData(hashSet);
    System.out.println(hashSet);

    // StringBuilder hat weder hashCode() noch equals()
    final Set<StringBuilder> hashSetSurprise = new HashSet<StringBuilder>();
    addStringBuilderDemoData(hashSetSurprise);
    System.out.println(hashSetSurprise);
}

private static void fillAndExploreTreeSet()
{
    // String implementiert Comparable
    final Set<String> treeSet = new TreeSet<String>();
    addStringDemoData(treeSet);
    System.out.println(treeSet);

    // StringBuilder implementiert Comparable nicht
    final Set<StringBuilder> treeSetSurprise = new TreeSet<StringBuilder>();
    addStringBuilderDemoData(treeSetSurprise);
    System.out.println(treeSetSurprise);
```

```
}

private static void addStringDemoData(final Set<String> set)
{
    set.add("Hallo");
    set.add("Welt");
    set.add("Welt");
}

private static void addStringBuilderDemoData(final Set<StringBuilder> set)
{
    set.add(new StringBuilder("Hallo"));
    set.add(new StringBuilder("Welt"));
    set.add(new StringBuilder("Welt"));
}
```

Listing 5.5 *Ausführbar als* **'FIRSTSETEXAMPLE'**

Starten wir das Programm FIRSTSETEXAMPLE, so kommt es zu folgenden Ausgaben:

```
[Hallo, Welt]
[Welt, Hallo, Welt]
[Hallo, Welt]
Exception in thread "main" java.lang.ClassCastException:
java.lang.StringBuilder
       cannot be cast to java.lang.Comparable
```

Das Beispiel zeigt, dass man zum sicheren Umgang mit den Mengen-Datenstrukturen verstehen sollte, welche Mechanismen die Eindeutigkeit von Elementen innerhalb einer Menge bewirken: Für Strings arbeitet alles wie erwartet, für den Typ `StringBuilder` werden Duplikate nicht erkannt und für `TreeSet<StringBuilder>` sogar eine Exception ausgelöst. Für zu speichernde Klassen ist eine korrekte und den jeweiligen Kontrakten folgende Implementierung einiger Methoden erforderlich. Für die Klasse `HashSet<E>` sind dies die Methoden `hashCode()` zum Auffinden von Elementen sowie `equals(Object)` zur Garantie von Eindeutigkeit. Die Klasse `TreeSet<E>` nutzt dazu die Methoden `compareTo(T)` bzw. `compare(T, T)` aus den Interfaces `Comparable<T>` bzw. `Comparator<T>`. Abschnitt 5.1.9 geht auf das Zusammenspiel der relevanten Methoden im Detail ein. In den Abschnitten 5.1.7 und 5.1.8 werden zuvor sowohl die Grundlagen von hashbasierten Containern (zum Verständnis der Klasse `HashSet<E>`) als auch die Grundlagen automatisch sortierender Container (als Basis für die Klasse `TreeSet<E>`) vorgestellt.

Bevor wir tiefer in die Details abtauchen, wollen wir zunächst einfache Beispiele für `HashSet<E>` und `TreeSet<E>` betrachten, um ein Gefühl für die Arbeit mit Mengen zu erhalten. Komplettiert wird das Ganze durch ein Praxisbeispiel: Wir erweitern die bereits realisierte Verzeichnisüberwachung und setzen dazu Mengen ein.

Fallstrick: Fehlende Angabe eines Sortierkriteriums

Zur Kompilierzeit wird für ein `TreeSet<E>` nicht geprüft, ob nur Objekte gespeichert werden, die das Interface `Comparable<T>` erfüllen. Das ist durchaus berechtigt, da auch ein `Comparator<T>` zur Beschreibung des Sortierkriteriums

dienen kann. Eine fehlende Angabe eines Sortierkriteriums macht sich daher erst zur Laufzeit beim Einfügen von Elementen durch eine `java.lang.ClassCastException` bemerkbar.

Die Klasse `HashSet`

Die Klasse `HashSet<E>` ist eine Spezialisierung der abstrakten Klasse `AbstractSet<E>` und speichert Elemente ungeordnet in einem Hashcontainer (genauer: in einer später in Abschnitt 5.1.10 vorgestellten `HashMap<K, V>`). Dadurch wird ein geringer Laufzeitbedarf für die Operationen `add(E)`, `remove(Object)`, `contains(Object)` usw. ermöglicht.

Betrachten wir ein kurzes Beispiel, in dem die Werte `1` bis `3` in absteigender Reihenfolge in ein `HashSet<Integer>` eingefügt werden:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new HashSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);    // 1, 2, 3
}
```

Listing 5.6 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE'**

Bei einem Blick auf die Ausgabe "`1, 2, 3`" scheint ein `HashSet<Integer>` die natürliche Ordnung der eingefügten Werte herzustellen. ***Dies ist jedoch nur ein zufälliger Effekt***. Dieser wird durch kleine Datenmengen und die gewählte Abbildung der zu speichernden Daten ausgelöst. Bei der Speicherung von Werten darf man sich bei einer *ungeordneten Menge*, wie sie von der Klasse `HashSet<E>` realisiert wird, *niemals* auf eine *definierte* Reihenfolge der Elemente verlassen. Dies wird deutlich, wenn man weitere Elemente einfügt, etwa die Werte `33`, `11` und `22`:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new HashSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);    // 1, 2, 3

    final Integer[] moreInts = new Integer[] { 33, 11, 22 };
    numberSet.addAll(Arrays.asList(moreInts));
    System.out.println("Add: " + numberSet);        // 1, 33, 2, 3, 22, 11
}
```

Listing 5.7 *Ausführbar als* **'HASHSETSTORAGEEXAMPLE2'**

Es kommt nun zu der zufällig wirkenden Ausgabe "`1, 33, 2, 3, 22, 11`", die durch die Verteilung im Container verursacht wird.

Benötigt man eine Ordnung der Elemente, so bietet sich der Einsatz der im Folgenden beschriebenen Klasse `TreeSet<E>` zur Speicherung von Elementen an.

Die Klasse `TreeSet`

Die Klasse `TreeSet<E>` implementiert das Interface `SortedSet<E>` und speichert Elemente sortiert. Die Sortierung wird entweder durch das Interface `Comparable<T>` oder einen explizit im Konstruktor übergebenen `Comparator<T>` festgelegt. Wir schauen auf ein ähnliches Beispiel wie für die Klasse `HashSet<E>`:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1 };
    final Set<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet); // 1, 2, 3

    final Integer[] moreInts = new Integer[] { 33, 11, 22 };
    numberSet.addAll(Arrays.asList(moreInts));
    System.out.println("Add: " + numberSet); // 1, 2, 3, 11, 22, 33
}
```

Listing 5.8 *Ausführbar als* **'TREESETSTORAGEEXAMPLE'**

Startet man das Programm TREESETSTORAGEEXAMPLE, so sieht man, dass die Elemente sortiert im Container gespeichert werden. Weil hier im Konstruktor kein `Comparator<T>` übergeben wurde, basiert die Sortierung auf `Comparable<T>`, das von der zu speichernden Klasse `Integer` erfüllt wird.

Das Interface `SortedSet<E>`

Die Klasse `TreeSet<E>` bietet neben der automatischen Sortierung folgende nützliche Funktionalität aus dem Interface `SortedSet<E>`:

- `E first()` und `E last()` – Mit diesen beiden Methoden kann das erste bzw. letzte Element der Menge ermittelt werden.
- `SortedSet<E> headSet(E toElement)` – Liefert die Teilmenge der Elemente, die kleiner als das übergebene Element `toElement` sind.
- `SortedSet<E> tailSet(E fromElement)` – Liefert die Teilmenge der Elemente, die größer oder gleich dem übergebenen Element `fromElement` sind: Ein übergebenes Element ist im Gegensatz zu `headSet(E)` in der zurückgelieferten Menge enthalten, wenn es Bestandteil der Originalmenge war.
- `SortedSet<E> subSet(E fromElement, E toElement)` – Liefert die Teilmenge der Elemente, startend von inklusive `fromElement` bis exklusive `toElement`.

Wir bauen unser Beispiel ein wenig aus und integrieren zwei Änderungsaktionen, um zu zeigen, dass es sich bei den durch die obigen Methoden gelieferten Sets jeweils um Sichten handelt, die Änderungen propagieren:

```
public static void main(final String[] args)
{
    final Integer[] ints = new Integer[] { 3, 2, 1, 33, 11, 22 };
    final SortedSet<Integer> numberSet = new TreeSet<>(Arrays.asList(ints));
    System.out.println("Initial: " + numberSet);   // 1, 2, 3, 11, 22, 33
```

```
    System.out.println("first:  " + numberSet.first());           // 1
    System.out.println("last:  " + numberSet.last());             // 33

    final SortedSet<Integer> headSet = numberSet.headSet(7);
    System.out.println("headSet:    " + headSet);                 // 1, 2, 3
    System.out.println("tailSet:    " + numberSet.tailSet(7));    // 11, 22, 33
    System.out.println("subSet:     " + numberSet.subSet(7, 23)); // 11, 22

    // Modifikationen an einem einzelnen Set
    headSet.remove(3);
    headSet.add(6);
    System.out.println("headSet:    " + headSet);   // 1, 2, 6
    System.out.println("numberSet:  " + numberSet); // 1, 2, 6, 11, 22, 33
}
```

Listing 5.9 *Ausführbar als* **'TREESETSTORAGEEXAMPLE2'**

Praxisbeispiel für den Einsatz von Mengen

In Abschnitt 4.6.1 wurde ein Verzeichnisüberwachungstool entwickelt, das lediglich eine Änderung der Anzahl in einem Verzeichnis enthaltener Dateien erkennen konnte. Die Kombination der Aktionen »Hinzufügen«, »Löschen« und »Umbenennen« von Dateien verändert die Anzahl der Elemente aber nicht zwingend. Die Klasse `DirectoryObserver` soll nun derart erweitert werden, dass Veränderungen spezifischer – in Form hinzugefügter bzw. gelöschter Dateien – ermittelt werden. Genauere Aussagen kann man nur dann treffen, wenn man den aktuellen sowie den vorherigen Verzeichnisinhalt speichert und Veränderungen bestimmt.

Im folgenden Beispiel der Klasse `DirectoryCheckerReportingChanges` nutzen wir ein `TreeSet<String>` zur Datenspeicherung des Verzeichnisinhalts. Dort werden Dateinamen als Objekte vom Typ `String` gespeichert und automatisch sortiert. Die Methode `checkForContentsChanged(int)` aus der Basisklasse wird überschrieben und in ihrer Funktionalität so erweitert, dass sie Mengenoperationen verwendet. Sie ermittelt durch Aufruf der bereits bekannten Methode `getContents()` den aktuellen Verzeichnisinhalt und wandelt diesen über den Umweg einer Liste in ein `TreeSet<String>` um. Dabei nutzen wir, dass alle konkreten Realisierungen des Interface `Collection<E>` einen speziellen Konstruktor mit einem Parameter vom Typ `Collection<E>` als Eingabe anbieten und sich dadurch in beliebige andere Realisierungen dieses Interface konvertieren lassen. Die Änderungen am Verzeichnisinhalt ermitteln wir über Mengenoperationen. Damit ergibt sich folgende Implementierung:

```
public class DirectoryCheckerReportingChanges extends DirectoryObserver
{
    private final Set<String> savedContent = new TreeSet<>();

    public DirectoryCheckerReportingChanges(final String
nameOfDirectoryToCheck)
            throws IOException
    {
        super(nameOfDirectoryToCheck);
```

```
    }

    @Override
    protected int checkForContentsChanged(final int numOfFiles)
    {
        final String[] content = FileUtils.getContents(getDirectoryToCheck());
        final List<String> contentAsList = Arrays.asList(content);
        final Set<String> contentAsSet = new TreeSet<>(contentAsList);

        // Neu = Differenzmenge Aktuell - Vorher
        final Set<String> newContent = new TreeSet<>(contentAsSet);
        newContent.removeAll(savedContent);

        // Gelöscht = Differenzmenge Vorher - Aktuell
        final Set<String> oldContent = new TreeSet<>(savedContent);
        oldContent.removeAll(contentAsSet);

        // Unverändert = Schnittmenge Vorher und Aktuell
        final Set<String> unchangedContent = new TreeSet<>(savedContent);
        unchangedContent.retainAll(contentAsSet);

        if (newContent.size() > 0 || oldContent.size() > 0)
        {
            onContentsChanged(contentAsSet.size(), savedContent.size());
            onFilesAdded(newContent);
            onFilesRemoved(oldContent);
        }

        savedContent.clear();
        savedContent.addAll(contentAsSet);

        return savedContent.size();
    }

    protected void onFilesAdded(final Set<String> newContent)
    {
        System.out.println("addedFiles = '" + newContent +"'");
    }

    protected void onFilesRemoved(final Set<String> removedContent)
    {
        System.out.println("removedFiles = '" + removedContent +"'");
    }

    public static void main(final String[] args) throws IOException
    {
         // Zugriff auf das systemspezifische tmp-Directory
         final String tmpDir = System.getProperty("java.io.tmpdir");
         new DirectoryCheckerReportingChanges(tmpDir).checkDirectory();
    }
}
```

Listing 5.10 *Ausführbar als* **'DIRECTORYCHECKERREPORTINGCHANGES'**

Wie aus dem Listing ersichtlich, kann man basierend auf dem aktuellen und dem zuletzt gespeicherten Verzeichnisinhalt mit den Methoden `removeAll(Collection<?>)` und `retainAll(Collection<?>)` problemlos folgende Mengen berechnen:

- Neu ***hinzugefügte*** Dateien bestimmt man aus der Differenz zwischen dem

aktuellen Inhalt und dem vorherigen Inhalt.

- Dateien, die ***gelöscht*** wurden, ergeben sich aus der Differenzmenge des vorherigen Verzeichnisinhalts und dem aktuellen.
- Die ***unveränderten*** Dateien kann man ermitteln, indem die Schnittmenge zwischen dem aktuellen und dem vorherigen Inhalt berechnet wird.

Nachdem so die entsprechenden Schnitt- und Differenzmengen bestimmt sind, können als Erweiterung zu der bereits bekannten Änderungsbenachrichtigung `onContentsChanged(int, int)` die zwei zusätzlichen Benachrichtigungsmethoden `onFilesAdded(Set<String>)` und `onFilesRemoved(Set<String>)` realisiert werden. Als Eingabe erhalten diese Methoden jeweils Mengen von Dateinamen. Zur Demonstration erfolgt im Programm lediglich eine Ausgabe auf der Konsole.

5.1.7 Grundlagen von hashbasierten Containern

Arrays und Listen haben einen in manchen Situationen unangenehmen Nachteil: Die Suche nach gespeicherten Daten und der Zugriff auf diese kann sehr aufwendig sein. Im Extremfall müssen alle enthaltenen Elemente betrachtet werden. Hashbasierte Container zeichnen sich dagegen dadurch aus, dass Suchen und diverse Operationen extrem performant ausgeführt werden können. Die Laufzeiten der Operationen Einfügen, Löschen und Zugriff sind von der Anzahl gespeicherter Elemente in der Regel (weitgehend) unabhängig. Allerdings erfordern hashbasierte Container einen zusätzlichen Aufwand, weil spezielle Hashwerte berechnet werden müssen, um diese Effizienz zu erreichen. Darum sind die hashbasierten Container etwas schwieriger zu verstehen als Arrays und Listen. Die im Folgenden beschriebenen Grundlagen helfen dabei, die hashbasierten Container gewinnbringend einzusetzen. Zum leichteren Einstieg beginne ich mit einer Analogie aus dem realen Leben und einer vereinfachten Darstellung der Arbeitsweise, die im Verlauf der Beschreibung immer weiter präzisiert wird.

Analogie aus dem realen Leben

Hashbasierte Container kann man sich wie riesige Schrankwände mit nummerierten Schubladen vorstellen. In diesen Schubladen ist wiederum Platz für beliebig viele Sachen. Diese speziellen Schubladen werden in der Informatik auch als ***Bucket*** (zu deutsch: Eimer) bezeichnet. Soll ein Objekt in der Schrankwand abgelegt werden, so wird diesem eine Schubladennummer zugeteilt – wobei diese von den Eigenschaften (Attributen) des Objekts abhängt, das abgelegt werden soll. Wenn man später wieder auf Objekte zugreifen möchte, kann man dies mit der zuvor zugewiesenen Nummer tun. Zum leichteren Verwalten von Dingen in einer Schrankwand können wir uns intuitiv folgende Auswirkungen klarmachen:

1. Benutzt man immer nur ein und dieselbe Schublade, so quillt diese bald über und man findet seine Sachen nur mühselig wieder: Erschwerend kommt hinzu, dass der Inhalt einer Schublade des Öfteren komplett zu durchsuchen ist.
2. Verteilt man die Sachen relativ gleichmäßig über möglichst viele Schubladen, so kann man Sachen (nahezu) ohne Suchaufwand finden – die Kenntnis der richtigen Schublade vorausgesetzt.
3. Wenn kein gezielter Zugriff auf die korrekte Schublade erfolgt, etwa weil man sich in der Schublade irrt, so muss man im Extremfall alle Schubladen durchsuchen, um die gewünschten Sachen zu finden.

Die Analogie erleichtert das Verständnis der Anforderungen an hashbasierte Container und vor allem an die Methode `hashCode()`:

1. Mithilfe der Methode `hashCode()` eines Objekts wird, vereinfacht gesagt, die Nummer für die Schublade berechnet, in der sich das Objekt befinden soll. Auch wenn es möglich und zulässig ist, dass `hashCode()` für unterschiedliche Objekte den gleichen Wert berechnet, sollte man das möglichst vermeiden. Wenn nämlich für zwei unterschiedliche Objekte derselbe Hashwert berechnet wird, so kommt es zu einer sogenannten ***Kollision***. Verschiedene Objekte werden dann im gleichen Bucket gespeichert und erfordern eine möglicherweise aufwendigere Suche innerhalb des Buckets.
2. Zu einer gleichmäßigen Verteilung von Objekten auf Buckets kommt es, wenn die `hashCode()`-Methode für verschiedene Objekte möglichst verschiedene Werte zurückgeben. Das erreicht man am einfachsten, wenn man die Attribute selbst wieder auf Zahlen abbildet und mit Primzahlfaktoren multipliziert, wie wir es später noch sehen werden.
3. Aus dem letzten Punkt der Analogie kann man schließen, dass man die Nummern nicht verlieren oder verwechseln sollte. ***Um Schwierigkeiten zu vermeiden, empfiehlt es sich, dass sich der über die Methode `hashCode()` für ein Objekt berechnete Hashwert zur Laufzeit möglichst nicht ändert.*** Wenn sich allerdings die Grundlagen zur Berechnung ändern, kann man natürlich Änderungen am Hashwert nicht vermeiden. Man sollte sich jedoch der möglicherweise entstehenden Probleme bewusst sein (vgl. folgenden Praxistipp).

Hinweis: Auswirkungen bei Änderungen im berechneten Hashwert

Wie gerade angedeutet, ist es teilweise der Fall, dass sich der für ein Objekt berechnete Hashwert ändert, weil sich der Wert zur Berechnung benutzter Attribute ändert. Das hat aber Konsequenzen, die man kennen sollte: Liefern zu unterschiedlichen Zeiten die Berechnungen des Hashwerts für ein Objekt unterschiedliche Ergebnisse, so kann das Element nicht mehr über seinen zuvor berechneten Wert im Hachcontainer gefunden werden, weil es durch die Wertänderung an der falschen Stelle gesucht wird. Darüber hinaus kann eine

Anderung im berechneten Hashwert zu der Inkonsistenz führen, dass mehrere gleiche Elemente in unterschiedlichen Buckets eingetragen werden, was ebenfalls verschiedenste andere Probleme mit sich bringt. ***Demnach ist es – wenn möglich – zu vermeiden, dass sich der berechnete Hashwert ändert.***

Realisierung in Java

Bis jetzt haben wir nicht explizit betrachtet, dass die Anzahl von Buckets beschränkt ist. Somit muss der durch `hashCode()` berechnete `int`-Wert auf die Anzahl der tatsächlich verfügbaren Buckets abgebildet werden. Die Speicherung der Buckets erfolgt als eindimensionales Array in einer sogenannten ***Hashtabelle***. Die Anzahl der dort vorhandenen Buckets wird ***Kapazität*** genannt. Jedes Bucket kann wiederum mehrere Elemente speichern. Dazu verwaltet es eine Liste, in der Elemente abgelegt werden.

Um für ein zu speicherndes Objekt die Bucket-Nummer, also den Index innerhalb der Hashtabelle, zu bestimmen, wird das in Abbildung 5-6 angedeutete Verfahren genutzt, das folgender Berechnungsabfolge entspricht:

$$Object \xrightarrow{\texttt{hashCode()}} Hashwert \xrightarrow{f(Hashwert)} Bucket\text{-}Nummer$$

Als Abbildungsfunktion *f*(*Hashwert*) zur Bestimmung der Bucket-Nummer wird von den Hashcontainern des JDKs eine Modulo-Operation angewendet: *f*(*Hashwert*) = *Hashwert* % *Kapazität*. Die vorgestellte Arbeitsweise hat gewisse Konsequenzen:

- Selbst wenn die für Objekte berechneten Hashwerte unterschiedlich sind, kann es aufgrund der Abbildungsfunktion *f* passieren, dass dieselbe Bucket-Nummer berechnet wird und es zu einer ***Kollision*** kommt: Verschiedene Objekte werden in dasselbe Bucket eingeordnet. Dort wird zur Speicherung eine Liste verwendet.

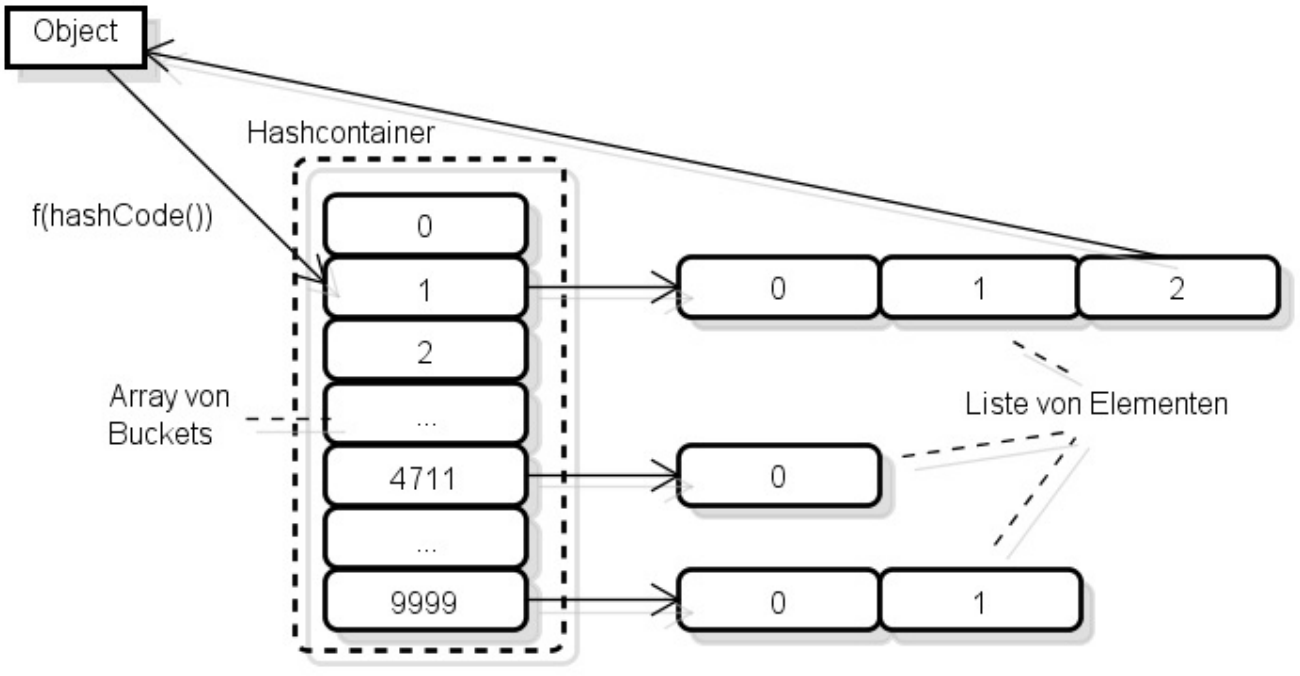


Abbildung 5-6 *Aufbau von hashbasierten Containern*

- Würden alle Objekte lediglich wenige unterschiedliche Bucket-Nummern (oder gar dieselbe) zurückliefern, so würde keine einigermaßen gleichmäßige Verteilung mehr erfolgen, sondern es käme zu einem Effekt, den man ***Clustering*** nennt. Damit bezeichnet man den Vorgang, dass in einigen Buckets sehr viele Elemente gespeichert werden und in anderen nahezu keine. Im Extremfall wird für alle Elemente die gleiche Bucket-Nummer berechnet. Der Hashcontainer würde dadurch auf eine einfache Liste reduziert werden – zusätzlich aber mit deutlichem Verwaltungsoverhead.

Ein Zugriff auf ein Element in einem Hashcontainer oder eine Suche danach erfordert durch den Hashcontainer ein zweistufiges Vorgehen:

1. Zunächst wird das Bucket bestimmt. Dazu wird die Methode `hashCode()` und die interne Abbildungsfunktion *f* des Hashcontainers benutzt.
2. Anschließend wird mit `equals(Object)` in der Liste des Buckets nach dem gewünschten Element gesucht.

Nach diesen grundsätzlichen Betrachtungen zur Arbeitsweise wollen wir uns konkret die Implikationen für Java-Klassen ansehen. Die Klasse `Object` stellt bekanntlich Defaultimplementierungen der Methoden `hashCode()` und `equals(Object)` bereit. Diese sind aber lediglich für sehr wenige Anwendungsfälle ausreichend. ***Darum sollten bei der Speicherung von Objekten eigener Klassen in hashbasierten Containern unbedingt immer deren Methoden `hashCode()` und `equals(Object)` konsistent zueinander überschrieben werden.***

Hinweis: Hashwerte in Mengen bzw. Schlüssel-Wert-Abbildungen

Der Hashwert wird mit der `hashCode()`-Methode entweder des Objekts selbst (bei Mengen) oder für Schlüssel-Wert-Abbildungen desjenigen Objekts, das als Schlüssel dient, berechnet. Damit ich beides im Anschluss nicht immer auseinanderhalten muss, beschreiben die folgenden Ausführungen zur einfacheren Darstellung den Ablauf für Mengen. Für Schlüssel-Wert-Abbildungen muss man sich abweichend davon nur gewahr sein, dass die `hashCode()`-Methode für Objekte des Typs des Schlüssels und zur späteren Suche im Bucket die `equals(Object)`-Methode derjenigen Klasse aufgerufen wird, die den Typ des Werts beschreibt.

Die Rolle von `hashCode()` beim Suchen

Konkretisieren wir die gerade gemachten Aussagen. Dazu wird das in Abschnitt 4.1.2 zur Demonstration der Methode `equals(Object)` verwendete Beispiel mit Objekten des Typs `Spielkarte` etwas abgewandelt: Statt einer Speicherung in einer `ArrayList<Spielkarte>` erfolgt diese nun in einem `HashSet<Spielkarte>`:

```
public static void main(final String[] args)
```

```
{
    final Collection<Spielkarte> spielkarten = new HashSet<>();
    spielkarten.add(new Spielkarte(Farbe.HERZ, 7));
    // PIK 8 einfügen
    spielkarten.add(new Spielkarte(Farbe.PIK, 8));
    spielkarten.add(new Spielkarte(Farbe.KARO, 9));

    // Finden wir eine PIK 8?
    final boolean gefunden = spielkarten.contains(new Spielkarte(Farbe.PIK,
8));
    System.out.println("gefunden = " + gefunden);
}
```

Listing 5.11 *Ausführbar als* **'SPIELKARTEINHASHSET'**

Wir erwarten, dass das Ergebnis einer Suche unabhängig davon ist, ob man Objekte in einem `HashSet<Spielkarte>` oder einer `ArrayList<Spielkarte>` speichert. Es stellt sich die Frage: Gefunden oder nicht gefunden? Prüfen wir den Wert der Variablen `gefunden`. Möglicherweise erleben wir dabei eine Überraschung. Die gesuchte »Pik 8« wird im Container nicht gefunden. Das ist merkwürdig, da die Methode `equals(Object)` der Klasse `Spielkarte` im oben genannten Abschnitt bereits korrekt implementiert wurde.

Ein kurzes Nachdenken bringt die Lösung: Beim Zugriff auf Hashcontainer wird zunächst durch Aufruf von `hashCode()` die Schublade berechnet, in der anschließend mit `equals(Object)` nach Objekten gesucht wird. Für die Klasse `Spielkarte` wurde die Methode `hashCode()` jedoch nicht überschrieben. Dadurch wird die Defaultimplementierung aus der Klasse `Object` ausgeführt, die typischerweise als `hashCode()` die Speicheradresse der Objektreferenz zurückliefert. Zur Suche wird aber ein neu erzeugtes Objekt verwendet, das zwar die gleiche Spielkarte darstellt, aber eine unterschiedliche Referenz besitzt. Dadurch sind die für die beiden Spielkartenobjekte berechneten Hashwerte unterschiedlich und es wird in zwei unterschiedlichen Buckets gesucht.

Wir müssen also die `hashCode()`-Methode der Klasse `Spielkarte` korrigieren. Schauen wir dazu zunächst auf den `hashCode()`-Kontrakt.

Der `hashCode()`-Kontrakt

Die Methode `hashCode()` bildet den Objektzustand (besser: den möglichst unveränderlichen Teil davon) auf eine Zahl ab und wird in der Regel dazu benötigt, Objekte in hashbasierten Containern verarbeiten zu können. Die Methode `hashCode()` ist durch die JLS mit folgender Signatur definiert:

```
public int hashCode()
```

Eine Implementierung sollte folgende Eigenschaften erfüllen:[5]

- **Eindeutigkeit** – Während der Ausführung eines Programms sollte der Aufruf der Methode `hashCode()` für ein Objekt, sofern möglich (d. h. falls sich keine relevanten Attribute ändern), denselben Wert zurückliefern.

- **Verträglichkeit mit `equals()`** – Wenn die Methode `equals(Object)` für zwei Objekte `true` zurückgibt, dann muss die Methode `hashCode()` für beide Objekte denselben Wert liefern. Umgekehrt gilt dies nicht: Bei gleichem Hashwert können zwei Objekte per `equals(Object)` verschieden sein.

Daraus können wir folgende Hinweise zur Realisierung der Methode `hashCode()` herleiten: Zur Berechnung sollten diejenigen (möglichst ***unveränderlichen***) Attribute verwendet werden, die auch in `equals(Object)` zur Bestimmung der Gleichheit genutzt werden. Dadurch werden Änderungen des Hashwerts vermieden bzw. auf nur tatsächlich benötigte Fälle eingeschränkt. Die Verträglichkeit mit `equals(Object)` ist automatisch dadurch gegeben, dass nur diejenigen Attribute zur Berechnung genutzt werden (oder auch nur ein Teil davon), die in `equals(Object)` verglichen werden.

Fallstricke bei der Implementierung von `hashCode()`

Ein typischer Fehler ist, dass die Methode `equals(Object)` überschrieben wird, die Methode `hashCode()` jedoch nicht. Dadurch wird in der Regel die Zusicherung verletzt, die besagt, dass für zwei laut `equals(Object)` gleiche Objekte auch der gleiche Wert durch `hashCode()` berechnet wird.

Auch sieht man Realisierungen von `hashCode()`, die veränderliche Attribute zur Berechnung verwenden. Das kann in einigen Fällen korrekt sein, aber manchmal Probleme bereiten.[6] Wir hatten bereits angesprochen, dass wir Objekte in Hashcontainern nicht mehr wiederfinden, wenn nach einer Änderung des Hashwerts im falschen Bucket gesucht wird. Aufgrund dessen sollte man einen kritischen Blick auf die Zusammensetzung der zur `hashCode()`-Berechnung verwendeten Attribute werfen und versuchen, bevorzugt unveränderliche Attribute zu nutzen. Damit kann man vermeiden, dass sich der berechnete Hashwert bei jeder Modifikation von Attributen des Objekts ändert.

Realisierung von `hashCode()` für die Klasse `Spielkarte`

Zur Korrektur der Klasse `Spielkarte` implementieren wir dort die Methode `hashCode()`. Dazu werfen wir einen Blick auf die Methode `equals(Object)`:

```
@Override
public boolean equals(Object other)
{
    if (other == null) // Null-Akzeptanz
        return false;
    if (this == other) // Reflexivität
        return true;
    if (this.getClass() != other.getClass()) // Typgleichheit
        return false;

    // int mit Wertevergleich, Enum mit equals()
    final Spielkarte karte = (Spielkarte) other;
    return this.wert == karte.wert && this.farbe.equals(karte.farbe);
```

```
}
```

Bei der Realisierung der Methode `hashCode()` dürfen in diesem Fall maximal die Attribute `wert` und `farbe` zur Berechnung verwendet werden. Die Verteilung auf die Buckets sollte möglichst gut gestreut werden. Das erreicht man, indem man mit Primzahlen als Multiplikatoren arbeitet: Jeder Spielkartenwert wird mit einem Primzahlfaktor multipliziert und dann der Hashwert der Farbe hinzu addiert, etwa wie folgt:

```
@Override
public int hashCode()
{
    final int PRIME = 37;
    return this.wert * PRIME + this.farbe.hashCode();
}
```

Man erreicht zwar so eine gleichmäßigen Verteilung, aber die Implementierung der Methode `hashCode()` wird doch schnell unübersichtlich und kompliziert. Als weitere Anforderung neben der gleichmäßigen Verteilung sollten die Hashfunktionen recht einfach und effizient zu berechnen sein, da sie unter Umständen sehr oft aufgerufen werden. Um sich darüber nicht allzu viele Gedanken machen zu müssen, ist der Einsatz einer passenden Utility-Klasse wünschenswert.

Entwurf einer Utility-Klasse zur Berechnung von `hashCode()`

Wir nutzen das in Abschnitt 4.2.4 gewonnene Wissen über Zahlen und Operationen und erstellen damit eine Utility-Klasse `HashUtils` mit überladenen Methoden `calcHashCode()` für primitive Datentypen und für Objektreferenzen folgendermaßen:

```
public final class HashUtils
{
    public static final int PRIME = 31;

    private HashUtils()
    {}

    public static final int calcHashCode(final int hash, final boolean input)
    {
        return PRIME * hash + (input ? 1 : 0);
    }

    public static final int calcHashCode(final int hash, final int input)
    {
        return PRIME * hash + input;
    }

    public static final int calcHashCode(final int hash, final long input)
    {
        return PRIME * hash + (int) (input ^ (input >>> 32));
    }

    public static final int calcHashCode(final int hash, final float input)
    {
        return calcHashCode(hash, Float.floatToIntBits(input));
```

```
    }

    public static final int calcHashCode(final int hash, final double input)
    {
        return calcHashCode(hash, Double.doubleToLongBits(input));
    }

    public static final int calcHashCode(final int hash, final Object input)
    {
        return (input == null) ? 0 : PRIME * hash + input.hashCode();
    }
}
```

Diese Realisierung orientiert sich an den von Joshua Bloch in seinem Buch »Effective Java« [9] entwickelten Ideen zur Implementierung der Methode `hashCode()`. Erwähnenswert ist, dass aufgrund des Rückgabetyps `int` bei der Berechnung verschiedene Konvertierungen und Wertebereichseinschränkungen durchgeführt werden müssen.

Einsatz einer Utility-Klasse

Um für die Klasse `Spielkarte` die Berechnungen in `hashCode()` zu vereinfachen, nutzen wir die zuvor erstellte Utility-Klasse `HashUtils`. Man ruft einfach die entsprechenden `calcHashCode()`-Methoden für die zu verwendenden Attribute hintereinander auf:

```
@Override
public int hashCode()
{
    int hash = 17;
    hash = HashUtils.calcHashCode(hash, farbe);
    hash = HashUtils.calcHashCode(hash, wert);
    return hash;
}
```

Vereinfachung mit JDK 7

In der mit Java 7 eingeführten Klasse `Objects` enthält das JDK verschiedene nützliche Funktionalitäten. Hier bietet sich der Einsatz der Methode `hash()` an, um die Berechnung einfach und übersichtlich zu schreiben:

```
@Override
public int hashCode()
{
   return Objects.hash(this.wert, this.farbe);
}
```

Realisierung von `hashCode()` für die Klasse `Person`

Mit dem bis hierher erlangten Wissen können wir nun auch die Klasse `Person` um eine passende, gut verständliche Realisierung von `hashCode()` erweitern:

```
@Override
public int hashCode()
```

```
{
    return Objects.hash(this.name, this.birthday, this.city);
}
```

Füllgrad (Load Factor)

Wir besitzen nun das Wissen, um `hashCode()` so zu implementieren, dass Kollisionen weitestgehend vermieden werden, indem eine möglichst gleichmäßige Verteilung der zu speichernden Elemente erfolgt. Das hängt einerseits von der gewählten Hashfunktion sowie andererseits von der Anzahl verfügbarer Buckets und gespeicherter Elemente ab: Werden fortlaufend immer mehr Elemente in einem hashbasierten Container gespeichert, so wächst die Wahrscheinlichkeit für und die Anzahl von Kollisionen. Ähnlich wie die Klasse `ArrayList<E>` führen auch Hashcontainer eine automatische Größenanpassung der Hashtabelle, d. h. eine Erweiterung um Buckets, durch, wenn die Anzahl gespeicherter Elemente einen Grenzwert übersteigt. Um zu bestimmen, ob eine Größenanpassung nötig ist, betrachtet man nicht den Inhalt aller Buckets im Einzelnen, sondern nutzt eine einfachere, aber effektive Variante: Dabei hilft als Kenngröße der sogenannte ***Füllgrad***, auch ***Load Factor*** genannt, der sich aus dem Quotienten der Anzahl gespeicherter Elemente und der Anzahl der Buckets ergibt. Dieser Wert beschreibt, wie voll der Hashcontainer werden darf, bis es zu einer Größenanpassung kommt. Bis zu einem Füllgrad von 75% sind Kollisionen erfahrungsgemäß relativ unwahrscheinlich (vgl. Javadoc-Dokumentation der Klassen `Hashtable<K, V>` und `HashMap<K, V>`).

Die Hashtabelle wird erweitert, wenn folgende Bedingung zwischen Füllgrad, Anzahl gespeicherter Elemente und der Kapazität der Hashtabelle erfüllt ist:

$$\textit{maximaler Füllgrad} * \textit{Kapazität} \geq \textit{Anzahl Elemente}$$

Wenn man zu dem Zeitpunkt, an dem die Hashtabelle angelegt wird, die ungefähre Anzahl der später zu speichernden Elemente kennt, kann man nachträgliche Größenanpassungen oftmals vermeiden, indem man die initiale Kapazität als Quotient aus der Anzahl der Elemente und dem maximal akzeptierten Füllgrad passend wählt:

$$\textit{initiale Kapazität} = \textit{Anzahl Elemente}/\textit{maximaler Füllgrad}$$

Für 1.000 Elemente ergibt sich bei einem maximalen Füllgrad von 75% die initiale Kapazität wie folgt: *initiale Kapazität* = 1.000/0,75 ≈ 1.333. Bei der Konstruktion eines Hashcontainers kann man den berechneten Wert der initialen Kapazität angeben, wobei dieser rund 1,3-mal größer als die geplante Anzahl der zu verwaltenden Elemente sein sollte. Bei einem angenommenen idealen Füllfaktor von 75% bedeutet dies, dass immer etwa 25% Kapazität unbelegt bleiben.

Der maximal erlaubte Füllgrad stellt damit eine Stellschraube von Hashcontainern dar, die sich auf Speicherplatz und Zugriffszeit auswirkt. Je geringer der Wert des maximal erlaubten Füllgrads, desto geringer ist die Wahrscheinlichkeit für

Kollisionen. Damit ist der Zugriff schneller, als wenn es Kollisionen gibt. Allerdings geht dies zulasten des benötigten Speichers und erhöht den Anteil unbenutzter Buckets. Umgekehrt »verschwendet« ein maximal erlaubter Füllgrad größer als 75% zwar weniger Speicher, jedoch steigt die Wahrscheinlichkeit für Kollisionen. Dadurch verschlechtern sich die Zugriffszeiten auf die Elemente.

Auswirkungen von Größenanpassungen

Werden mehr Elemente in einem Hashcontainer gespeichert als zunächst erwartet, und übersteigt der momentane Füllgrad die durch den maximal erlaubten Füllgrad angegebene Schwelle, so wird die Hashtabelle automatisch vergrößert. Es stehen daraufhin mehr Buckets zur Verfügung. Im Gegensatz zu Array-basierten Listen, die neue Daten nach einem solchen Vergrößerungsschritt einfach am Ende anfügen können, ist der Sachverhalt für hashbasierte Container komplizierter. ***Nach einer erfolgten Größenanpassung muss die Hashtabelle vollständig neu organisiert werden***, da die Abbildungsfunktion für zuvor gespeicherte Werte nicht mehr korrekt arbeitet: *Die Modulo-Operation liefert nun in der Regel andere Werte als zuvor*. Für jedes Element in der Hashtabelle muss das entsprechende aufnehmende Bucket neu ermittelt werden. Diesen Umsortierungsvorgang nennt man ***Rehashing***. Da jedes gespeicherte Element betrachtet werden muss, ist dieser Vorgang relativ aufwendig. Als Optimierung wird zu jedem Element dessen zuvor über die Methode `hashCode()` berechnete Wert zwischengespeichert. Dieser ändert sich bei einem Rehashing nicht. Dadurch werden zusätzliche Performance-Einbußen durch die Neuberechnung der Hashwerte durch Aufrufe von `hashCode()` vermieden. Das neue, aufnehmende Bucket kann auf Basis des zwischengespeicherten Hashwerts bestimmt werden. Das Rehashing kostet Rechenzeit, macht spätere Zugriffe aber wieder performanter, weil dadurch weniger Kollisionen auftreten.

Neben dem Rehashing gibt es Folgendes zu bedenken: Falls in einem Hashcontainer irgendwann einmal sehr viele Elemente gespeichert wurden, so kam es höchstwahrscheinlich zu einigen Vergrößerungsschritten und Rehashing-Vorgängen. Werden später viele Elementen gelöscht, wird die Größe der Hashtabelle nicht automatisch verkleinert und der Speicher bleibt (unnütz) belegt. Schlimmer noch: ***Im Gegensatz zu Listen gibt es für die Hashcontainer kein Pendant zu*** **`trimToSize()`**, ***das es nach einer mittlerweile hinfälligen Expansion erlaubt, den Speicherverbrauch zu beschneiden***. Als Abhilfe kann man einen neuen Hashcontainer mit passend gewählter Größe anlegen, der mit dem Inhalt des bisherigen gefüllt wird.

Um wieder das Beispiel einer Schrankwand zu bemühen: Analog zu den Vergrößerungen wird diese um einen Anbausatz und damit weiteren Stauraum ergänzt, wenn der Platz eng wird. Ein Umräumvorgang sorgt für eine bessere Verteilung der Sachen auch auf die neuen Schubladen und erleichtert eine spätere Suche, da wieder mehr Ordnung herrscht und in jeder Schublade weniger Dinge gelagert sind. Bezogen auf die Speicherverschwendung gilt in etwa folgende Analogie: Nach einem Frühjahrsputz sind beispielsweise mehr als die Hälfte aller Schubladen

des Schranks leer. Der Anbausatz wird aber nicht abmontiert, sondern nimmt dann einfach nur noch Platz weg.

5.1.8 Grundlagen automatisch sortierender Container

Für einige Anwendungsfälle ist es praktisch, wenn die in einer Containerklasse verwalteten Daten sortiert vorliegen. Bekanntermaßen gibt es die Containerklassen `TreeSet<E>` bzw. `TreeMap<K, V>`, die automatisch die Sortierung von Elementen ohne weiteren Implementierungsaufwand im Applikationscode herstellen. Für Arrays und Listen gibt es so etwas im JDK nicht. Um diese sortiert zu halten, wird ein manueller Schritt notwendig. Hierbei unterstützen die Methoden `sort()` aus den Utility-Klassen `Arrays` und `Collections` aus dem Package `java.util` (vgl. Abschnitt 5.2.2).

Aber unabhängig von automatischer oder manueller Sortierung muss immer eine Ordnung festgelegt werden, um bei Vergleichen von Objekten »kleiner« bzw. »größer« oder »gleich« ausdrücken zu können. Das kann man mithilfe von Implementierungen der Interfaces `Comparable<T>` und `Comparator<T>` beschreiben:

- **Natürliche Ordnung und `Comparable`** – Sofern zu speichernde Objekte das Interface `Comparable<T>` erfüllen, können sie darüber ihre Ordnung, d. h. ihre Reihenfolge untereinander, beschreiben. Diese Reihenfolge wird auch als ***natürliche Ordnung*** bezeichnet, da sie durch die Objekte selbst bestimmt wird.[7]
- **Weitere Ordnungen und `Comparator`** – Teilweise benötigt man zusätzlich zur natürlichen Ordnung weitere oder alternative Sortierungen, etwa wenn man Personen nicht nach Nachname, sondern alternativ nach Vorname und Geburtsdatum ordnen möchte. Diese ergänzenden Sortierungen können mithilfe von Implementierungen des Interface `Comparator<T>` festgelegt werden. Dadurch lassen sich von der natürlichen Ordnung abweichende Sortierungen für Objekte einer Klasse realisieren und auch Klassen sortieren, für die keine natürliche Ordnung definiert ist, weil sie das Interface `Comparable<T>` nicht implementieren.

Sortierungen und das Interface `Comparable`

Oftmals besitzen Werte oder Objekte eine natürliche Ordnung: Das gilt etwa für Zahlen und Strings. Für komplexe Typen ist die Aussage »kleiner« bzw. »größer« nicht immer sofort ersichtlich, lässt sich aber selbst definieren.

Dazu erlaubt das Interface `Comparable<T>` typsichere Vergleiche und deklariert die Methode `compareTo(T)` folgendermaßen:

```
public interface Comparable<T>
{
    public int compareTo(T o);
```

```
}
```

Das Vorzeichen des Rückgabewerts bestimmt die Reihenfolge der Sortierung:

- = 0: Der Wert 0 bedeutet Gleichheit des aktuellen und des übergebenen Objekts.
- < 0: Das aktuelle Objekt ist kleiner als das übergebene Objekt.
- > 0: Das aktuelle Objekt ist größer als das übergebene Objekt.

Diverse Klassen im JDK (alle Wrapper-Klassen, `String`, `Date` usw.) implementieren das Interface `Comparable<T>` und sind damit automatisch sortierbar.

Implementieren von `compareTo()` in eigenen Klassen

Wie man das Interface `Comparable<T>` für eigene Klassen implementiert, zeige ich nun für die folgende Klasse `Person`. Dort wird anstatt des Geburtsdatums als `Date`-Objekt das Alter bewusst als primitiver Typ gespeichert, um einige Varianten bei der Realisierung des Interface `Comparable<Person>` zu verdeutlichen:

```
public final class Person implements Comparable<Person>
{
    private final String name;
    private final String city;
    private final int    age;

    public Person(final String name, final String city, final int age)
    {
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.city = Objects.requireNonNull(city, "city must not be null");
        this.age = age;
    }
```

Eine erste Implementierung von `compareTo(Person)` könnte die natürliche Ordnung für `Person`-Objekten (eher leicht unintuitiv) ausschließlich über deren Namen realisieren:

```
@Override
public int compareTo(final Person otherPerson)
{
    Objects.requireNonNull(otherPerson, "otherPerson must not be null");

    return getName().compareTo(otherPerson.getName());
}
```

Wir benötigen eine `null`-Prüfung lediglich für den Methodenparameter, nicht aber für die Attribute, da wir für diese im Konstruktor `null`-Werte ausgeschlossen haben. Zudem nutzen wir, dass die Klasse `String` das Interface `Comparable<String>` erfüllt.

Betrachten wir den Einsatz unserer Methode `compareTo(Person)` und nehmen dazu an, eine Kundenliste `customers` enthielte etwa folgende Einträge:

```
customers.add(new Person("Müller", "Bremen", 27));
customers.add(new Person("Müller", "Kiel", 37));
```

Nutzen wir die obige Umsetzung, so werden laut `compareTo(Person)` alle Objekte vom Typ `Person` mit gleichem Namen als gleich angesehen. Dass dies keine wirklich gelungene Realisierung einer natürlichen Ordnung für Personen darstellt, wird nach kurzer Überlegung offensichtlich: Herr Müller aus Kiel ist nicht Herr Müller aus Bremen. Wie geht es also besser? Einen guten Anhaltspunkt zur Verbesserung stellt oftmals die Methode `equals(Object)` bzw. die dort zum Vergleich verwendeten Attribute dar. Nachfolgend werden hier neben dem Namen zusätzlich die Attribute `city` und `age` zur Gleichheitsprüfung herangezogen:

```
public boolean equals(final Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    final Person other = (Person) obj;
    return name.equals(other.name) && city.equals(other.city) &&
           age == other.age;
}
```

Vorgehen zur Implementierung von `compareTo()`

Im Allgemeinen kann man sich beim Implementieren von `compareTo(T)` an einer bestehenden Realisierung der Methode `equals(Object)` der jeweiligen Klasse orientieren. Alle dort verglichenen Attribute sind in der Regel auch für die Sortierung gemäß der natürlichen Ordnung relevant.[8] Diese Attribute werden wie folgt verglichen:

1. Referenztypen, die das Interface `Comparable<T>` implementieren, verwenden deren `compareTo(T)` -Methoden.
2. Für primitive Datentypen lassen sich die Vergleichsoperatoren '<','=' und '>' einsetzen, nachfolgend für einen Vergleich des Attributs `age`:[9]

   ```
   int result = 0;
   if (this.getAge() < otherPerson.getAge())
   {
       result = -1;
   }
   if (this.getAge() > otherPerson.getAge())
   {
       result = 1;
   }
   ```

   Statt die drei Fälle größer, kleiner und gleich selbst abzufragen und den passenden Rückgabewert bereitzustellen, ist es sinnvoller, die Methoden `compare()` der jeweiligen Wrapper-Klasse zu nutzen, weil diese einem Arbeit abnehmen und der Vergleich wie folgt kürzer und klarer notiert werden kann:

   ```
   int result = Integer.compare(this.getAge(), otherPerson.getAge());
   ```

3. Für alle Attribute anderen Typs muss der Vergleich selbst implementiert werden. Wenn man die Klasse des Attributs im Zugriff hat, kann man diese derart erweitern, dass sie das Interface `Comparable<T>` erfüllt und in der Realisierung die gewünschten Attribute vergleicht. Hat man eine Klasse jedoch nicht im Zugriff oder soll/darf diese nicht verändert werden, so muss der Vergleich der relevanten Attribute dieser Klasse gemäß der Schritte 1 und 2 selbst programmiert werden.

Konsistenz von `compareTo()` und `equals()`

Die Methoden `compareTo(T)` und `equals(Object)` sollten so implementiert werden, dass `x.compareTo(y)` genau dann den Wert 0 zurückgibt, wenn der Vergleich `x.equals(y)` den Wert `true` liefert. Wird gegen diese Regel verstoßen, so empfiehlt es sich, dies im Javadoc zu vermerken.

Für unser Beispiel ist die Forderung nicht eingehalten, weil `compareTo(Person)` schwächer prüft als `equals(Object)`. Um nicht für Verwirrung beim Einsatz zu sorgen, korrigieren wir die Implementierung dahingehend, dass `compareTo(Person)` auch die Attribute `city` und `age` beim Vergleich heranzieht:

```
@Override
public int compareTo(final Person otherPerson)
{
    Objects.requireNonNull(otherPerson, "otherPerson must not be null");

    int ret = getName().compareTo(otherPerson.getName());
    if (ret == 0)
    {
        ret = getCity().compareTo(otherPerson.getCity());
    }
    if (ret == 0)
    {
        ret = Integer.compare(getAge(), otherPerson.getAge()));
    }
    return ret;
}
```

Falls man in den Methoden `compareTo(T)` und `equals(Object)` dieselben Attribute nutzt, lässt sich Sourcecode-Duplikation vermeiden, indem man in `equals(Object)` die Methode `compareTo(T)` aufruft:

```
@Override
public boolean equals(final Object obj)
{
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    final Person other = (Person) obj;
    return compareTo(other) == 0;         // Vergleich mittels
compareTo(Person)
}
```

Diese Art der Realisierung vermeidet einerseits Konsistenzprobleme zwischen beiden Methoden und führt andererseits dazu, dass die Vergleichslogik nur einmal in der Methode `compareTo(T)` realisiert wird. Dies ist wiederum hilfreich, wenn Änderungen an der Klasse erfolgen, etwa Attribute hinzugefügt werden. Schnell wird in einem solchen Fall übersehen, beide Implementierungen anzupassen. Auch hier erkennt man den Vorteil, eine Funktionalität möglichst nur einmal zu realisieren. Andrew Hunt und David Thomas beschreiben dies in ihrem Buch »Der Pragmatische Programmierer« [48] als das sogenannte DRY-Prinzip (**D**on't **R**epeat **Y**ourself).

Entwicklung: `compareTo(T)` basierend auf `equals(Object)`

Häufig entwickelt man bei einer Neuimplementierung einer Klasse zunächst eine `equals(Object)`-Methode. Wird im Verlauf der Entwicklung eine natürliche Ordnung durch Erfüllen des Interface `Comparable<T>` erforderlich, so bietet es sich oftmals an, `equals(Object)` durch Aufruf von `compareTo(T)` zu realisieren und die Vergleichslogik in `compareTo(T)` zu verlagern.

Sortierungen und das Interface `Comparator`

Wir haben zur Beschreibung der natürlichen Ordnung das Interface `Comparable<T>` kennengelernt. Darüber lässt sich lediglich *eine* spezielle Sortierung beschreiben. In vielen Anwendungsfällen sind weitere Sortierungen wünschenswert, z. B. möchte man in Tabellen häufig nach jeder beliebigen Spalte sortieren können. Dies wird durch den Einsatz der im Folgenden beschriebenen ***Komparatoren*** möglich. Der Vorteil dieses Vorgehens ist, dass man Anwendungsklassen nicht mit Sortierfunktionalität überfrachtet, sondern diese in ***eigenständigen Vergleichsklassen*** definiert wird. Dazu müssen diese das Interface `Comparator<T>` erfüllen und die gewünschte Sortierung realisieren. Als Hinweis sei angemerkt, dass die dafür benötigten Attribute bzw. deren Zugriffsmethoden in ihrer Sichtbarkeit möglicherweise eingeschränkt und im `Comparator<T>` nicht zugreifbar sind. Mit `Comparable<T>` hat man immer Zugriff auf alle Attribute.

Das Interface `Comparator`

Das Interface `Comparator<T>` beschreibt einen Baustein zum Vergleich von Objekten des Typs `T`. Hierfür wird die Methode `int compare(T, T)` angeboten, die dazu dient, zwei beliebige Objekte des Typs `T` miteinander zu vergleichen:

```
public interface Comparator<T>
{
    int compare(T o1, T o2);
}
```

Über den Rückgabewert wird die Reihenfolge der Sortierung bestimmt. Es gilt:

- = 0: Der Wert 0 bedeutet Gleichheit der beiden Objekte `o1` und `o2`.

- < 0: Das erste Objekt `o1` ist als kleiner als das zweite Objekt `o2` anzusehen.
- > 0: Bei positiven Zahlen ist das erste Objekt `o1` als größer anzusehen.

Grundgerüst eines einfachen Komparators

Stellen wir uns vor, unsere Aufgabe bestünde darin, eine Liste mit `Person`-Objekten nach verschiedenen Kriterien zu sortieren, etwa nach Name, Wohnort oder Alter. Der grundsätzliche Aufbau einer Realisierung für Komparatoren für einen Typ `T` folgt immer einem gleichen Schema: In der `compare(T, T)`-Methode werden die benötigten Vergleiche durchgeführt. Einen Vergleich auf Namen realisiert man beispielsweise wie folgt:

```
public final class PersonNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        Objects.requireNonNull(person1, "person1 must not be null");
        Objects.requireNonNull(person2, "person2 must not be null");

        return person1.getName().compareTo(person2.getName());
    }
}
```

Diskussion: Konsistenz von `compare()` und `equals()`

Obwohl im `Comparator<T>` im Javadoc von `compare(T, T)` empfohlen wird, dass `(compare(x, y) == 0) == (x.equals(y))` gelten sollte, ist dies in der Praxis häufig nicht der Fall. Eine entsprechende Forderung wurde bereits bezüglich des Interface `Comparable<T>` aufgestellt. Dabei gibt es zwischen beiden Forderungen die folgenden, entscheidenden Unterschiede.

Unterschiede der Forderungen von `compareTo()` und `compare()`

Realisierungen des Interface `Comparable<T>` sind meistens bijektiv, d. h., es existiert eine »Genau-dann-wenn«-Beziehung: Aus einer Gleichheit bezüglich `compareTo(T)` folgt eine Gleichheit bezüglich `equals(Object)` und auch umgekehrt: Ergibt `equals(Object)` Gleichheit, so gilt dies auch für `compareTo(T)`.

Realisierungen des Interface `Comparator<T>` sind dagegen oftmals injektiv, d. h., es wird eine »Darausfolgt«-Beziehung beschrieben: Aus einer Gleichheit gemäß `equals(Object)` kann man (in der Regel) auf eine Gleichheit bezüglich `compare(T, T)` schließen. Aus einer Gleichheit gemäß `compare(T, T)` folgt jedoch meist *keine* Gleichheit bezüglich `equals(Object)`. Anhand von Komparatoren für `Person`-Objekte, die die Attribute `name` bzw. `city` vergleichen, kann man sich dies verdeutlichen: Zwei gleichnamige oder in der gleichen Stadt wohnende Personen werden über die jeweilige Realisierung von

`compare(Person, Person)` als gleich angesehen, für `equals(Object)` gilt das logischerweise nicht, da hier noch weitere Attribute wie z. B. Geburtstag oder Größe verglichen werden.

Hintergrundwissen: Arbeitsweise sortierender Datenstrukturen

Zum besseren Verständnis der Arbeitsweise der Containerklassen `TreeSet<E>` bzw. `TreeMap<K, V>` betrachten wir ein `TreeSet<Long>`, das initial die Werte 1, 2 und 3 speichert und in das anschließend die Werte 4, 5 und 6 eingefügt werden. Bevor ich auf Details beim Einfügen eingehe, erläutere ich kurz die zugrunde liegende Datenstruktur.

Es wird ein sogenannter ***binärer Baum*** genutzt, der sich dadurch auszeichnet, dass es einen speziellen Startknoten (***Wurzel*** genannt) gibt, der maximal einen direkten linken und einen direkten rechten Kindknoten besitzt. Diese Kindknoten können wiederum jeweils maximal zwei direkte Kindknoten haben, dies aber beliebig fortgesetzt, sodass ein Knoten beliebig viele Nachfahren besitzen kann. Die ***Tiefe des Baums*** ist als die maximale Anzahl der Knoten von der Wurzel bis zu einem Knoten ohne Nachfahren (auch ***Blatt*** genannt) definiert. Per Definition werden ausgehend von der Wurzel jeweils in den linken Kindknoten diejenigen Elemente eingefügt, die in der Wertebelegung ihrer Attribute als kleiner als der momentane Knoten anzusehen sind. Analog gilt dies für »größere« Elemente, die im rechten Teilbaum gespeichert werden.[10]

Für unser Beispiel des `TreeSet<Long>` ergibt sich mit diesem Wissen und den initialen Werten ein Baum, dessen Wurzelknoten den Wert 2 hat und einen linken sowie rechten Nachfolger mit den Werten 1 bzw. 3. Um die Arbeitsweise beim Einfügen von Elementen zu verdeutlichen, werden dann sukzessive die Elemente 4, 5 und 6 eingefügt. Bei Einfügeoperationen (und selbstverständlich auch bei den hier nicht gezeigten Löschoperationen) wird einerseits immer die gewünschte Sortierung hergestellt und andererseits durch ***Balancierung*** (Höhenausgleich der Teilbäume) für eine ausgeglichene Verteilung der innerhalb der Datenstruktur gespeicherten Elemente gesorgt. Die Auswirkungen verschiedener Aktionen auf den Baum zeigt Abbildung 5-7.

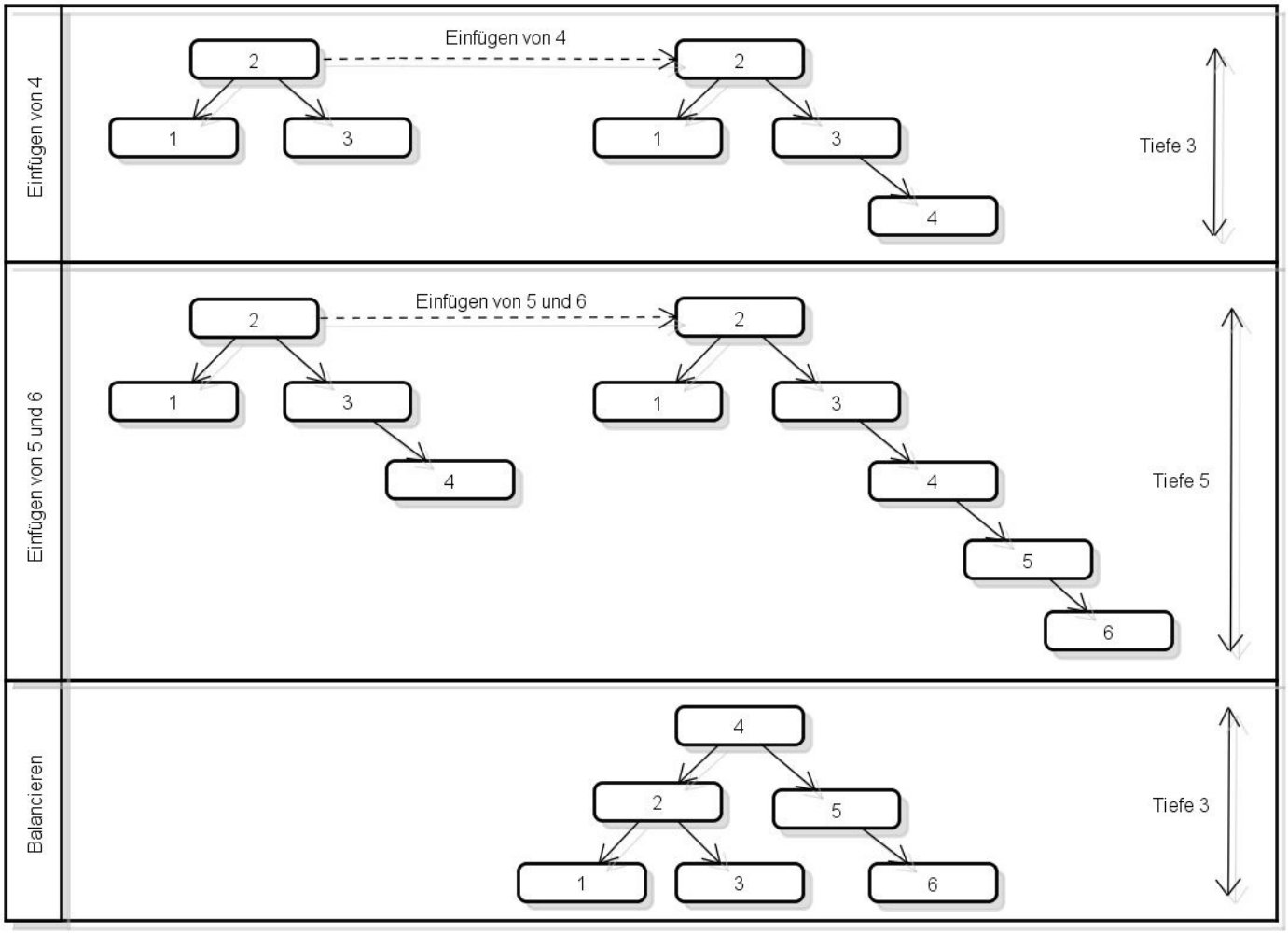


Abbildung 5-7 *Arbeitsweise eines balancierten Baums*

Da die Elemente 4, 5 und 6 größer als die Wurzel sind, werden sie zunächst im rechten Teilbaum einsortiert. Durch Einfügen des Werts 4 entsteht lediglich eine Höhendifferenz von 1 zwischen dem linken und dem rechten Teilbaum. Eine solche Differenz führt nicht zu einem Ausgleichsvorgang, weil sich eine derartige Dysbalance nicht in jedem Fall vermeiden lässt – beispielsweise ist bei zwei gespeicherten Elementen immer ein Teilbaum leer. Durch das Einfügen der Werte 5 und 6 im obigen Beispiel ergibt sich allerdings eine Unausgewogenheit in der Tiefe der Teilbäume, deren Differenz größer als eins ist. Wie in der Abbildung ersichtlich, erhält man nach einem Einfügen möglicherweise fast so etwas wie eine lineare Liste. Um performante Suchen zu gewährleisten, ist es das Ziel, die Tiefe minimal zu halten, den Baum also möglichst auszugleichen. Dies wird durch ein Rotieren der Knoten erreicht, wodurch auch eine Degeneration vermieden wird. Hier wird der Knoten mit dem Wert 4 zur neuen Wurzel.

Durch die beschriebenen Ausgleichsvorgänge wird die Tiefe des Baums nahezu identisch für den linken und den rechten Teilbaum gehalten – genauer: Die Höhendifferenz überschreitet nie den Wert eins. Damit bleibt die maximale Tiefe immer logarithmisch zur Anzahl der im Baum gespeicherten Elemente. Die Ausgeglichenheit sorgt dafür, die maximale Suchgeschwindigkeit auf logarithmische Komplexität zu begrenzen. Das ermöglicht sehr performante Suchvorgänge: Bei 1.000 Elementen beträgt die Tiefe 10 und definiert damit auch die maximale Anzahl

an Suchschritten bis zum Auffinden des gesuchten Elements bzw. zum Erkennen, dass kein solches existiert. Selbst bei 1 Million gespeicherter Elemente sind dadurch maximal 20 Schritte notwendig.

5.1.9 Die Methoden `equals()`, `hashCode()` und `compareTo()` im Zusammenspiel

In C++ gibt es das sogenannte »***Law of the Big Three***«: Wenn für eine Klasse entweder ein Copy-Konstruktor, ein Zuweisungsoperator oder ein Destruktor benötigt wird, so sind in der Regel alle drei zu implementieren. Eine ähnliche Aussage gilt in Java für die drei Methoden `equals(Object)`, `hashCode()` und `compareTo(T)`. Hier ist es aber vor allem wichtig, diese Methoden konsistent zueinander zu implementieren, wobei `compareTo(T)` nicht in jedem Fall angeboten werden muss. Wenn es aber existiert, dann sollte es konsistent zu `equals(Object)` sein.[11] Beachtet man die Forderung nach Konsistenz nicht, kann es zu Fehlern kommen, die sich nur schwierig reproduzieren lassen und sich in merkwürdigem Programmverhalten äußern.

Betrachten wir dies anhand der Verwaltung einiger Objekte der folgenden Klasse `SimplePerson` mithilfe der Datenstrukturen `HashSet<SimplePerson>` und `TreeSet<SimplePerson>`. Die Klasse `SimplePerson` implementiert das Interface `Comparable<SimplePerson>` und ist wie folgt definiert:

```
private static class SimplePerson implements Comparable<SimplePerson>
{
    private final String name;

    SimplePerson(final String name)
    {
        this.name = name;
    }

    @Override
    public int compareTo(final SimplePerson other)
    {
        return name.compareTo(other.name);
    }
}
```

Das folgende Listing zeigt, wie zwei inhaltlich gleiche `SimplePerson`-Objekte erzeugt und per `add(SimplePerson)` in einem `HashSet<SimplePerson>` und einem `TreeSet<SimplePerson>` gespeichert werden. Anschließend ermitteln wir durch Aufruf der Methode `size()` die Anzahl der gespeicherten Elemente im jeweiligen Container:

```
public static void main(final String[] args)
{
    final Set<SimplePerson> hashSet = new HashSet<>();
    hashSet.add(new SimplePerson("Test"));
    hashSet.add(new SimplePerson("Test"));
    System.out.println("HashSet size = " + hashSet.size()); // Size = 2
```

```
    final Set<SimplePerson> treeSet = new TreeSet<>();
    treeSet.add(new SimplePerson("Test"));
    treeSet.add(new SimplePerson("Test"));
    System.out.println("TreeSet size = " + treeSet.size()); // Size = 1
}
```

Listing 5.12 *Ausführbar als* **'LAWOFBIG3EXAMPLE'**

Zunächst ist überraschend, dass im `HashSet<SimplePerson>` zwei Elemente vorhanden sind und nicht wie im `TreeSet<SimplePerson>` nur eins. Wie ist das zu erklären? Das liegt daran, dass die Methode `equals(Object)`, die zur Bestimmung der Gleichheit von Einträgen innerhalb von Buckets verwendet wird, in der Klasse `SimplePerson` nicht implementiert ist. Somit findet ein Referenzvergleich statt, wenn zwei `SimplePerson`-Objekte verglichen werden. Für die Klasse `TreeSet<SimplePerson>` wird beim Hinzufügen zum Ausschluss doppelter Einträge und damit zum Erhalt der Integrität der Menge die Methode `compareTo(SimplePerson)` anstelle von `equals(Object)` verwendet. In diesem Beispiel besteht demnach das Problem, dass die Methode `compareTo(SimplePerson)` [12] nicht mit `equals(Object)` kompatibel ist, wie dies in Abschnitt 5.1.8 gefordert wurde. Es fehlt eine entsprechende Implementierung von `equals(Object)` in der Klasse `SimplePerson`:

```
@Override
public boolean equals(final Object other)
{
    if (other == null)                              // Null-Akzeptanz
        return false;
    if (this == other)                              // Reflexivität
        return true;
    if (this.getClass() != other.getClass())        // Typgleichheit
        return false;

    final SimplePerson otherPerson = (SimplePerson) other;
    return compareTo(otherPerson) == 0;             // Vergleich mit compareTo()
}
```

Listing 5.13 *Ausführbar als* **'LAWOFBIG3EXAMPLE2'**

Ein erneuter Testlauf liefert immer noch zwei Elemente im `HashSet<SimplePerson>` und eins im `TreeSet<SimplePerson>`. Wie kann das sein, nachdem wir auch die `equals(Object)`-Methode korrigiert haben? Überlegen wir kurz.

Die Erklärung ist einfach: Zwar werden nun zwei `SimplePerson`-Objekte als gleich angesehen, wenn sie denselben Inhalt besitzen, aber zu diesem Vergleich kommt es erst gar nicht. Wie bereits in Abschnitt 5.1.7 angedeutet, berechnet die Methode `hashCode()` zunächst das Bucket zur Speicherung der Objekte. Da keine eigene Implementierung der Methode `hashCode()` existiert, werden die von `equals(Object)` als gleich angesehenen `SimplePerson`-Objekte in unterschiedlichen Buckets gespeichert. Dies widerspricht dem `hashCode()`-Kontrakt und führt dazu, dass das gleiche `SimplePerson`-Objekt zweimal in das `HashSet<SimplePerson>` eingefügt wird. Als Korrektur realisieren wir die

hashCode() -Methode wie folgt:

```
@Override
public int hashCode()
{
    return name.hashCode();
}
```

Listing 5.14 *Ausführbar als* **'LAWOFBIG3EXAMPLE3'**

Ein erneuter Testlauf bestätigt, dass als Folge dieser Korrektur nun sowohl das HashSet<SimplePerson> als auch das TreeSet<SimplePerson> nur noch einen Eintrag enthalten.

Fazit

Dieses einfache Beispiel verdeutlicht das Zusammenspiel der drei Methoden und die Notwendigkeit, die Forderungen der jeweiligen Methodenkontrakte einzuhalten, um Überraschungen oder Merkwürdigkeiten zu vermeiden.

5.1.10 Schlüssel-Wert-Abbildungen und das Interface Map

Nachdem wir bisher die konkreten Realisierungen des Interface Collection<E> besprochen haben, wenden wir uns nun den Implementierungen des Interface Map<K, V> zu. Sie realisieren, wie bereits erwähnt, Abbildungen von Schlüsseln auf Werte. Häufig werden Maps deshalb auch als ***Dictionary*** oder ***Look-up-Tabelle*** bezeichnet.

Die zugrunde liegende Idee ist, jedem gespeicherten Wert einen eindeutigen Schlüssel zuzuordnen. Ein intuitiv verständliches Beispiel sind Telefonbücher: Hier werden Namen auf Telefonnummern abgebildet. Eine Suche über einen Namen (Schlüssel) liefert meistens recht schnell eine Telefonnummer (Wert). Da jedoch keine Rückabbildung von Telefonnummer auf Name existiert, wird das Ermitteln eines Namens zu einer Telefonnummer recht aufwendig.

Das Interface Map

Maps speichern Schlüssel-Wert-Paare. Jeder Eintrag in einer Map wird durch das innere Interface Map.Entry<K, V> repräsentiert, das die Abbildung zwischen Schlüsseln (Typparameter K) und Werten (Typparameter V) realisiert. Die Methoden im Interface Map<K, V> sind daher auf diese spezielle Form der Speicherung von Schlüssel-Wert-Abbildungen ausgelegt, ähneln aber denen des Interface Collection<E>. Das Interface Map<K, V> bietet unter anderem folgende Methoden:

- V put(K key, V value) – Fügt dieser Map eine Abbildung (Schlüssel auf

Wert) als Eintrag hinzu. Falls zu dem übergebenen Schlüssel bereits ein Wert gespeichert ist, so wird dieser mit dem neuen Wert überschrieben. Die Methode gibt den zuvor mit diesem Schlüssel verbundenen Wert zurück, sofern es einen derartigen Eintrag gab, ansonsten wird `null` zurückgegeben.

- `void putAll(Map<? extends K, ? extends V> map)` – Fügt alle Einträge aus der übergebenen Map in diese Map ein. Werte bereits existierender Einträge werden, analog zur Arbeitsweise der Methode `put(K,V)`, überschrieben.
- `V remove(Object key)` – Löscht einen Eintrag (Schlüssel und zugehörigen Wert) aus der Map. Als Rückgabe erhält man den zum Schlüssel `key` gehörenden Wert oder `null`, wenn zu diesem Schlüssel kein Eintrag gespeichert war.
- `V get(Object key)` – Ermittelt zu einem Schlüssel `key` den assoziierten Wert. Existiert kein Eintrag zu dem Schlüssel, so wird `null` zurückgegeben.
- `boolean containsKey(Object key)` – Prüft, ob der Schlüssel `key` in der Map gespeichert ist und liefert genau dann `true`, wenn dies der Fall ist.
- `boolean containsValue(Object value)` – Prüft, ob der Wert `value` in der Map gespeichert ist und liefert genau dann `true`, wenn dies der Fall ist.
- `void clear()` – Löscht alle Einträge der Map.
- `int size()` – Ermittelt die Anzahl der in der Map gespeicherten Einträge.
- `boolean isEmpty()` – Prüft, ob die Map leer ist.

Folgende Methoden bieten Zugriff auf gespeicherte Schlüssel, Werte und Einträge:

- `Set<K> keySet()` – Liefert eine Menge mit allen Schlüsseln.
- `Collection<V> values()` – Liefert die Werte in Form einer Collection.
- `Set<Map.Entry<K,V>» entrySet()` – Liefert die Menge aller Einträge. Dadurch hat man sowohl Zugriff auf die Schlüssel als auch auf die Werte.

Diese drei Methoden liefern jeweils Sichten auf die Daten. Erfolgen Veränderungen in der zugrunde liegenden Map, so werden diese in den Sichten widergespiegelt. ***Beachten Sie bitte, dass Änderungen in der Sicht ebenfalls in die Map übertragen werden***. Ähnliches haben wir für Listen und Sets kennengelernt.

Tipp: Der Wert `null` als Schlüssel und als Wert

Liefert die Methode `get(Object)` den Wert `null`, so wird dies vielfach als Nichtvorhandensein eines Eintrags in der Map gedeutet. Diese Schlussfolgerung ist allerdings nicht immer korrekt: In einigen Realisierungen des Interface `Map<K,V>` ist `null` als Wert und sogar als Schlüssel erlaubt. Für `null`-Werte kann man dadurch die Fälle »kein Wert« und »Speicherung des Werts `null`« anhand der Rückgabe von `get()` nicht voneinander unterscheiden. Für diesen Zweck gibt es

die Methode `containsKey(Object)`.

Beispiel: Maps im Einsatz

Bevor wir uns die konkreten Realisierungen des Interface `Map<K, V>` anschauen, wollen wir durch ein kleines Beispiel ein wenig vertrauter mit Maps werden. Wir bauen eine Art Telefonbuch nach bzw. realisieren eine Abbildung von `String` auf `Integer`:

```
public static void main(final String[] args)
{
   final Map<String, Integer> nameToNumber = new TreeMap<>();
   nameToNumber.put("Micha", 4711);
   nameToNumber.put("Tim", 0714);
   nameToNumber.put("Jens", 1234);
   nameToNumber.put("Tim", 1508);       // Zweites put() für "Tim"
   nameToNumber.put("Ralph", 2208);

   // Verschiedene Aktionen ausführen
   System.out.println(nameToNumber);
   System.out.println(nameToNumber.containsKey("Tim")); // Prüfe Schlüssel
   System.out.println(nameToNumber.get("Jens"));        // Zugriff per
Schlüssel
   System.out.println(nameToNumber.size());             // Anzahl der Einträge
   System.out.println(nameToNumber.keySet());           // Alle Schlüssel
   System.out.println(nameToNumber.values());           // Alle Werte
}
```

Listing 5.15 *Ausführbar als* **'FIRSTMAPEXAMPLE'**

Starten wir das Programm FIRSTMAPEXAMPLE, so kommt es zu folgenden Ausgaben, die uns schon ein paar Dinge über Maps verraten, nämlich etwa, dass Werte überschrieben werden, wenn mehrmals Daten zum gleichen Schlüssel eingefügt werden:

```
{Jens=1234, Micha=4711, Ralph=2208, Tim=1508}
true
1234
4
[Jens, Micha, Ralph, Tim]
[1234, 4711, 2208, 1508]
```

Die Klasse `HashMap`

Die Klasse `HashMap<K, V>` ist eine Realisierung der abstrakten Klasse `AbstractMap<K, V>`, die das Interface `Map<K, V>` implementiert. Die Datenhaltung geschieht in einer Hashtabelle und ermöglicht dadurch eine effiziente Ausführung gebräuchlicher Operationen wie `get(Object)`, `put(K, V)`, `containsKey(Object)` und `size()`. Die Reihenfolge der Elemente bei einer Iteration wirkt zufällig. Tatsächlich wird sie durch den jeweiligen Hashwert sowie die Verteilung auf die Buckets bestimmt, wie dies bereits für das `HashSet<E>` besprochen wurde (vgl. Abschnitt 5.1.7).

Beispiel

Zur Demonstration der Klasse `HashMap<K, V>` wollen wir einen in der Praxis häufig anzutreffenden Anwendungsfall betrachten, bei dem eine Menge von Eingabewerten auf eine Menge von Ausgabewerten abgebildet werden soll. Dazu sieht man häufig `if`- oder `switch`-Anweisungen wie die folgende:

```
private static Color mapToColor(final String colorName)
{
    switch (colorName)
    {
    case "BLACK":
        return Color.BLACK;
    case "RED":
        return Color.RED;
    case "GREEN":
        return Color.GREEN;
    // … viele mehr …

    default:
        throw new IllegalArgumentException("No color for: '" + colorName +
"'");
    }
}
```

Sind nur ein paar wenige Fälle abzudecken, kann diese Realisierung durchaus akzeptabel sein, je mehr Fälle jedoch aufeinander abgebildet werden sollen, desto umfangreicher und schwieriger wartbar werden solche Konstrukte. Als Abhilfe kann man sich eine Abbildungstabelle in Form einer `HashMap<K, V>` definieren und die Abbildung wird durch einen Zugriff mit dem entsprechenden Schlüssel realisiert:

```
private static final Map<String, Color> nameToColor = new HashMap<>();
static
{
    initMapping(nameToColor);
}

public static void main(final String[] args)
{
    System.out.println(mapToColor("RED"));      //
java.awt.Color[r=255,g=0,b=0]
    System.out.println(mapToColor("GREEN"));    //
java.awt.Color[r=0,g=255,b=0]
    System.out.println(mapToColor("UNKNOWN")); // => Exception
}

private static Color mapToColor(final String colorName)
{
    if (nameToColor.containsKey(colorName))
    {
        return nameToColor.get(colorName);
    }
    throw new IllegalArgumentException("No color for: '" + colorName + "'");
}

private static void initMapping(final Map<String, Color> nameToColor)
{
    nameToColor.put("BLACK", Color.BLACK);
```

```
    nameToColor.put("RED", Color.RED);
    nameToColor.put("GREEN", Color.GREEN);
    // … viele mehr …
}
```

Listing 5.16 *Ausführbar als* **'HASHMAPEXAMPLE'**

Diese Art der Realisierung hält die Funktionalität der Abbildung in der Applikation selbst kurz, einfach und übersichtlich. Hier im Beispiel wird aus Gründen der Einfachheit eine statische Definition und Initialisierung genutzt. Sofern benötigt kann die Initialisierung auch ausgelagert werden und mithilfe einer externen Datenquelle, etwa einer Datei, erfolgen. Dadurch erzielt man eine größere Flexibilität.

Die Klasse `LinkedHashMap`

Die Klasse `LinkedHashMap<K, V>` bietet die Funktionalität einer `HashMap<K, V>` und erweitert diese um die Möglichkeit, Elemente in einer definierten Reihenfolge (wahlweise Einfüge- bzw. Zugriffsreihenfolge) zu speichern und abrufen zu können.

Zum einen kann dies nützlich sein, wenn man eine feste Reihenfolge bei der Iteration benötigt – für `HashMap<K, V>` ist die Ausgabe recht willkürlich. Zum anderen und für die Praxis relevanter ist es, dass man mithilfe der Klasse `LinkedHashMap<K, V>` auf einfache Weise Zwischenspeicher, auch ***Cache*** genannt, realisieren kann. Diese sind immer dann nützlich, wenn man beispielsweise wiederholt auf ähnliche Daten aus dem Dateisystem oder einer Datenbank zugreift. Diese Zugriffe sind teuer, d. h., sie sind aufwendig und führen durch Latenzzeiten auch zu Verzögerungen in der Abarbeitung des Programms. Als Optimierung kann man Caches für die relevantesten Daten im Speicher halten, um auf diese direkt zugreifen zu können. Häufig sind das die zuletzt zugegriffenen Daten.

Im Folgenden betrachten wir zunächst die Realisierung einer Größenbeschränkung, wobei hier das älteste Element anhand der Reihenfolge des Einfügens bestimmt wird. Neuere Daten verdrängen so früher eingefügte.

Steuerung durch Callback-Methode

Die Klasse `LinkedHashMap<K, V>` bietet die folgende Callback-Methode, die beim Einfügen von Elementen aufgerufen wird:

```
protected boolean removeEldestEntry(Map.Entry<K,V> eldest)
```

Der Rückgabewert bestimmt, ob das jeweils älteste Element aus der Map entfernt werden soll. Die Defaultimplementierung dieser Methode liefert den Wert `false` und sorgt damit dafür, dass beim Hinzufügen von Elementen kein Element gelöscht wird. Soll dieses Verhalten geändert werden, so muss die Methode überschrieben und für zu löschende Elemente den Wert `true` zurückgegeben werden. Das Löschen geschieht dann automatisch durch die Implementierung der Map selbst.

Hinweis: Aussagekräftige Methodennamen im API

Da die Methode `removeEldestEntry(Map.Entry<K,V> eldest)` kein Element löscht, sondern lediglich bestimmt, ob dies geschehen soll, hätte man sie besser `shouldRemoveEldestEntry(Map.Entry<K,V> eldest)` genannt.

Beispiel: Realisierung einer Größenbeschränkung

Mithilfe der gerade vorgestellten Callback-Methode kann man leicht eine in ihrer Größe beschränkte Map implementieren, die ältere Elemente entfernt, wenn eine gewisse Größe überschritten ist und dann Elemente eingefügt werden. Die folgende Klasse `FixedSizeLinkedHashMap<K,V>` zeigt, wie einfach eine derartige Größenbeschränkung zu realisieren ist:

```
public final class FixedSizeLinkedHashMap<K, V> extends LinkedHashMap<K, V>
{
    private final int maxEntryCount;

    public FixedSizeLinkedHashMap(final int maxEntryCount)
    {
        this.maxEntryCount = maxEntryCount;
    }

    @Override
    protected boolean removeEldestEntry(final Map.Entry<K, V> customer)
    {
        return size() > maxEntryCount;
    }
}
```

Für die Größenbeschränkung überschreibt man lediglich die Methode `removeEldestEntry(Map.Entry<K,V>)` und prüft in der Implementierung, ob die Anzahl der gespeicherten Elemente eine zuvor festgelegte Größe übersteigt.

Da hier keine anderweitige Parametrisierung der zugrunde liegenden `LinkedHashMap<K,V>` erfolgt, wird das älteste Element anhand der Reihenfolge des Einfügens bestimmt. Bei Überschreiten der angegebenen Größe wird das laut Einfügereihenfolge älteste, d. h. das zuerst eingefügte Element gelöscht und damit die Größenbeschränkung erhalten.

Im nachfolgenden Beispiel wird die Größe des realisierten Containers zu Demonstrationszwecken auf den Wert 3 festgelegt. Anschließend werden fünf Abbildungen von Namen auf `Customer`-Objekte in der Map abgelegt.

```
public static void main(final String[] args)
{
    // Größenbeschränkung auf drei Elemente
    final int MAX_ELEMENT_COUNT = 3;
    final FixedSizeLinkedHashMap<String, Customer> fixedSizeMap =
              new FixedSizeLinkedHashMap<String, Customer>(MAX_ELEMENT_COUNT);

    // Initial befüllen
    fixedSizeMap.put("Erster", new Customer("Erster", "Stuhr", 11));
    fixedSizeMap.put("Zweiter", new Customer("Zweiter", "Hamburg", 22));
```

```
    fixedSizeMap.put("M. Inden", new Customer("Inden", "Aachen", 39));
    printCustomerList("Initial", fixedSizeMap.values());

    // Änderungen durchführen und ausgeben
    fixedSizeMap.put("New1", new Customer("New_1", "London", 44));
    printCustomerList("After insertion of 'New_1'", fixedSizeMap.values());

    fixedSizeMap.put("New2", new Customer("New_2", "San Francisco", 55));
    printCustomerList("After insertion of 'New_2'", fixedSizeMap.values());
}

private static void printCustomerList(final String title,
                                      final Collection<Customer> customers)
{
    System.out.println(title);
    for (final Customer customer : customers)
        System.out.println(customer);
}
```

Listing 5.17 *Ausführbar als* **'FIXEDSIZELINKEDHASHMAPEXAMPLE'**

Führt man das Programm FIXEDSIZELINKEDHASHMAPEXAMPLE aus, wird die Ersetzungsstrategie deutlich: Die beiden zuerst eingefügten Elemente "Erster" und "Zweiter" werden durch die neu hinzugefügten Elemente "New1" und "New2" verdrängt:

```
[...]
After insertion of 'New_1'
Customer [name=Zweiter, city=Hamburg, age=22]
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
After insertion of 'New_2'
Customer [name=Inden, city=Aachen, age=39]
Customer [name=New_1, city=London, age=44]
Customer [name=New_2, city=San Francisco, age=55]
```

Beispiel: Realisierung eines LRU-Caches

Statt der Reihenfolge des Einfügens als Verbleibkriterium zu nutzen, ist es oftmals sinnvoller, zu betrachten, welche Elemente zuletzt verwendet wurden.[13] Man realisiert dazu einen sogenannten ***LRU-Cache*** (Least-Recently-Used), der die zuletzt benutzten Objekte zwischenspeichert, indem er die am längsten nicht mehr zugegriffenen Elemente im Cache austauscht:

```
public final class LruLinkedHashMap<K, V> extends LinkedHashMap<K, V>
{
    // Kopie der Package-privaten Definitionen aus der Klasse HashMap
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    private static final float DEFAULT_LOAD_FACTOR = 0.75f;
    private static final boolean USE_ACCESS_ORDER = true;

    private final int maxEntryCount;

    public LruLinkedHashMap(final int maxEntryCount)
    {
        // Unschön: Um die Eigenschaft accessOrder anzugeben, müssen wir Werte
        // an den Konstruktor übergeben, die wir nicht spezifizieren wollen
        super(DEFAULT_INITIAL_CAPACITY, DEFAULT_LOAD_FACTOR,
```

```
USE_ACCESS_ORDER);
        this.maxEntryCount = maxEntryCount;
    }

    @Override
    protected boolean removeEldestEntry(final Map.Entry<K, V> customer)
    {
        return size() > maxEntryCount;
    }
}
```

Zur Verdeutlichung der Arbeitsweise der Klasse `LruLinkedHashMap` speichern wir dort wieder einige `Customer`-Objekte. Danach wird dann nur auf drei der vier zuvor gespeicherten Einträge zugegriffen. Durch das Einfügen eines weiteren Eintrags wird der am längsten nicht verwendete Eintrag ersetzt. Im folgenden Beispiel wird der Eintrag »`M. Inden`« durch den Eintrag »`D. Dummy`« ersetzt.

```
public static void main(final String[] args)
{
   // Größenbeschränkung auf vier Elemente
   final int MAX_ELEMENT_COUNT = 4;
   final LruLinkedHashMap<String, Customer> lruMap =
                                    new LruLinkedHashMap<>(MAX_ELEMENT_COUNT);

   lruMap.put("A. Mustermann", new Customer("A. Mustermann", "Stuhr", 16));
   lruMap.put("B. Mustermann", new Customer("B. Mustermann", "Hamburg", 32));
   lruMap.put("C. Mustermann", new Customer("C. Mustermann", "Zürich", 64));
   lruMap.put("M. Inden", new Customer("M. Inden", "Kiel", 32));

   printCustomerList("Initial", lruMap.values());

   // Zugriff auf alle bis auf M. Inden
   lruMap.get("A. Mustermann");
   lruMap.get("B. Mustermann");
   lruMap.get("C. Mustermann");

   // Neuer Eintrag sollte M. Inden ersetzen
   lruMap.put("Dummy", new Customer("D. Dummy", "Oldenburg", 128));

   printCustomerList("Nach Zugriffen", lruMap.values());
}
```

Listing 5.18 *Ausführbar als* **'LRULINKEDHASHMAPEXAMPLE'**

Ein Start des Programms LRULINKEDHASHMAPEXAMPLE zeigt dies und produziert die hier gekürzte Ausgabe:

```
[...]
Nach Zugriffen
Customer [name=A. Mustermann, city=Stuhr, age=16]
Customer [name=B. Mustermann, city=Hamburg, age=32]
Customer [name=C. Mustermann, city=Zürich, age=64]
Customer [name=D. Dummy, city=Oldenburg, age=128]
```

Die Klasse `TreeMap`

Die Klasse `TreeMap<K, V>` ist eine Erweiterung der abstrakten Klasse `AbstractMap<K, V>` und implementiert das Interface `SortedMap<K, V>`. Eine

`TreeMap<K, V>` stellt automatisch die Ordnung der gespeicherten Schlüssel her, und nutzt dazu entweder das Interface `Comparable<T>` oder einem im Konstruktor übergebenen `Comparator<T>`. Außerdem implementiert die Klasse `TreeMap<K, V>` das Interface `NavigableMap<K, V>`, das einige nützliche Methoden definiert: Durch Aufruf der Methode `ceilingKey(K)` erhält man einen passenden Schlüssel, der größer oder gleich dem übergebenen Schlüssel ist. Korrespondierende Methoden `floorKey(K)`, `lowerKey(K)` und `higherKey(K)` liefern Schlüssel, die kleiner oder gleich, kleiner und größer als der angegebene Schlüssel sind. Weiterhin kann man dazugehörende Einträge der Map über korrespondierende `xyzEntry(K)`-Methoden ermitteln, wobei `xyz` für `lower`, `higher` usw. steht.

Beispiel

In folgendem Beispiel nutzen wir die genannten Methoden, um eine Abbildung von Namen auf das Alter zu erreichen und passende Schlüssel bzw. Einträge zu einem übergebenen Namenskürzel zu ermitteln:

```
public static void main(final String[] args)
{
    final NavigableMap<String, Integer> nameToAgeMap = new TreeMap<>();
    nameToAgeMap.put("Max", 47);
    nameToAgeMap.put("Moritz", 39);
    nameToAgeMap.put("Micha", 43);

    System.out.println("floor   Ma: " + nameToAgeMap.floorKey("Ma"));
    System.out.println("higher  Ma: " + nameToAgeMap.higherKey("Ma"));
    System.out.println("lower   Mz: " + nameToAgeMap.lowerKey("Mz"));
    System.out.println("ceiling Mc: " + nameToAgeMap.ceilingEntry("Mc"));
}
```

Listing 5.19 *Ausführbar als* **'TREEMAPEXAMPLE'**

Führt man das Programm TREEMAPEXAMPLE aus, so erhält man diese Ausgabe:

```
floor   Ma: null
higher  Ma: Max
lower   Mz: Moritz
ceiling Mc: Micha=43
```

Die Werte verdeutlichen die vorangegangenen Beschreibungen: Der Aufruf von `floorKey("Ma")` liefert den Vorgängereintrag von `"Ma"`. Weil dieser nicht existiert, wird `null` zurückgeliefert. Ein Aufruf von `higherKey("Ma")` liefert den Nachfolgereintrag von `"Ma"` und dies ist der Schlüssel `"Max"`. Analog arbeiten die anderen Zugriffe.

5.1.11 Erweiterungen am Beispiel der Klasse `HashMap`

Manchmal möchte man die bestehenden Containerklassen um etwas Funktionalität erweitern. Nachfolgend wollen wir das exemplarisch für die Klasse `HashMap<K, V>`

tun. Dort soll die Normierung von Schlüsseln gezeigt werden. Das dient z. B. dazu, Benutzereingaben dezent zu korrigieren, etwa führende oder abschließende Leerzeichen zu entfernen, damit nach einheitlichen Schlüsseln gesucht werden kann.

Anwendungskontext

Verwenden wir die Klasse `HashMap<K, V>`, um Bilder von Typ `java.awt.Image` über einen symbolischen Namen statt über ihren Dateipfad zu referenzieren. Für eine erste, typsichere Umsetzung kann man im einfachsten Fall direkt eine `HashMap<String, Image>` nutzen. Ein Textfeld erlaubt die Eingabe von beliebigen Namen mit dem Ziel, ein Bild darzustellen, wenn zu dem eingegebenen Namen ein entsprechender Eintrag in der Map vorhanden ist.

Das folgende Listing zeigt das initiale Befüllen der Abbildungstabelle und einige Zugriffe darauf, insbesondere den Umgang mit `null` als Schlüssel und als Wert sowie den Spezialfall eines Schlüssels mit Leerzeichen. Die Kommentare im Listing sowie die Ausgaben auf der Konsole helfen, ein erstes Verständnis aufzubauen:

```
public static void main(final String[] args) throws IOException
{
    // Typsichere Definition
    final Map<String, Image> nameToImageMap = new HashMap<>();

    // Speicherung einiger Mappings Name -> Bild
    nameToImageMap.put("Fußball", readImageFile("tile_gras_1.jpg"));
    nameToImageMap.put("Wasserball", readImageFile("tile_water.jpg"));
    nameToImageMap.put("Klettern", readImageFile("tile_rock_2.jpg"));
    // Zugriff liefert BufferedImage
    System.out.println("'Fußball' " + nameToImageMap.get("Fußball"));

    // Spezialfall: Schlüssel mit Leerzeichen und null als Wert
    nameToImageMap.put("Skaten ", null);
    // Zugriff liefert null, obwohl Schlüssel vorhanden
    System.out.println("'Skaten ' " + nameToImageMap.get("Skaten "));
    // containsKey() wertet dies korrekt mit true aus
    System.out.println("'Skaten ' " + nameToImageMap.containsKey("Skaten "));

    // Füge eine Abbildung von null auf Bild hinzu
    nameToImageMap.put(null, readImageFile("tile_gras_2.jpg"));

    // Ausgabe aller Schlüssel und Werte
    System.out.println("Keys = " + nameToImageMap.keySet());
    System.out.println("Values = " + nameToImageMap.values());

    final JFrame frame = new AppFrame("NameToImageMap-Demo", nameToImageMap);
    frame.setVisible(true);
}
```

Listing 5.20 *Ausführbar als* **'NAMETOIMAGEMAPEXAMPLE'**

Das Programm NAMETOIMAGEMAPEXAMPLE produziert folgende Ausgaben:

```
'Fußball' BufferedImage@4fca772d: type = 5 ColorModel: #pixelBits = 24…
'Skaten ' null
'Skaten ' true
Keys = [Fußball, null, Wasserball, Skaten , Klettern]
```

```
Values = [BufferedImage@4fca772d: type = 5 ColorModel: #pixelBits = 24…
```

Darüber hinaus wird durch das Programm ein einfaches GUI bereitgestellt. Dort stehen in einer Combobox alle vorhandenen Schlüssel-Wert-Abbildungen zur Auswahl. Das zeigt Abbildung 5-8.

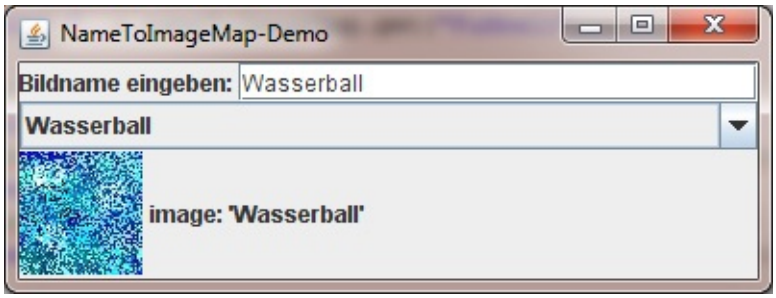


Abbildung 5-8 *Beispielapplikation* NAMETOIMAGEMAPEXAMPLE

Hinweis: Benutzereingaben als Schlüssel

Beim Experimentieren mit eigenen Eingaben wird ein mögliches Problem bei der Verarbeitung von Benutzereingaben sichtbar: Werden diese ungeprüft, unbearbeitet und ohne Korrekturen direkt zur Abfrage als Schlüssel einer Map verwendet, so ist es nicht möglich, gespeicherte Werte zuverlässig wiederzufinden. Fehler treten z. B. dann auf, wenn man ein Wort oder einen Buchstaben kleinschreibt oder die Eingabe versehentlich ein führendes oder nachfolgendes Leerzeichen enthält. Sollen textuelle Werte als Referenz auf Schlüssel einer Map dienen, ist es daher wichtig, eine konsistente Umwandlung oder Normalisierung (z. B. Abschneiden von Leerzeichen) der eingegebenen Texte in eine festgelegte Darstellungsform (beispielsweise eine Darstellung komplett in Groß- oder Kleinbuchstaben) zu definieren. Im Folgenden gehe ich auf derartige Erweiterungen ein.

Lösungsvarianten

Weil die Schlüssel aus Benutzereingaben stammen können, besteht die Gefahr von Inkonsistenzen. Daher soll eine konsistente Normalisierung von Schlüsseln in eine festgelegte Darstellungsform erfolgen. Weiterhin ist es wünschenswert, dies in der zu erstellenden Containerklasse einmal zentral zu realisieren. Zudem soll die Namensabbildung für nutzende Applikationen unsichtbar und ohne Aufwand einsetzbar sein. Um die gewünschten Erweiterungen umzusetzen, existieren folgende zwei Alternativen:

1. ***Aggregation*** einer Containerklasse und ***Delegation*** an deren Methoden
2. ***Ableitung*** von einer Containerklasse und ***Überschreiben*** von Methoden

Aggregation und Delegation

Verwendet man Delegation, so muss die benötigte Funktionalität über Methodenaufrufe an die aggregierte Containerklasse selbst programmiert werden. Eine Realisierung könnte wie folgt aussehen:

```
public final class NameToImageMapUsingDelegation
{
    private final Map<String, Image> nameToImage = new HashMap<>();

    public void put(final String name, final Image image)
    {
        final String key = name != null ? name.toUpperCase().trim() : null;
        nameToImage.put(key, image);
    }

    public Image get(final String name)
    {
        final String key = name != null ? name.toUpperCase().trim() : null;
        return nameToImage.get(key);
    }

    public void clear()
    {
        nameToImage.clear();
    }
}
```

Diese Art der Realisierung besitzt folgende Auswirkungen:

- Es wird häufig – wie im Beispiel auch – zunächst nur diejenige Funktionalität bereitgestellt, die man initial benötigt. Ist später mehr Containerfunktionalität erforderlich, so muss diese passend realisiert werden. Im Speziellen kann man aber auch bewusst gewisse Methoden in der Schnittstelle *nicht* anbieten.[14]
- Ein derart realisierte Klasse erschwert die Handhabung, weil sie nicht gut mit dem Collections-Framework harmoniert: Da weder ein Basisinterface erfüllt noch eine Basisklasse erweitert wird, etwa `Map<K, V>`, lässt sich die von der Utility-Klasse `Collections` angebotene Funktionalität nicht oder nur eingeschränkt nutzen.
- Ein weiteres Problem ist, dass Methodensignaturen umdefiniert werden können. Dies führt aber zu weiteren Inkompatibilitäten mit dem Collections-Framework.

Ableitung und Überschreiben

Seit JDK 5 kann man typsichere Containerklassen auf elegantere Art und Weise als bisher gezeigt realisieren. Man nutzt dazu eine Kombination von Ableitung und Einsatz von Generics. Mithilfe kovarianter Rückgabewerte werden sogar typsichere Rückgabewerte möglich (vgl. Abschnitt 3.6.2). Außerdem bleibt man durch diese Art der Realisierung kompatibel zu allen Funktionalitäten, die durch die Utility-Klasse `Collections` zur Verfügung gestellt werden.

Folgendes Beispiel zeigt den ersten Versuch, die geforderte Funktionalität auf

Basis einer typsicheren `HashMap<String, Image>` umzusetzen:

```
// ACHTUNG: Fehlerhafter erster Versuch!
public final class NameToImageMap extends HashMap<String, Image>
{
    @Override
    public Image put(final String name, final Image image)
    {
        return super.put(name.toUpperCase().trim(), image);
    }

    // @Override => nicht möglich da Signatur get(Object)
    public Image get(final String name)
    {
        return super.get(name.toUpperCase().trim());
    }
    // …
}
```

Auf den ersten Blick ist kein Fehler zu erkennen. Tatsächlich enthält diese Art der Umsetzung jedoch folgende Probleme:

1. Die Realisierung unterstützt keine `null`-Werte als Schlüssel, sondern führt stattdessen sogar zu einer `NullPointerException`. ***Damit verstößt diese Umsetzung gegen die Methodenkontrakte und verhält sich nicht korrekt wie eine Subklasse***. Das Problem ist relativ einfach dadurch zu lösen, dass man eine Hilfsmethode `normalizeKey(String)` wie folgt implementiert:

   ```
   private String normalizeKey(final String key)
   {
       if (key == null)
           return null;

       return key.toUpperCase().trim();
   }
   ```

2. Die oben im Listing gezeigte Methode `put(String, Image)` ist korrekt überschrieben, die Methode `get(Object)` jedoch nicht! Hier findet vielmehr ein versehentliches Überladen von `get(Object)` statt. Ohne Nutzung der Annotation `@Override` kann das leicht passieren, da im Collections-Framework leider einige Methoden mit Parametern vom Typ `Object` statt des Typs `K` des Schlüssels definiert sind. Diese Besonderheit gilt auch für `get()`-Methoden und erfordert Vorsicht, um Typfehler zu vermeiden. Um Fehler beim Überschreiben durch den Compiler aufdecken zu können, bietet es sich an, ***alle Methoden, die man überschreiben möchte, mit der Annotation `@Override` zu kennzeichnen***.

3. Damit sich die Klasse korrekt als Spezialisierung verhält, müssen alle Methoden angepasst werden, die einen Schlüssel als Parameter erwarten. Geschieht dies nicht, kann man ansonsten zwar problemlos Elemente speichern, eine Abfrage über `containsKey(Object)` oder ein Löschen über `remove(Object)` würde jedoch nicht richtig arbeiten. Ohne Anpassungen in einer Überschreibung werden lediglich die Methoden der Oberklasse aufgerufen. ***Es wird zuvor eine Umwandlung der Schlüssel benötigt, um garantiert mit passenden Schlüsseln zu suchen***.

Man kann die Klasse derart verallgemeinern, dass statt des Typs `Image` beliebige Typen gespeichert werden können, indem man eine generische Definition mit dem Typkürzel `V` nutzt. Zudem werden alle Methoden, die auf Schlüssel zugreifen, entsprechend angepasst, wodurch sich die Klasse wie eine Spezialisierung einer `HashMap<K, V>` verhält, die als Besonderheit jedoch die Schlüssel normalisiert. Folgende Klasse `UpperCaseNormalizedHashMap<V>` behebt die angesprochenen Mängel:

```
public final class UpperCaseNormalizedHashMap<V> extends HashMap<String, V>
{
    @Override
    public V put(final String key, final V value)
    {
        return super.put(normalizeKey(key), value);
    }

    @Override
    public V get(final Object key)
    {
        return super.get(normalizeKey((String) key));
    }

    @Override
    public boolean containsKey(final Object key)
    {
        return super.containsKey(normalizeKey((String) key));
    }

    // …

    private String normalizeKey(final String key)
    {
        if (key == null)
            return null;

        return key.toUpperCase().trim();
    }
}
```

Listing 5.21 *Ausführbar als* **'UPPERCASENORMALIZEDHASHMAP'**

Diese Klasse erfüllt die Anforderungen, passt sich ins Collections-Framework ein und stellt somit eine gelungenere Realisierung als diejenige durch Aggregation dar.

5.1.12 Entscheidungshilfe zur Wahl von Datenstrukturen

Wir haben mittlerweile eine Vielzahl von Containerklassen und dabei auch Details zu deren Arbeitsweise kennengelernt. Nachfolgend möchte ich daraus eine Entscheidungshilfe ableiten, weil die adäquate Wahl der für ein Problem geeigneten Datenstruktur große Auswirkungen sowohl auf die Lesbarkeit und Verständlichkeit als auch auf die Performance haben kann: Nehmen wir an, man würde statt einer Liste ein Array verwenden, obwohl die nutzenden Programmteile

Listenfunktionalität benötigen. Diese Funktionalität müsste dann jeweils an der einsetzenden Stelle vom Entwickler selbst programmiert werden. Durch diese für das Problem unpassend gewählte Datenstruktur kommt es zu mehr Sourcecode und mehr Komplexität. Zudem steigt die Wahrscheinlichkeit für Fehler, weil Eigenimplementierungen weniger ausgereift und gut getestet sind als die Containerklassen des JDKs.

Abbildung 5-9 bietet eine Entscheidungshilfe zur Auswahl einer geeigneten Datenstruktur aus dem Collections-Framework. Als Faustregel gilt, dass für Listen die `ArrayList<E>` und für Maps die `HashMap<K, V>` vielfach eine gute Wahl sind, unter anderem auch weil sie in der Regel die beste Performance (vgl. Abschnitt 22.2.1) liefern. Arrays nutzt man z. B. zur Verwaltung primitiver Typen. Insbesondere bei Mehrdimensionalität stellt ein Array-Zugriff vielfach die natürlichste Zugriffsvariante dar.

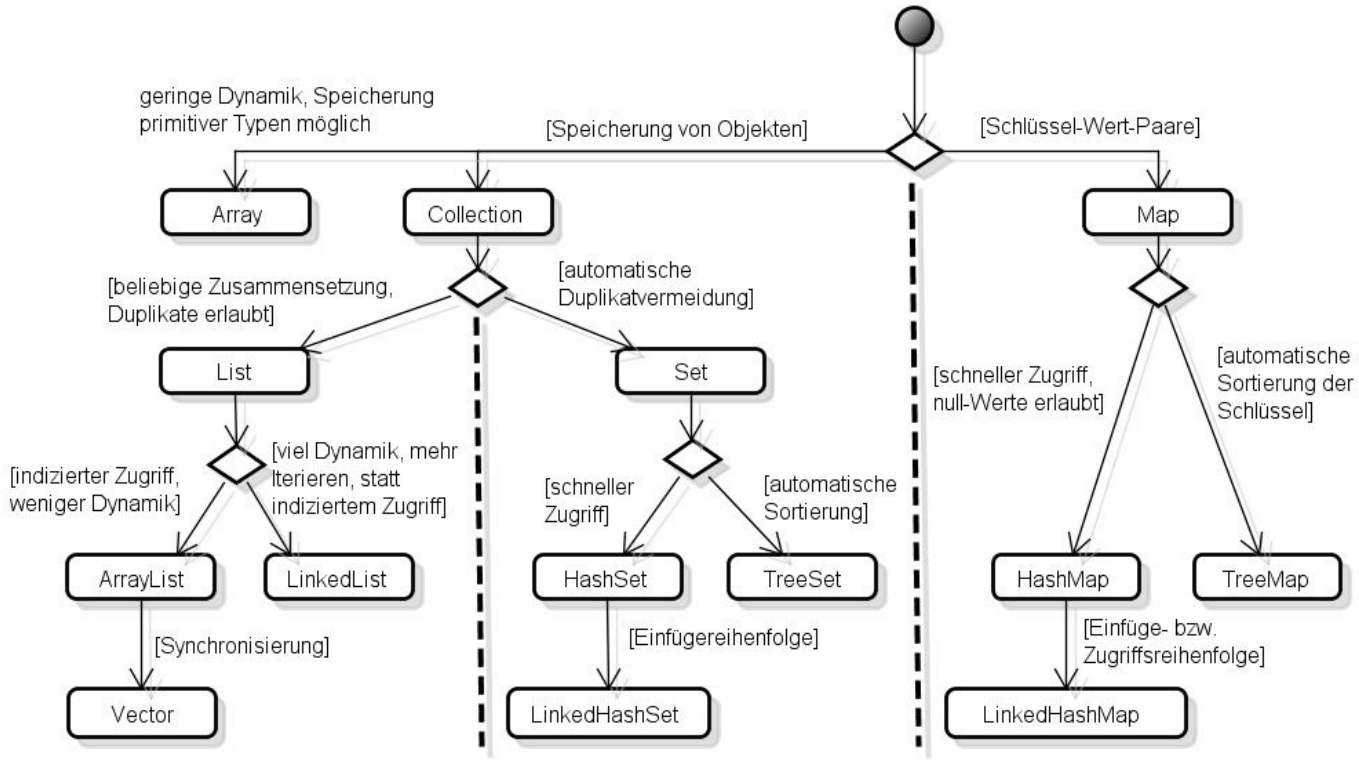


Abbildung 5-9 *Entscheidungshilfe zur Wahl von Datenstrukturen*

Die Grafik lehnt sich an Ideen aus dem Buch »Java 2 – Designmuster und Zertifizierungswissen« [22] von Friedrich Esser an und beschränkt sich aus Gründen der Übersichtlichkeit nur auf die zuvor besprochenen Klassen.

5.2 Suchen, Sortieren und Filtern

Nachdem wir bisher hauptsächlich das Einfügen und Löschen von Daten in Containern betrachtet haben, wollen wir uns im Folgenden mit den Themen Suchen und Sortieren beschäftigen. Das sind zwei elementare Themen der Informatik im Bereich der Algorithmen und Datenstrukturen. Das Collections-Framework setzt beide um und nimmt einem dadurch viel Arbeit ab. Allerdings ist eine wichtige und in

der Praxis häufig benötigte Funktionalität nicht enthalten: das Filtern.

Zunächst betrachten wir das Suchen in Abschnitt 5.2.1. Danach behandeln die Abschnitte 5.2.2 sowie 5.2.3 das Sortieren von Arrays und Listen sowie mithilfe von Komparatoren. Zum Schluss geht Abschnitt 5.2.4 auf das Filtern von Collections nach verschiedenen Kriterien ein.[15]

5.2.1 Suchen

Praktischerweise besitzen alle Containerklassen Methoden, mit denen man nach Elementen suchen kann und auch um zu prüfen, ob Elemente enthalten sind.

Suchen mit `contains()`

Wenn Containerklassen über den allgemeinen Typ `Collection<E>` angesprochen werden, so kann mithilfe der Methode `contains(Object)` lediglich ermittelt werden, ob gewünschte Elemente enthalten sind. Darüber hinaus kann mit `containsAll(Collection<?>)` geprüft werden, ob eine Menge von Elementen enthalten ist. Dabei wird über die gespeicherten Elemente iteriert, die mithilfe von `equals(Object)` auf Gleichheit mit dem übergebenen Element bzw. den übergebenen Elementen geprüft werden. Für Maps existieren – wie bereits erwähnt – korrespondierende Methoden `containsKey(Object)` und `containsValue(Object)`.

Suchen mit `indexOf()` und `lastIndexOf()`

Für Listen gibt es ergänzend zu der Prüfung auf Existenz mit `contains(Object)` die Methoden `indexOf(Object)` und `lastIndexOf(Object)`, um die Position eines gesuchten Elements zu ermitteln. Die erste Methode beginnt die Suche am Anfang einer Liste und die zweite beginnt an deren Ende. Auf diese Weise kann, sofern vorhanden, entweder das erste bzw. letzte Vorkommen ermittelt werden. Gleichheit wird wiederum durch `equals(Object)` überprüft.

Suchen mit `binarySearch()`

Neben den gerade genannten Suchmethoden, die iterativ so lange alle Elemente der Datenstruktur betrachten, bis sie fündig geworden sind, wird für die Datenstrukturen Array und `List<E>` außerdem eine extrem effiziente Suche, die sogenannte ***Binärsuche***, angeboten. ***Das setzt allerdings zwingend eine sortierte Datenstruktur voraus.*** Den Vorteil der Binärsuche gegenüber einer linearen Suche erkennt man bei größeren Datenvolumina: Die Binärsuche ist extrem performant und benötigt zum Suchen eines Elements eine mit der Anzahl der gespeicherten Elemente logarithmisch wachsende Zeit. Das liegt daran, dass der Algorithmus der Suche die

jeweils zu untersuchenden Suchabschnitte halbiert und danach im passenden Teilstück weitersucht. Die beschriebene Binärsuche wird im JDK durch die überladene Methode `binarySearch()` in den Utility-Klassen `Arrays` bzw. `Collections` realisiert. Abbildung 5-10 stellt den prinzipiellen Ablauf dar, wobei aussortierte Teilstücke grau markiert sind.

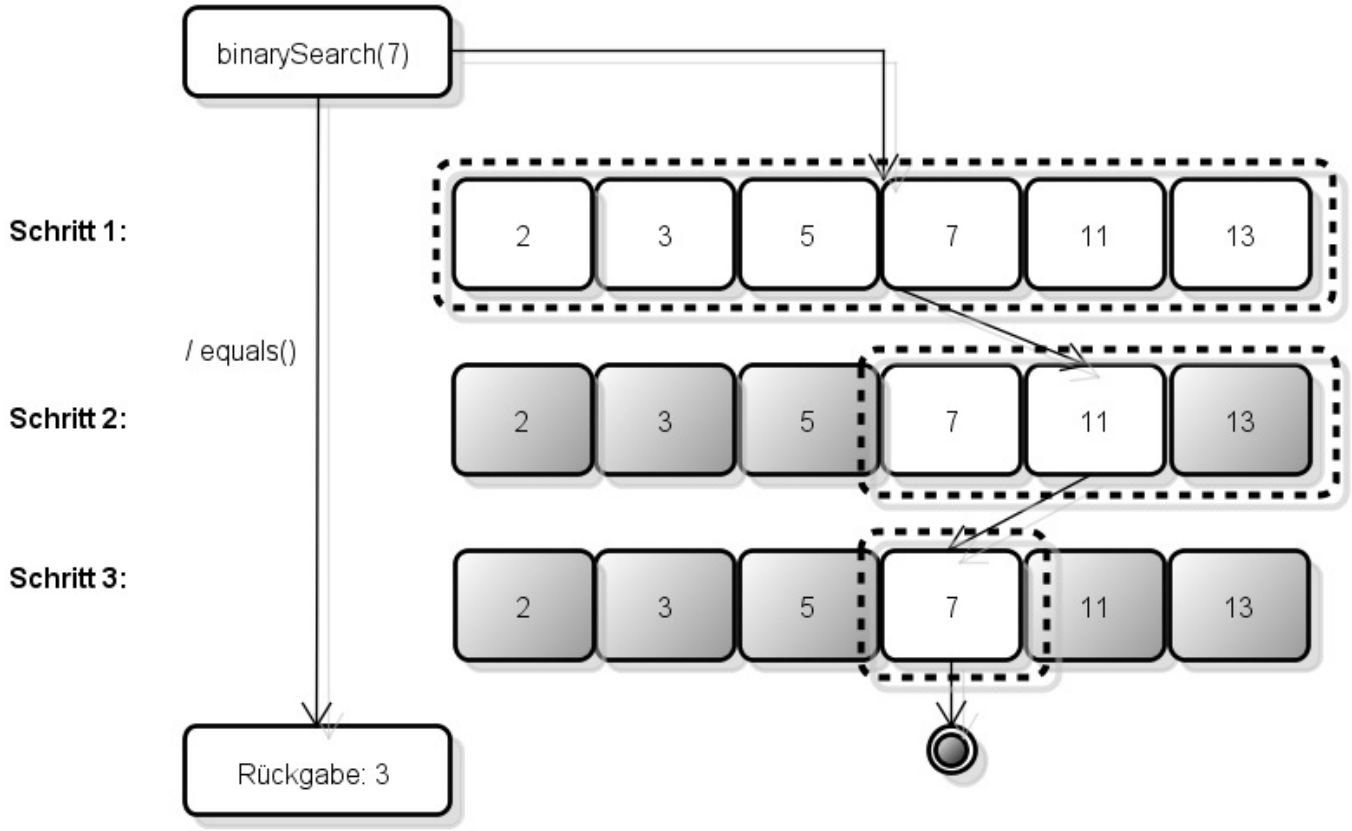


Abbildung 5-10 *Schematischer Ablauf bei der Binärsuche*

Analogie: Binärsuche im realen Leben

Für einige eher abstrakte Algorithmen lassen sich schöne Analogien aus dem realen Leben finden. Dies gilt auch für die Binärsuche. Betrachten wir dazu die Suche nach einer Telefonnummer in einem Telefonbuch, etwa für Aachen. Wollten wir dort die Telefonnummer von Michael Inden finden, so würden wir das Telefonbuch etwa in der Mitte aufschlagen, da wir wissen, dass die Einträge alphabetisch nach Nachnamen geordnet sind und demnach Inden ziemlich mittig zu finden sein sollte – jedenfalls wenn man in etwa eine Gleichverteilung der Nachnamen und deren Anfangsbuchstaben annimmt. Möglicherweise haben wir nicht exakt getroffen und sind bei Max Mustermann gelandet. Dann gehen wir etwas nach vorne und schlagen dort auf. Wir landen etwa bei Nachnamen mit dem Anfangsbuchstaben H. Dann gehen wir wieder ein Stück nach hinten und landen etwa beim Buchstaben J. Das setzen wir fort, bis wir den Autor gefunden haben. Nach diesem Prinzip arbeitet die Binärsuche und ist damit deutlich effizienter, als die Suche beim Buchstaben A zu starten und sich sukzessive bis I vorzuarbeiten.

Binärsuche in Arrays und Listen

Betrachten wir als Beispiel eine Suche in einer `List<Integer>`, die einige Primzahlen in aufsteigender Ordnung speichert. Dort suchen wir nach den Zahlen 7 und 14. Im Anschluss daran kehren wir die Reihenfolge der Elemente um und wiederholen die Suche. Das Ganze wird folgendermaßen realisiert:

```
public static void main(final String[] args)
{
    final List<Integer> primes = Arrays.asList(2, 3, 5, 7, 11, 13);

    System.out.println(primes); // [2, 3, 5, 7, 11, 13]

    // Suche nach 7 und 14
    System.out.println(Collections.binarySearch(primes, 7));  // 3
    System.out.println(Collections.binarySearch(primes, 14)); // -7

    // Anderes Sortierkriterium anwenden, um Versagen der Suche zu provozieren
    Collections.sort(primes, Collections.reverseOrder());
    System.out.println(primes); // [13, 11, 7, 5, 3, 2]

    // Suche nach 7 und 14
    System.out.println(Collections.binarySearch(primes, 7));  // 2
    System.out.println(Collections.binarySearch(primes, 14)); // -7
}
```

Listing 5.22 *Ausführbar als* **'BINARYSEARCHEXAMPLE'**

Führt man das Programm BINARYSEARCHEXAMPLE aus, so wird der Wert 7 wie erwartet an Position drei (0-basiert) gefunden. Der gesuchte Wert 14 ist jedoch nicht gespeichert. Die Methode `binarySearch()` liefert als Ergebnis den Wert -7. Sind die Rückgabewerte der Methode `binarySearch()` größer oder gleich 0, so entspricht dies dem 0-basierten Index, an dem der gesuchte Wert gefunden wurde. Ein negativer Wert bedeutet, dass das gesuchte Element nicht in der Datenstruktur gespeichert ist. Über den Rückgabewert wird die Position beschrieben, an der der gesuchte Wert in den Container eingefügt werden kann, ohne die Sortierung zu zerstören. Demnach codiert der Rückgabewert die Position wie folgt: *Position* = – *Rückgabewert* – 1. Die Zahl 14 könnte demnach an Position 6 eingefügt werden, also hinter der 13.

Bis jetzt scheint alles recht einfach. Um auf Fallstricke hinzuweisen, wird für das Beispiel anschließend eine absteigende Sortierung durch einen Aufruf der statischen Methode `Collections.sort(List<T>, Comparator<? super T>)` hergestellt. Dazu nutzen wird einen Komparator, den man durch Aufruf der statischen Methode `reverseOrder()` erzeugt. Nun wird die Suche auf der absteigend sortierten Liste wiederholt. Der Wert 7 wird dann an Position 2 gefunden, was jedoch ein Zufallstreffer ist, der durch die Teilung der Eingabewerte und die Mittenposition des Werts 7 entsteht.[16] Auch der Rückgabewert -7 für die Suche nach dem Wert 14 verwundert, denn in der umsortierten Liste wäre der Wert -1 die richtige Rückgabe für Position 0 gewesen, an der 14 eingefügt werden müsste (siehe nachfolgenden Kasten »Fallstrick«).

Fallstrick: Unterschiedliche Reihenfolgen bei Sortierung und

Suche

Wie im Beispiel gesehen, ist es elementar wichtig, dass ***sowohl die zur Sortierung genutzte als auch die bei der Suche verwendete Reihenfolge identisch sind.***[a] Ansonsten ist das Ergebnis einer solchen Suche nicht definiert, liefert jedoch häufig den Wert -1. Da die Binärsuche ohne Angabe eines Sortierkriteriums von einer Sortierung gemäß der natürlichen Ordnung »ausgeht«, kommt es im obigen Beispiel durch die abweichende Sortierung aber auch zu der falschen Angabe von -7 für eine Suche der Zahl 14 in der absteigend sortierten Liste von Primzahlen.

[a] Dies ist eine »beliebte« Fangfrage bei OCPJP/SCJP-Prüfungen.

Binärsuche in Sets und Maps?

Intuitiv fragt man sich: Wie sucht man effizient in Sets und Maps? Mit dem bisher erlangten Wissen können wir diese Frage leicht beantworten: Die hashbasierten Container bieten aufgrund der Speicherung in einer Hashtabelle bereits einen extrem performanten Zugriff (sofern die Hashfunktion gut und nicht zu aufwendig zu berechnen ist), sodass eine zusätzliche *eigenständige* Realisierung einer derartigen Suche nicht sinnvoll ist.[17] Das Gleiche gilt für die sortierten Container `TreeSet<E>` und `TreeMap<K, V>`. Wie in Abschnitt 5.1.8 beschrieben, verwenden diese Datenstrukturen einen balancierten Baum, in dem die Suche nach dem gleichen Prinzip wie bei der Binärsuche erfolgt.

5.2.2 Sortieren von Arrays und Listen

Zum Sortieren kann man im Collections-Framework neben verschiedenen Realisierungen der automatisch sortierenden Container `SortedSet<E>` und `SortedMap<K, V>` auch Arrays und Listen durch einen Aufruf der jeweiligen Methode `sort()` aus den Utility-Klassen `Arrays` bzw. `Collections` bei Bedarf sortieren.

Bevor wir im nächsten Abschnitt Komparatoren, die die Reihenfolge der Elemente steuern, genauer betrachten, wollen wir anhand der manuellen Sortierung von Listen den Einsatz von Komparatoren rekapitulieren und danach auch einen Blick auf die Sortierung von Arrays werfen.

Warnhinweis: `null`-Prüfungen in Komparatoren

Um die nachfolgenden Ausführungen zu den Komparatoren nicht mit `null`-Prüfungen unübersichtlich zu machen, werde ich darauf (weitestgehend) verzichten. In eigenen Applikationen ist es jedoch sinnvoll, diese Prüfungen vorzunehmen. Noch besser ist es, die Eingabedaten `null`-frei zu halten, sofern dies möglich ist.

Beispiel: Sortieren einer Liste

Zur Sortierung von Listen nutzen wir die Methode `sort()` sowie die Interfaces `Comparable<T>` und `Comparator<T>`, die die Reihenfolgen von Objekten festlegen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Tim",
"Stefan");

   // Comparable-basierte Sortierung
   Collections.sort(names);
   System.out.println(names);

   final Comparator<String> byLength = new Comparator<String>()
   {
      @Override
      public int compare(final String str1, final String str2)
      {
         return Integer.compare(str1.length(), str2.length());
      }
   };

   // Comparator-bestimmte Sortierung
   Collections.sort(names, byLength);
   System.out.println(names);
}
```

Listing 5.23 *Ausführbar als* **'LISTSORTEXAMPLE'**

Führen wir das Programm LISTSORTEXAMPLE aus, so zeigt die Ausgabe zunächst eine alphabetische Sortierung und dann eine nach der Länge der Strings:

```
[Andy, Michael, Stefan, Tim]
[Tim, Andy, Stefan, Michael]
```

Vereinfachungen mit JDK 8

JDK 8 bietet verschiedenste Neuerungen, die bei Sortierungen nützlich sind. Das sind unter anderem Lambdas, Methodenreferenzen und Defaultmethoden (vgl. Kapitel 11) – aber auch Streams können gewinnbringend eingesetzt werden. Zudem wurde auch das Interface `Comparator` erweitert, sodass sich nun mit sehr wenig Aufwand eigene Komparatoren erstellen lassen. Details finden Sie in Abschnitt 15.1. Nachfolgend ist dazu nur ein kurzes Beispiel angegeben:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Tim",
"Stefan");

   // Comparable-basierte Sortierung
   names.sort(Comparator.naturalOrder());
   System.out.println(names);

   // Comparator-bestimmte Sortierung mit Lambda
   final Comparator<String> byLength =
```

```
            (str1, str2) -> Integer.compare(str1.length(), str2.length());
    names.sort(byLength);
    System.out.println(names);
}
```

Beispiel: Sortieren eines Arrays

Analog zu dem zuvor gezeigten Sortieren von Listen kann man Arrays sortieren. Dafür nutzt man die Methode `Arrays.sort()` – allerdings existiert hier nur eine `Comparable<T>`-basierte Sortierung:

```
public static void main(final String[] args)
{
    final String[] names = { "Andy", "Michael", "Tim", "Stefan" };

    // Comparable-basierte Sortierung
    Arrays.sort(names);
    System.out.println(Arrays.toString(names));
}
```

Vereinfachungen mit JDK 8

Auch für Arrays wurde das Sortieren mit JDK 8 erweitert. Hier wird sogar eine parallele Ausführung der Sortierung möglich. Details finden Sie in Abschnitt 15.3. Nachfolgend ist lediglich die prägnante Schreibweise dargestellt, damit Sie schon einen Eindruck der Vorzüge von Java 8 bekommen:

```
public static void main(final String[] args)
{
    final String[] manyNames = { "Andy", "Michael", "Tim", /* … */ "Stefan" };

    // Comparable-basierte, parallele Sortierung
    Arrays.parallelSort(manyNames);
    System.out.println(Arrays.toString(manyNames));

    // Comparator-bestimmte, parallele Sortierung mit Lambda
    final Comparator<String> byLength =
          (str1, str2) -> Integer.compare(str1.length(), str2.length());
    Arrays.parallelSort(manyNames, byLength);
    System.out.println(Arrays.toString(manyNames));
}
```

Listing 5.24 *Ausführbar als* **'ARRAYSSORTJDK8EXAMPLE'**

Führen wir das Programm ARRAYSSORTJDK8EXAMPLE aus, so kommt es erwartungsgemäß zu folgender Ausgabe, die derjenigen der Listensortierung entspricht:

```
[Andy, Michael, Stefan, Tim]
[Tim, Andy, Stefan, Michael]
```

5.2.3 Sortieren mit Komparatoren

Im Folgenden wird der Einsatz von Komparatoren etwas genauer betrachtet, um

verschiedene Sortierungen und Kombinationen davon zu realisieren. Stellen wir uns vor, wir hätten eine Liste mit `Person`-Objekten, die nach verschiedenen Kriterien sortiert werden sollen, etwa nach Name, Wohnort oder Alter. ***Um die Beispiele einfach halten zu können, gehe ich davon aus, dass die zu sortierenden Collections keine `null`-Werte enthalten und man daher auf `null`-Prüfungen verzichten kann.***

Definition eines Komparators (nach Name)

Zunächst wollen wir Personen nach Namen sortieren. Dazu implementieren wir eine Klasse `PersonNameComparator`, die zur Realisierung die Methode `compareTo(String)` der Klasse `String` verwendet:

```
public final class PersonNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        return person1.getName().compareTo(person2.getName());
    }
}
```

Definition eines Komparators (nach Alter)

Schnell kommt der Wunsch auf, Personen auch nach deren Alter sortieren zu können. Für primitive Datentypen existieren keine `compareTo()`-Methoden. Häufig sieht man dann folgende Art der Realisierung des Vergleichs von Hand:

```
public final class PersonAgeComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        if (person1.getAge() < person2.getAge())
        {
            return -1;
        }
        if (person1.getAge() > person2.getAge())
        {
            return 1;
        }
        return 0;
    }
}
```

Diese Implementierung löst das Problem, erfordert allerdings einige Zeilen Sourcecode. Zum Teil sieht man eine Subtraktion der zu vergleichenden Werte:

```
return person1.getAge() - person2.getAge();
```

Diese Form der Berechnung scheint zunächst kürzer und einfacher zu sein. ***Sie ist jedoch zu vermeiden, da sie nicht das zu lösende Problem widerspiegelt***. Darüber hinaus kann diese Art des Vergleichs zu Problemen durch Wertebereichsverletzungen führen, wenn die Differenz zweier Werte aus dem Wertebereich des verwendeten Datentyps `int` herausfällt. Für Altersangaben kann man dies wohl ausschließen.

Nutzt man die Methode `Integer.compare(int, int)`, die einen Rückgabewert kompatibel zu dem von Komparatoren liefert, erhält man als Vorteil eine kurze Realisierung mit einer guten Lesbarkeit und Verständlichkeit:

```
return Integer.compare(person1.getAge(), person2.getAge());
```

Kombination von Sortierkriterien

Sind Sortierungen nach unterschiedlichen Kriterien möglich, so ist der Wunsch nach einer Kombination verschiedener Sortierungen eine naheliegende Folge, etwa nach einer Kombination der Sortierungen erst nach Alter und dann nach Name.

Beim Sortieren anhand verschiedener Kriterien gibt es vorrangige und nachrangige Sortierkriterien. Nur wenn das erste (vorrangige) Kriterium Gleichheit ergibt, wird anhand des nächsten (nachrangigen) Kriteriums weiter sortiert. Im Falle der Sortierung zuerst nach Alter und dann nach Name heißt dies, dass der Name nur dann zum Tragen kommt, wenn das Alter gleich ist. Verallgemeinert gilt, dass bei Gleichheit eines Sortierkriteriums so lange das jeweils folgende Kriterium betrachtet werden muss, bis entweder alle Sortierkriterien ausgewertet sind oder vorher eine Ungleichheit festgestellt wird. Schauen wir zur Verdeutlichung auf eine Realisierung der Kombination der Sortierungen nach Alter und Name:

```
public final class PersonAgeAndNameComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        int ret = Integer.compare(person1.getAge(), person2.getAge());

        if (ret == 0)
        {
            ret = person1.getName().compareTo(person2.getName());
        }

        return ret;
    }
}
```

Häufig sind Anforderungen nicht präzise formuliert. Nehmen wir an, es bestände zusätzlich der Wunsch, auch in der anderen Kombination, d. h. zuerst nach Name und dann nach Alter, sortieren zu können. Die Realisierung dieser Funktionalität ist einfach:

```
public final class PersonNameAndAgeComparator implements Comparator<Person>
{
    public int compare(final Person person1, final Person person2)
    {
        int ret = person1.getName().compareTo(person2.getName());

        if (ret == 0)
        {
            ret = Integer.compare(person1.getAge(), person2.getAge());
        }

        return ret;
```

```
    }
}
```

Schnell erkennt man, dass die hier eingesetzten Vergleichsoperationen denen der beiden eingangs vorgestellten Komparatoren sehr ähneln. Diese Duplikation ist hier zwar nicht ausgeprägt, aber trotzdem nicht optimal, wenn man qualitativ hochwertigen Sourcecode schreiben möchte. Was kann man also verbessern?

Zusammengesetzte Komparatoren

Die Klassen `PersonAgeComparator` und `PersonNameComparator` sollten ohne Copy-Paste-Duplikation der entsprechenden Zeilen zum Entwurf von neuen Komparatoren wiederzuverwenden sein. Es sollte eine geschicktere Umsetzung geben, als dies für die zwei bisher vorgestellten Sortierkombinationen der Fall war. Man könnte die bereits definierten Komparatoren wie folgt kombinieren:

```
public final class PersonNameAndAgeComparatorV2 implements Comparator<Person>
{
   private final Comparator<Person> ageComparator  = new
PersonAgeComparator();
   private final Comparator<Person> nameComparator = new
PersonNameComparator();

   public int compare( final Person person1, final Person person2)
   {
       int ret = nameComparator.compare( person1, person2);

       if ( ret == 0)
       {
          ret = ageComparator.compare( person1, person2);
       }

       return ret;
   }
}
```

Diese Technik des Zusammenfassens von einzelnen Komparatoren zu neuen speziellen Komparatoren kann man für beliebige Kombinationen von Sortierkriterien nutzen. Allerdings wird dann für jede Kombination von Sortierkriterien eine eigene Klasse benötigt. Man stößt schnell an Grenzen für sinnvolle Namen für die Klassen. Insgesamt entsteht ein wenig flexibles Design, das sich nur aufwendig und umständlich erweitern lässt. Das erinnert an die kombinatorische Explosion von Klassen durch die Realisierung von Eigenschaften und Verhalten mittels Vererbung (vgl. Abschnitt 3.3.2).

Hintereinanderschaltung von Komparatoren

Der eben beschriebene Ansatz lässt sich praktischerweise leicht in einen flexiblen, erweiterbaren und generischen Ansatz umgestalten – ganz nebenbei werden die besprochenen Nachteile der Namenskonflikte und der explodierenden Klassenhierarchien behoben: Wir entwerfen eine Klasse

`PersonUniversalComparator`. Diese verwaltet eine Liste von beliebigen `Comparator<Person>`-Objekten, die hintereinander ausgeführt werden. Die Klasse bietet dazu einen Konstruktor, der eine Liste mit gewünschten `Comparator<Person>`-Objekten entgegennimmt. Weil oftmals genau zwei Komparatoren den Vergleich festlegen, bieten wir einen dafür passenden speziellen Konstruktor an, um die Arbeit für Klienten zu erleichtern. Alternativ können nutzende Klassen beliebige Kombinationen von Komparatoren individuell zusammenstellen. Die Implementierung der Klasse `PersonUniversalComparator` zeigt folgendes Listing:

```
public final class PersonUniversalComparator implements Comparator<Person>
{
    private final List<Comparator<Person≫ comparators = new ArrayList<>();

    public PersonUniversalComparator(final List<Comparator<Person≫
comparators)
    {
        comparators.addAll(comparators);
    }

    // Convenience
    public PersonUniversalComparator(final Comparator<Person> comparator1,
                                     final Comparator<Person> comparator2)
    {
        comparators.add(comparator1);
        comparators.add(comparator2);
    }

    public int compare(final Person person1, final Person person2)
    {
        int ret = 0;

        for (final Comparator<Person> comparator : comparators)
        {
            ret = comparator.compare(person1, person2);
            if (ret != 0)
                break;
        }

        return ret;
    }
}
```

Diese Realisierung macht den Einsatz für Klienten einfach und flexibel. Betrachten wir dies anhand einer kurzen Kundenliste und der Definition zweier zusammengesetzter Komparatoren. Im folgenden Listing ist lediglich die Definition der Komparatoren gezeigt. Die Arbeitsweise wird deutlich, wenn man das Programm PERSONUNIVERSALCOMPARATOREXAMPLE ausführt.

```
public static void main(final String[] args)
{
    // Bereitstellen der Daten
    final List<Person> customers = new ArrayList<>();
    customers.add(new Person("Werner", "Stuhr", 43));
    customers.add(new Person("Tim", "Kiel", 33));
    customers.add(new Person("Tim", "Aachen", 43));
    customers.add(new Person("Reinhard", "Osterrönfeld", 50));
```

```
    customers.add(new Person("Peter", "Oldenburg", 33));

    // Definition von Einzelkomparatoren
    final Comparator<Person> ageComparator = new PersonAgeComparator();
    final Comparator<Person> nameComparator = new PersonNameComparator();

    // Definition zusammengesetzter Komparatoren
    final Comparator<Person> ageAndNameComparator =
                new PersonUniversalComparator(ageComparator, nameComparator);
    final Comparator<Person> nameAndAgeComparator =
                new PersonUniversalComparator(nameComparator, ageComparator);

    // Einsatz der Komparatoren
    final List<Person> ageAndNameSortedList = new ArrayList<>(customers);

    Collections.sort(ageAndNameSortedList, ageAndNameComparator);
    printCustomerList("Sorted by Age And Name:", ageAndNameSortedList);

    System.out.println("-----------------");

    final List<Person> nameAndAgeSortedList = new ArrayList<>(customers);
    Collections.sort(nameAndAgeSortedList, nameAndAgeComparator);
    printCustomerList("Sorted by Name And Age:", nameAndAgeSortedList);
}
```

Listing 5.25 *Ausführbar als* **'PERSONUNIVERSALCOMPARATOREXAMPLE'**

Die Ausgabe des Programms PERSONUNIVERSALCOMPARATOREXAMPLE lässt die Arbeitsweise der zusammengesetzten Komparatoren gut erkennen:

```
Sorted by Age And Name:
Person: Name='Peter' City='Oldenburg' Age='33'
Person: Name='Tim' City='Kiel' Age='33'
Person: Name='Tim' City='Aachen' Age='43'
Person: Name='Werner' City='Stuhr' Age='43'
Person: Name='Reinhard' City='Osterrönfeld' Age='50'
-----------------
Sorted by Name And Age:
Person: Name='Peter' City='Oldenburg' Age='33'
Person: Name='Reinhard' City='Osterrönfeld' Age='50'
Person: Name='Tim' City='Kiel' Age='33'
Person: Name='Tim' City='Aachen' Age='43'
Person: Name='Werner' City='Stuhr' Age='43'
```

Umkehren der Sortierreihenfolge per Komparator

Für Tabellen ist es ein üblicher Anwendungsfall, nach beliebigen Kriterien (Spalten) sortieren zu können. Häufig möchte man dabei zudem die Sortierreihenfolge zwischen aufsteigend und absteigend wechseln können.

Eine Realisierungsidee für den Wechsel der Sortierreihenfolge ist, den Rückgabewert eines Komparators zu negieren. Das funktioniert in der Regel.[18] Einfacher und semantisch klarer als durch Negation lässt sich die Umkehrung der Reihenfolge durch die Bibliotheksklasse `ReverseComparator` und die zwei überladenen Hilfsmethoden `reverseOrder()` ausdrücken. Eine davon basiert auf dem Interface `Comparable<T>` und erlaubt, die natürliche Sortierung umzukehren. Die zweite Version ist für das Interface `Comparator<T>` ausgelegt. Damit können

Sortierungen invertiert werden.

Wir greifen hier das vorherige Beispiel auf und definieren eine absteigende Sortierung nach Alter und Name wie folgt:

```
// Definition des ReverseComparators
final Comparator<Person> reverseAgeAndNameComparator =
                         Collections.reverseOrder(ageAndNameComparator);
Collections.sort(ageAndNameSortedList, reverseAgeAndNameComparator);
```

Listing 5.26 *Ausführbar als* **'PERSONREVERSECOMPARATOREXAMPLE'**

Hiermit beschließe zunächst ich die Ausführungen zu Komparatoren. Mit JDK 8 hat sich in dem Bereich viel getan: Insbesondere in Kombination mit Lambda-Ausdrücken und Methodenreferenzen lassen sich auf einfache Weise selbst komplexere Komparatoren implementieren. Für Details verweise ich auf Abschnitt 15.1.

5.2.4 Filtern von Collections

Das Collections-Framework bietet zwar diverse Algorithmen out-of-the-Box, jedoch fehlt bis JDK 8 das gezielte Filtern von Elementen nach beliebigen Kriterien. Dieser Abschnitt stellt das Thema vor. Damit sind Sie für den Fall gerüstet, mit JDK 7 eine Filterung realisieren zu müssen. Neben Erleichterungen für Komparatoren existieren in JDK 8 mächtige Möglichkeiten zur Filterung mit Streams und dem Filter-Map-Reduce-Framework (vgl. Abschnitt 12.4).

Grundlagen

Der hier beschriebene Ansatz zum Filtern von Collections basiert auf der Definition eines generischen Interface `Filterable<T>`, das eine Methode `accept(T)` enthält. Diese entscheidet, ob ein übergebenes Element akzeptiert wird, und muss für jedes Element der Collection aufgerufen werden.

```
public interface Filterable<T>
{
    boolean accept(final T value);
}
```

Ein Filter, der durch Aufruf der Methode `equals(Object)` auf Gleichheit prüft, kann wie folgt realisiert werden:

```
public class EqualsFilter<T> implements Filterable<T>
{
    private final T acceptedValue;

    public EqualsFilter(final T acceptedValue)
    {
        this.acceptedValue = Objects.requireNonNull(acceptedValue,
                                    "acceptedValue must not be null");
    }

    @Override
```

```
    public boolean accept(final T value)
    {
        return acceptedValue.equals(value);
    }
}
```

Um gültige Werte sicherzustellen, setzen wir hier und in den weiteren Beispielen die statische Hilfsmethode `requireNonNull(T, String)` aus der Utility-Klasse `Objects` ein. Die genannte Methode prüft, ob ein übergebener Parameter ungleich `null` ist. Ist dies nicht der Fall, wird eine Exception ausgelöst.

In einer Utility-Klasse `FilterUtils` erstellen wir folgende Methode `applyFilter(List<T>, Filterable<T>)`:

```
public final class FilterUtils
{
    public static <T> List<T> applyfilter(final List<T> values,
                                          final Filterable<T> filter)
    {
        Objects.requireNonNull(values, "values must not be null");
        Objects.requireNonNull(filter, "filter must not be null");

        final List<T> filteredValues = new ArrayList<>();
        for (final T current : values)
        {
            if (filter.accept(current))
                filteredValues.add(current);
        }

        return filteredValues;
    }
}
```

Die Implementierung folgt dem STRATEGIE-Muster (vgl. Abschnitt 18.3.4) und erlaubt dadurch ohne Anpassungen des generellen Ablaufs, verschiedene Filterungen zu realisieren. Dazu wird die übergebene Liste durchlaufen und für jedes Element die durch das übergebene `Filterable<T>`-Objekt beschriebene Filterbedingung ausgewertet. Dann wird gegebenenfalls das entsprechende Element in die Ergebnisliste `filteredValues` aufgenommen.

Folgende `main()`-Methode nutzt die bisher vorgestellte Basisfunktionalität, um aus einer Liste von `Integer`-Objekten dasjenige mit dem Wert `2` herauszufiltern:

```
public static void main(final String[] args)
{
    final List<Integer> intValues = Arrays.asList(1, 2, 3, 4, 5, 6, 7);

    // int-Zahlenfilter auf den Wert 2
    final Filterable<Integer> numberFilter = new EqualsFilter<>(2);
    final List<Integer> filteredValues = FilterUtils.applyFilter(intValues,
                                                                  numberFilter);
    System.out.println(filteredValues);
}
```

Listing 5.27 *Ausführbar als* **'SIMPLEFILTEREXAMPLE'**

Vergleiche

Häufig soll nicht nach einem exakten Wert gefiltert werden, sondern nach Bedingungen oder Wertebereichen. Man möchte etwa alle Werte erhalten, die größer bzw. kleiner als ein bestimmter Wert sind. Für derartige Vergleiche haben wir bereits das Interface `Comparable<T>` kennengelernt.

Zur Realisierung typsicherer Vergleiche verwenden wir eine generische Definition von Filtern, die basierend auf dem Interface `Comparable<T>` eine Typeinschränkung `<T extends Object & Comparable<T≫` (vgl. Abschnitt 3.7.1) festlegen. Dadurch lassen sich sehr einfach die Vergleiche »größer« bzw. »kleiner« realisieren und in die Bibliothek der angebotenen Filterbedingungen integrieren. Das folgende Listing zeigt exemplarisch die Klasse `Greater` als Implementierung eines »Größer als«-Filters:

```
public class Greater<T extends Object & Comparable<T≫ implements
Filterable<T>
{
    private final T lowerBound;

    public Greater(final T lowerBound)
    {
        this.lowerBound = Objects.requireNonNull(lowerBound,
                                                 "lowerBound must not be
null");
    }

    @Override
    public boolean accept(final T value)
    {
        return lowerBound.compareTo(value) < 0;
    }
}
```

Neben Vergleichen mit einem Grenzwert sind häufig auch Wertebereiche von Interesse. Auch deren Definition wird einfach über folgende Filterklasse `Between` möglich:

```
public class Between<T extends Object & Comparable<T≫ implements
Filterable<T>
{
   private final T lowerBound;
   private final T upperBound;

   public Between(final T lowerBound, final T upperBound)
   {
      this.lowerBound = Objects.requireNonNull(lowerBound,
                                               "lowerBound must not be null");
      this.upperBound = Objects.requireNonNull(upperBound,
                                               "upperBound must not be null");

      if (!(lowerBound.compareTo(upperBound) <= 0))
          throw new IllegalArgumentException("lowerBound " + lowerBound +
                                " must be <= upperBound " + upperBound);
   }

   @Override
   public boolean accept(final T value)
   {
```

```
        return lowerBound.compareTo(value) <= 0 &&
               upperBound.compareTo(value) >= 0;
    }
}
```

Logische Verknüpfungen

Häufig sollen Bedingungen miteinander verknüpft werden. Bei Personen könnte man etwa nach solchen mit dem Namen »Meyer« und einem Alter zwischen 20 und 40 Jahren filtern wollen.

Die logischen Verknüpfungen AND, OR und NOT lassen sich auf einfache Weise als Filter realisieren, wie dies folgendes Listing für die Implementierung von OR zeigt:

```
public class Or<T> implements Filterable<T>
{
    private final Filterable<T> filter1;
    private final Filterable<T> filter2;

    public Or(final Filterable<T> filter1, final Filterable<T> filter2)
    {
        this.filter1 = Objects.requireNonNull(filter1,
                                              "filter1 must not be null");

        this.filter2 = Objects.requireNonNull(filter2,
                                              "filter2 must not be null");
    }

    @Override
    public boolean accept(final T value)
    {
        return (filter1.accept(value) || filter2.accept(value));
    }
}
```

Die Implementierung von AND verwendet lediglich eine andere Verknüpfung (Operator '`&&`') in der `accept(T)` -Methode. Die Filterklasse für NOT benötigt nur einen Eingabeparameter und ist wie folgt realisiert:

```
public class Not<T> implements Filterable<T>
{
    private final Filterable<T> filter;

    public Not(final Filterable<T> filter)
    {
        this.filter = Objects.requireNonNull(filter,
                                             "filter must not be null");
    }

    @Override
    public boolean accept(final T value)
    {
        return !(filter.accept(value));
    }
}
```

In den Implementierungen von AND und OR nutzen wir das KOMPOSITUM-Muster.

Zur Realisierung von NOT kommt das DEKORIERER-Muster zum Einsatz. Detaillierte Informationen zu diesen Mustern liefern die Abschnitte 18.2.3 und 18.2.4.

Spezialisierungen für verschiedene Datentypen

Neben diesen allgemeingültigen Filtern ist es für einige Datentypen wünschenswert, spezielle Filter zu definieren. Für Strings kann etwa ein Enthaltensein-Filter wie folgt realisiert werden:

```
public class StringContains implements Filterable<String>
{
    private final String necessarySubstring;

    public StringContains(final String necessarySubstring)
    {
        this.necessarySubstring = Objects.requireNonNull(necessarySubstring,
                                  "necessarySubstring must not be null");
    }

    @Override
    public boolean accept(final String value)
    {
        if (value == null)
            return false;

        return value.contains(necessarySubstring);
    }
}
```

Die Filter im Einsatz

Nachfolgend nutzen wir viele der bisher erstellten Filter und realisieren damit zwei komplexere Filterbedingungen: Durch die Hintereinanderschaltung mehrerer Filterbedingungen kann man Wertebereiche invertieren oder Filter kombinieren, z. B. die Filter »Wertebereich 3 – 7« sowie »Werte größer als 12«:

```
public static void main(final String[] args)
{
    final List<Integer> intValues = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9,
10,
                                                  11, 12, 13, 14, 15);

    // Ermittle alle Werte, die "NICHT im Bereich 4-11" liegen
    final Filterable<Integer> notRange4_11 = new Not<>(new Between<>(4,11));
    final List<Integer> filteredValues1 =
                        FilterUtils.applyFilter(intValues, notRange4_11);
    System.out.println(filteredValues1);

    // Ermittle alle Werte, die "im Bereich 3-7 liegen oder größer als 12"
sind
    final Filterable<Integer> range2_7OrGreater12 = new Or<>(
                                                    new Between<>(3,7),
                                                    new Greater<>(12));
    final List<Integer> filteredValues2 =
```

```
                        FilterUtils.applyFilter(intValues,
range2_7OrGreater12);
    System.out.println(filteredValues2);
```

Listing 5.28 *Ausführbar als* **'SIMPLEFILTEREXAMPLE2'**

Erwartungsgemäß kommt es zu folgenden Ausgaben für die Ausgangsdaten 1 – 15:

```
[1, 2, 3, 12, 13, 14, 15]
[3, 4, 5, 6, 7, 13, 14, 15]
```

Erweiterung

In der Regel sind in Collections nicht, wie bisher dargestellt, Objekte von Klassen gespeichert, die sich derart einfach filtern lassen. Zum einen implementieren die Klassen möglicherweise das Interface `Comparable<T>` nicht. Zum anderen möchte man vielfach nicht auf Klassenbasis, sondern feingranular auf Basis der Wertebelegungen verschiedener Attribute filtern. In beiden Fällen benötigt man dafür in den Filtern Zugriffe auf Attribute.

Betrachten wir folgende Klasse `SimplePerson`, um das Filtern nach verschiedenen Kriterien nachzuvollziehen:

```
public final class SimplePerson
{
    private final String name;
    private final int age;

    public SimplePerson(final String name, final int age)
    {
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.age = age;
    }

    @Override
    public String toString()
    {
        return "SimplePerson [age=" + age + ", name=" + name + "]";
    }

    public int getAge()
    {
        return age;
    }
    // …
```

Um die bisher definierten Filter wiederverwenden zu können, benötigen wir für jedes Attribut einen speziellen ***Attributermittlungsfilter***. Im folgenden Listing ist ein solcher für das Attribut `age` gezeigt – eine Realisierung für das Namensattribut erfolgt analog.

```
public class SimplePersonAgeFilter implements Filterable<SimplePerson>
{
    private final Filterable<Integer> intFilter;

    public SimplePersonAgeFilter(final Filterable<Integer> intFilter)
    {
        this.intFilter = Objects.requireNonNull(intFilter,
```

```
                                                    "intFilter must not be null");
    }

    @Override
    public boolean accept(final SimplePerson person)
    {
        return intFilter.accept(person.getAge());
    }
}
```

Durch den Einsatz dieser Attributermittlungsfilter kann man auf einfache Weise entsprechende `SimplePerson`-Objekte mit dem Namen »Meyer« und einem Alter zwischen 20 und 40 Jahren ermitteln:

```
public static void main(final String[] args)
{
   final SimplePerson demo1 = new SimplePerson("Meyer1", 11);
   final SimplePerson demo2 = new SimplePerson("Meyer2", 22);
   final SimplePerson demo3 = new SimplePerson("Meyer3", 33);
   final SimplePerson demo4 = new SimplePerson("Meyer4", 44);
   final SimplePerson demo5 = new SimplePerson("Müller", 34);

   final List<SimplePerson> demoDataValues = Arrays.asList(demo1, demo2,
demo3,
                                                           demo4, demo5);

   // Filter für "Alter 20 - 40"
   final Filterable<SimplePerson> ageRangeFilter =
                            new SimplePersonAgeFilter(new Between<>(20, 40));

   // Filter für "Name enthält Meyer"
   final Filterable<SimplePerson> nameFilter =
                       new SimplePersonNameFilter(new StringContains("Meyer"));

   // Kombination der Filter
   final Filterable<SimplePerson> ageAndNamefilter = new And<>(nameFilter,
                                                              ageRangeFilter);

   final List<SimplePerson> filteredValues =
                            FilterUtils.applyFilter(demoDataValues,
                                                    ageAndNamefilter);
   System.out.println(filteredValues);
}
```

Listing 5.29 *Ausführbar als* **'FILTEREXAMPLE'**

Erwartungsgemäß kommt es zu folgender Ausgabe:

```
[SimplePerson [age=22, name=Meyer2], SimplePerson [age=33, name=Meyer3]]
```

Fazit

Dieser Abschnitt hat einen kurzen Einstieg in das Thema Filtern von Datenstrukturen gegeben. Dabei haben wir erfahren, wie durch den Einsatz von problemangepassten Entwurfsmustern kurze, verständliche und leicht erweiter- und kombinierbare Filterklassen realisiert werden können. Bitte bedenken Sie aber, dass mit JDK 8 vielfältige Filtermöglichkeiten bereitgestellt werden (vgl. Abschnitt 12.4).

5.3 Utility-Klassen und Hilfsmethoden

Das Collections-Framework bietet mit den Klassen `Collections` und `Arrays` zwei mächtige Utility-Klassen an, die diverse Algorithmen und Erweiterungen bereitstellen, von denen einige wichtige im Folgenden kurz vorgestellt werden.

5.3.1 Nützliche Hilfsmethoden

In diesem Abschnitt wollen wir einen Blick auf verschiedene Methoden rund um die Erzeugung von Collections werfen.

Listen befüllen und `Arrays.asList()`

Manchmal soll eine Liste aus einer Menge an vordefinierten Werten erstellt werden. Eine zur Initialisierung gebräuchliche Realisierung ist folgende:

```
final List<String> names = new ArrayList<>();
names.add("Max");
names.add("Michael");
names.add("Carsten");
```

Diese Schreibweise ist recht geschwätzig, da der Name der Collection-Variable immer wieder beim Hinzufügen angegeben werden muss. Etwas eleganter kann man das Ganze schreiben, wenn man zwei Java-Techniken kombiniert, nämlich die Definition einer anonymen Klasse und einen Initializer-Block:

```
final List<String> names = new ArrayList<String>()
{{
    add("Max");
    add("Michael");
    add("Carsten");
}};
```

Damit erspart man sich, den Namen der Collection ständig wiederholen zu müssen. Noch besser lesbar kann man diese Befüllung mit Daten durch Aufruf der statischen Methode `Arrays.asList(T...)` folgendermaßen schreiben:

```
final List<String> names = Arrays.asList("Max", "Michael", "Carsten");
```

Hier nutzt man, dass Java seit Version 5 das Sprachfeature Varargs bietet, wodurch die kommaseparierte Angabe beliebig vieler Werte als Parameter möglich wird. Allerdings muss man zwei Besonderheiten beachten, auf die ich nun eingehe.

Hinweis: Syntaktische Besonderheiten im Beispiel

Zur Konstruktion einer Liste mit einer Menge an vordefinierten Werten haben wir eben verschiedene Varianten gesehen. Auf Variante zwei möchte ich noch einmal explizit eingehen:

```
// Drei Besonderheiten
final List<String> names = new ArrayList<String>() // #1
```

```
{{ // #2
    add("Max");       // statt names.add("Max"); #3
    add("Michael");
    add("Carsten");
}}; // #2
```

Dieses kurze Beispiel enthält tatsächlich drei syntaktische Besonderheiten, über die Sie sich eventuell schon beim Betrachten gewundert haben.

1. Wir konstruieren eine `ArrayList<E>` und müssen dabei den Typ angeben, weil hier eine anonyme innere Klassen entsteht. Als erste Besonderheit sehen wir, dass der Diamond Operator <> für anonyme innere Klassen nicht anwendbar ist. Überall sonst dient er als Schreibweisenabkürzung für Generics.
2. Durch die spezielle Doppel-Klammer-Syntax mit `{{ }}` entsteht durch die äußere Klammerung eine anonyme innere Klasse mit dem Basistyp `ArrayList<E>`.
3. Das innere Paar `{}` stellt einen Instance Initializer dar, der während der Konstruktion ausgeführt wird. Zudem können die Aufrufe von `add(E)` ohne Collection-Namen erfolgen.

Besonderheiten

Die von der Methode `Arrays.asList(T...)` erzeugte Liste weist im Gegensatz zu einer `ArrayList<T>` einige Unterschiede auf. Es wird nämlich eine unveränderliche Liste erzeugt, wodurch Änderungen an der Zusammensetzung, etwa über die Methoden `add(T)` bzw. `remove(Object)`, zu einer `UnsupportedOperationException` führen:

```
public static void main(final String[] args)
{
    final String[] valuesArray = { "Value 1", "Value 2", "Value 3" };

    // Änderungen an der Liste (Inhalt, Zusammensetzung)
    final List<String> valuesAsList = Arrays.asList(valuesArray);
    valuesAsList.set(1, "Value 7"); // Inhalt ändern
    // valuesAsList.add("Value 4"); // UnsupportedOperationException

    System.out.println("valuesArray:   " + Arrays.toString(valuesArray));
    System.out.println("valuesAsList:  " + valuesAsList);

    // Änderungen am Inhalt des Arrays
    valuesArray[0] = "Michael changed";
    valuesArray[1] = "Value 1 & 2";

    System.out.println("valuesArray:   " + Arrays.toString(valuesArray));
    System.out.println("valuesAsList:  " + valuesAsList);
}
```

Listing 5.30 *Ausführbar als* **'ARRAYSASLISTEXAMPLE'**

Das Programm ARRAYSASLISTEXAMPLE produziert folgenden Ausgaben:

```
valuesArray:  [Value 1, Value 7, Value 3]
valuesAsList: [Value 1, Value 7, Value 3]
valuesArray:  [Michael changed, Value 1 & 2, Value 3]
```

```
valuesAsList: [Michael changed, Value 1 & 2, Value 3]
```

Als Besonderheit sehen wir zudem, dass Werte inhaltlich geändert werden können und sich dies jeweils in der anderen Datenstruktur widerspiegelt.

Einelementige Collections und `Collections.singletonList()`

Wenn man eine Liste übergeben oder zurückliefern soll, jedoch nur ein einzelnes Datenelement hat, so kann man dieses durch Aufruf der statischen Methode `Collections.singletonList(T)` in eine Liste mit einem einzigen gespeicherten Element umwandeln, die zudem unmodifizierbar ist:

```
final List<Image> thumbnailImages = Collections.singletonList(thumbnailImage);
```

Häufig sieht man jedoch zur Konstruktion einelementiger Listen folgende Zeilen:

```
final List<Image> thumbnailImages = new ArrayList<>();

thumbnailImages.add(thumbnailImage);
```

Nutzt man stattdessen die Methode `Collections.singletonList(T)` wird die Lesbarkeit des Sourcecodes besser. Es wird zudem klarer kommuniziert, dass es sich um eine einelementige, unveränderliche Liste handelt, der nachträglich keine weiteren Elemente hinzugefügt werden können.

Tipp: API-Unschönheit

Im Collections-Framework sind zum Erzeugen einelementiger Collections folgende generische und typsichere Methoden definiert:

- `Set<T> singleton(T o)`
- `List<T> singletonList(T o)`
- `Map<K,V> singletonMap(K key, V value)`

Beachten Sie die Inkonsistenz: Die erste Methode zum Erzeugen eines einelementigen Sets heißt nicht, wie man erwarten dürfte, `singletonSet()`, sondern lediglich `singleton()`.

Null-Objekte mit `Collections.emptyList()/emptyIterator()` usw.

Über die Methoden `emptyList()`, `emptyMap()`, `emptySet()` werden typsichere Konstanten für leere Container bereitgestellt. Sie korrespondieren zu den untypisierten Konstanten `EMPTY_LIST`, `EMPTY_MAP` und `EMPTY_SET` und sind gemäß dem Entwurfsmuster NULL-OBJEKT (vgl. Abschnitt 18.3.2) realisiert. Es lassen sich auf diese Weise leere, unveränderliche Container realisieren.

Mit JDK 7 wurde die Utility-Klasse `Collections` um folgende Null-Objekte für Iteratoren bzw. Enumerationen erweitert:

```
public static <T> Iterator<T> emptyIterator()
public static <T> ListIterator<T> emptyListIterator()
public static <T> Enumeration<T> emptyEnumeration()
```

5.3.2 Dekorierer `synchronized`, `unmodifiable` und `checked`

Im Kontext von Multithreading ist es oftmals hilfreich, Containerklassen unveränderlich zu machen und Zugriffe darauf zu synchronisieren. Wenn man mit älteren, noch auf JDK 1.4 basierenden und damit untypisierten Programmteilen zusammenarbeiten muss, empfiehlt es sich zur Vermeidung von Fehlern, eine Typprüfung zur Laufzeit zu ergänzen.

Das Collections-Framework stellt für die genannten Anwendungsfälle spezifische Hilfsklassen bereit. Diese Erweiterungen erfolgen gemäß dem DEKORIERER-Muster (vgl. Abschnitt 18.2.3) und sind dadurch kombinierbar.

`synchronized`-Collections

Die in diesem Kapitel vorgestellten Containerklassen sind nicht Thread-sicher, unter anderem, weil ihre Methoden unsynchronisiert realisiert sind. Dadurch kann es bei Multithreading zu Inkonsistenzen durch konkurrierende Zugriffe kommen. Zur Abhilfe lässt sich jeder einzelne Methodenzugriff mithilfe der `synchronized`-Wrapper um Synchronisierung erweitern.

Leicht wird allerdings der Fehler gemacht, derart ummantelte Container bzw. die von Hause aus synchronisierten Container `Vector<E>` und `Hashtable<K, V>` als vollständig Thread-sicher zu betrachten. Jeder einzelne Methodenzugriff erfolgt zwar synchronisiert, wodurch er für sich gesehen geschützt ist und nach jedem Methodenaufruf den Container in einem gültigen Zustand hinterlässt. Trotzdem kann es zu Inkonsistenzen kommen, ***weil beim Multithreading die Kombination Thread-sicherer Methoden keine Thread-Sicherheit garantiert***. Das klingt zunächst paradox. Sie werden aber an den folgenden zwei einfachen Beispielen die Problematik sehr schnell erkennen. Weitere Details beschreibt Kapitel 7.

Zugriff auf Elemente

Betrachten wir eine typische Situation beim Zugriff auf eine Collection. Zunächst wird mithilfe der Methode `size()` die Anzahl der Elemente geprüft und anschließend per `get(int)` indiziert zugegriffen:

```
final Vector<Person> personsAsVector = new Vector<>(getAllMembers());
final int elementCount = personsAsVector.size();
```

```
// Sicherstellen, dass Elemente vorhanden sind
if (elementCount > 0)
{
    // IndexOutOfBoundsException wenn parallel clear() aufgerufen wird
    final Person first = personsAsVector.get(0);

    // IndexOutOfBoundsException wenn irgendeine Veränderung erfolgt,
    // etwa durch Aufruf von removeAt() oder clear()
    final Person last = personsAsVector.get(elementCount - 1);
}
```

Nachdem über die Methode `size()` die Anzahl der Elemente ermittelt wurde, kann man bei Multithreading nicht in jedem Fall davon ausgehen, dass der Wert zum Zeitpunkt des nächsten eigenen Zugriffs auf die Collection unverändert ist. Dies liegt daran, dass die Ausführung eines laufenden Threads nahezu jederzeit unterbrochen werden kann. Im ungünstigsten Fall geschieht dies nach Abfrage der Größe und bevor man lesend per `get(int)` zugreift. Wird die Collection von einem parallel laufenden Thread verändert, so kommt es beim Zugriff entweder zu Dateninkonsistenzen (z. B. durch falsche Indizes) oder zu Exceptions.

Iteration für Collections von synchronisierten Wrappern

Während ein Thread eine Iteration über eine synchronisiert ummantelte Collection durchführt, sind konkurrierende Zugriffe durch andere Threads nicht ausgeschlossen, da, wie bereits erwähnt, nur jeder Zugriff für sich, nicht aber eine Folge geschützt abläuft. Wird demnach ein anderer Thread während der Iteration aktiv, so könnte er die Datenstruktur ändern und dadurch beim nächsten Zugriff des Iterators eine `ConcurrentModificationException` auslösen. Man nutzt dann das IDIOM DER THREAD-SICHEREN ITERATION und sichert mit dem Schlüsselwort `synchronized` die Iteration als Ganzes:

```
final List<Person> persons = getAllMembers();
final List<Person> syncPersons = Collections.synchronizedList(persons);
synchronized (syncPersons)
{
    final Iterator<Person> it = syncPersons.iterator();
    while (it.hasNext())
    {
        final Person person = it.next();
        // Aktionen auf dem Person-Objekt
    }
}
```

Das `synchronized (syncPersons)` verhindert, dass die Iteration an einer beliebigen Stelle unterbrochen werden kann. Allerdings werden so auch alle anderen über `syncPersons` synchronisierten Zugriffe auf die Datenstruktur für die Dauer der Iteration blockiert. Dadurch kann eine Parallelverarbeitung (selbst parallele, rein lesende Zugriffe) empfindlich gestört werden. Als Abhilfe können die in Abschnitt 7.6.1 beschriebenen Concurrent Collections zum Einsatz kommen.

`unmodifiable`-Collections

Die `unmodifiable`-Wrapper dienen dazu, eine unveränderliche Sicht, also ein Read-only-Interface, auf eine Collection bereitzustellen. Im Gegensatz zu den zuvor vorgestellten `synchronized`-Wrappern, die Synchronisierungsfunktionalität hinzufügen, werden durch die `unmodifiable`-Wrapper Veränderungsmöglichkeiten »entfernt«.[19] Dies kann zum einen gewünscht sein, um eine Collection nach ihrer Erstellung unveränderlich zu machen. Zum anderen hilft es, Konsistenz im Zusammenspiel verschiedener Komponenten zu wahren. Klienten besitzen lediglich eine unveränderliche Sicht auf die Daten. Die eigene Klasse hat weiter vollen Zugriff. Aufgrund der bereits ausführlich diskutierten Referenzsemantik kann allerdings nur ein Teil der durch den Namen suggerierten Intention erfüllt werden: Nur die Collection selbst wird so vor Veränderungen geschützt, dort gespeicherte Elemente sind jedoch potenziell weiterhin veränderlich.

Beispiel

Betrachten wir zum Verständnis folgendes Beispiel eines Datenmodells `DataModel`, das ein Read-only-Interface `IDataAccessRO` zum lesenden Datenzugriff anbietet. In diesem Modell soll eine beliebige Anzahl von Objekten des Typs `ModelElement` verwaltet werden können. Abbildung 5-11 zeigt das UML-Klassendiagramm.

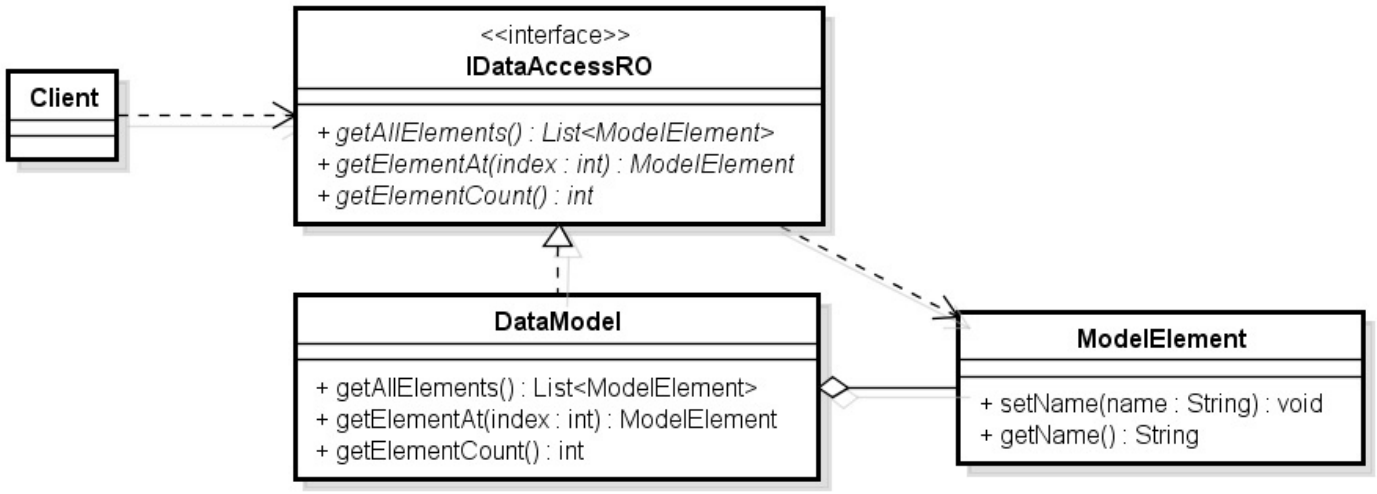


Abbildung 5-11 *Read-only-Datenmodell*

Die Zugriffsmethode `getAllElements()` liefert eine Referenz auf eine Liste von `ModelElement`-Objekten:

```
public List<ModelElement> getAllElements()
{
    return modelElements;
}
```

Die Rückgabe einer Referenz auf diese Liste erlaubt es Aufrufern, die interne Zusammensetzung des Datenmodells beliebig zu verändern. Durch Aufruf von Methoden des `List<ModelElement>`-Interface können unerwartet `ModelElement`-Objekte neu eingefügt oder gelöscht werden, obwohl das Datenmodell für Klienten lediglich Methoden mit rein lesendem Zugriff bereitstellt. Wie bereits in Kapitel 3 besprochen, kann als Abhilfe entweder eine Kopie der Liste

erstellt oder die Technik READ-ONLYINTERFACE für Container angewendet werden. Das Collections-Framework bietet dazu verschiedene Klassen. Wir nutzen hier einen Aufruf von `unmodifiableList(List<? extends T>)`:

```
public List<ModelElement> getAllElements()
{
    return Collections.unmodifiableList(modelElements);
}
```

Ein Versuch, die zurückgegebene Liste zu modifizieren, würde nun eine Exception auslösen. Dieser Schutz vor Modifikationen ist für einige Anwendungsfälle bereits ausreichend. Aufgrund der Referenzsemantik kann jedoch nur ein Teil der Intention einer unveränderlichen Datenstruktur erfüllt werden. Die gespeicherten Elemente sind weiterhin veränderlich, sofern sie nicht als Immutable-Klasse realisiert sind. Das wurde bereits ausführlich in Abschnitt 3.4.1 diskutiert.

Es gibt aber noch einen unerwarteten Effekt. Es wird nämlich lediglich eine unveränderliche Sicht angeboten, die Änderungen an den Ausgangsdaten reflektiert.

```
public static void main(final String[] arguments)
{
    final List<String> originalStrings = new ArrayList<>();
    originalStrings.add("Tim");
    originalStrings.add("Tom");
    originalStrings.add("Peter");

    // JDK und Guava für unmodifizierbare Collections nutzen
    final List<String> unmodifiables =
                        Collections.unmodifiableList(originalStrings);
    final ImmutableList<String> immutables =
                                ImmutableList.copyOf(originalStrings);
    System.out.println("Initial List of Strings:      " + unmodifiables);

    // Peter im Original entfernen … indirekt auch in unmodifiableList()
    originalStrings.remove("Peter");

    System.out.println("Original List of Strings:     " + originalStrings);
    System.out.println("Unmodifiable List of Strings: " + unmodifiables);
    System.out.println("Immutable List of Strings:    " + immutables);

    // Entfernen nicht erlaubt
    unmodifiables.remove("Tom"); // java.lang.UnsupportedOperationException
}
```

Listing 5.31 *Ausführbar als* **'UNMODIFIABLEUNEXPECTEDEXAMPLE'**

Nach dem Start des Programms UNMODIFIABLEUNEXPECTEDEXAMPLE wird man – ohne Kenntnis des einleitenden Satzes – ein wenig verwundert sein, dass die als unveränderlich konstruierte Liste durch Änderungen am Original auch modifiziert wird, hier symbolisch durch das Entfernen vom Eintrag "Peter" dargestellt:

```
Initial List of Strings:      [Tim, Tom, Peter]
Original List of Strings:     [Tim, Tom]
Unmodifiable List of Strings: [Tim, Tom]
Immutable List of Strings:    [Tim, Tom, Peter]
Exception in thread "main" java.lang.UnsupportedOperationException
  at
java.util.Collections\$UnmodifiableCollection.remove(Collections.java:1118)
```

In der Regel ist das nicht konform zu der Erwartungshaltung einer unveränderlichen Datenstruktur. Für diese Fälle kann man Google Guava einsetzen, wodurch man eine wirklich unveränderliche Kopie der Listenzusammensetzung erhält – wie es das Beispiel zeigt. Aufgrund der Referenzsemantik sind die einzelnen Listenelemente hier allerdings nach wie vor veränderlich, sofern sie nicht selbst wieder als Immutable-Klasse realisiert sind.

Tipp: Einbinden von Google Guava

Für viele elementare, aber auch komplexere Funktionalitäten existieren Fremdbibliotheken. Google Guava enthält eine große Menge nützlicher Utility-Klassen. Die derzeit aktuelle Version 18.0 dieser Bibliothek binden wir in unser Projekt ein, indem wir deren Abhängigkeit in der Datei `build.gradle` wie folgt angeben:

```
dependencies
{
    compile 'com.google.guava:guava:18.0'
}
```

`checked`-Collections

Beim Einsatz von Collections ohne Generics kann man sich nicht sicher sein, von welchem Typ gespeicherte Werte sind und ob nur Elemente des erwarteten Typs verarbeitet werden. Durch den Einsatz von Generics kann man Typsicherheit sicherstellen – allerdings unter der Prämisse, dass es beim Kompilieren zu keinerlei Typwarnungen kommt. Aufgrund der Realisierung durch Type Erasure erfolgen zur Laufzeit keine Prüfungen mehr (vgl. Abschnitt 3.7.2). Dies kann vor allem im Zusammenspiel mit sogenanntem Legacy-Code[20] problematisch werden. In diesem Fall ist damit Sourcecode gemeint, der keine Generics verwendet. Als Folge kommt es immer zu Typwarnungen zur Kompilierzeit – durch den Compiler kann Typsicherheit nicht mehr garantiert werden.

Die `checked`-Wrapper fügen bestehenden Collections eine ergänzende Laufzeitprüfung hinzu. Sie stellen bei jedem Aufruf sicher, dass bei Aktionen mit inkompatiblen Elementtypen eine `ClassCastException` ausgelöst wird. Dadurch erreicht man eine Datenstruktur, die zur Laufzeit größtmögliche[21] Typsicherheit bietet. Folgende Methoden dienen dazu, Typsicherheit beim Mix von typisierten und untypisierten Containerklassen in diesem Sinne sicherzustellen:

- `checkedCollection(Collection<E>, Class<E>)` – Versieht eine beliebige Collection mit einer Typprüfung auf den angegebenen Typ.
- `checkedList(List<E>, Class<E>)` – Identisch, sichert aber eine Liste.
- `checkedMap(Map<K,V>, Class<K>, Class<V>)` – Identisch, sichert aber eine Map.
- `checkedSortedMap(Map<K,V>, Class<K>, Class<V>)` – Identisch,

sichert aber eine Map mit dem Basisinterface `SortedMap<K, V>`.

- `checkedSet(Set<E>, Class<E>)` – Identisch, sichert aber ein Set.
- `checkedSortedSet(SortedSet<E>, Class<E>)` – Identisch, sichert aber ein Set mit dem Basisinterface `SortedSet<E>`.

Beispiel

Führen wir das vorherige Beispiel eines Datenmodells zur Verwaltung von `ModelElement`-Objekten fort und schauen auf eine Utility-Klasse `OldStyleUtilityClass`, die mit JDK 1.4 entwickelt wurde und die wir nicht ändern können. Es existieren dort unter anderem die Methoden `addSampleModelElement(List)` und `printListElements(String, List)`. Obwohl der Namen der ersteren Methode suggeriert, dass der übergebenen Liste ein Objekt vom Typ `ModelElement` hinzugefügt wird, wird tatsächlich (und vermutlich versehentlich) aber ein Stringobjekt in eine übergebene untypisierte Liste eingefügt:

```
// Auf das Wesentliche gekürzte, ältere Utility-Klasse
public class OldStyleUtilityClass
{
    public static final void addSampleModelElement(final List elements)
    {
        // Hinzufügen eines Strings statt eines ModelElements
        elements.add("Unexpected element of type string!");
    }

    public static void printListElements(final String title,
                                         final List elements)
    {
        System.out.println(title);
        for (int i = 0; i < elements.size(); i++)
            System.out.println(i + ": " + elements.get(i));
    }
    // …
}
```

Um die Arbeitsweise der `checked`-Collections zu verdeutlichen, verwenden wir eine typisierte Liste `List<ModelElement>`, rufen dann die Methode `addSampleModelElement(List)` auf und geben die dort gespeicherten Elemente per `printListElements(String, List)` aus. Diese Aktionen laufen noch ohne Typprüfung ab. Zu deren Aktivierung erzeugen wir dann mit dem Wrapper `checkedList(List<E>, Class<E>)` eine typprüfende Ummantelung und führen die Aktionen erneut aus:

```
public static void main(final String[] args)
{
    final List<ModelElement> elements = new ArrayList<>();
    elements.add(new ModelElement("Modelelement"));

    // Hinzufügen auf typisierter Liste -> String wird gespeichert !!
    System.out.println("Adding string to List<ModelElement>");
    OldStyleUtilityClass.addSampleModelElement(elements);
```

```
    OldStyleUtilityClass.printListElements("UNCHECKED", elements);

    // Erzeugen einer dynamisch typsicheren Sicht auf die Liste
    final List<ModelElement> checkedElements =
Collections.checkedList(elements,
            ModelElement.class);

    // Hinzufügen auf typisierter Liste -> Exception !!
    System.out.println("Adding string to Checked List");
    OldStyleUtilityClass.addSampleModelElement(checkedElements);
    OldStyleUtilityClass.printListElements("TYPE-CHECKED", checkedElements);
}
```

Listing 5.32 *Ausführbar als* **'CHECKEDCOLLECTIONSEXAMPLE'**

Beim Betrachten des Listings und Ausführen des Programms erkennt man die zuvor geschilderte Problematik möglicher Typinkompatibilitäten: Wird die Methode `addSampleModelElement(List)` mit einer Eingabe vom Typ `List<ModelElement>` aufgerufen, so wird die Methode `add(Object)` auf der übergebenen, untypisierten Listenreferenz `elements` ohne Fehler durchgeführt, obwohl hier eigentlich eine Typverletzung vorliegt: Es wird tatsächlich ein Stringobjekt in eine auf den Typ `ModelElement` beschränkte Liste eingefügt. Erinnern wir uns daran, dass Generics durch Type Erasure realisiert sind, so wird das Verhalten klar: Zur Programmlaufzeit existiert tatsächlich nur eine untypisierte Form der Containerklassen, hier `List.class`.

Ummantelt man diese Liste jedoch mit einer `checked`-Collection, so findet bei jedem Aufruf einer Methode zunächst eine Typprüfung statt. Ruft man mit einer derart geschützten Referenz die Methode `addSampleModelElement(List)` auf, so wird ein Einfügen verhindert, und es kommt zu einer `ClassCastException`. Folgende, gekürzte Ausgabe des obigen Programms verdeutlicht dies:

```
Adding string to List<ModelElement>
UNCHECKED
0: ModelElement name = 'Modelelement'
1: Unexpected element of wrong string type!
Adding string to Checked List
Exception in thread "main" java.lang.ClassCastException: Attempt to insert
class
       java.lang.String element into collection with element type class
      collections.ModelElement
[…]
```

5.3.3 Vordefinierte Algorithmen

Im Collections-Framework stehen bereits diverse Algorithmen zur Verfügung, die bei der täglichen Arbeit gewinnbringend einzusetzen sind.

Anhand der Speicherung und Verarbeitung von `Person`-Objekten wollen wir einen kurzen Blick auf einige der angebotenen Algorithmen werfen. Nehmen wir dazu an, dass die Klasse `Person` das Interface `Comparable<Person>` implementiert.

Die Methoden `nCopies()` und `frequency()`

Durch Aufruf der Methode `nCopies(int, T)` wird eine `List<T>` erzeugt, die *n*-mal das übergebene Objekt vom Typ `T` enthält. Mit einem Aufruf der Methode `frequency(Collection<?>, Object)` kann man die Anzahl der gemäß `equals(Object)` gleichen Objekte innerhalb einer Collection bestimmen.

Wir definieren zunächst drei `Person`-Objekte `MALE`, `FEMALE` und `MISTER_X`. Anschließend rufen wir die Methode `nCopies(int, T)` auf, um in der Methode `initPersonList()` mehrere Instanzen zum Befüllen einer Liste zu erzeugen: einen `MISTER_X`-, zwei `MALE`- und drei `FEMALE`-Einträge. Durch einen Aufruf von `frequency(Collection<?>, Object)` ermitteln wir in der nachfolgenden `main()`-Methode die Anzahl der gespeicherten `MALE`-Objekte:

```
public final class AlgorithmsExample
{
    private static final Person MALE     = new Person("Male", "Bremen", 42);
    private static final Person FEMALE   = new Person("Female", "New York",
43);
    private static final Person MISTER_X = new Person("Mister X", "Sydney",
44);

    private static List<Person> initPersonList()
    {
        final List<Person> maleList = Collections.nCopies(2, MALE);
        final List<Person> femaleList = Collections.nCopies(3, FEMALE);

        final List<Person> persons = new LinkedList<>();
        persons.addAll(maleList);
        persons.addAll(femaleList);
        persons.add(MISTER_X);
        return persons;
    }

    public static void main(final String[] args)
    {
        final List<Person> persons = initPersonList();

        final int maleCount = Collections.frequency(persons, MALE);
        System.out.println("Male Persons: " + maleCount);
        System.out.println("All Persons: " + persons);
    }
}
```

Listing 5.33 *Ausführbar als* **'ALGORITHMSEXAMPLE'**

Wie erwartet, kommt es zu folgender Ausgabe auf der Konsole, die zwei `MALE`s und drei `FEMALE`s zeigt:

```
Male Persons: 2
All Persons: [Person: Name='Male' City='Bremen' Age='42', Person: Name='Male'
      City='Bremen' Age='42', Person: Name='Female' City='New York' Age='43',
      Person: Name='Female' City='New York' Age='43', Person: Name='Female'
City=
      'New York' Age='43', Person: Name='Mister X' City='Sydney' Age='44']
```

Die Methoden `min()` und `max()`

In der Klasse `Math` gibt es die statischen Methoden `min()` und `max()`. Diese ermitteln den kleinsten bzw. größten Wert zweier Zahlen. Gleichnamige Methoden sind in der Klasse `Collections` zur Bestimmung minimaler und maximaler Elemente innerhalb einer Collection definiert. Diese Methoden arbeiten standardmäßig mit der natürlichen Ordnung basierend auf `Comparable<T>`, können aber auch selbst definierte Komparatoren nutzen.

Für ein Beispiel fügen wir in eine `List<Person>` drei `Person`-Objekte ein. Dann nutzen wir zunächst die natürliche Ordnung, also `Comparable<Person>`. Diese ist durch die Implementierung der Klasse `Person` so realisiert, dass hier nach Name, Ort und Alter sortiert wird. Damit werden die Methoden `min()` und `max()` ausgeführt. Anschließend kommt ein selbst definierter Komparator zum Einsatz, der die Städtenamen vergleicht und daraus das Maximum bestimmt:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Anton", "Tirol", 11));
    persons.add(new Person("Micha", "Zürich", 43));
    persons.add(new Person("Stefan", "Kiel", 43));

    // Bestimmung min() mit Comparable
    final Person min = Collections.min(persons);
    System.out.println("Min: " + min);
    // Bestimmung max() mit Comparable
    final Person max = Collections.max(persons);
    System.out.println("Max: " + max);

    // Bestimmung max() mit eigenem Komparator
    // Besonderheit: Diamond Operator nicht für annonyme innere Klassen
erlaubt
    final Comparator<Person> cityComparator = new Comparator<Person>()
    {
        public int compare(final Person person1, final Person person2)
        {
            return person1.getCity().compareTo(person2.getCity());
        }
    };
    final Person maxCity = Collections.max(persons, cityComparator);
    System.out.println("Max city: " + maxCity);
}
```

Listing 5.34 *Ausführbar als* **'ALGORITHMSEXAMPLEMINMAX'**

Das Programm ALGORITHMSEXAMPLEMINMAX produziert folgende Ausgaben:

```
Min: Person: Name='Anton' City='Tirol' Age='11'
Max: Person: Name='Stefan' City='Kiel' Age='43'
Max city: Person: Name='Micha' City='Zürich' Age='43'
```

Die Methoden `shuffle()` und `replaceAll()`

Für einige Anwendungsfälle ist es praktisch, die Anordnung innerhalb einer

Collection umzuordnen, etwa zur Präsentation eines zufälligen Vorschlags aus einer Bestenliste wie aus den beliebtesten Pizzen des Monats oder den Top-10-Bestsellern. Das kann über die Methode `shuffle(List<?>)` zufallsbasiert geschehen. Darüber hinaus lassen sich über `replaceAll(List<T>, T, T)` Elemente ersetzen.

Folgendes Listing zeigt den Einsatz beider Methoden. Zunächst ersetzen wir die Zahlen 1–3 mit dem Wert 7 und danach durchmischen wir die Elemente:

```
public static void main(final String[] args)
{
    // Eingabe von 1 - 10
    final List<Integer> numbers = Arrays.asList(1,2,3,4,5,6,7,8,9,10);

    // Ersetzungen
    Collections.replaceAll(numbers, 1, 7);
    Collections.replaceAll(numbers, 2, 7);
    Collections.replaceAll(numbers, 3, 7);
    System.out.println("All numbers after replace: " + numbers);

    // Umordnen
    Collections.shuffle(numbers);
    System.out.println("All numbers after shuffle: " + numbers);

    System.out.println("#7: " + Collections.frequency(numbers, 7));
    System.out.println("#3: " + Collections.frequency(numbers, 3));
}
```

Listing 5.35 *Ausführbar als* **'ALGORITHMSEXAMPLESHUFFLEREPLACEALL'**

Führt man das Programm ALGORITHMSEXAMPLESHUFFLEREPLACEALL aus, so kommt es in etwa zu folgenden Ausgaben:

```
All numbers after replace: [7, 7, 7, 4, 5, 6, 7, 8, 9, 10]
All numbers after shuffle: [6, 7, 9, 5, 7, 10, 7, 4, 8, 7]
#7: 4
#3: 0
```

Es findet offensichtlich zunächst eine Ersetzung statt, wodurch es keine 1, 2 und 3 mehr, aber viermal die 7 gibt. Danach zeigt das Ergebnis von `shuffle()` die Umordnung.

5.4 Containerklassen: Generics und Varianz

Um einige Besonderheiten im Kontext von Generics und Polymorphie kennenzulernen, nutzen wir wieder grafischer Figuren mit einer gemeinsamen Basisklasse `BaseFigure` als Beispiel – ähnliche Klassenhierarchien wurden auch schon in verschiedenen Abschnitten im Kapitel 3 zu OO-Design genutzt.

Bisher haben wir Generics und Collections ohne viel Nachdenken verwendet und dabei ganz natürlich Zuweisungen wie Folgende geschrieben:

```
final List<String> names = new ArrayList<String>();
final Set<BaseFigure> figures = new HashSet<BaseFigure>();

// oder mit Diamond Operator aus JDK 7 kürzer wie folgt:
final List<String> namesJDK7 = new ArrayList<>();
```

```
final Set<BaseFigure> figuresJDK7 = new HashSet<>();
```

In diesem Abschnitt wollen wir uns ein paar Details von Generics widmen, um mögliche Probleme beim Einsatz generischer Typen im Zusammenhang mit Collections und der Verarbeitung von Typen einer Vererbungshierarchie zu vermeiden.

Im obigen Beispiel sind die Containerklassen auf einen bestimmten Typ fixiert. In einer solchen Collection lassen sich trotzdem auch Objekte von Subtypen speichern, z. B. statt vom Typ `BaseFigure` auch solche vom Typ `Circle`. Für Arrays kann man problemlos folgende Konstrukte verwenden:

```
final Object[] names = new String[10];
final BaseFigure[] figures = new Circle[10];
```

Problematisch wird es aber, diese mit Generics abzubilden. Intuitiv könnte man Folgendes schreiben wollen:

```
// Achtung: Kompilerfehler
final List<Object> names = new ArrayList<String>();
final Set<BaseFigure> figures = new HashSet<Circle>();
```

Diese kovariante Definition ist so für generische Container nicht erlaubt und führt zu Kompilierfehlern. In den nachfolgenden Abschnitten werden wir diese Thematik und Hintergründe dazu genauer behandeln.

Generics, Invarianz, Kovarianz und Polymorphie

Nachfolgendes Beispiel zeigt die gewohnte Definition typsicherer Containerklassen: Man kann eine Instanz eines spezifischeren Typs eines Containers, hier `HashSet<BaseFigure>`, einer Variablen eines allgemeineren Typs eines Containers, hier `Set<BaseFigure>`, zuweisen. Dabei sind die Typparameter invariant und die Typen der Container kovariant:

```
final Set<BaseFigure> figures = new HashSet<BaseFigure>();
//     ^      ^                      ^          ^
//     |      | _____ Typparameter invariant ____|
//     |                             |
//     | ______ Typen kovariant ______|
```

Mit ***Invarianz*** ist für Generics gemeint, dass ***die Typparameter bei Deklaration und Definition exakt übereinstimmen.*** Diese Forderung ist sehr restriktiv. ***Kovarianz*** bedeutet, dass die Typen der Vererbungshierarchie folgen. Dies ist hier für das Interface `Set` und die konkrete Realisierung `HashSet` gegeben.

Bemerkenswert ist, dass sich die Polymorphie durch Kovarianz bei generischen Definitionen nur auf den Typ der Containerklasse bezieht und nicht auf die Typparameter (und hat damit eigentlich gar nichts mit Generics zu tun)!

Polymorphie für Typen

So weit scheint die obige Definition recht selbstverständlich zu sein. Allerdings muss man bei Generics penibel darauf achten, dass der Typparameter invariant ist. Weil dies – wie wir später noch ausführlicher sehen werden – ein häufiger Stolperstein beim Einsatz von Generics ist, schauen wir uns das nun etwas genauer an. Dazu betrachten wir die Ableitungshierarchie `RectFigure extends BaseFigure`. Man könnte annehmen, dass Folgendes erlaubt wäre:

```
// Compile-Error
final Set<BaseFigure> figures = new HashSet<RectFigure>();
//            ^                                 ^
//            | ____ Typparameter kovariant _____|
```

Für eine solche Definition wären die gezeigten Typparameter nicht mehr invariant, sondern kovariant, da der Typ `RectFigure` von `BaseFigure` abgeleitet ist. Diese Schreibweise scheint intuitiv korrekt, birgt aber Probleme bei der Typsicherheit. Der folgende Abschnitt geht auf drohende Probleme ein.

Mögliche Probleme durch Kovarianz

Erinnern wir uns daran, dass Arrays kovariant und Generics invariant arbeiten. Betrachten wir, was das genau bedeutet. Beginnen wir dabei mit Arrays und schauen, was dort mit Kovarianz gemeint ist.

Wie schon aus Kapitel 3 bekannt, kann ein `Object[]` beispielsweise sowohl ein `String[]` als auch ein `BaseFigure[]` repräsentieren, jedoch nicht gleichzeitig.

```
// kovariante Zuweisung an Object[]; jetzt Referenz auf String[]
Object[] objects = new String[] { "Test1", "Test2" };
objects[0] = "Some string content";
```

Obwohl im Listing das `Object[]` zunächst mit einem `String[]` initialisiert wird, kann später im Programmablauf problemlos eine Zuweisung mit dem Typ `BaseFigure[]` an das Array erfolgen:

```
// kovariante Zuweisung an Object[]; jetzt Referenz auf BaseFigure[]
objects = new BaseFigure[10];
// Laufzeitfehler: java.lang.ArrayStoreException: java.lang.String
objects[0] = "This is not a BaseFigure";
```

Im Listing sehen wir, dass an Position 0 des Arrays ein String zugewiesen wird, ohne dass dies einen Fehler beim Kompilieren auslöst: Semantisch ist die gezeigte Aktion natürlich nicht sinnvoll, jedoch finden für Arrays mit dem Basistyp `Object[]` zur Kompilierzeit keine Typprüfungen statt. Eine derartige Array-Referenz kann zur Laufzeit demnach beliebige, nicht primitive Array-Typen repräsentieren, etwa `String[]` oder auch `BaseFigure[]`. Typverstöße oder mögliche Zuweisungsfehler werden erst zur Laufzeit erkannt. Daher benötigen und besitzen Arrays zur Laufzeit Typinformationen über die in ihnen gespeicherten Elemente. Zuweisungen an die Array-Elemente werden erst zur Laufzeit auf Typsicherheit geprüft und lösen bei Inkompatibilitäten eine `ArrayStoreException` aus. Eine solche Fehlverwendung von Arrays habe ich aber in über 15 Berufsjahren bisher nicht gesehen.

Trotzdem war Sun beim Entwurf der Generics angetreten, eine möglichst wasserdichte Typprüfung zu erreichen. Nehmen wir einmal an, wir könnten auch für Generics kovariante Definitionen vornehmen. Damit wäre dann etwa Folgendes möglich:

```
// Compile-Error
final List<Object> names = new ArrayList<String>();
//            ^                          ^
//            | __ Typparameter kovariant ___|

names.add(new RectFigure());  // Typinkompatibilität
```

Ebenso wie für Arrays sollte dies dann zur Laufzeit zu einem Fehler führen, wenn man einer `List<Object>` mal eine `List<String>` oder `List<BaseFigure>` zuweist und Elemente hinzufügt. Um den Typverstoß aber erkennen zu können, müsste man zur Laufzeit Typinformationen zu den gespeicherten Elementen besitzen. Nun wurde aber beim Entwurf von Generics die Entscheidung getroffen, Generics mithilfe von Type Erasure umzusetzen. Aufgrund dessen ist eine Typprüfung erst zur Laufzeit keine Option, da nach dem Kompilieren keine Typinformation mehr vorhanden ist, die ausgewertet werden könnte. Wäre obige kovariante Definition also erlaubt, so könnte der Compiler hier nicht mehr sicherstellen, dass zur Laufzeit keine Typfehler auftreten. Daher führt die gezeigte Definition bereits beim Kompilieren zu einem Fehler.

Weil Kovarianz als potenziell gefährlich eingestuft wurde und es ein wesentliches Ziel beim Entwurf von Generics war, Laufzeitfehler zu vermeiden, gilt hier eine strengere Forderung, nämlich die nach ***Invarianz*** der Typparameter. Manchmal ist Kovarianz allerdings nützlich, wie wir nun sehen werden.

Warum ist Kovarianz so wünschenswert?

Kovarianz ist immer dann wünschenswert, wenn eine »is-a«-Beziehung zwischen den zu speichernden Typen besteht. Die mit Generics standardmäßig realisierte Invarianz erschwert dann den Umgang.

Betrachten wir das anhand eines Beispiels und nutzen zur Datenspeicherung sowohl ein Array `BaseFigure[]` als auch eine Liste `List<BaseFigure>` wie folgt:

```
final BaseFigure[] arrayOfFigures = new BaseFigure[10];
final List<BaseFigure> listOfFigures = new ArrayList<BaseFigure>();
```

Nehmen wir weiterhin an, eine Methode `printInfo(BaseFigure)` zur Ausgabe von Informationen zu einer Figur sei in einer Utility-Klasse `FigureUtilities` für einzelne Elemente vom Typ `BaseFigure` folgendermaßen definiert:

```
public void printInfo(final BaseFigure figure)
{
    figure.printDetails();
}
```

Schauen wir uns nun an, wie einfach es ist oder auf welche Probleme man stößt, wenn

man die Utility-Klasse so erweitern soll, dass eine überladene Methode für Arrays sowie auch für Collections von Figuren genutzt werden kann. Die geforderte Funktionalität lässt sich für Arrays wie folgt leicht realisieren:

```
public void printInfo(final BaseFigure[] figures)
{
    for (final BaseFigure figure : figures)
        printInfo(figure);
}
```

Für Listen schreibt man analog:

```
public void printInfo(final List<BaseFigure> figures)
{
    for (final BaseFigure figure : figures)
        printInfo(figure);
}
```

In beiden Varianten werden alle Elemente durchlaufen und für jede Figur polymorph die Methode `printInfo(BaseFigure)` aufgerufen. Das ist in beiden Fällen gleich. Aber es gibt einen funktionalen Unterschied, wie wir dies nachfolgend sehen werden.

Beginnen wir damit, die Methoden mit unterschiedlichen Eingabewerten aufzurufen. Dazu erzeugen wir Objekte unterschiedlicher Figurenklassen. Betrachten wir dies für folgende drei Arrays mit Kreisen, Rechtecken und einem Mix daraus:

```
public static void main(final String[] args)
{
    // Definitionen von Arrays unterschiedlichen Typs
    final CircleFigure[] circles = { new CircleFigure(), new CircleFigure() };
    final RectFigure[] rects = { new RectFigure(), new RectFigure() };
    final BaseFigure[] figures = { new CircleFigure(),  new RectFigure() };

    // Ausgabe über die Methode printInfo(BaseFigure[])
    printInfo(circles);
    printInfo(rects);
    printInfo(figures);
}
```

Listing 5.36 *Ausführbar als* **'GENERICSARRAYPOLYMORPHIEEXAMPLE'**

Für Arrays des jeweiligen Figurentyps bzw. des Basistyps `BaseFigure` kann die zuvor definierte Methode `printInfo(BaseFigure[])` problemlos aufgerufen werden.

Das scheint alles ganz einfach zu sein. Versuchen wir daher, das Ganze auf generische Collections zu übertragen. Intuitiv könnte man auf die im folgenden Beispiel dargestellte Umsetzung kommen, die analog zur vorherigen Realisierung arbeitet, jedoch mithilfe von `Arrays.asList(T...)` zunächst eine Umwandlung der typisierten Arrays in typsichere Listen vornimmt, um dann die Methode `printInfo(List<BaseFigure>)` aufzurufen:

```
public static void main(final String[] args)
{
    // Identische Definitionen
    final CircleFigure[] circles = { new CircleFigure(), new CircleFigure() };
    final RectFigure[] rects = { new RectFigure(), new RectFigure() };
```

```
    final BaseFigure[] figures = { new CircleFigure(), new RectFigure() };

    // Umwandlung Array -> Liste (Arrays.asList(T…))
    final List<BaseFigure> figureList = Arrays.asList(figures);
    printInfo(figureList);

    // Compile-Error: Type mismatch: cannot convert from
    // List<CircleFigure> to List<BaseFigure>
    // final List<BaseFigure> circleList = Arrays.asList(circles);

    // Compile-Error: The method printInfo(List<BaseFigure>) in the type
    // GenericsArrayPolymorphie2Example is not applicable
    // for the arguments (List<RectFigure>)
    // printInfo(Arrays.asList(rects));
}
```

Listing 5.37 *Ausführbar als* **'GENERICSARRAYPOLYMORPHIE2EXAMPLE'**

Nur das in eine Liste umgewandelte `BaseFigure[]` erlaubt problemlos den Aufruf der Methode `printInfo(List<BaseFigure>)`, da hier der Typparameter invariant ist. Bei den beiden Arrays der spezielleren Typen kommt es zu Kompilierfehlern. Dabei ist es unbedeutend, ob zunächst eine Umwandlung in eine `List<BaseFigure>` erfolgt oder ein direkter Aufruf. Es stellt sich die Frage: »Wie kommt es dazu und wie lösen wir das Problem?«

Problemlösung

Um die mitunter wünschenswerte Kovarianz für Typparameter bei Generics zu ermöglichen, existiert eine spezielle Notation. Allerdings darf aufgrund der Type Erasure und der fehlenden Möglichkeit einer Typprüfung zur Laufzeit nur eine »sichere Form« der Kovarianz unterstützt werden, bei der garantiert wird, dass der verwendete Typ alle Operationen des Basistyps implementiert. Gleiches gilt für Kontravarianz. Für beide kommt eine sogenannte Wildcard zum Einsatz. Die Wildcard '?' erlaubt beliebige Typen und kann zur Kennzeichnung von Ko- und Kontravarianz wie folgt genutzt werden:

1. **Kovarianz** – Die Wildcard '`? extends` **`basetype`**' wird ***Upper Type Bound*** genannt und ermöglicht ***Kovarianz***. Es dürfen dadurch generische Klassen verwendet werden, die Subtypen von `basetype` als Typparameter nutzen.[22]
2. **Kontravarianz** – Die Wildcard '`? super` **`subtype`**' heißt ***Lower Type Bound*** und ermöglicht ***Kontravarianz***. Dadurch sind alle diejenigen generischen Klassen erlaubt, die als Typparameter einen Basistyp von `subtype` besitzen. Es werden für die Typparameter also *alle* Basistypen akzeptiert, jedoch *keine* Subtypen mehr.

Kovarianz – Upper Type Bound

Weil durch die Type Erasure zur Laufzeit keine Typinformationen existieren, ist Kovarianz bei Generics nur mit Einschränkungen möglich. Probleme durch inkompatible Typen müssen bereits zur Kompilierzeit ausgeschlossen werden

können. Typparameter müssen dazu mit der Syntax '`? extends basetype`' explizit als ***kovariant*** gekennzeichnet werden.

Nutzen wir dieses Wissen, um das vorherige Beispiel alternativ sowohl für Figuren vom Typ `CircleFigure` als auch `RectFigure` ohne Fehler kompilieren zu können:

```
final List<? extends BaseFigure> figureList1 = new ArrayList<CircleFigure>();
final List<? extends BaseFigure> figureList2 = new ArrayList<RectFigure>();
```

Einer derart definierten `List<? extends BaseFigure>` kann problemlos eine `ArrayList<CircleFigure>` oder eine `ArrayList<RectFigure>` zugewiesen werden. Worin besteht denn dann der Unterschied zu der Kovarianz bei Arrays? Die Antwort ist einfach: In der Angabe des Typparameters! Zuweisungsfehler können bei Arrays erst zur Laufzeit erkannt werden. Für Generics werden Schreibzugriffe auf Elemente bereits beim Kompilieren ausgeschlossen und problematische kovariante Zuweisungen vom Compiler verboten.

Einfügen von Elementen

Eingangs haben wir erkannt, dass Schreibzugriffe in eine kovariant definierte Collection problematisch sein können. Um Typsicherheit garantieren zu können, werden Schreibzugriffe vom Compiler unterbunden und führen beim Kompilieren zu Fehlern. Dadurch vermeidet man beispielsweise, dass in einer als `ArrayList<RectFigure>` konstruierten Liste (fälschlicherweise) ein `CircleFigure`-Objekt gespeichert werden kann. Als einzige Ausnahme kann immer `null` gespeichert werden, da dieser Wert zu allen Referenztypen zuweisungskompatibel ist.

Zugriff auf Elemente

Für den Zugriff auf Elemente gilt, dass es immer problemlos möglich ist, Referenzen vom Typ `BaseFigure` auszulesen, da die gespeicherten Elemente garantiert mindestens diesen Typ besitzen:

```
final BaseFigure value = figureList.get(index);
```

Das Auslesen eines Subtyps von `BaseFigure`, z. B. `CircleFigure`, ist jedoch nicht möglich, da die durch '`? extends BaseFigure`' repräsentierte, konkrete Klasse unbekannt ist. Damit wird etwa verhindert, dass aus einer `ArrayList<RectFigure>` ein `CircleFigure`-Objekt gelesen werden kann:

```
// Compile-Error: Type mismatch: cannot convert from capture#3-of ? extends
// BaseFigure to RectFigure
final RectFigure value = figureList.get(index);
```

Kontravarianz – Lower Type Bound

Die Forderung nach Kovarianz ist relativ natürlich und basiert auf dem

Substitutionsprinzip. Der Einsatz von Kontravarianz ist weniger eingängig, weil diese eine Kompatibilität entgegengesetzt zur Vererbungshierarchie der Typparameter ermöglicht. Kontravarianz bei Generics wird durch die Notation '`? super subtype`' ausgedrückt:

```
final List<? super BaseFigure> figureList = new ArrayList<BaseFigure>();
```

Im Speziellen ist sogar folgende Zuweisung möglich:

```
final List<? super BaseFigure> figureList = new ArrayList<Object>();
```

Durch die Beschränkung des Typs nach »unten« führt folgende Definition zu einem Fehler beim Kompilieren:

```
// Compile-Error
// cannot convert ArrayList<RectFigure> to ArrayList<? super BaseFigure>
final List<? super BaseFigure> figureList = new ArrayList<RectFigure>();
```

Einfügen von Elementen

Für eine kontravariant deklarierte Liste ist das Hinzufügen von Elementen möglich. Im nachfolgenden Beispiel nutzen wir als begrenzenden Typ `BaseFigure` und fügen mit `RectFigure` eine Spezialisierung davon ein:

```
final List<? super BaseFigure> figureList = new ArrayList<BaseFigure>();
figureList.add(new RectFigure());
```

In diesem Beispiel fällt eine Besonderheit auf, die für Verwirrung sorgen kann. Obwohl die Liste mit dem Typparameter `BaseFigure` definiert wurde, kann tatsächlich (zunächst irritierenderweise) jeder Subtyp von `BaseFigure` gespeichert werden, wie dies bereits im obigen Beispiel durch den Aufruf von `figureList.add(new RectFigure())` gezeigt wurde. Folgendes ist wichtig zu wissen: ***Die Varianz bezieht sich auf die Typparameter der generischen Klasse (`ArrayList<E>`) und nicht auf die Typen der tatsächlich gespeicherten Elemente.***

Zugriff auf Elemente

Beim Einsatz von Kontravarianz kann der Compiler nicht feststellen, von welchem Typ die Elemente der Collection tatsächlich sind. ***Aus einer solchen Collection kann deshalb nicht typsicher gelesen werden***.

```
// Compile-Error: Type mismatch: cannot convert from capture#3-of ? super
// BaseFigure to RectFigure
final RectFigure value = list.get(index);
```

Allerdings besitzen die enthaltenen Elemente immer den Basistyp `Object`, wodurch folgender Lesezugriff möglich ist:

```
final Object obj = list.get(index);
```

Invarianz

Bekanntermaßen sind bei der Invarianz für Generics die Typparameter bei

Deklaration und Definition gleich. Im folgenden Beispiel ist dies für eine `ArrayList` von `BaseFigure`-Objekten gezeigt:

```
final List<BaseFigure> graphicObjects = new ArrayList<BaseFigure>();
```

Eine auf diese Weise definierte `ArrayList<BaseFigure>` lässt sich so nutzen, als ob die Signaturen der Methoden tatsächlich für `BaseFigure`-Objekte definiert wären: Beim Auslesen mit `get(int)` liefert sie ein Objekt vom Typ `BaseFigure`. Beim Hinzufügen lässt sich mit `add(BaseFigure)` ein Element vom Typ `BaseFigure` (bzw. sogar Subtypen davon) speichern:

```
final BaseFigure baseFigure = graphicObjects.get(index);
graphicObjects.add(new RectFigure());
```

Info: Invarianz und Auto-Boxing

Für Collections von Wrapper-Klassen, etwa `ArrayList<Long>`, erfolgt automatisch Auto-Boxing/-Unboxing beim Hinzufügen oder Auslesen von Elementen:

```
final List<Integer> integerList = new ArrayList<>();
integerList.set(7, 4711);                          // Auto-Boxing
final int valueAtPosition7 = integerList.get(7);   // Auto-Unboxing
```

Auswirkungen der Varianzformen

Die Vorstellung der Varianzformen war möglicherweise ein wenig theoretisch. Der Nutzen in der Praxis lässt sich gut an einem Beispiel verdeutlichen.

Nehmen wir an, die bereits kurz genannte Utility-Klasse `FigureUtilities` soll um eine Methode erweitert werden, mit der man grafische Figuren aus einer Liste in eine andere Liste kopieren kann. Eine erste Variante wäre, die Methoden folgendermaßen zu realisieren:

```
public static void copy(final List<BaseFigure> src,
                        final List<BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

So plausibel diese Realisierung zunächst auch aussehen mag, sie besitzt einige Einschränkungen und ist wenig hilfreich. Schauen wir nun auf mögliche Probleme und was man dagegen unternehmen kann.

Problem Invarianz

Das erste Problem der gezeigten Realisierung besteht in der invarianten Definition der Typparameter von Quelle und Ziel:

```
final List<BaseFigure> src = new ArrayList<BaseFigure>();
// Füllen der Liste …
final List<BaseFigure> dest = new ArrayList<BaseFigure>();
```

```
FigureUtilites.copy(src, dest);
```

Dadurch sind wir darauf beschränkt, beim Kopieren jeweils nur Listen mit identischem Typparameter zu nutzen. Das ist aber störend, wenn man Subtypbeziehungen und Polymorphie nutzt.

Rekapitulieren wir kurz das Verhalten für Referenzen und Arrays: Man kann z. B. eine Referenz `CircleFigure` an den Typ `BaseFigure` zuweisen und so auf allgemeinerer Ebene arbeiten. Für Arrays wäre es auch möglich, einem `BaseFigure[]` ein spezielleres `CircleFigure[]` zuzuweisen. Wie für Arrays wäre dies auch für generische Container wünschenswert: Ein möglicher Nutzer der Utility-Klasse könnte dann eine `List<CircleFigure>` in eine Liste mit dem allgemeineren Typ `BaseFigure` übertragen. Das ist jedoch mit der zuvor gezeigten `copy()`-Methode nicht möglich, stattdessen kommt es zu einem Kompilierfehler, wie dies folgende Zeilen zeigen:

```
final List<CircleFigure> src = new ArrayList<CircleFigure>();
// Füllen der Liste …
final List<BaseFigure> dest = new ArrayList<BaseFigure>();

// Compile-Error: The method copy(List<BaseFigure>, List<BaseFigure>)
// in the type FigureUtilites is not applicable for the arguments
// (List<CircleFigure>, List<BaseFigure>)
FigureUtilites.copy(src, dest);
```

Motivation für Kovarianz für den Parameter der Quelle

Als Abhilfe nutzen wir eine kovariante Definition folgendermaßen:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

Damit lässt sich die eben noch an einem Kompilierfehler scheiternde Kopieraktion von einer `ArrayList<CircleFigure>` in eine `ArrayList<BaseFigure>` durchführen.

Kovarianz auch für den Parameter des Ziels?

Intuitiv fragt man sich: »Wäre es nicht praktisch, auch den Parameter für das Ziel kovariant zu definieren?« Das klingt eigentlich ganz vernünftig. Damit ergäbe sich folgende Realisierung:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<? extends BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);   // Compile-Error
}
```

Tatsächlich kommt es zu einem Kompilierfehler bei `add(BaseFigure)`. Das ist auch gut so, denn dadurch verhindert man möglicherweise problematische Schreibzugriffe, wie wir dies bereits bei der Diskussion der Kovarianz bei Arrays besprochen haben. Nehmen wir kurz an, diese kovariante Methodendefinition wäre erlaubt, dann wäre z. B. folgende Kopie einer Liste von Kreisen in eine Liste von Rechtecken möglich:

```
final List<CircleFigure> src = new ArrayList<>();
final List<RectFigure> dest = new ArrayList<>();

FigureUtilites.copy(circles, rects);
```

Das würde aber zu Typfehlern führen, weil man Elemente vom Typ `CircleFigure` in eine Liste, festgelegt auf Elemente vom Typ `RectFigure`, einfügen würde. Um Derartiges grundsätzlich zu verhindern, sind Schreibzugriffe für kovariant definierte Containerklassen nicht erlaubt.

Motivation für Kontravarianz für den Parameter des Ziels

Die initial genutzte Invarianz bot wenig Flexibilität. Auch für Kovarianz haben wir erkannt, dass es keinen Sinn ergibt, das Kopierziel derart zu definieren. Was ist nun mit Kontravarianz?

Beim Kopieren wäre es teilweise durchaus wünschenswert, ein allgemeineres Ziel angeben zu können, also statt einer `List<BaseFigure>` etwa eine `List<Object>`. Es ist typsicher, wenn man etwa eine Liste von Rechteckfiguren in einer Liste speichert, deren Typparameter `BaseFigure` bzw. `Object` ist. Durch Kontravarianz wird es möglich, auch eine auf einen weniger speziellen Typ festgelegte generische Containerklasse als Kopierziel zu nutzen. Das erlaubt es, die Zieldatenstruktur flexibler angeben zu können. Betrachten wir die neue Variante `copy()`-Methode, die dies nutzt:

```
public static void copy(final List<? extends BaseFigure> src,
                        final List<? super BaseFigure> dest)
{
    for (final BaseFigure figure : src)
        dest.add(figure);
}
```

Mit dieser Methode haben wir unser Ziel einer flexiblen Kopiermethode erreicht: Wir können mithilfe der obigen `copy()`-Methode eine typsichere Kopie erstellen, und dabei sowohl speziellere Eingabe- als auch allgemeinere Rückgabetypen nutzen. Dadurch wird die Handhabung für Klienten vereinfacht, sodass nun etwa folgende Aufrufe möglich sind:

```
// Definition von Zieldatenstrukturen
final List<BaseFigure> baseFigures = new ArrayList<>();
final List<Object> objects = new ArrayList<>();

// Kovariante Eingabe und kontravarianter Zielparameter
FigureUtilites.copy(circles, baseFigures);
FigureUtilites.copy(rects, objects);
```

Sind wir damit wirklich am Ziel? Nein, aber fast. Was fehlt denn noch? Bei der gesamten Diskussion um Super- und Subtypen haben wir einen Fall bisher nicht betrachtet: Man sollte jede Liste eines speziellen Typs, etwa `CircleFigure`, selbstverständlich auch wieder in eine `List<CircleFigure>` kopieren können. Obwohl das banal klingt, ist es allerdings momentan mit der realisierten `copy()`-Methode noch nicht möglich. Stattdessen kommt es zu dem im nachfolgenden Listing gezeigten Kompilierfehler:

```
final List<CircleFigure> circlesSrc = new ArrayList<>();
// Füllen der Liste …
final List<CircleFigure> circlesDest = new ArrayList<>();

// Compile-Error: The method copy(List<? extends BaseFigure>,
// List<? super BaseFigure>) in the type FigureUtilites is not
// applicable for the arguments List<CircleFigure>, List<CircleFigure>)
FigureUtilites.copy(circlesSrc, circlesDest);
```

Korrekterweise wird erkannt, dass der Typ `List<CircleFigure>` des Ziels nicht dem Typ `List<? super BaseFigure>` zugewiesen werden kann. Es ist aber eine natürliche Forderung und wünschenswert, Elemente aus Listen des gleichen Typs kopieren zu können. Um dies zu ermöglichen, nutzen wir die bereits vorgestellte Syntax für Typeinschränkungen für statische Methoden wie folgt:

```
public static <T extends BaseFigure> void copy(final List<T> src,
                                                final List<? super T> dest)
{
    for (final T figure : src)
        dest.add(figure);
}
```

Nach dieser Modifikation haben wir nun unter Einsatz von Ko- und Kontravarianz eine typsichere, generische Kopiermethode implementiert.

Schlussfolgerung

In diesem Beispiel haben wir diverse kleinere Fallstricke und Besonderheiten beim Einsatz von Generics und Collections, aber auch Möglichkeiten zur Lösung kennengelernt. Damit haben Sie ein recht gutes Wissen erlangt, um eigene Experimente zu starten.

Der Einsatz von Kovarianz bietet sich an, um Eingaben allgemeiner zu gestalten. Kontravarianz erlaubt dies für Rückgaben bzw. Zuweisungen. Um Typsicherheit zu gewährleisten, sind Schreibzugriffe bei Kovarianz verboten, und durch den Compiler wird ein Read-only-Umgang forciert. Der Einsatz von Kontravarianz verbietet Lesezugriffe, dafür werden Schreibzugriffe ermöglicht. Tabelle 5-1 fasst die obigen Aussagen zum Zugriff und Einfügen von Elementen zusammen.

Tabelle 5-1 *Konsequenzen der Varianzformen*

Varianzform	Zugriff	Einfügen
Kontravarianz	- (nur `Object`)	✓
Kovarianz	✓	- (nur `null`)
Invarianz	✓	✓

Tipp: Varianz nur für Eingabeparameter, niemals für Rückgabewerte

Joshua Bloch gibt in »Effective Java« [9] den Hinweis, die beiden Varianzformen

- ? extends T (Kovarianz)
- ? super T (Kontravarianz)

nur für Eingabeparameter, niemals aber für Rückgabetypen zu verwenden. Der Grund ist folgender: Für Methodeneingaben erreicht man für mögliche Aufrufer mehr Flexibilität. Für Rückgabewerte gilt dies nicht. Im Gegenteil: Es erschwert sogar die Handhabung, da die Wildcards dann auch in den Sourcecode der Klienten aufgenommen werden müssen.

Info: Generics und Wildcards

Nachdem wir die Verwendung von Wildcards kennengelernt und diese bereits in einigen Beispielen eingesetzt haben, ist uns aufgefallen, dass deren Verwendung mit einigen Merkwürdigkeiten in der Schreibweise verbunden ist: Dies gilt im Speziellen, wenn man komplexere generische Typen oder Methoden definiert, die eine Kombination aus `extends` und `super` nutzen, etwa wie folgt:

```
static <T> int binarySearch( List<? extends Comparable<? super T≫ list,
                             T key)
```

Eine interessante Diskussion finden Sie unter `http://www.artima.com/weblogs/viewpost.jsp?thread=222021`.
Einen Videovortrag von Joshua Bloch zu diesem Thema kann man sich unter `http://beta.parleys.com/#id=116&st=5&sl=37` anschauen.

Spezielle Syntax bei Generics

Wir haben mit Ko- und Kontravarianz und der ?-Notation schon eine recht kryptische Syntax bei Generics kennengelernt. Leider gibt es sogar noch eine Steigerung. Diese kommt dann zum Einsatz, wenn man Methoden mit generischen Parametern aufrufen möchte, etwa folgendermaßen:

```
public void print( final String title,
```

```
                    final List<TimeSeriesData> timeSeriesData);
```

Diese Signatur sieht harmlos aus, aber was passiert, wenn wir eine leere Liste als Parameter übergeben wollen? Dazu nutzen wir `Collections.emptyList()` wie folgt:

```
print("title", Collections.emptyList());
```

Auch wenn dieser Aufruf korrekt und logisch aussieht, so kommt es durch die Type Erasure zu folgender Fehlermeldung: »`The method print(String, List<TimeSeriesData>) in the type XYZ is not applicable for the arguments (String, List<Object>)`«. Als Abhilfe muss man den Typ direkt angeben, und zwar mithilfe folgender kovarianter Syntax:

```
print(…, Collections.<TimeSeriesData>emptyList());
```

Unterschied zwischen `List`, `List<Object>` und `List<?>`

Nachdem nun die Formen der Varianz bekannt sind, möchte ich abschließend auf einige Unterschiede in den Deklarationen `List`, `List<Object>` und `List<?>` eingehen. Obwohl alle recht ähnlich aussehen, besitzen sie doch voneinander abweichende Eigenschaften, die wir nun erkunden.

`List` vs. `List<Object>`

Etwas vereinfachend kann man sagen, dass die Deklarationen `List` und `List<Object>` nahezu gleich sind. Derart definiert können Elemente beliebigen Typs gespeichert werden. Der Unterschied besteht lediglich darin, welche Zuweisungen an die so definierten Referenzvariablen vorgenommen werden können: Eine Deklaration `List<Object>` erlaubt tatsächlich nur, dass exakt so definierte Listen zugewiesen werden. Das folgt aufgrund der Invarianz:

```
// beim Auslesen nicht typsicher
final List plainList = new ArrayList();
plainList.add(new Integer(4711));
plainList.add("Test");
final Integer sum = plainList.get(0) +
                    plainList.get(1); // Laufzeitfehler, weil
                                      // Integer und String enthalten sind

// ähnlich wie List, total generisch
final List<Object> objectList = new ArrayList<Object>();
// final List<Object> objectList = new ArrayList<String>(); // Compile-Error
objectList.add("Test");
objectList.add(new Integer(4711));
```

`List<Object>` vs. `List<?>`

Im Gegensatz zu den gerade betrachteten Deklarationen bestehen zwischen den Deklarationen `List<Object>` und `List<?>` sehr große Unterschiede. Wie eben erwähnt, können in einer `List<Object>` Elemente beliebigen Typs gespeichert

werden, aufgrund der Invarianz jedoch nur Listen zugewiesen werden, die auch exakt den generischen Typ `Object` nutzen.

Für die `List<?>` gilt das Gegenteil: Die Wildcard '?' erlaubt beliebige Typen und somit können Listen mit beliebigen Typparametern zugewiesen werden:

```
final List<?> anyTypeList1 = new ArrayList<Person>();
final List<?> anyTypeList2 = new ArrayList<CircleFigure>();

// typsichere Liste, bei der der Typ unbekannt ist
final List<?> anyTypeList = new ArrayList<String>();

// Compile-Error: The method add(capture#1-of ?) in the type
// List<capture#1-of ?> is not applicable for the arguments (String)/(Object)
// anyTypeList.add("Test");
// anyTypeList.add("Object");

anyTypeList.add(null);     // erlaubt
```

Allerdings kann aufgrund der fehlenden Typinformation der zu speichernden Elemente keine Typsicherheit garantiert werden. ***Für `List<?>` ist daher ein Einfügen von Elementen generell nicht möglich***. Es kommt zu der im Listing gezeigten Fehlermeldung.

Zusammenfassung

Fassen wir alles nochmal zusammen: Wie eingangs erwähnt, verhält sich eine `List<Object>` nahezu wie der nicht typisierte Raw Type `List` und definiert eine ***heterogene Liste***. Allerdings kann einer `List<Object>` aufgrund der Forderung nach Invarianz *keine* `List<String>` zugewiesen werden; es können der `List<Object>` aber durchaus Stringobjekte hinzugefügt werden.

Mit einer `List<?>` wird eine ***homogene Liste*** beschrieben: Es kann zwar eine `List<String>` oder `List<Person>` zugewiesen werden; es können aber *keine* Stringobjekte bzw. `Person`-Objekte hinzugefügt werden. Dies klingt nach einer starken Einschränkung. Trotzdem ist `List<?>` aber für Utility-Methoden nützlich, die nur auf der Listenfunktionalität unabhängig vom Typ arbeiten sollen, etwa `size(List<?>)`.

5.5 Fallstricke im Collections-Framework

Beim Collections-Framework handelt es sich um eine gelungene und umfangreiche Sammlung gebräuchlicher Algorithmen und Datenstrukturen. Allerdings existieren doch kleinere Fallstricke, die man kennen sollte, um nicht über sie zu stolpern.

5.5.1 Wissenswertes zu Arrays

Arrays und Vergleiche

Die für verschiedene primitive Typen und den Typ `Object[]` überladene Methode `Arrays.equals()` verhält sich für verschachtelte Arrays nicht so, wie man es intuitiv erwarten würde. Folgendes Listing zeigt das Problem: Es werden zwei Arrays verglichen, die jeweils gleiche Arrays von Stringliteralen speichern. In diesem Beispiel werden zur Verdeutlichung die Indexpositionen der Arrays als Wert gespeichert:

```
public static void main(final String[] args)
{
    final String[][] array1 = { { "0.0", "0.1" }, { "1.0", "1.1" } };
    final String[][] array2 = { { "0.0", "0.1" }, { "1.0", "1.1" } };

    final boolean arrayEquals = Arrays.equals(array1, array2);
    final boolean deepEquals = Arrays.deepEquals(array1, array2);

    System.out.println("equals = " + arrayEquals);      // false !!!
    System.out.println("deepEquals = " + deepEquals);   // true
}
```

Listing 5.38 *Ausführbar als* **'ARRAYCOMPAREEXAMPLE'**

Ganz intuitiv erwartet man bei einem Vergleich von Arrays mit offensichtlich gleichem Inhalt auch das Feststellen von Gleichheit. Tatsächlich werden die obigen Arrays als unterschiedlich erkannt. Dies ist dadurch bedingt, dass die Implementierung von `Arrays.equals(Object[], Object[])` die enthaltenen Subarrays nicht rekursiv vergleicht. Das wurde aus Performance-Gründen so realisiert. ***Zum korrekten Vergleich verschachtelter Arrays ist daher immer die Methode `Arrays.deepEquals(Object[], Object[])` zu verwenden.***[23]

Ähnliche Fallstricke lauern bei der Berechnung des Hashwerts über `Arrays.hashCode(Object[])` und bei der Erzeugung einer Stringrepräsentation mit `Arrays.toString(Object[])`. Es existieren in der Utility-Klasse `Arrays` korrespondierende Methoden `deepHashCode(Object[])` und `deepToString(Object[])`.

Rückgabe von Arrays und die Gefahr von Inkonsistenzen

Zur Bereitstellung von Berechnungsergebnissen oder einer Menge vordefinierter Werte setzt man mitunter Arrays ein, ohne sich darüber allzu viele Gedanken zu machen. Allerdings besteht bei der Rückgabe von Arrays auf interne Daten die Gefahr von Inkonsistenzen, weil gespeicherte Werte unerwartet verändert werden können.

Folgendes Beispiel macht dies deutlich und zeigt mögliche Probleme beim Zugriff auf ein Array `CAPITAL_CITIES` vom Typ `String[]`. Es sind folgende zwei Zugriffsmethoden `getCities()` und `getCityIterator()` definiert:

```
public final class BesserIteratorAlsArray
{
    private static final String[] CAPITAL_CITIES = new String[]
                                { "Berlin", "London", "Paris", "Wien" };

    public static final String[] getCities()
```

```
    {
        return CAPITAL_CITIES;
    }

    public static final Iterator<String> getCityIterator()
    {
        return Arrays.asList(CAPITAL_CITIES).iterator();
    }

    public static void main(final String[] args)
    {
        System.out.println("CITIES " + Arrays.toString(CAPITAL_CITIES));

        final String[] cities = getCities();
        // unerwartete Modifikation (auch im Original-Array!)
        cities[1] = "London has changed!";

        System.out.println("CITIES " + Arrays.toString(cities));
        System.out.println("CITIES " + Arrays.toString(CAPITAL_CITIES));

        // keine Modifikation möglich
        final Iterator<String> cityIterator = getCityIterator();
        while (cityIterator.hasNext())
            System.out.println(cityIterator.next());
    }
}
```

Listing 5.39 *Ausführbar als* **'BESSERITERATORALSARRAY'**

In diesem Beispiel wird die Ausgabe der Städte auf zwei unterschiedliche Arten realisiert: Beim Zugriff über die Methode `getCities()` wird zwar nur eine Referenz auf ein `String[]` zurückgeliefert, allerdings werden unerwartet Modifikationen im `private static final` definierten Array `CAPITAL_CITIES` möglich. Die Begründung dafür ist einfach: Zwar ist die Referenz auf das Array `final` und damit unveränderlich, jedoch gilt das nicht für die dort gespeicherten Referenzen. Das spiegelt sich in folgender gekürzter Konsolenausgabe wider:

```
CITIES [Berlin, London, Paris, Wien]
CITIES [Berlin, London has changed!, Paris, Wien]
[…]
```

Der Eintrag "`London`" wird unerwartet zu "`London has changed!`". Demnach gilt: ***Die Rückgabe von Referenzen auf Arrays birgt die Gefahr von Modifikationen und verschlechtert die Datenkapselung.***

Iterator als Abhilfe

Die zweite Variante nutzt einen Iterator zum Zugriff. Das führt zu einer besseren Kapselung, verhindert automatisch Modifikationen der Zusammensetzung als auch der Daten und löst damit das Problem ungewünschter Änderungen im Array: Allerdings gilt dies nur in diesem Fall für unveränderliche Stringobjekte. Für andere gespeicherte Objektreferenzen erreicht man so lediglich einen Schutz ähnlich zu dem der `unmodifiable`-Wrapper: Die Zusammensetzung des Arrays lässt sich nicht ändern, aber die Daten der gespeicherten Objekte möglicherweise schon.

Arrays und `clone()`

Arrays bieten eine öffentliche `clone()`-Methode, die eine flache Kopie eines Arrays erzeugt: Das bedeutet, dass jedoch lediglich ein neues Array mit der Kopie der ersten Ebene der Elemente erzeugt wird. Die Subarrays werden zwischen den Arrays »geteilt«. Betrachten wir dazu ein Beispiel:

```
public final class ArrayCloneExample
{
    private static final String[][] original = { { "0.0", "0.1" },
                                                 { "1.0", "1.1" } };

    // Kopie des Arrays erzeugen
    private static final String[][] clone = original.clone();

    public static void main(final String[] args)
    {
        // Protokolliere Ausgangszustand
        System.out.println("before modification:");
        System.out.println("original:  " + Arrays.deepToString(original));
        System.out.println("clone:     " + Arrays.deepToString(clone));

        // Verändere Einträge im Clone
        System.out.println("after modification:");
        clone[0][1] = "New 0.1";
        clone[1] = new String[] { "New Sub-Array 1" };

        System.out.println("original:  " + Arrays.deepToString(original));
        System.out.println("clone:     " + Arrays.deepToString(clone));
    }
}
```

Listing 5.40 *Ausführbar als* **'ARRAYCLONEEXAMPLE'**

Es kommt zu folgender Ausgabe:

```
before modification:
original: [[0.0, 0.1], [1.0, 1.1]]
clone:    [[0.0, 0.1], [1.0, 1.1]]
after modification:
original: [[0.0, New 0.1], [1.0, 1.1]]
clone:    [[0.0, New 0.1], [New Sub-Array 1]]
```

Hierzu muss allerdings die Methode `Arrays.deepToString(String[])` verwendet werden, da es sich um verschachtelte `String`-Arrays handelt.

Das Klonen führt zu einer Kopie der ersten Array-Ebene. In diesem Beispiel sind demnach die Arrays `original` und `clone` verschieden, aber die Subarrays sind gleich. Dadurch kann man mit einer Zuweisung an `clone[1]` unabhängig einen anderen Wert setzen. Ein Schreibzugriff auf die zweite Ebene `clone[0][1]` verändert allerdings beide Arrays. Abbildung 5-12 zeigt dies.

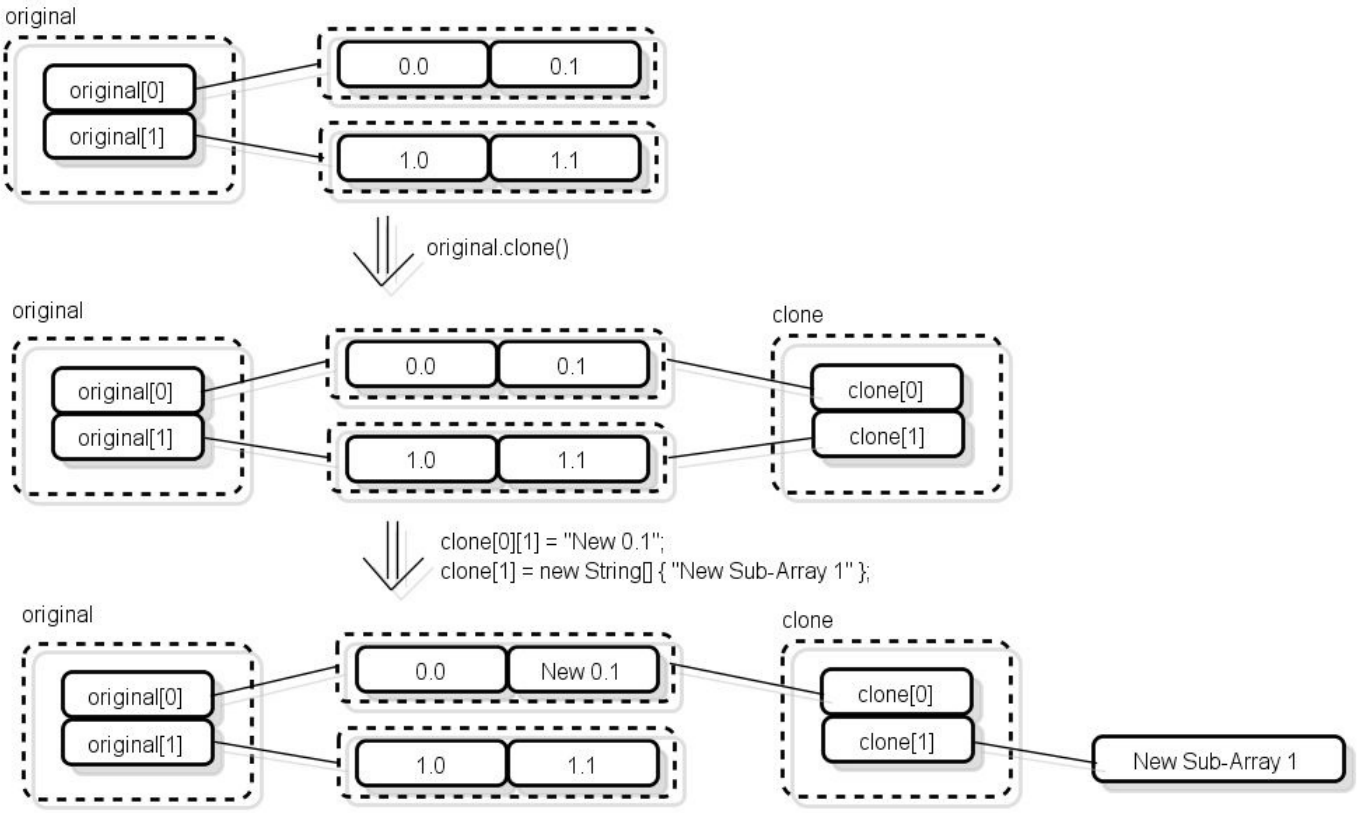


Abbildung 5-12 *Arbeitsweise der Methode* `clone()` *für Arrays*

5.5.2 Wissenswertes zu `Stack`, `Queue` und `Deque`

Bislang wurden die Klassen `Stack<E>`, `Queue<E>` und `Deque<E>` nur am Rande erwähnt. In diesem Abschnitt erfolgt zunächst eine kurze Einführung in deren Arbeitsweise und anschließend gehe ich auf einige Fallstricke ein.

Die Arbeitsweise der Klasse `Stack<E>` (zu deutsch: Stapel) kann man sich wie einen Stapel Papier in einer Schreibtischablage vorstellen. Neue Aufträge werden oben abgelegt und der oberste Auftrag wird als Nächstes bearbeitet. Diese Bearbeitungsreihenfolge ist in der Informatik unter dem Begriff ***LIFO*** (***Last-In-First-Out***) bekannt.

Die Klasse `Queue<E>` repräsentiert eine Warteschlange. Man kann sich die Verarbeitung wie beim Anstellen an der Kasse im Supermarkt vorstellen. Neue Wartende reihen sich immer hinten in die Schlange der Wartenden ein und jeder kommt der Reihe nach dran. Diese Art der Bearbeitung wird in der Informatik ***FIFO*** (***First-In-First-Out***) oder auch ***FCFS*** (***First-Come-First-Serve***) genannt. Tatsächlich kennt man es aus der Realität, dass sich immer mal wieder Personen vordrängeln oder aber von anderen vorgelassen werden. Diesen Sachverhalt kann man durch sogenannte ***Prioritätswarteschlangen*** abbilden. Sie werden durch das Interface `PriorityQueue<E>` realisiert. Dort übersteuern Prioritäten die Bedienreihenfolge. Die Arbeitsweise der Datenstrukturen wird in Abbildung 5-13 angedeutet.

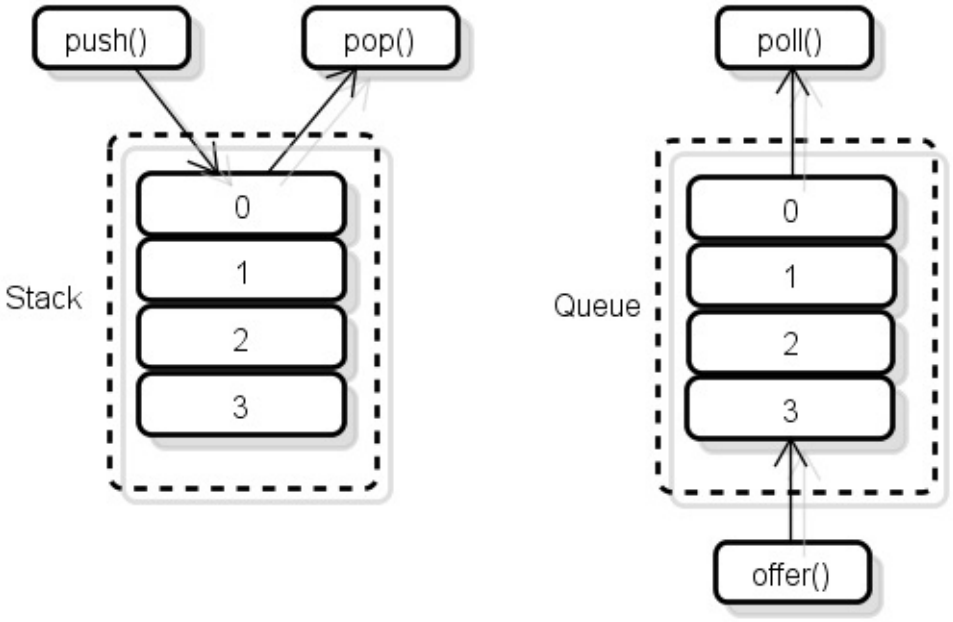


Abbildung 5-13 *Arbeitsweise von Stacks und Queues*

Fallstricke in der Klasse `Stack`

Die Klasse `Stack<E>` wird durch eine Implementierungsvererbung von der Klasse `Vector<E>` realisiert. Sie bietet daher alle Methoden ihrer Basisklasse `Vector<E>`. Darüber hinaus werden folgende fünf stackspezifischen Methoden bereitgestellt:

- `E push(E element)` – Legt ein Element oben auf den Stack und gibt es zurück.
- `E peek()` – Liefert das oberste Element, *ohne* es aus dem Stack zu entfernen. Falls keine Elemente existieren, kommt es zu einer `java.util.EmptyStackException`.
- `E pop()` – Gibt das oberste Element zurück und *entfernt* es aus dem Stack. Wirft eine `EmptyStackException`, falls keine Elemente vorhanden sind.
- `boolean empty()` – Prüft, ob der Stack leer ist.
- `int search(Object object)` – Sucht nach dem übergebenen Element und gibt dessen Position zurück. Diese Funktionalität ist nützlich, um herauszufinden, wie weit ein Element noch von der obersten Position entfernt ist. Die Suche liefert eine Position, die 1-basiert ist: Für das Element ganz oben auf dem Stack gibt die Methode den Wert 1 zurück.

Durch die Implementierungsvererbung von `Vector<E>` sind für die Klasse `Stack<E>` leider auch Methodenaufrufe möglich, die der Arbeitsweise eines Stacks widersprechen. Es können z. B. die Methoden `get(int)`, `indexOf(Object)` oder sogar verändernde Zugriffe aufgerufen werden, etwa `add(int, E)` oder `remove(int)`. Weiterhin existieren mehrere Methoden mit leicht unterschiedlichem Namen, die dasselbe tun, beispielsweise `empty()` und `isEmpty()`. Diese Beispiele sollen Sie für die Problematik der

Implementierungsvererbung sensibilisieren.

Deutlich erkennbar werden die resultierenden Fallstricke, wenn man sowohl die Methoden der Basisklasse `Vector<E>` als auch diejenigen der Klasse `Stack<E>` benutzt. Nehmen wir an, es würden einige Elemente statt durch den Aufruf von `push(E)` durch Aufruf der Methode `add(E)` hinzugefügt. Intuitiv erwartet man ein Anfügen an letzter Position. Die in Abbildung 5-13 dargestellten Positionen stimmen jedoch nicht mit den Indexwerten im `Vector<E>` überein. Tatsächlich entspricht das oberste Element eines Stacks dem letzten Element der zugrunde liegenden Basisklasse `Vector<E>`. Diese Details sind beim Einsatz der eigentlichen Schnittstelle der Klasse `Stack<E>` nicht von Interesse. Beim Aufruf von Methoden der Klasse `Vector<E>` muss man diese Details jedoch kennen, ihre Auswirkungen beachten und Vorsicht walten lassen.

Derartige Probleme beim Einsatz eines nicht intuitiven APIs erschweren das Programmieren und lenken von der eigentlich zu realisierenden Aufgabe ab. Eine weitere wesentliche Erkenntnis aus der obigen Diskussion ist: ***Implementierungsvererbung stellt keine geeignete Wahl dar, um funktionale Erweiterungen zu realisieren***. Eine korrekte Realisierung, die das Substitutionsprinzip (»is-a«-Beziehung) nicht verletzt, wäre sehr einfach möglich gewesen: Eine objektorientierte Umsetzung verwendet eine Referenz auf die Klasse `Vector<E>` (oder einer beliebigen Liste) und nutzt Delegation, um die zuvor genannten Methoden eines Stacks zu realisieren. Das kann man in etwa wie folgt machen:

```
public class Stack<E>
{
    // Komposition statt Vererbung, hier konkreter Typ LinkedList<E, weil das
    // Interface List<E> nur add(int, E) und remove(int) bietet, hier aber …
    private final LinkedList<E> elements = new LinkedList<>();

    public void push(final E item)
    {
        // … Delegation an besser passende Methode
        element.addFirst(item);
    }

    public E pop()
    {
        if (elements.size())
            throw new EmptyStackException();

        // … Delegation an besser passende Methode
        return elements.removeFirst()
    }
    // …
}
```

Merkwürdigkeiten im Interface `Queue`

Zum Datenaustausch zwischen verschiedenen Programmkomponenten können Implementierungen des generischen Interface `Queue<E>` verwendet werden. Dieses

erweitert das bereits vorgestellte Interface `Collection<E>` und bietet ergänzend dazu folgende Methoden[24]:

- `boolean add(E element)`,
- `boolean offer(E element)` – Beide Methoden fügen ein Element am Ende der Queue ein. Für größenbeschränkte Queues existiert zwischen beiden Methoden folgender Unterschied: Der Aufruf von `add(E)` löst teilweise – abhängig von der konkreten Realisierung des Interface `Queue<E>` – eine `IllegalStateException` aus, wenn die Queue voll ist. Ein Aufruf von `offer(E)` liefert stattdessen den Wert `false`.
- `E element()`,
- `E peek()` – Beide Methoden geben das erste Element zurück, *ohne* es aus der Queue zu entfernen. `element()` löst eine `NoSuchElementException` aus, wenn die Queue leer ist. `peek()` liefert in diesem Fall den Wert `null`.
- `E poll()`,
- `E remove()` – Beide Methoden geben das erste Element zurück und *entfernen* es aus der Queue. `remove()` löst eine `NoSuchElementException` aus, wenn die Queue leer ist. `poll()` liefert in diesem Fall den Wert `null`.

Da das Interface `Queue<E>` eine Erweiterung des Interface `Collection<E>` ist, fügt es sich nahtlos in das Collections-Framework ein. Unter anderem bedeutet das, dass sich die vordefinierten Algorithmen anwenden lassen. Wie die zuvor kritisierte Realisierung der Klasse `Stack<E>` enthält auch die Definition des Interface `Queue<E>` aufgrund der Vererbungsbeziehung zum Interface `Collection<E>` einige Methoden, die nicht zur Funktionsweise der Datenstruktur Queue passen: Dies ist beispielsweise eine Methode `remove(Object)`, die es erlaubt, beliebige Objekte zu löschen. Außerdem sind dies sogenannte Bulk-Operationen, die auf einer Menge von Elementen arbeiten, etwa `addAll()`, `containsAll()`, `removeAll()` oder `retainAll()`. Die Kritik wird dadurch verstärkt, dass die Klasse `LinkedList<E>` das Interface `Queue<E>` erfüllt. Ein schöneres, objektorientiertes Design hätte darin bestanden, eine eigenständige Klasse zu definieren, die eine Liste nutzt.

Unklarheiten im Interface `Deque`

Eine sogenannte Deque ist eine doppelseitige Warteschlange und wird durch das generische Interface `Deque<E>` beschrieben, das das zuvor vorgestellte Interface `Queue<E>` erweitert. Ergänzend zum Interface `Queue<E>` werden hier Methoden zum Hinzufügen und Löschen am Anfang und am Ende der Datenstruktur bereitgestellt. Dadurch gibt es aber mehrere Methodenpaare, die gleiche Funktionalität anbieten, aber unterschiedliche Namen tragen, etwa `add(E)` und `addLast(E)` sowie `offer(E)` und `offerLast(E)`. Die sich daraus ergebenden Probleme wurden schon in den Beschreibungen der Klasse `Stack<E>` und des

Interface `Queue<E>` vorgestellt.

5.6 Weiterführende Literatur

Dieses Kapitel hat eine Einführung in Datenstrukturen und das Collections-Framework gegeben. Weitere Informationen finden Sie in folgenden Büchern:

- »**Java Standard Libraries**« von Markus Gumbel, Marcus Vetter und Carlos Cardenas [35]

  Dieses Buch bietet einen umfassenden Einstieg in das Collections-Framework und beschreibt ausführlich die Datenstrukturen Listen, Mengen und Schlüssel-Wert-Abbildungen. Allerdings wird nur das JDK 1.3 behandelt. Für einen Einstieg in die Materie ist dies jedoch ausreichend.

- »**Java Generics and Collections**« von Maurice Naftalin und Philip Wadler [69]

  Dieses Werk ist die ideale Weiterführung des zuvor genannten Buchs, da es besonders auf die Neuerungen im JDK 5 eingeht – im Speziellen sind dies Generics und einige Erweiterungen des Collections-Frameworks.

Weiterführende Informationen und Grundlagen zu Algorithmen und Datenstrukturen können in folgenden Büchern nachgelesen werden:

- »**Data Structures and Algorithms with Object-Oriented Design Patterns in Java**« von Bruno R. Preiss [75]

  Dieses Buch zeigt, wie man Datenstrukturen in Java realisieren kann. Man erhält hier das notwendige Hintergrundwissen, wenn man Details des Collections-Frameworks verstehen möchte.

- »**Introduction to Algorithms**« von Thomas H. Cormen et al. [18]

  Ähnlich wie das zuvor genannte Buch von Bruno R. Preiss wird hier ein guter, aber eher formaler Überblick über Algorithmen und Datenstrukturen geboten.

- »**Algorithmen**« von Robert Sedgewick [79]

  Robert Sedgewick stellt fundiert verschiedene Algorithmen und Datenstrukturen vor. Das geschieht weniger formal als bei Thomas H. Cormen et al. Einige anschauliche Abbildungen helfen dabei, die Algorithmen nachzuvollziehen.

6 Applikationsbausteine

Applikationen auf Basis vorgefertigter, funktionierender Bausteinen zu konstruieren, statt Dinge immer wieder von Grund auf neu zu bauen, macht die Softwareentwicklung zu einer Ingenieurdisziplin. Man vermeidet so, das Rad ständig neu zu erfinden. In diesem Kapitel werden sowohl zu einem kleinen Teil eigene wiederverwendbare Bausteine konstruiert als auch insbesondere Bausteine aus frei verfügbaren Bibliotheken beschrieben und eingesetzt. Dazu lohnt sich immer zunächst eine Internetrecherche, bevor man loslegt. Jon Bentley bemerkt in seinem empfehlenswerten Buch »Perlen der Programmierkunst« [6] dazu treffend: »Wenn eine vom System bereitgestellte […] Funktion Ihren Bedürfnissen genügt, sollten Sie noch nicht einmal daran denken, eine eigene zu schreiben.« Auf diesen Sachverhalt weist auch Joshua Bloch in seinem hervorragenden Buch »Effective Java« [9] in Item 47 »Know and use the libraries« hin. Demzufolge bedienen wir uns nachfolgend ausgiebig in verschiedenen Bibliotheken.

Während ich in den ersten beiden Auflagen dieses Buchs vor allem eigene Entwicklungen von Utility-Klassen gezeigt habe, werde ich dies in dieser dritten Auflage nur noch exemplarisch tun und mich stattdessen auf die Bibliothek Google Guava konzentrieren und ab und an auf Apache Commons als Alternative und Ergänzung eingehen. Für den Fall, dass man doch einmal eine eigene Utility-Klasse erstellt, sollte die Funktionalität durch Unit Tests gründlich geprüft werden. Die später vorgestellten Bausteine der Bibliothek Google Guava sind mit fast 300.000 (!) Tests abgesichert und werden zudem in einer Vielzahl von Google-Produkten (und mittlerweile auch vielen anderen) genutzt. Eine derartige Qualität wird man wohl kaum selbst erreichen können.

In Abschnitt 6.1 motiviere ich, dass der gezielte Einsatz von Bibliotheken die Realisierung von Aufgaben viel einfacher und den entstehenden Sourcecode besser lesbar machen kann. Das wird in Abschnitt 6.2 unter Verwendung der Bibliothek Google Guava weiter konkretisiert. Dazu gebe ich einen Überblick über verschiedene nützliche Funktionalitäten, die einem das Java-Entwickler-Leben erleichtern können. Ein weiteres Beispiel sind Wertebereichs- und Parameterprüfungen, die dabei helfen, ungültige Parameterwerte und Objektzustände zu vermeiden. Das thematisiert Abschnitt 6.3. Aufgedeckte Probleme sollte man für spätere Analysen oder Fehlersuchen protokollieren, wozu sich ein Logging-Framework anbietet. Dieses dient auch der Aufzeichnung von Aktionen oder Wertebelegungen in Fehlersituationen. Auf die Thematik Logging geht Abschnitt 6.4 genauer ein. Abschließend stellt Abschnitt 6.5 das Thema Konfiguration vor, unter anderem die Auswertung von Kommandozeilenparametern und die Parametrierung mithilfe der JDK-Klassen `Properties` und `Preferences`.

6.1 Einsatz von Bibliotheken

Der Umfang des JDKs und die Anzahl frei verfügbarer Bibliotheken nimmt ständig zu, und es wird dadurch immer schwieriger, die Übersicht zu behalten. Auf jeden Fall sollte man einige zentrale Elemente des JDKs kennen und sicher anwenden können, unter anderem das bereits besprochene Collections-Framework.

Muss man sich in ein Thema einarbeiten, so bietet sich eine Internetrecherche an. Die Oracle-Seiten liefern einen ersten Überblick. Allerdings sind die dort vorgestellten Beispiele zum Teil nicht für den Einsatz in Produktivsoftware geeignet, sondern stellen eher sogenannte Schönwettersoftware dar. Diese Beispiele enthalten nämlich meistens keine angemessene Fehlerbehandlung und sollten daher nur als Ideen für eigene Applikationsbausteine dienen.

Eine umfangreiche Quelle im Bereich der Utility-Klassen ist die Sammlung der Apache Commons-Bibliotheken[1]. Die dort angebotenen Bausteine können einem das Programmiererdasein sehr erleichtern. Ähnliches gilt für die Bibliothek Google Guava[2]. Während Apache Commons ihre Funktionalität in jeweils separaten JAR-Dateien bereitstellt, umfasst Google Guava nur eine JAR-Datei. Exemplarisch sind gebräuchliche Bestandteile bzw. deren Referenzierung für die Gradle-Build-Datei nachfolgend angegeben:

```
// Apache Commons
compile 'org.apache.commons:commons-lang3:3.3.2'
compile 'commons-io:commons-io:2.4'
compile 'org.apache.commons:commons-collections4:4.0'

// Google Guava
compile 'com.google.guava:guava:18.0'
```

Motivation zum Einsatz von Bibliotheken

Beim Programmieren sollte man das Ziel verfolgen, möglichst verständlich und elegant zu entwickeln. Damit ist unter anderem gemeint, dass keine unnötige Komplexität entsteht, der Sourcecode gut lesbar ist usw. Die Realität sieht aber leider häufig anders aus. Man findet unkommentierte, teils schlecht lesbare oder auch wüst strukturierte sowie oftmals viel zu umfangreiche Methoden. Darüber hinaus muss man sich mit diversen Implementierungsdetails wie Indexberechnungen in `for`-Schleifen, Sonderbehandlungen von Randfällen, tief verschachtelte `if`-Zweige mit komplexen Bedingungen usw. beschäftigen. Gebilde wie das Folgende kennen Sie bestimmt:

```
// Beispiel: Zu viel Komplexität
if (!(str == null || str.isEmpty()))
{
    if (skipValidation || (str.length() > 5 && lengthCheckActivated))
    {
```

All dies lenkt von der eigentlichen Business-Funktionalität ab und macht den Sourcecode teilweise so schlecht nachvollziehbar, dass der Sinn und zum Teil der Ablauf nicht mehr erkennbar ist. Die Bibliotheken Apache Commons und Google Guava enthalten viele praktische Funktionalitäten. Durch deren Einsatz kann man

mit weniger eigenem Sourcecode viel mehr erreichen, als wenn man alles selbst programmieren würde – außerdem muss man sich nicht zum 100sten Mal um Dinge kümmern, die schon längst gelöst sind.

Betrachten wir als Beispiel eine Gültigkeitsprüfung, um die obige Argumentation nachzuvollziehen. Nehmen wir an, wir sollten für Stringeingaben absichern, dass diese weder `null` noch leer sind und auch nicht nur aus Whitespaces bestehen (z. B. für GUI-Textfelder). Dazu nutzen wir Java-Bordmittel, Google Guava und Apache Commons:

```
public static boolean isValidString_Using_JDK(final String input)
{
   return input != null && !input.trim().isEmpty();
}

public static boolean isValidString_Using_Guava(final String input)
{
   return !Strings.isNullOrEmpty(input) && !input.trim().isEmpty();
}

public static boolean isValidString_Using_Commons(final String input)
{
   return StringUtils.isNotBlank(string);
}
```

Wie man sieht, nimmt die Komplexität beim Einsatz einer passenden Bibliothek ab und die Lesbarkeit und Verständlichkeit steigt.[3] Offensichtlich ist die Klasse `StringUtils` am besten für die hier benötigte Prüfung geeignet. In anderen Anwendungsfällen ist Google Guava teilweise ein wenig angenehmer in der Handhabung.

Hinweis: Lizenzproblematik beim Einsatz von Fremdbibliotheken

Auch wenn in der Regel der Einsatz von Bibliotheken empfehlenswert ist, um gewisse Funktionalitäten nicht selbst realisieren zu müssen, so sollten Sie sich über das Lizenzmodell der jeweiligen Bibliothek genau informieren, bevor Sie sie nutzen. Wichtige Lizenzen sind unter anderem GPL (GNU Public License), LGPL (Lesser GPL) und auch die Apache License. Stellen Sie sicher, dass die Lizenz den Einsatz der Bibliothek in Ihrem Kontext erlaubt. GPL beispielsweise fordert, dass die eigenen Sourcen veröffentlicht werden müssen, falls man die GPL-lizenzierten Bibliotheken in eigene Programme einbindet. Für viele kommerzielle Projekte ist das nicht akzeptabel.

Die hier beschriebenen Bibliotheken Apache Commons und Google Guava stehen unter der recht lockeren Apache License Version 2.0, wodurch sie problemlos auch in kommerziellen Projekten verwendet können, ohne dass man die eigenen Sourcen veröffentlichen muss.

Beispiel: Apache Commons Lang `StringUtils`

Verschiedene Aktionen und Prüfungen auf Strings sind sehr gebräuchlich, z. B. der Test auf einen leeren String, das links- oder rechtsbündige Ausrichten u. v. m. Früher wurde wohl in nahezu jedem Projekt eine eigene Utility-Klasse erstellt. Diesen »Eigengewächsen« ist etwa die Klasse `org.apache.commons.lang3.StringUtils` aus den Apache Commons Lang vorzuziehen. Sie bietet u. a. folgende Methoden:

- `isEmpty(String)` – Prüft, ob der übergebene String leer (`""` oder `null`) ist.
- `isBlank(String)` – Prüft, ob der übergebene String nur Whitespaces enthält, leer oder gar `null` ist.
- `leftPad(String, int, String)` bzw. `rightPad(String, int, String)` – Füllt einen String vom Anfang bzw. vom Ende mit dem angegebenen Zeichen bzw. der Zeichenfolge bis zur gewünschten Länge auf.
- `abbreviate(String, int)` – Verkürzt einen String auf die übergebene Länge, indem der hintere Teil abgeschnitten und durch drei Auslassungspunkte, eine sogenannte Ellipsis (`...`), ersetzt wird.
- `abbreviateMiddle(String, String, int)` – Verkürzt einen String auf die übergebene Länge, indem in der Mitte Zeichen entfernt werden und als Auslassungszeichenfolge die übergebene genutzt wird.

Setzen wir diese Methoden wie folgt ein:

```
public static void main(final String[] args)
{
    // Spezielle Prüfungen
    System.out.println("isEmpty:          " + StringUtils.isEmpty(" "));
    System.out.println("isBlank:          " + StringUtils.isBlank(" "));
    System.out.println("isBlank/null:     " + StringUtils.isBlank(null));

    // Ausrichtung
    final String rightAligned = StringUtils.leftPad("Right", 15, "*");
    System.out.println("leftPad:          " + rightAligned);

    // Abkürzungen
    final int maxWidth = 15;
    final String longText = "This is a long text that has to be shortened!";
    final String shortened = StringUtils.abbreviate(longText, maxWidth);
    final String shortened2 = StringUtils.abbreviateMiddle(longText,
                                                           "…", maxWidth);
    System.out.println("abbreviate:       " + shortened);
    System.out.println("abbreviateMiddle: " + shortened2);
}
```

Listing 6.1 *Ausführbar als* **'STRINGUTILSEXAMPLE'**

Vom Programm StringUtilsExample werden folgende Ausgaben produziert:

```
isEmpty:          false
isBlank:          true
isBlank/null:     true
leftPad:          **********Right
abbreviate:       This is a lo…
abbreviateMiddle: This i…tened!
```

Fallstrick – Do It Yourself

Obwohl es mittlerweile ausgezeichnete Bibliotheken mit umfangreicher und sehr gut getesteter Funktionalität gibt, findet man immer noch einige Projekte, in denen Entwickler lieber verschiedene Hilfsfunktionen selbst programmieren, anstatt auf die oben genannten Bibliotheken zurückzugreifen. Bestimmt kennen Sie diverse Varianten von selbst geschriebenen Klassen namens `StringUtils`. Nahezu alles, was Entwickler dort selbst realisiert haben, findet man auch in Apache Commons oder Google Guava – mit dem Vorteil, dass die dort implementierte Funktionalität erprobt und ausgereift ist. Aber erst die komfortable Verwaltung von Abhängigkeiten durch Tools wie Maven oder Gradle hat das Einbinden derart vereinfacht, dass kaum noch eigene String-Utilities entstehen – hoffentlich jedenfalls.

Hinweis: Korrekturen in (eigenen) Utility-Klassen

Früher hat man vielfach Utility-Funktionalität selbst erstellt. Mitunter hat sich dabei (trotz Unit Tests) auch mal ein Fehler eingeschlichen. Wenn man diesen nun behebt, gilt es Folgendes zu bedenken: Möglicherweise haben Klienten um einige Fehler herumprogrammiert oder ein Fehlverhalten sogar als Feature missbraucht. Dann führt eine Korrektur in der Bibliothek zu Problemen bei Nutzern. Eine allgemeingültige Lösung für diese Problematik gibt es nicht. Im besten Fall, wenn alle Aufrufer bekannt sind, können diese über eine Korrektur informiert werden, sodass benötigte Anpassungen in nutzendem Sourcecode erfolgen können.

In der Regel kennt man jedoch nicht alle Klienten. Dadurch wird die Lage etwas diffiziler. Wenn die Fehlerkorrektur in den eigenen Utility-Klassen jedoch Priorität vor Rückwärtskompatibilität hat, kann man nicht immer Rücksicht auf sich falsch verhaltende Nutzer nehmen. Eine mögliche Variante, die weniger Probleme bei Klienten macht, besteht darin, eine fehlerhafte Methode als veraltetet zu markieren (Annotation `@Deprecated` und optional Javadoc-Kommentar `@deprecated`) und eine neue Methode ins API aufzunehmen.

Die bisherige Argumentation führt aber auch zu dem Schluss, das ähnliche Probleme mit Fremdbibliotheken auftreten können. Durch ein Update auf eine neuere Version handelt man sich unter Umständen unerwartet Inkompatibilitäten ein. Demnach sollte man nicht blindlings Fremdbibliotheken einbinden.

6.2 Google Guava im Kurzüberblick

Nach einem ersten kurzen Blick auf Apache Commons wollen wir uns nun mit Google Guava beschäftigen. Dabei werde ich auf ähnliche Funktionalität in Apache Commons hinweisen.

Obwohl die im JDK bereitgestellte Funktionalität durchaus beachtlich ist, gibt es immer wieder mal Bereiche und Anwendungsfälle, wo man spezifische

Hilfsfunktionalität benötigt, die im JDK nicht vorhanden ist. Dann empfiehlt sich der Einsatz von diesen Bibliotheken. Google Guava hat sich als ergiebige Quelle nützlicher Dinge entwickelt, unter anderem Folgender:

- String-Aktionen
- String-Konkatenation und -Extraktion
- Erweiterungen für Collections
- Weitere Utility-Funktionalitäten

Verbesserungen durch JDK 7 und 8

Mit JDK 7 und insbesondere auch JDK 8 findet man viel Funktionalität, die zuvor nur in Fremdbibliotheken existierte, nun auch in ähnlicher Form im JDK selbst. Mit JDK 7 wurden unter anderem `null`-Prüfungen, Ressourcenfreigabe mit ARM sowie Dateiaktionen verbessert. JDK 8 erleichtert die Verarbeitung von Strings und optionalen Werten. Gleiches gilt für die Unterstützung von Base64-Codierungen. Herausragend ist aber, dass mit Java 8 die funktionale Programmierung mit Lambdas Einzug in die Sprache gefunden hat. Im Bereich der funktionalen Programmierung besitzen die Lambdas leichte Vorteile gegenüber den Varianten aus Google Guava und Apache Commons. Statt der mit JDK 8 erfolgten Integration in die Sprache werden in den Bibliotheken sogenannte Functors als kleine funktionale Objekte mithilfe anonymer Klassen realisiert. Das hat den Vorteil, dass man funktionale Programmierung auch dann nutzen kann, wenn man noch nicht auf JDK 8 umgestiegen ist. Zudem erlernt man schon einmal die neue funktionale Denkweise als nützliche Ergänzung zur Objektorientierung, wodurch der Umstieg auf Java 8 dann erleichtert wird. Der grundsätzliche Ansatz besteht darin, Funktionalität in kleine Einheiten zu kapseln, was die Modularität verbessert.

Folgende Tabelle 6-1 zeigt, wie sich Funktionalitäten aus Guava auf die neuen Bestandteile in JDK 7 und 8 abbilden lassen:

Tabelle 6-1 *Abbildung von Guava auf JDK-Funktionalität*

Klasse in Guava	Analogon in JDK 7 / 8
com.google.common.base.Objects	java.util.Objects
com.google.common.base.Preconditions	java.util.Objects
com.google.common.collect.Ordering	java.util.Comparator
com.google.common.base.Optional	java.util.Optional
com.google.common.io.Files	NIO 2
com.google.common.base.Function	Lambdas

Hinweis:

Bitte beachten Sie vor einer möglichen Migration, dass die Funktionalität eventuell nicht 100% kompatibel ist – so ist die Klasse `Optional` aus dem JDK 8 nicht serialisierbar, was im EJB-Kontext zu Problemen führen kann.

6.2.1 String-Aktionen

In Google Guava stehen diverse Stringmanipulationsmöglichkeiten bereit – jedoch sind diese nicht so umfangreich wie die in der schon kurz vorgestellten Klasse `StringUtils` aus den Apache Commons. Schauen wir uns nun die Klasse `Strings` aus Guava an, die unter anderem folgende Methoden bietet:

- `isNullOrEmpty(String)` – Prüft, ob ein String leer oder `null` ist.
- `nullToEmpty(String)` – Wandelt eine `null`-Referenz in einen Leerstring um.
- `emptyToNull(String)` – Ein Leerstring wird auf den Wert `null` abgebildet.
- `padStart(String, int, char)` bzw. `padEnd(String, int, char)` – Fügt einem String ein bestimmtes Zeichen solange am Anfang bzw. am Ende hinzu, bis der String die gewünschte angegebene Länge erreicht hat. Damit kann man eine links- bzw. rechtsbündige Ausrichtung erzielen.

Ähnlich wie schon zuvor schreiben wir ein kleines Programm zur Demonstration der Funktionalität der Prüfmethoden, des Mappings und der Ausrichtung:

```
public static void main(final String[] args)
{
    // Prüfmethoden
    System.out.println(Strings.isNullOrEmpty(""));
    System.out.println(Strings.isNullOrEmpty("   "));
```

```
    // Mapping
    System.out.println("'" + Strings.emptyToNull("") + "'");
    System.out.println("'" + Strings.nullToEmpty(null) + "'");

    // Ausrichtung
    final String rightAligned = Strings.padStart("Right", 15, '*');
    System.out.println(rightAligned);
}
```

Listing 6.2 *Ausführbar als* **'STRINGSEXAMPLE'**

Startet man das Programm STRINGSEXAMPLE, so kommt es zu folgender Ausgabe:

```
true
false
'null'
''
**********Right
```

Analog in Apache Commons

Ähnliches zur Klasse `Strings` findet man in Apache Commons mit der bereits vorgestellten Klasse `StringUtils`. Letztere bietet noch mehr Funktionalität, etwa diverse weitere Prüfungen sowie die Manipulation von Strings wie das Abkürzen von Texten. Für weitere Manipulationsfunktionalität lohnt sich ein Blick auf die Klasse `WordUtils` in Apache Commons Lang.

6.2.2 String-Konkatenation und -Extraktion

Im JDK existiert wenig Funktionalität zur Stringmanipulation – insbesondere zur Konkatenation sowie Extraktion. Google Guava bietet die folgenden zwei Klassen:

- `Joiner` – Verknüpfung von Strings mit optionaler Parametrierung
- `Splitter` – Extraktion basierend auf einer Parametrierung und Konfiguration

Die Klasse `Joiner`

Die Klasse `com.google.common.base.Joiner` ermöglicht mit der Methode `join()` das Zusammenfügen von Zeichenketten unter Verwendung von konfigurierbaren Trennzeichen. Konfigurierbar ist auch, wie `null`-Werte verarbeitet werden. Wird dies nicht spezifiziert und enthält die Eingabe `null`-Werte, wird beim Aufruf von `join()` eine `NullPointerException` ausgelöst. Manchmal sollen die einzelnen Werte in Hochkommata eingeschlossen werden – das ist etwa dann nützlich, wenn man Leerzeichen in den Werten sichtbar machen möchte. Dazu sind lediglich Hochkommata vor und nach dem Aufruf sowie im Verknüpfungsmuster selbst anzugeben:

```
public static void main(final String[] args)
{
   // Verschiedene Demodaten
```

```
    final List<String> words = Arrays.asList("Text", "is", "concatenated");
    final List<String> withNulls = Arrays.asList("Skip", "null", null,
"values");
    final List<String> names = Arrays.asList("Tim B. ", " Mike I.", " Andy S.
");

    // Einfache Verknüpfung mit Trennzeichenfolge
    System.out.println(Joiner.on("-+-").join(words));

    // Verknüpfung mit null-Werten
    System.out.println(Joiner.on(" ").skipNulls().join(withNulls));

    // Spezielle Markierung von Werten mit Hochkommata
    final String markedValues = "'" + Joiner.on("',").join(names) + "'";
    System.out.println(markedValues);
}
```

Listing 6.3 *Ausführbar als* **'JOINEREXAMPLE'**

Im Listing sehen wir die Fabrikmethode `on()`, über die das Trennzeichen bzw. die Trennzeichenfolge festgelegt wird. Das Verbinden der einzelnen Bestandteile geschieht dann durch Aufruf der Methode `join()` und liefert im obigen Fall folgendes Ergebnis:

```
Text-+-is-+-concatenated
Skip null values
'Tim B. ',' Mike I.',' Andy S. '
```

Ausblick auf Java 8

Das Zusammenfügen von Texten war bis Java 8 leider nicht Bestandteil des JDKs. Die in JDK 8 neu eingeführten Streams erlauben es in Zusammenarbeit mit einem Lambda-Ausdruck und der Utility-Funktionalität aus der Klasse `Collectors`, die letzte Aktion wie folgt zu schreiben:

```
final String markedValues = names.stream().map(input -> "'" + input + "'").
                                          collect(Collectors.joining(","));
```

Die mit einem `Joiner` realisierte Umsetzung ist vielleicht etwas besser lesbar – wobei die Flexibilität bei der JDK-8-Umsetzung höher ist (vgl. Abschnitt 12.3). Alternativ kann man auf die Methode `join()` der Klasse `String` zurückgreifen. Diese ist jedoch nur für einfache Verknüpfungen sinnvoll einsetzbar, bereits bei `null`-Werten gibt es möglicherweise unerwartete Ausgaben. Probieren wir einmal folgende Aktionen aus:

```
System.out.println(String.join("-+-", words));
System.out.println(String.join(" ", withNulls));
```

Führt man diese Anweisungen aus, so kommt es zu folgender Ausgabe:

```
Text-+-is-+-concatenated
Skip null null values
```

Die Klasse `Splitter`

Mithilfe eines `com.google.common.base.Splitters` kann man einen

Eingabestring in eine Folge von Tokens zerlegen. Ähnliches kennen wir aus dem JDK mit einem `StringTokenizer` oder einem `Scanner` (vgl. Abschnitte 4.3.4 und 4.6.4).

Betrachten wir ein Beispiel einer kommaseparierten Eingabe von Namen, wobei bewusst auch ein paar Leereinträge eingefügt sind, um im Anschluss auf einige Besonderheiten eingehen zu können.

```
public static void main(final String[] args)
{
    final String input = " Mike,,Florian ,     Tim ,, Erkan ";

    final Iterable<String> splittedNames = Splitter.on(',').split(input);
    for (final String name : splittedNames)
    {
       System.out.println(name);
    }
}
```

Listing 6.4 *Ausführbar als* **'SPLITTEREXAMPLE'**

Wie beim `Joiner` dient auch beim `Splitter` die Methode `on()` dazu, die Trennzeichen festzulegen. Das Programm SPLITTEREXAMPLE erzeugt folgende Ausgabe:

```
   Mike

Florian
    Tim

Erkan
```

Es fällt auf, dass zum einen auch leere Eingaben als Leerstring behandelt werden und zum anderen alle Whitespaces rund um die Eingaben erhalten bleiben. Beides ist häufig in der Praxis so nicht gewünscht. Derartiges zu entfernen bedarf dann wieder einiger Sonderbehandlungen, wenn man nur Funktionalität aus dem JDK nutzt.

Schauen wir, wie wir die Anforderungen durch Parametrierung des `Splitters` lösen können. Dieser bietet unter anderem folgende Methoden: `trimResults()`, `omitEmptyStrings()` sowie `limit(int limit)`:

```
public static void main(final String[] args)
{
    final String input = " Mike,,Florian , Tim ,, Erkan ";

    final List<String> splittedNames = Splitter.on(',').trimResults().
                                       omitEmptyStrings().limit(3).
                                       splitToList(input);

    for (final String name : splittedNames)
    {
        System.out.println(name);
    }
}
```

Listing 6.5 *Ausführbar als* **'SPLITTERIMPROVEDEXAMPLE'**

Mit der im Listing gezeigten Parametrierung überspringen wir leere Eingaben (`omitEmptyStrings()`) und beschneiden die Ergebnisse (`trimResults()`).

Weiterhin begrenzen wir die zu liefernden Treffer auf 3 (`limit(3)`). Hier verwenden wir den Aufruf von `splitToList(String)`, der als Ergebnis eine `List<String>` liefert. Das ist teilweise praktischer als ein `Iterable<String>`, wie es die zuvor genutzte Methode `split(String)` zurückgibt. Mit der Beschränkung auf drei Elemente erfolgt nach Erkennen von drei Vorkommen keine weitere Verarbeitung des Reststrings, wodurch bei der obigen Eingabe diese Ausgaben produziert werden:

```
Mike
Florian
Tim ,, Erkan
```

Ich habe schon Äußerungen gehört, in denen über den letzten Teil der Ausgabe, der durch die fehlende weitere Auswertung zustandekommt, Verwunderung ausgedrückt wurde. Diese Personen argumentieren, dass die Ausgabe wie folgt hätte aussehen sollen, wenn man die Funktionalität der Extraktion ähnlich wie bei einer Subliste sieht:

```
Mike
Florian
Tim
```

Analog in Apache Commons

Ähnliches zu den Klassen `Joiner` und `Splitter` findet man in Apache Commons in der Klasse `StringUtils` und deren Methoden `split()` sowie `join()`. Schauen Sie in die Javadoc der Methoden, um Herauszufinden, welche am besten auf Ihren Anwendungsfall passt. Dieser Tipp gilt generell für Google Guava und Apache Commons – eine Kombination kann teilweise sinnvoll sein.

Praxisbeispiel

Eingangs des Kapitels habe ich betont, dass der Einsatz von Bibliotheken die Arbeit erleichtern kann. Oftmals lässt sich eine Aufgabe einfacher realisieren, weil man sich mit weniger Details und Sonderfällen zu beschäftigen hat. Das wollen wir nun an den schon genutzten kommaseparierten Daten (CSV) nachvollziehen.[4] Einige Ideen hierzu habe ich aus Holger Staudachers Blog (`http://eclipsesource.com/blogs/2012/07/26/having-fun-with-guavas-string-helpers/`) mit dessen persönlicher Erlaubnis übernommen.

CSV-Daten sind in der Praxis weit verbreitet. Daher ist es nicht ungewöhnlich, dass man Klassen zur Konvertierung von Daten in und aus CSV erstellt. Obwohl das einfach klingt, gibt es dabei den einen oder anderen Stolperstein oder Spezialfall zu beachten: Bei der Aufbereitung von CSV muss nach dem letzten Eintrag kein Komma mehr geschrieben werden. Zum Teil sollen bei der Wandlung in CSV `null`-Werte nicht in die Ausgabe übernommen werden – das kann problematisch werden, wenn die Positionen spezielle Bedeutungen tragen oder Attributen zugeordnet sind. Dann darf man `null`-Werte nicht auslassen. Das gilt ebenso für `omitEmptyStrings()`

beim Splitting. Bitte beachten Sie, dass die hier gezeigte, vereinfachte CSV-Verarbeitung nur dann korrekt arbeitet, wenn die Eingaben selbst ***keine*** Kommas enthalten.

CSV-Verarbeitung mit JDK-Klassen

Folgende Methode nutzt lediglich JDK-Bordmittel, um die Funktionalität zu realisieren, und sensibilisiert für Probleme:

```
public static String toCommaSeparatedString(final List<String> inputs)
{
    if (inputs == null || inputs.isEmpty())
        return "";

    final StringBuilder result = new StringBuilder();
    final Iterator<String> it = inputs.iterator();
    while (it.hasNext())
    {
        final String value = it.next();
        if (value != null)  // skip null
        {
            result.append(value);
            if (it.hasNext())
                result.append(",");
        }
    }
    return result.toString();
}
```

Es ist leicht ersichtlich, dass hier einige `if`-Abfragen und Sonderbehandlungen erfolgen, um leere Eingaben, `null`-Werte usw. korrekt zu behandeln. Dadurch ist die eigentliche Logik der kommaseparierten Aufbereitung nur noch mit Mühe ersichtlich.

Der Vollständigkeit halber schauen wir kurz auf die inverse Aktion, die aus einem kommaseparierten String eine `List<String>` erzeugt:

```
public static List<String> fromCommaSeparatedString(final String input)
{
    if (input == null || input.isEmpty())
        return new ArrayList<String>();

    final List<String> result = new ArrayList<>();
    final String[] tokens = input.split( "," );
    for (final String token : tokens)
    {
        result.add(token.trim());
    }
    return result;
}
```

Beide Methoden sind funktional so weit in Ordnung, aber reicht das? Es hängt stark davon ab: Die tiefe Implementierungsebene, die Sonderbehandlungen, die Komplexität und die Länge sind in beiden Fällen schon ein wenig zu kritisieren. Aber was ist beispielsweise mit der Situation, wenn Erweiterungen erfolgen sollen oder das Verhalten für leere Eingabewerte geändert werden muss? Diese Änderungen ohne

Fehler zu realisieren, ist nicht ganz so einfach. Dann profitiert man davon, wenn es einige Unit Tests gibt, die die Funktionalität prüfen. Details zum Unit-Testen finden Sie in Kapitel 20.

CSV-Verarbeitung mit Guava

Betrachten wir die Umsetzung unter Einsatz von Google Guava, die hier ohne weitere Erklärung gezeigt wird, da wir die beiden genutzten Klassen `Joiner` und `Splitter` bereits zuvor etwas genauer kennengelernt haben.

```
public static String toCommaSeparatedString(final List<String> inputs)
{
    return Joiner.on(",").skipNulls().join(inputs);
}

public static List<String> fromCommaSeparatedString(final String input)
{
    return
Splitter.on(",").omitEmptyStrings().trimResults().splitToList(input);
}
```

Ganz offensichtlich ist diese Lösung viel kürzer, besitzt nahezu keine sichtbare Komplexität und kommuniziert sehr klar, was gemacht werden soll.

6.2.3 Erweiterungen für Collections

In den nachfolgenden Abschnitten schauen wir uns einige nützliche Funktionalitäten aus Guava rund um Collections an.

Mengenoperationen mit Sets

Im JDK sind im Interface `Set<E>` alle Methoden vorhanden, um verschiedenste Mengenoperationen durchzuführen, etwa die Bestimmung der Differenz-, Schnitt- sowie Vereinigungsmenge (vgl. Abschnitt 5.1.3). Die dazu genutzten Methodennamen wie `retainAll()`, `removeAll()` usw. sind allerdings eher technischer Natur und beschreiben das Algorithmische, nicht aber die mathematische Entsprechung – je nach Problemstellung ist das eine oder andere besser passend. Im mathematischen Kontext ist Guava von der Namensgebung her klarer und intuitiver. Hier gibt es eine Utility-Klasse `com.google.common.collect.Sets` mit unter anderen folgenden Methoden:

- `difference(Set<E> set1, Set<?> set2)` – Berechnet die Differenzmenge, also all diejenigen Elemente, die in `set1` sind, nicht aber in `set2`.
- `intersection(Set<E> set1, Set<?> set2)` – Ermittelt die Schnittmenge. Das sind die Elemente, die sowohl in `set1` als auch in `set2` enthalten sind.

- `union(Set<? extends E> set1, Set<? extends E> set2)` – Berechnet die Vereinigungsmenge, die alle Elemente aus `set1` und `set2` enthält.
- `symmetricDifference(Set<? extends E> set1, Set<? extends E> set2)` – Ermittelt die symmetrische Differenz. Diese ist definiert als die Menge der Elemente, die entweder in `set1` oder in `set2`, nicht aber in beiden enthalten sind. Formal ist es die Kombination von *difference*(*union*, *intersection*).

Schreiben wir ein kleines Programm, um die Funktionalität zu erkunden:

```
public static void main(final String[] args)
{
    final Set<String> names = ImmutableSet.of("Andy", "Mike", "Tim", "Peter");
    final Set<String> readers = ImmutableSet.of("Sagi", "Mike", "Tim",
"Jörg");

    final Set<String> intersection = Sets.intersection(names, readers);
    final Set<String> union = Sets.union(names, readers);
    final Set<String> difference1 = Sets.difference(names, readers);
    final Set<String> difference2 = Sets.difference(readers, names);
    final Set<String> symDifference1 = Sets.symmetricDifference(names,
readers);
    final Set<String> symDifference2 = Sets.symmetricDifference(readers,
names);
    final Set<String> symDifference3 = Sets.difference(union, intersection);

    System.out.println("intersection:    " + intersection);
    System.out.println("union:           " + union);
    System.out.println("difference1:     " + difference1);
    System.out.println("difference2:     " + difference2);
    System.out.println("symDifference1:  " + symDifference1);
    System.out.println("symDifference2:  " + symDifference2);
    System.out.println("symDifference3:  " + symDifference3);
}
```

Listing 6.6 *Ausführbar als* **'SETSEXAMPLE'**

Die Programmausgaben verdeutlichen die oben beschriebene Arbeitsweise:

```
intersection:    [Mike, Tim]
union:           [Andy, Mike, Tim, Peter, Sagi, Jörg]
difference1:     [Andy, Peter]
difference2:     [Sagi, Jörg]
symDifference1:  [Andy, Peter, Sagi, Jörg]
symDifference2:  [Sagi, Jörg, Andy, Peter]
symDifference3:  [Andy, Peter, Sagi, Jörg]
```

Analoges für Listen und Maps

Google Guava enthält zwei weitere Utility-Klassen namens `Lists` und `Maps`, die für Listen und Maps adäquate Funktionalitäten anbieten. Diese Klassen findet man im Package `com.google.common.collect`.

Analog in Apache Commons

Ähnliches zu den drei Klassen `Sets`, `Lists` und `Maps` findet man in Apache

Commons Collections. Dort gibt es die Utility-Klasse `org.apache.commons.collections4.CollectionUtils`, die den allgemeineren Typ `Collection<E>` nutzen, also keine Einschränkung auf den Typ `Set<E>` vorgenommen wird. Das hat zum einen den Vorteil, für mehr Typen anwendbar zu sein, zum anderen aber den Nachteil, dass etwa die Vereinigung oder Differenz auf Listen nicht sofort intuitiv sind, z. B. im Hinblick auf doppelte Einträge.

Das Interface `Multimap`

Immer wieder gibt es Situationen, in denen man etwas komplizierte Abbildungen in Form einer Map realisieren muss. Oftmals ist es dabei erforderlich, dass einem Schlüssel eine Menge von Werten zugewiesen werden kann. Im JDK steht für diese Anforderung keine passende Klasse bereit. Guava bietet dafür das Interface `com.google.common.collect.Multimap` und Spezialisierungen davon.

Das folgende Beispiel zeigt eine Abbildung von Ländern auf deren größte Städte:

```
public static void main(final String[] args)
{
    final Multimap<String, String> countryToBigCities =
                                                    ArrayListMultimap.create();
    countryToBigCities.put("Germany", "Berlin");
    countryToBigCities.put("Germany", "Hamburg");
    countryToBigCities.put("Germany", "Munich");
    countryToBigCities.put("Germany", "Cologne");
    countryToBigCities.put("Switzerland", "Basel");
    countryToBigCities.put("Switzerland", "Berne");
    countryToBigCities.put("Switzerland", "Zurich");
    countryToBigCities.put("Switzerland", "Geneve");

    System.out.println("Switzerland: " +
countryToBigCities.get("Switzerland"));
    System.out.println("Germany:     " + countryToBigCities.get("Germany"));
}
```

Listing 6.7 *Ausführbar als* **'MULTIMAPEXAMPLE'**

Startet man das obige Programm MULTIMAPEXAMPLE, so kommt es zu folgenden Ausgaben, die die Mehrfachverknüpfung zeigen:

```
Switzerland: [Basel, Berne, Zurich, Geneve]
Germany:     [Berlin, Hamburg, Munich, Cologne]
```

Analog in Apache Commons

Eine bis auf die Schreibweise gleichnamige Alternative zur Klasse `Multimap` findet man in Apache Commons mit dem Basisinterface `org.apache.commons.collections4.MultiMap`.

Umsetzung mit JDK-Mitteln

Damit Sie ein Gefühl dafür bekommen, dass es mit JDK-Bordmitteln schwieriger als

mit Guava ist, zeige ich hier exemplarisch eine Umsetzung mit einer Map, deren Werte eine Liste beinhalten. Beim Hinzufügen von Werten rufen wir eine Methode `putSpecial()` auf, die die Besonderheit mehrerer Werte für einen Schlüssel kapselt und bei Bedarf eine Liste zur Datenhaltung anlegt:

```
public static void main(final String[] args)
{
    final Map<String, List<String≫ countryToBigCities = new HashMap<>();
    putSpecial(countryToBigCities, "Germany", "Berlin");
    putSpecial(countryToBigCities, "Germany", "Hamburg");
    putSpecial(countryToBigCities, "Germany", "Munich");
    putSpecial(countryToBigCities, "Germany", "Cologne");
    putSpecial(countryToBigCities, "Switzerland", "Basel");
    putSpecial(countryToBigCities, "Switzerland", "Berne");
    putSpecial(countryToBigCities, "Switzerland", "Zurich");
    putSpecial(countryToBigCities, "Switzerland", "Geneve");

    System.out.println("Switzerland: " +
countryToBigCities.get("Switzerland"));
    System.out.println("Germany:     " + countryToBigCities.get("Germany"));
}

private static void putSpecial(final Map<String, List<String≫ map,
                               final String key, final String value)
{
    if (!map.containsKey(key))
    {
        map.put(key, new ArrayList<>());
    }

    final List<String> cities = map.get(key);
    cities.add(value);
}
```

Listing 6.8 *Ausführbar als* **'MULTIMAPJDKEXAMPLE'**

Wenn man an wiederverwendbare Bausteine denkt, dann würde man eine Klasse realisieren, die diese Details versteckt. Das wollen wir ganz bewusst hier nicht machen, da es die Funktionalität ja bereits in Google Guava und Apache Commons gibt.

Das Interface `BiMap`

Neben dem zuvor geschilderten Fall, einem Schlüssel mehrere Werte zuordnen zu können, besteht zum Teil die Anforderung einer bidirektionalen Abbildung, also einem Schlüssel einen Wert zuzuordnen und diesen wieder auf den Schlüssel abzubilden. Das kann man mithilfe zweier separater Maps zwar selbst realisieren, jedoch sind dabei dann ein paar Spezialfälle zu bedenken. Einfacher ist es, Spezialisierungen des Interface `com.google.common.collect.BiMap` aus Guava zu nutzen. Der normale Zugriff erfolgt wie üblich mit `get()`. Die Rückabbildung erfragt man auch über `get()`, jedoch nachdem man zuvor `inverse()` auf der Map aufgerufen hat.

Nachfolgend zeige ich diese Zugriffe für eine bidirektionale Abbildung von

Ländern auf Hauptstädte (und zurück):

```
public static void main(final String[] args)
{
    final BiMap<String, String> countryToCity = HashBiMap.create();
    countryToCity.put("Germany", "Berlin");
    countryToCity.put("Switzerland", "Berne");

    System.out.println("Switzerland:  " + countryToCity.get("Switzerland"));
    System.out.println("Berlin:       " +
countryToCity.inverse().get("Berlin"));
}
```

Listing 6.9 *Ausführbar als* **'BIMAPEXAMPLE'**

Folgende Programmausgaben zeigen die bidirektionale Verknüpfung:

```
Switzerland:  Berne
Berlin:       Germany
```

Analog in Apache Commons

Ähnliches zur `BiMap` findet man in Apache Commons mit dem Basisinterface `org.apache.commons.collections.BidiMap`.

6.2.4 Weitere Utility-Funktionalitäten

In diesem Abschnitt möchte ich Ihnen verschiedene weitere Utility-Funktionalitäten vorstellen. Dabei verzichte ich abgesehen vom folgenden einleitenden Beispiel zur Klasse `Objects` auf die Darstellung von Guava-Varianten, wenn es eine passende Alternative aus dem JDK gibt. Wie ähnlich sich Guava und die zumeist später in das JDK aufgenommenen Funktionalitäten sind, möchte ich am Beispiel der Utility-Klasse `Objects` zeigen, die wir sowohl in Guava als auch im JDK (ab Version 7) finden.

Die Klasse `Objects`

Die Klasse `com.google.common.base.Objects` erleichtert die Implementierung der Methoden `equals(Object)` und `hashCode()`. Wir nutzen folgende Klasse `Person` und die Aufzählung `MaritalStatus` zum Familienstand als Ausgangsbasis:

```
enum MaritalStatus
{
    SINGLE, MARRIED, DIVORCED, WIDOWED, UNKNOWN;
}

public final class  Person
{
    private final  String name;
    private final  Color eyecolor;
    private  MaritalStatus maritalStatus = MaritalStatus.SINGLE;
```

```
    private  String nickname =  null;
    private int  sizeInCm =  0;

    // …
```

Hinweis: Unzulänglichkeiten der Eclipse-Automatiken

Man kann sich mit Eclipse zwar recht bequem über das Menü SOURCE –> GENERATE HASHCODE() AND EQUALS()… die Implementierungen für die Methoden `equals(Object)` und `hashCode()` erstellen lassen. Das entstehende Resultat ist aber komplex und unleserlich. Deshalb empfiehlt sich oftmals der Einsatz der Klasse `Objects` entweder aus Guava oder dem JDK. Schauen wir zum besseren Verständnis auf den von Eclipse erzeugten Sourcecode:

```
@Override
public int hashCode()  {
   final int prime = 31;
   int result = 1;
   result = prime * result + ((eyecolor == null) ? 0 : eyecolor.hashCode());
   result = prime * result + ((name == null) ? 0 : name.hashCode());
   result = prime * result + ((nickname == null) ? 0 : nickname.hashCode());
   result = prime * result + sizeInCm;
   return result;
}

@Override
public boolean equals(Object obj) {
   if (this == obj)
      return true;
   if (obj == null)
      return false;
   if (getClass() != obj.getClass())
      return false;
   Person other = (Person) obj;
   if (eyecolor == null) {
      if (other.eyecolor != null)
         return false;
   } else if (!eyecolor.equals(other.eyecolor))
      return false;
   if (name == null) {
      if (other.name != null)
         return false;
   } else if (!name.equals(other.name))
      return false;
   if (nickname == null) {
      if (other.nickname != null)
         return false;
   } else if (!nickname.equals(other.nickname))
      return false;
   if (sizeInCm != other.sizeInCm)
      return false;
   return true;
}
```

Ein weiterer Negativpunkt der Automatik ist, dass immer auch `null`-Prüfungen in den generierten Sourcecode aufgenommen werden, da Eclipse nicht entscheiden kann, wo dies notwendig ist und wo nicht. Auch besteht die Gefahr, dass

veränderliche Attribute zur Hashcode-Berechnung herangezogen werden. Wenn sich deren Wert ändert, soll das Objekt aber nicht plötzlich in anderen Buckets einsortiert werden. Weitere Details zu Fallstricken bei `hashCode()` und veränderlichen Attributen liefert Abschnitt 5.1.7.

Wie gezeigt, gibt es zwar in Eclipse die Möglichkeit, die Methoden `equals(Object)` und `hashCode()` generieren zu lassen, jedoch ist der entstehende Sourcecode recht umfangreich und nicht so gut lesbar. Es gibt darüber hinaus noch einen guten Grund dafür, die Hilfsfunktionalität aus der Klasse `Objects` zu nutzen. In Abschnitt 4.1.1 hatte ich angedeutet, dass eine menschenlesbare Stringrepräsentation eines Objekts bei (Log-)Ausgaben oder beim Nachvollziehen hilfreich sein kann. Ganz besonders gilt dies auch beim Debugging. Bei der Implementierung der Methode `toString()` ist die Methode `toStringHelper()`, die man in der Utility-Klasse `MoreObjects` findet, nützlich. Die zuvor genannten Funktionalitäten aus Guava werden in der Klasse `Person` wie folgt genutzt:

```
public final class Person
{
    private final String name;
    private final Color eyecolor;
    private MaritalStatus maritalStatus = MaritalStatus.SINGLE;
    private String nickname = null;
    private int sizeInCm = 0;

    // …

    @Override
    public boolean equals(final Object object)
    {
       // Kurzform, kompatibel zum Kontrakt, wenn die Klasse final ist
       if (object instanceof Person)
       {
           final Person other = (Person) object;
           return Objects.equal(this.name, other.name) &&
                  Objects.equal(this.eyecolor, other.eyecolor) &&
                  Objects.equal(this.nickname, other.nickname) &&
                  Objects.equal(this.sizeInCm, other.sizeInCm);
       }
       return false;
    }

    @Override
    public int hashCode()
    {
        // Familienstand nicht in die Hash-Berechnung
        // einfließen lassen, weil veränderlich
        return Objects.hashCode(name, eyecolor, nickname, sizeInCm);
    }

    @Override
    public String toString()
    {
        return MoreObjects.toStringHelper(this)
                          .add("name", name)
                          .add("eyecolor", eyecolor)
```

```
                            .add("maritalStatus", maritalStatus)
                            .add("nickname", nickname)
                            .add("sizeInCm", sizeInCm)
                            .omitNullvalues();
                            .toString();
    }
}
```

Die obige `toString()`-Methode liefert folgende Ausgabe, wenn man ein `Person`-Objekt mit `new Person("Marc", Color.BLUE, 188)` erzeugt:

```
Person{name=Marc, eyecolor=java.awt.Color[r=0,g=0,b=255],
maritalStatus=SINGLE,
      sizeInCm=188}
```

Die Klasse `Objects` aus dem JDK als Alternative

Seit JDK 7 kann man alternativ zur Klasse `Objects` aus Guava die gleichnamige Utility-Klasse aus dem JDK nutzen – wobei der Unterschied im Sourcecode minimal ist:

```
public class Person
{
    private final String name;
    private final Color eyecolor;
    private MaritalStatus maritalStatus = MaritalStatus.SINGLE;
    private String nickname = null;
    private int sizeInCm = 0;

    // ..,

    @Override
    public boolean equals(final Object object)
    {
        // Kurzform, kompatibel zum Kontrakt, wenn die Klasse final ist
        if (object instanceof Person)
        {
            final Person other = (Person) object;
            return Objects.equals(this.name, other.name) &&
                   Objects.equals(this.eyecolor, other.eyecolor) &&
                   Objects.equals(this.maritalStatus, other.maritalStatus) &&
                   Objects.equals(this.nickname, other.nickname) &&
                   Objects.equals(this.sizeInCm, other.sizeInCm);
        }
        return false;
    }

    @Override
    public int hashCode()
    {
        // Familienstand nicht nutzen, weil veränderlich
        return Objects.hash(name, eyecolor, nickname, sizeInCm);
    }

    @Override
    public String toString()
    {
        return "Person [name=" + name + ", eyecolor=" + eyecolor +
               ", maritalStatus=" + maritalStatus + ", nickname=" +
```

```
                nickname + ", sizeInCm=" + sizeInCm + "]";
    }
}
```

Für die Implementierung von `toString()` gibt es in `Objects` des JDKs wenig Unterstützung. Nützlich ist dann die Automatik von Eclipse (vgl. Abschnitt 4.1.1). Der entstehende Sourcecode wird allerdings recht unschön, wenn man die Option SKIP NULL VALUES anwählt. Dann wird eine wenig lesbare Implementierung generiert:

```
@Override
public String toString()
{
    return "Person [" + (name != null ? "name=" + name + ", " : "") +
           (eyecolor != null ? "eyecolor=" + eyecolor + ", " : "") +
           (maritalStatus != null ? "maritalStatus=" + maritalStatus +
           ", " : "") + (nickname != null ? "nickname=" + nickname +
           ", " : "") + "sizeInCm=" + sizeInCm + "]";
}
```

Zum einen erkennt man, dass hier `null`-Prüfungen für Attribute generiert werden, die nicht optional sind und keine `null`-Werte enthalten dürfen. Zum anderen ist kaum auszumachen, welche Attribute tatsächlich ausgegeben werden. Die Variante mit Guava ist deutlich besser lesbar. Sie ist zudem leichter erweiter- und veränderbar.

Tipp: Vereinfachte Nutzung von APIs durch Utility-Klassen

Zum Teil sind die JDK-APIs wenig intuitiv oder kompliziert anzuwenden. Wie bereits aus den Kapiteln 4 und 5 bekannt, kann auch die Implementierung der gebräuchlichen Objektmethoden `toString()`, `equals(Object)`, `hashCode()` und `compareTo(T)` mühsam sein.

Durch den Einsatz nicht intuitiver APIs erhöht sich in der Regel der Sourcecode-Umfang, da es zu komplizierten oder merkwürdigen Lösungen kommt. Wenn um API-Probleme herum programmiert wird, so verschlechtert sich dadurch die Lesbarkeit. ***Soll der eigentliche Applikationscode frei von solchen Implementierungsdetails gehalten werden, schreiben oder nutzen wir Utility-Klassen, die die Verwendung unhandlicher APIs erleichtern.***

Die Klasse `Preconditions`

In den vorangegangenen Kapiteln habe ich es immer mal wieder anklingen lassen, dass die Wertebelegung von Attributen oder Variablen bestimmten Bedingungen genügen sollte, etwa dass Eingaben nicht `null` sein dürfen oder einem bestimmten Wertebereich entstammen müssen. Auch hatte ich im Rahmen der Beschreibung des Objektzustands auf Prüfungen hingewiesen, ebenso wie bei der Behandlung von Fehlersituationen.

Zur Verdeutlichung betrachten wir folgende Methode `filter()`, die ihre Parameter »von Hand« wie folgt prüft:

```
public static <T> void filter(final List<T> list, final Condition<T>
```

```
condition)
{
    if (condition == null)
        throw new IllegalArgumentException("condition must not be null");
    if (list == null)
        throw new IllegalArgumentException("list must not be null");
    if (list.isEmpty())
        throw new IllegalArgumentException("list must not be empty");

    performFilter(list, condition);
}
```

Obwohl Sicherheitsprüfungen dabei helfen können, Fehler schneller zu entdecken, so verschlechtern sie hier doch die Lesbarkeit, da sie auf tieferer Implementierungsebene realisiert sind. Die Verständlichkeit lässt sich durch den Einsatz der Klasse `com.google.common.base.Preconditions` aus Guava deutlich verbessern:

```
public static <T> void filter(final List<T> list, final Condition<T>
condition)
{
    Preconditions.checkNotNull(condition, "condition must not be null");
    Preconditions.checkNotNull(list, "list must not be null");
    Preconditions.checkArgument(!list.isEmpty(), "list must not be empty");

    performFilter(list, condition);
}
```

Das ist schon recht gut, aber die Lesbarkeit lässt sich durch den statischen Import nochmals verbessern – das gilt übrigens auch für diverse andere Utility-Klassen und deren Funktionalität:

```
public static <T> void filter(final List<T> list, final Condition<T>
condition)
{
    checkNotNull(condition, "condition must not be null");
    checkNotNull(list, "list must not be null");
    checkArgument(!list.isEmpty(), "list must not be empty");

    performFilter(list, condition);
}
```

Eine Kleinigkeit muss man bei der Methode `checkArgument()` beachten: Die boolesche Bedingung ist für derartige Prüfungen leicht unleserlich, weil man die teilweise benötigte Negation schnell mal übersieht.

Alternative im JDK

Seit JDK 7 kann man einige der Prüfungen über die Methode `Objects.requireNonNull()` lösen. Die von der Klasse `Preconditions` gebotenen Möglichkeiten sind jedoch umfangreicher.

Analog in Apache Commons

Auch in Apache Commons lassen sich Preconditions prüfen. Dazu gibt es dort die Klasse `org.apache.commons.lang3.Validate`. Im abschließenden Beispiel

möchte ich nochmals zeigen, dass Guava die leicht elegantere Variante bietet. Vergleichen Sie die vorherige mit dieser Methode:

```
public static <T> void filter(final List<T> list, final Condition<T>
condition)
{
    notNull(condition, "condition must not be null");
    notNull(list, "list must not be null");
    isTrue(!list.isEmpty(), "list must not be empty");

    performFilter(list, condition);
}
```

Tipp: Precondition-Prüfungen und schnelles Fehlschlagen

Ganz allgemein kann man sagen, dass es (während der Entwicklung) praktisch ist, wenn Dinge möglichst schnell schiefgehen (Motto Fail-fast), weil man so Fehlersituationen direkt entdecken und gut beheben kann. Dabei ist es hilfreich, wenn die Exception sachdienliche Hinweisen auf die Ursachen eines Problems enthält. Fail-fast ist sicher besser, als irgendwo im Programm aufgrund einer fehlerhaften Eingabe mit einer für den Aufrufer unerwarteten Exception abzustürzen. Schlimmer noch sind sogenannte Silent Fails, die den Fehler verschlucken und einfach so tun, als ob alles in Butter wäre, aber ein falsches Berechnungsergebnis produzieren.

Zusätzliche Vereinfachungen und Erweiterungen

Google Guava bietet über die hier beschriebene Funktionalität hinaus verschiedene weitere Vereinfachungen, etwa bei der Definition von Komparatoren oder der Ausführung von Dateiaktionen. Ebenso gilt dies für die Darstellung optionaler Werte. Auf diese Themen bzw. die korrespondierenden Funktionalitäten in Google Guava gehe ich nachfolgend nicht weiter ein, weil es in JDK 7 bzw. JDK 8 adäquate Alternativen gibt. Für die funktionale Programmierung wurden in Java 8 die Sprachfeatures Lambda-Ausdrücke, Methodenreferenzen und funktionale Interfaces neu hinzugefügt (vgl. Kapitel 11). Auch die Funktionalität von Komparatoren wurde erweitert (vgl. Abschnitt 15.1). Darüber hinaus gibt es nun die Klasse `Optional<T>` im JDK, die optionale Werte als Objekte modelliert (vgl. Abschnitt 15.2).

6.3 Wertebereichs- und Parameterprüfungen

In diesem Abschnitt stelle ich einige Bausteine zur Wertebereichsprüfung vor. Eine solche Prüfung kann schnell recht aufwendig werden, etwa wenn der erlaubte Wertebereich nicht zusammenhängend ist und man die Prüfung selbst implementiert. Einige Programmierer scheuen dann den mit der Abfrage und der Ausgabe von fehlerhaften Werten verbundenen Aufwand, wodurch eine spätere Fehlersuche allerdings erschwert wird. Mögliche Probleme werden als BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN in Abschnitt 16.3.8 diskutiert.

Häufig kann bereits der Einsatz von `enum`-Aufzählungen oder des ENUM-Musters (vgl. Abschnitt 3.4.4) die Arbeit erleichtern. Sind einige Konstanten beispielsweise als `int`-Literale definiert, so können wir gemäß dem Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM (vgl. Abschnitt 17.4.12) vorgehen, um einen eigenständigen Typ zu definieren. Bei umfangreichen Wertebereichen ist der Einsatz dieser Techniken jedoch nicht immer sinnvoll möglich, da man die Werte nicht einzeln benennen möchte oder kann.

Die im Folgenden vorgestellten Bausteine helfen bei Wertebereichsprüfungen. Zunächst entwickle ich exemplarisch einige Prüfmethoden. Wünschenswert ist es, die Prüfung auf Wertemengen und schlussendlich auf nicht zusammenhängende Wertebereiche zu erweitern. Dass dies aufwendig werden kann, wurde in den ersten beiden Auflagen dieses Buchs gezeigt. Weil es deutlich sinnvoller ist, gut getestete Basisbausteine statt selbst geschriebener Hilfsmethoden zu verwenden, nutzen wir die Klassen `Range` und `RangeSet` aus dem Package `com.google.common.collect` aus Google Guava.

6.3.1 Prüfung einfacher Wertebereiche und Wertemengen

Muss man prüfen, ob ein Wert innerhalb eines durch Minimal- und Maximalwert definierten zusammenhängenden Wertebereichs liegt, so kann man folgende Methode `isValueInRange(long, long, long)` verwenden. Aufgrund der Definition für `long`-Werte ist sie auch für alle anderen Ganzzahltypen anwendbar:

```
public final class RangeCheckUtils
{
   public static boolean isValueInRange(final long value, final long minValue,
                                         final long maxValue)
   {
      checkArgument(minValue <= maxValue, "minValue: " + minValue + " must " +
                                          "be <= maxValue: " + maxValue);

      return (minValue <= value && value <= maxValue);
   }
}
```

Eine Prüfung für den Typ `double` erfolgt analog. Allerdings muss man bei Wertebereichsprüfungen von Gleitkommazahlen immer mögliche Rundungsprobleme bedenken: Aufgrund der internen Darstellung weisen Werte eventuell eine Abweichung in einer Nachkommastelle auf und werden dadurch dann fälschlicherweise als ungültig zurückgewiesen. Als Faustregel gilt: ***Wenn man exakte Vergleiche durchführen und anhand derer gewisse Aussagen treffen möchte, ist der Einsatz von Gleitkommazahlen zu vermeiden***. Das wurde bereits in Abschnitt 4.1.2 besprochen.

Für beliebige Referenztypen kann man eine Wertebereichsprüfung analog zu der für die primitiven Typen `long` und `double` formulieren, indem man das

`Comparable<T>`-Interface nutzt. Für benutzerdefinierte und komplexere Typen sind die Aussagen »größer« bzw. »kleiner« nicht wie für Zahlen intuitiv klar – zur Festlegung sind die Werte verschiedener Attribute der verglichenen Objekte ausschlaggebend (vgl. Abschnitt 5.1.8). Die für `Comparable<T>` ausgelegte Methode `isValueInRange( T, T, T)` sieht wie folgt aus:

```
public static <T extends Comparable<T≫ boolean isValueInRange( final T value,
                                                final T minValue, final T
maxValue)
{
    checkNotNull( value,  "value must no be null");
    checkNotNull( minValue, "minValue must no be null");
    checkNotNull( maxValue, "maxValue must no be null");
    checkArgument( minValue.compareTo( maxValue) <= 0, "minValue: " + minValue +
                                          " must be <= maxValue: " + maxValue);

   return ( minValue.compareTo( value) <= 0 && value.compareTo( maxValue) <= 0);
}
```

Erstellen erster Unit Tests

Da wir den Aspekt der Wiederverwendbarkeit berücksichtigt haben, wurden die obigen Methoden in einer Klasse `RangeCheckUtils` definiert. Nun legen wir eine korrespondierende Testklasse `RangeCheckUtilsTest` mit Unit Tests an. Exemplarisch sind einige Testfälle realisiert, die alle bisher definierten Methoden nutzen. Dabei prüfen wir jeweils einen Vertreter aus dem Wertebereich sowie die beiden Randwerte. Ebenso existiert jeweils ein Testfall für den Negativfall, wo die geprüften Werte (gerade) nicht mehr Bestandteil des Wertebereichs sind:

```
public final class RangeCheckUtilsTest
{
    @Test
    public void testValueInRange_With_Borders()
    {
        assertTrue( "7 in [2 .. 9]", isValueInRange( 7, 2, 9));
        assertTrue( "2 in [2 .. 9]", isValueInRange( 2, 2, 9));
        assertTrue( "9 in [2 .. 9]", isValueInRange( 9, 2, 9));
    }

    @Test
    public void testValueInRange_Value_Not_Included()
    {
        assertFalse( "1 not in [2 .. 3]", isValueInRange( 1, 2, 3));
        assertFalse( "4 not in [2 .. 3]", isValueInRange( 4, 2, 3));
    }

   @Test
   public void testValueInRangeDouble_With_Borders()
   {
       assertTrue( "7.2 in [7.1 .. 7.3]", isValueInRange( 7.2, 7.1, 7.3));
       assertTrue( "7.1 in [7.1 .. 7.3]", isValueInRange( 7.1, 7.1, 7.3));
       assertTrue( "7.3 in [7.1 .. 7.3]", isValueInRange( 7.3, 7.2, 7.3));
   }

   @Test
   public void testValueInRangeDouble_Value_Not_Included()
```

```
    {
        assertFalse("1.0 not in [1.1 .. 1.2]", isValueInRange(1.0, 1.1, 1.2));
        assertFalse("1.3 not in [1.1 .. 1.2]", isValueInRange(1.3, 1.1, 1.2));
    }

    @Test
    public void testValueInRangeComparable_With_Borders()
    {
        assertTrue("'BB' in [AA .. CC]", isValueInRange("BB", "AA", "CC"));
        assertTrue("'AA' in [AA .. CC]", isValueInRange("AA", "AA", "CC"));
        assertTrue("'CC' in [AA .. CC]", isValueInRange("CC", "AA", "CC"));

    }

    @Test
    public void testValueInRangeComparable_Value_Not_Included()
    {
        assertFalse("'A ' not in [AA .. CC]", isValueInRange("A ", "AA",
"CC"));
        assertFalse("'DD' not in [AA .. CC]", isValueInRange("DD", "AA",
"CC"));
    }
}
```

Listing 6.10 *Ausführbar als* **'RANGECHECKUTILSTEST'**

Zur besseren Lesbarkeit des Aufrufs der Prüfmethoden nutze ich hier den statischen Import für die Utility-Klasse `RangeCheckUtils`. Für Testklassen und Utility-Klassen ist dieses Sprachfeature sehr praktisch.[5]

Anders als hier und bei den folgenden Beispielen dargestellt, sollte man speziell für Applikationsbausteine eine wesentlich größere Anzahl an Unit Tests bereitstellen, als es hier aus Platzgründen möglich ist. Weitere Informationen zum Thema Unit-Testen finden Sie in Kapitel 20.

6.3.2 Prüfung komplexerer Wertebereiche

Die bisher vorgestellten Methoden bilden den Grundstock, um komplexere Prüfungen realisieren zu können. Allerdings ist die Umsetzung bis jetzt wenig objektorientiert, sondern eher funktional in einer Utility-Klasse `RangeCheckUtils` implementiert. Dieses Vorgehen ist für Utility-Klassen durchaus akzeptabel. Werden die Wertebereiche komplexer, so bietet sich eine objektorientierte Realisierung an. Grundlage dafür ist die Definition von Wertebereichen in Form einer Klasse. Das wollen wir jedoch nicht selbst realisieren, weil Google Guava dafür mit der Klasse `Range` und dem Interface `RangeSet` sowie seinen Spezialisierungen bereits vorgefertigte und gut erprobte Bausteine anbietet.

Die Klasse `Range`

Die Klasse `com.google.common.collect.Range` erlaubt es auf einfache Weise, Wertebereiche zu beschreiben. Dazu müssen die zu verarbeitenden Typen das Interface `Comparable<T>` erfüllen. Dann lassen sich mithilfe verschiedener

Erzeugungsmethoden offene und geschlossene Intervalle konstruieren.[6] Laut mathematischer Definition beinhaltet ein offenes Intervall die Intervallgrenzen nicht, ein geschlossenes schon:

```
public static void main(final String[] args)
{
    final Range<Integer> closed = Range.closed(0, 100);
    final Range<Integer> open = Range.open(0, 100);
    final Range<Integer> openClosed = Range.openClosed(0, 100);
    final Range<Integer> closedOpen = Range.closedOpen(0, 100);

    System.out.println("Closed:     " + closed);
    System.out.println("Open:       " + open);
    System.out.println("openClosed: " + openClosed);
    System.out.println("closedOpen: " + closedOpen);
}
```

Listing 6.11 *Ausführbar als* **'RANGEEXAMPLE'**

Startet man das obige Programm RANGEEXAMPLE, so werden die verschiedenen Varianten des Wertebereichs von 0 bis 100 ausgegeben, wobei leider in Windows aufgrund von Zeichensatzunzulänglichkeiten die `toString()`-Ausgabe ein Fragezeichen statt ein '–' als Bereichsangabe nutzt:

```
Closed:     [0?100]
Open:       (0?100)
openClosed: (0?100]
closedOpen: [0?100)
```

Ähnliches haben wir bereits im Collections-Kapitel in Abschnitt 5.2.4 für einfache Filterbedingungen selbst realisiert. Die Klasse `Range` bietet aber deutlich mehr, so kann man etwa auch nach unten oder oben unbegrenzte Wertebereiche definieren sowie auch abfragen, ob einzelne Werte oder eine Wertemenge in einem `Range` enthalten sind:

```
public static void main(final String[] args)
{
    final Range<Integer> lessThan100 = Range.atMost(99);
    final Range<Integer> moreThan10 = Range.atLeast(11);

    // Prüfe Enthaltensein
    System.out.println("-50 in [...99]:  " + lessThan100.contains(-50));
    System.out.println("500 in [...99]:  " + lessThan100.contains(500));
    System.out.println("10  in [11...]:  " + moreThan10.contains(10));
    System.out.println("100 in [11...]:  " + moreThan10.contains(100));

    // Vereinigung
    final Range<Integer> intersection = moreThan10.intersection(lessThan100);
    System.out.println("Intersection:     " + intersection);

    // Obermenge
    final Range<Integer> range_10_25 = Range.closed(10, 25);
    final Range<Integer> range_40_60 = Range.closed(40, 60);
    final Range<Integer> span = range_10_25.span(range_40_60);
    System.out.println("span:  " + span);

    // Prüfe, ob der Wertebreich umschlossen wird oder erzeuge eine solchen
    final Range<Integer> lessThan1000 = Range.atMost(999);
    System.out.println(lessThan1000.encloses(lessThan100));
```

```
    System.out.println(Range.encloseAll(Arrays.asList(5, 100, 500)));
}
```

***Listing 6.12** Ausführbar als* **'RANGEIMPROVEDEXAMPLE'**

Startet man das Programm RANGEIMPROVEDEXAMPLE, so werden einige Aktionen ausgeführt, die folgende Konsolenausgaben produzieren:

```
-50 in [ …99]: true
500 in [ …99]: false
10  in [ 11…]: false
100 in [ 11…]: true
Intersection:   [ 11?99]
span:           [ 10?60]
true
[ 5?500]
```

Man erkennt sehr schön die Prüfungen auf Enthaltensein sowie die Vereinigung von Wertebereichen als auch die Erweiterung zweier Wertebereiche auf einen umschließenden Wertebereich mit `span()`. Auch kann man sich mit `encloseAll()` einen Wertebereich basierend auf einer Menge einzelner Werte erstellen.

Das Interface `RangeSet`

Spezialisierungen von `com.google.common.collect.RangeSet` beschreiben eine Menge nicht verbundener Wertebereiche, die in Form der zuvor vorgestellten Klasse `Range` definiert sind. Man kann einem `RangeSet` verschiedene Wertebereiche hinzufügen, wieder entfernen und darauf prüfen, ob Werte in der Wertebereichsmenge enthalten sind. Als Besonderheit werden beim Hinzufügen überlappende Wertebereiche miteinander verbunden. Das sehen wir in folgendem Listing. Insgesamt soll die Wertebereichskombination 0–9 und 50–70 definiert werden. Anstatt diese direkt anzugeben, gestalten wir alles etwas komplizierter, um die Möglichkeiten mit dem Interface `RangeSet` zu erkunden: Der Wertebereich 0–9 wird in Form zweier Bestandteile 0–5 und 6–9 definiert. Letzteres wird wiederum als Zusammenschluss der Intervalle [6 – 8) und [8 – 10) realisiert. Hier kommt die mathematische Notation mit eckigen und runden Klammern zum Einsatz. Zur Erinnerung nochmal: Eine eckige Klammer besagt, dass der Wert ein Teil des Wertebereichs ist, eine runde Klammer besagt, dass der Wert nicht Bestandteil des Wertebereichs ist.

Das eben erlangte Wissen wollen wir nutzen, um verschiedene Aktionen auf einem `RangeSet` auszuführen. Dazu schauen wir nun auf das folgende Beispiel:

```
public static void main(final String[] args)
{
    final RangeSet<Integer> rangeSet = TreeRangeSet.create();

    // Bereich 0-9 aus den Teilen 0-5 und 6-9 erstellen
    rangeSet.add(Range.closed(0, 5));
    System.out.println(rangeSet);

    // Zweiter Bereich aus zwei Subbereichen erstellen
```

```
        rangeSet.add(Range.closedOpen(6, 8));  // eigenständiger Teil
        rangeSet.add(Range.closedOpen(8, 10)); // wird verbunden zu [6, 10)
        System.out.println(rangeSet);

        // Wertebereich (40-70]
        rangeSet.add(Range.openClosed(40, 70));
        System.out.println(rangeSet);

        // Teilbereich (40-50) entfernen
        rangeSet.remove(Range.open(40, 50));
        System.out.println(rangeSet);

        // Prüfungen auf Enthaltensein
        System.out.println(rangeSet.contains(7));
        System.out.println(rangeSet.contains(55));
        System.out.println(rangeSet.contains(777));
    }
```

Listing 6.13 *Ausführbar als* **'RANGESETEXAMPLE'**

Startet man das Programm RANGESETEXAMPLE, so erhält man folgende Ausgaben, die die Arbeitsweise verdeutlichen, aber auch zeigen, dass die Stringrepräsentation von `RangeSet` nicht ganz so hübsch ist, wenn man nicht den passenden Zeichensatz nutzt:

```
[[0?5]]
[[0?5], [6?10)]
[[0?5], [6?10), (40?70]]
[[0?5], [6?10), [50?70]]
true
true
value 777 is not included in [[0?5], [6?10), [50?70]]
```

Fazit

Aus Platzgründen konnte nur ein Einstieg gegeben werden, aber selbst damit wird deutlich, dass man mithilfe der Klasse `Range` und des Interface `RangeSet` und seinen Spezialisierungen auch komplexere Wertebereichsprüfungen auf einfache und nachvollziehbare Weise lösen kann. Wollte man dies per Hand realisieren, würde das in deutlich mehr Sourcecode enden und auch wesentlich schwieriger wart- und erweiterbar sein.

6.4 Logging-Frameworks

Neben dem Debugging (vgl. Abschnitt 2.5) stellt das sogenannte ***Logging***, also Programmausgaben über den Zustand des Programms selbst, ein wichtiges Hilfsmittel zur Fehlersuche dar: Durch Log-Ausgaben können die internen Abläufe, derzeitige Wertebelegungen usw. im Detail dargestellt werden. Sinnvolle Log-Ausgaben erleichtern sowohl eine mögliche Fehlersuche als auch das Nachvollziehen und Verständnis des Programmablaufs. Diverse Java-Anwendungen produzieren Ausgaben und nutzen dazu im einfachsten Fall `System.out`. Empfehlenswert ist jedoch der Einsatz von Logging-Frameworks. Dieser Abschnitt beginnt mit einer

kurzen Darstellung der Vorteile bei der Verwendung eines solchen Frameworks. Dann stelle ich in Abschnitt 6.4.1 mit log4j ein weitverbreitetes Framework vor. Den Abschluss bildet in Abschnitt 6.4.2 eine Zusammenstellung verschiedener Tipps und Tricks beim Einsatz von Logging.

Gründe für Logging-Frameworks

Ausgaben über `System.out` können zwar in kleinen Testsystemen ausreichend sein, für die meisten Anwendungen sind sie es aus vielfältigen Gründen aber nicht. Der entscheidende Nachteil ist, dass die Ausgaben nicht persistent gespeichert werden, sondern lediglich über die Konsole »rauschen« und damit unwiederbringlich verschwinden. Zudem macht es einen unprofessionellen Eindruck, wenn ein Anwender alle möglichen (Fehler-)Meldungen auf der Konsole sieht. Daher ist es hilfreich, ein Logging-Framework zu nutzen: Dadurch kann man Informationen strukturiert, in verschiedenen Detaillierungsgraden und in verschiedenen Log-Dateien konfigurierbarer Größe und Anzahl sammeln. Nutzt man zur Protokollierung lediglich Ausgaben über `System.out`, so besitzt man diese Flexibilität nicht: Sollen mehr oder weniger Informationen ausgegeben werden oder will man diese komplett ein- oder ausschalten, so bedarf es Programmänderungen. Das erfordert dann auch eine Neukompilierung und eine neue Auslieferung der Software. Bei Logging-Frameworks ist das nicht erforderlich, denn diese erlauben eine Konfiguration zum Detailgrad von Ausgaben sogar zur Laufzeit. Damit kann man temporär die Protokollierung zur Fehlersuche detaillierter gestalten. Außerdem kann das Format der Log-Ausgaben beeinflusst werden, z. B. Thread-und Datumsinformationen automatisch ergänzt werden.

6.4.1 Apache log4j

Apache log4j ist ein recht populäres Logging-Framework. Die benötigten Dateien können unter `http://logging.apache.org/log4j/docs/download.html` heruntergeladen und in den `CLASSPATH` Ihres Projektverzeichnisses aufgenommen werden. Einfacher ist es, folgende Zeilen in die Build-Datei hinzuzufügen, wodurch dann Gradle dies für Sie erledigt:

```
compile group: 'org.apache.logging.log4j', name: 'log4j-api', version: '2.0.2'
compile group: 'org.apache.logging.log4j', name: 'log4j-core', version:
'2.0.2'
```

log4j-Grundbegriffe

log4j ist zwar ziemlich mächtig, aber manchmal für den ersten Einsatz etwas schwierig zu konfigurieren und zu verstehen. Bevor ich Details der Konfiguration beschreibe, gebe ich einen Überblick über die zentralen Klassen und Interfaces `Logger`, `Appender` und `Layout` aus dem Package `org.apache.log4j`. Diese

Klassen und deren Beziehungen untereinander sind in Abbildung 6-1 visualisiert.

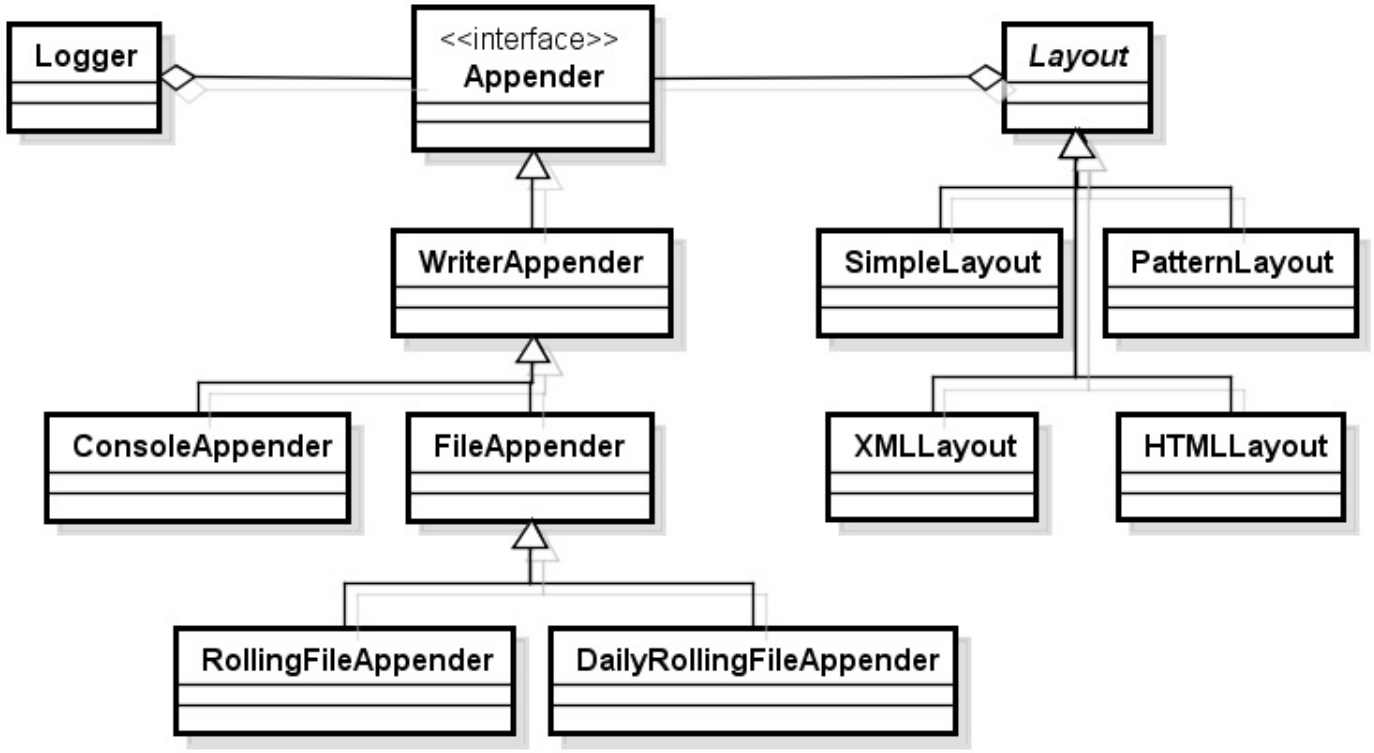


Abbildung 6-1 *Das log4j-Framework*

Ein sogenannter ***Logger*** ist die zentrale Stelle zur Ausgabe von Meldungen und wird durch eine Instanz der Klasse `Logger` definiert. Über ihn werden die von einem Progamm auszugebenden Meldungen im Sourcecode abgesetzt. Weiterhin versteckt ein Logger den komplizierten Ausgabeprozess und bietet eine einheitliche Schnittstelle zur formatierten Ausgabe auf verschiedene Ausgabemedien mithilfe sogenannter ***Appender***.

Das folgende Listing zeigt ein Beispiel für den Einsatz der Klasse `Logger` und ihrer Methoden `info(String)` und `error(String)` zur Ausgabe von Meldungen:

```
public final class LoggingExample
{
    private static final Logger LOGGER = Logger.getRootLogger();

    public static void main(final String[] args)
    {
        // Standardkonfiguration mit Konsolenausgabe
        BasicConfigurator.configure();

        // Log-Meldungen ausgeben
        LOGGER.info("Info-Meldung aus LoggingExample.");
        LOGGER.error("Error-Meldung aus LoggingExample.");
    }
}
```

Listing 6.14 *Ausführbar als* **'LOGGINGEXAMPLE'**

Einem Logger werden Subtypen von `Appender` zugeordnet, die einem Ausgabemedium entsprechen, etwa der Konsole oder einer Datei. Im Beispiel geschieht dies indirekt durch den Aufruf von

`BasicConfigurator.configure()`. Damit wird eine Standardkonfiguration für die Logging-Umgebung hergestellt, die einen `org.apache.log4j.ConsoleAppender` erzeugt, der auf die Konsole schreibt. Es existieren weitere, spezielle `Appender`-Klassen: Die Klassen `RollingFileAppender` und `DailyRollingFileAppender` aus dem Package `org.apache.log4j` erlauben beispielsweise, Log-Ausgaben in neue Dateien zu schreiben, wenn eine gewisse Größe der Log-Datei überschritten wurde oder ein neuer Tag angebrochen ist.

Ein sogenanntes ***Layout*** legt die Formatierung der Log-Ausgabe fest. Es existieren vier verschiedene Spezialisierungen. Aus dem Package `org.apache.log4j` stammen die Klassen `SimpleLayout`, `PatternLayout` und `HTMLLayout`. Das `XMLLayout` findet man im Package `org.apache.log4j.xml`.

Zusammenspiel von Logger, Appender und Layout

Die Konfiguration und die Zuordnung der Instanzen der hier vorgestellten Klassen kann wahlweise im Sourcecode, über eine Property- oder eine XML-Datei erfolgen. Die letzteren beiden Möglichkeiten werden in der Praxis bevorzugt eingesetzt, weil sie eine flexible Konfiguration und nachträgliche Änderung erlauben, die auch ohne Neustart oder Neukompilierung wirksam wird.

Zur Verdeutlichung der Zusammenhänge in den später gezeigten Konfigurationsdateien stelle ich hier jedoch zunächst ein Beispiel einer Realisierung in Sourcecode vor. Im folgenden Listing wird ein Logger definiert und diesem werden zwei Appender zugeordnet. Danach erfolgen einige Ausgaben von Log-Meldungen:

```
private static final Logger  LOGGER  =
Logger.getLogger(LogConfigExample.class);
private static final String  LOGFILE = "LogDatei.log";
private static final boolean APPEND  = true;

public static void main(final String[] args)
{
    // Layout erzeugen
    final SimpleLayout layout = new SimpleLayout();

    // ConsoleAppender dem Logger zuordnen
    LOGGER.addAppender(new ConsoleAppender(layout));
    // FileAppender erzeugen und dem Logger zuordnen
    try
    {
        LOGGER.addAppender(new FileAppender(layout, LOGFILE, APPEND));
    }
    catch (final IOException ex)
    {
        LOGGER.warn("Can't create FileAppender for " + new File(LOGFILE).
              getAbsolutePath(), ex);
    }
```

```
    // Log-Level auf WARN und dann testweise Meldungen ausgeben
    LOGGER.setLevel(Level.WARN);
    LOGGER.info("Filtered - Not displayed");
    LOGGER.warn("Warning should be printed");
}
```

Listing 6.15 *Ausführbar als* **'LOGCONFIGEXAMPLE'**

Durch Aufruf von `Logger.getLogger()` erzeugt man einen Logger – im Beispiel einen klassenspezifischen. Zur Konfiguration der Ausgabe wird diesem hier ein `ConsoleAppender` zugeordnet. Zusätzlich wird ein `org.apache.log4j.FileAppender` registriert, der in die angegebene Log-Datei `LogDatei.log` im aktuellen Verzeichnis loggt. Ein solcher `FileAppender` lässt sich so konfigurieren, dass bei jedem Start der Anwendung entweder alle vorherigen Log-Meldungen überschrieben oder neue Meldungen zu den bisherigen hinzugefügt werden. In der Praxis verwendet man meistens das Anhängen, da man an der Historie der Ausgaben interessiert ist.

Mit der Methode `setLevel(Level)` legt man den gewünschten Detaillierungsgrad der Log-Ausgabe fest und filtert damit alle Meldungen, die mit einer feineren Granularität ausgegeben werden: Weist man log4j an, im Level `WARN` zu loggen, so werden alle Ausgaben, die in `DEBUG` oder `INFO` erfolgen, nicht an die Appender übertragen.

Tipp: Korrekte und konsistente Wahl der Log-Level

Beim Logging kann über das Log-Level die Granularität und Wichtigkeit der auszugebenden Informationen bestimmt werden. Eine möglichst einheitliche Verwendung erleichtert eine spätere Auswertung von Meldungen. Es haben sich die in der folgenden Aufzählung genannten Vorschläge bewährt. Die Log-Level sind in aufsteigender Reihenfolge genannt – der gravierendste zuletzt:

- `TRACE` – Ausgaben auf dieser niedrigsten Ebene sind nur für Entwickler zum Nachvollziehen von Programmabläufen interessant, die man durch Debugging eventuell nicht gut verfolgen kann. Sehr ähnlich, aber etwas grobgranularer sind Ausgaben im anschließend beschriebenen `DEBUG`-Level.[a]
- `DEBUG` – Ausgaben auf dieser Ebene sind nur für Entwickler interessant und sollten im Normalbetrieb ausgeschaltet sein. Sie können aber zur Laufzeit bei Bedarf selektiv eingeschaltet werden, um bestimmte Fehlerursachen aufspüren zu können.
- `INFO` – Auf dieser Ebene werden Informationen ausgegeben, die den Programmfluss sichtbar und verständlich machen. Dazu gehört etwa die Protokollierung von Methodenaufrufen an externen Schnittstellen.
- `WARN` – Wie der Name schon sagt, werden hier Warnungen ausgegeben: Es werden also Betriebssituationen gemeldet, die man zwar berücksichtigt hat, die aber von der erwarteten Situation abweichen, etwa dass eine Datei nicht geöffnet werden kann. Das System zeigt als Folge möglicherweise ein etwas

anderes Verhalten als gewünscht.

- `ERROR` – Abgefangene Exceptions sollten auf dieser Ebene ausgegeben werden, wenn der folgende Programmablauf dadurch wahrscheinlich gestört wird, etwa bei unerwarteten oder inkonsistenten Objektzuständen.
- `FATAL` – Der Log-Level `FATAL` bedeutet, dass ein schwerwiegender Fehler aufgetreten ist und die sinnvolle Programmausführung nicht mehr möglich ist. In diesem Fall sollte das Programm beendet werden.

Zur Analyse von Programmfehlern ist es hilfreich, die obigen Hinweise zu beachten. Die Ausgabe von Informationen auf einem unangebrachten Log-Level kann eine Fehlersuche in eine falsche Richtung lenken. Schauen auch Dritte auf Log-Ausgaben, kann bei einer großen Anzahl von Ausgaben im `WARN`- oder `ERROR`-Level ein falscher Eindruck der Programmstabilität entstehen.

[a] Ich selbst nutze `TRACE` eher selten, sondern bevorzuge `DEBUG`, da diese Granularität für meine Bedürfnisse nahezu immer ausreichend ist. Für Testsysteme kann man mit `TRACE` aber für spezielle Debug-Einsätze noch feiner protokollieren.

Log-Konfiguration

Für die Konfiguration von log4j über eine Property-Datei wird standardmäßig der Dateiname `log4j.properties` verwendet. Mithilfe dieser Konfigurationsdatei definiert man für eine Applikation gewisse Logger und zugehörige Appender. Dort erfolgt auch die Konfiguration von Log-Leveln für einzelne Packages und Klassen.

Im folgenden Beispiel wird der `rootLogger` so konfiguriert, dass nur Meldungen auf dem Level `INFO` oder bedeutender ausgegeben werden. Die Ausgabe erfolgt sowohl auf der Konsole als auch in eine Datei. Dazu werden die entsprechenden Appender über deren Namen `SimpleConsoleAppender` und `PDFEditorRollingFileAppender` referenziert. Deren eigentliche Definition und Konfiguration erfolgt im Anschluss in der `log4j.properties`-Datei über die Angabe von Schlüssel-Wert-Paaren: Alle zu konfigurierenden Schlüssel sind als voll qualifizierte Namen `org.apache.log4j.*` gefolgt von ihrem Wert anzugeben.

In der Praxis wird häufig das im Folgenden beschriebene `PatternLayout` eingesetzt, da dieses umfangreiche Möglichkeiten zur Konfiguration der Ausgabetexte erlaubt. Eine `log4j.properties`-Datei könnte damit wie folgt aussehen, wenn sie die obigen zwei eigenen Appender nutzt:

```
log4j.rootLogger=INFO, SimpleConsoleAppender, PDFEditorRollingFileAppender

log4j.appender.SimpleConsoleAppender=org.apache.log4j.ConsoleAppender
log4j.appender.SimpleConsoleAppender.layout=org.apache.log4j.PatternLayout
log4j.appender.SimpleConsoleAppender.layout.ConversionPattern=%d{MM-dd
HH:mm:ss}
       %-5p [%t] %c: %m%n

log4j.appender.PDFEditorRollingFileAppender=org.apache.log4j.
      DailyRollingFileAppender
```

…

Einstellungen im `PatternLayout`

Das `PatternLayout` erlaubt es, die LogAusgaben an eigene Bedürfnisse anzupassen. Dazu wird im `ConversionPattern` das Format über angegebene Platzhalter festgelegt. Die aktuellste Dokumentation findet man online unter `http://logging.apache.org/log4j/docs/api/org/apache/log4j/Layout.html`. Hier erfolgt daher nur eine kurze Vorstellung der zuvor verwendeten Platzhalter:

- `%d{MM-dd HH:mm:ss}` – Zeitstempel
- `%-5p` – Log-Level (z. B. `INFO`), linksbündige, auf 5 Zeichen verlängerte Ausgabe, wodurch z. B. `INFO`- und `ERROR`-Ausgaben gleich ausgerichtet werden. Man verhindert dadurch Flattersatz in der Log-Datei.
- `%t` – Thread-Name
- `%c` – Kategorie- bzw. Logger-Name
- `%m` – Log-Ausgabe: Dieser Wert wird von dem einsetzenden Programm spezifiziert, alle anderen Werte werden automatisch ermittelt und gefüllt.
- `%n` – Zeilenende

Einlesen der Konfigurationsdatei

Die log4j-Konfiguration kann in Form der Konfigurationsdatei `log4j.properties` der Applikation zur Verfügung gestellt werden. Dazu wird beim Start über die Klasse `PropertyConfigurator` die zugehörige Konfiguration mithilfe der Methode `configureAndWatch(String, long)` eingelesen. Diese erhält als Eingabe den Dateinamen der Konfigurationsdatei und einen Wert für ein Überwachungsintervall. Dieses legt fest, in welchen Zeitabständen die Datei auf Änderungen geprüft und bei Bedarf neu eingelesen wird. Das erlaubt eine Änderung der Konfiguration ohne Neustart der Applikation. Folgendes Beispiel prüft alle fünf Sekunden auf Änderungen und gibt alle aktivierten Log-Level im Takt von zwei Sekunden aus:

```
public static void main(final String[] args)
{
   PropertyConfigurator.configureAndWatch("config/log4j.properties",
FIVE_SECS);
   logger.info("LogReadConfigExample started");

   while (!Thread.currentThread().isInterrupted())
   {
       // Auf allen Leveln ausgeben, anpassbar in log4j.properties
       logger.trace("TRACE");
       logger.debug("DEBUG");
       logger.info("INFO");
       logger.warn("WARN");
       logger.error("ERROR");
```

```
        logger.fatal("FATAL");

        try
        {
            Thread.sleep(TWO_SECS);
        }
        catch (final InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }
}
```

Listing 6.16 *Ausführbar als* **'LOGREADCONFIGEXAMPLE'**

Die Ausgabe steuern wir über entsprechende Einträge in der Konfigurationsdatei. In diesem Fall ist folgende Zeile von Interesse:

```
log4j.logger.ch06_applikationsbausteine.LogReadConfigExample=WARN
```

Das `ConversionPattern 'd{MM-dd HH:mm:ss} -5p [t] c: m%n'` führt zu Ausgaben, die ähnlich zu folgender sind:

```
04-06 21:58:50  WARN [main] ch06_applikationsbausteine.LogReadConfigExample: …
04-06 21:58:50 ERROR [main] ch06_applikationsbausteine.LogReadConfigExample: …
04-06 21:58:50 FATAL [main] ch06_applikationsbausteine.LogReadConfigExample: …
```

Erweiterungen für `FileAppender`

Bei umfangreichen Log-Ausgaben werden die Log-Dateien schnell recht groß. Um eine Weiterverarbeitung und Auswertung zu vereinfachen, ist es daher sinnvoll, die Log-Dateien in handliche »Häppchen« aufzuteilen. Mögliche Kriterien sind dabei die Dateigröße oder das Datum der Log-Ausgaben. Statt eine Verwaltung mühsam selbst zu programmieren, nutzt man die Klassen `RollingFileAppender` und `DailyRollingFileAppender`.

Mithilfe der Klasse `RollingFileAppender` können bei Bedarf fortlaufend neue Log-Dateien erzeugt werden. Das heißt, wenn die Größe der momentanen Log-Datei einen gewissen Wert übersteigt, dann wird automatisch eine neue Log-Datei erzeugt und die bisherige umbenannt. Die Anzahl der so entstehenden Backup-Dateien und die maximal gewünschte Dateigröße sind über die Werte `MaxBackupIndex` und `MaxFileSize` einstellbar. Eine mögliche Konfiguration eines entsprechenden Appenders `MyRollingFile`, die bis zu 31 Log-Dateien jeweils der Größe 5 MB konfiguriert, ist im Folgenden gezeigt:

```
log4j.appender.MyRollingFile=org.apache.log4j.RollingFileAppender
log4j.appender.MyRollingFile.File=RollingLogDatei.log
log4j.appender.MyRollingFile.MaxFileSize=5000KB
log4j.appender.MyRollingFile.MaxBackupIndex=31
log4j.appender.MyRollingFile.layout=org.apache.log4j.PatternLayout
log4j.appender.MyRollingFile.layout.ConversionPattern=%d{MM-dd HH:mm:ss} %-5p
[%
      t] %c: %m%n
```

Ich möchte nochmal explizit darauf hinweisen, dass der Name des eigenen Appenders nach dem Präfix `log4j.appender` folgt und danach die Konfiguration in Form von

Schlüssel-Wert-Paaren angegeben wird.

Erweiterungen für `DailyRollingFileAppender`

Wenn man die Klasse `DailyRollingFileAppender` nutzt, so werden in beliebigen, vorgegebenen Zeitintervallen neue Log-Dateien erzeugt (und nicht nur täglich, wie es der Name der Klasse suggeriert). Nach einer konfigurierbaren Zeit wird dann eine neue Log-Datei angelegt und die bisherigen Informationen werden in einer Log-Datei mit einem konfigurierbaren Zeitstempel gespeichert: Ältere Dateien werden durch einen Zeitstempel im Namen gekennzeichnet. Sowohl der Zeitstempel als auch das Zeitintervall werden über das Attribut `datePattern` festgelegt. Folgende Aufzählung zeigt einige mögliche Varianten:

- **'.'**`yyyy-ww` – Startet wöchentlich eine neue Log-Datei.
- **'.'**`yyyy-MM-dd` – ***Startet täglich eine neue Log-Datei. Dies hat sich in der Praxis für viele Anwendungen bewährt.***
- **'.'**`yyyy-MM-dd_HH` – Startet stündlich eine neue Log-Datei. Normalerweise sollte dies für alle Anwendungen ausreichen. ***Im Muster darf man nicht das Zeichen ':'*** *angeben*[7], da die angegebenen Zeichen Bestandteil des Dateinamens der Log-Datei werden und somit den Dateinamen der Log-Datei ungültig machen.

Folgende Zeilen konfigurieren einen `DailyRollingFileAppender` derart, dass täglich eine neue Datei erzeugt wird:

```
log4j.appender.PDFEditorRollingFileAppender=
                org.apache.log4j.DailyRollingFileAppender
log4j.appender.PDFEditorRollingFileAppender.datePattern='.'yyyy-MM-dd
log4j.appender.PDFEditorRollingFileAppender.file=DailyLogDatei.log
log4j.appender.PDFEditorRollingFileAppender.layout=org.apache.log4j.
      PatternLayout
log4j.appender.PDFEditorRollingFileAppender.layout.ConversionPattern=
                %d{MM-dd HH:mm:ss} %-5p [%t] %c: %m%n
```

Konfiguration der Log-Ausgaben für Packages und Klassen

Neben der Konfiguration der Logger, Appender und Layouts können spezielle Log-Level für bestimmte Packages oder Klassen in der Konfigurationsdatei `log4j.properties` nach folgendem Muster als Schlüssel-Wert-Paare definiert werden:

```
log4j.logger.<meinpackage>.<klasse>=<Log-Level>
```

Manchmal sieht man Einträge, die statt des Wortes `logger` den Begriff `category` verwenden. Dies ist die veraltete, aber immer noch gebräuchliche Form. In beiden Fällen kann dadurch die Ausgabe von Log-Meldungen bis auf Klassenebene feingranular in ihrem Log-Level eingestellt werden. Dabei kann statt eines Packages oder eines konkreten Klassennamens auch die Angabe einer logischen Kategorie, etwa `AudioIn` oder `AudioOut`, erfolgen. Diese werden im folgenden Abschnitt

benutzt, um spezielle, klassenübergreifende Log-Ausgaben zu steuern.

```
%% Hier folgt die Konfiguration für die Packages und Klassen
log4j.logger.ch06_applikationsbausteine.LogCategoryExample=DEBUG

%% Logische Log-Kategorien
log4j.logger.AudioIn=INFO
log4j.logger.AudioOut=INFO
log4j.logger.REPLAY=INFO

%% Hier ändern für LogReadConfigExample-Beispiel
log4j.logger.applikationsbausteine.LogReadConfigExample=WARN
```

Achtung: Fallstricke bei der Angabe von Klassen- und Packagenamen

Ganz wichtig ist es, den voll qualifizierten Namen der Klasse oder des Packages korrekt anzugeben. Wird hierbei ein Tippfehler gemacht, so wirken sich die Konfigurationen nicht aus. In der Praxis ist das ein häufiges Problem, wenn nach Refactorings zum Teil vergessen wird, den korrespondierenden Pfad oder Klassennamen in der Konfigurationsdatei `log4j.properties` anzupassen. Im obigen Beispiel fehlt in einem Fall das Präfix `ch06_` vor `applikationsbausteine`.

6.4.2 Tipps und Tricks zum Einsatz von Logging mit log4j

Abfrage der Log-Level

Für Log-Ausgaben wird der Log-String standardmäßig immer aufbereitet. Gehört er einem Log-Level an, der nicht ausgegeben werden soll, so war diese Stringerzeugung überflüssig. Durch vorherige Abfrage des Log-Levels, etwa `isDebugEnabled()`, kann man die Aufbereitung aufwendiger Log-Ausgaben nur bei Bedarf durchführen und auch nur dann in die Log-Datei schreiben.

Die Abfrage der Log-Level sollte man insbesondere vor der Aufbereitung komplexer Log-Ausgaben und nicht einfach immer durchführen, da ein Programm ansonsten schnell unübersichtlich wird. Dies gilt im Speziellen, wenn Abfragen sehr häufig im Sourcecode auftauchen. Auch hier gilt es, die richtige Balance zu finden. ***Im Zweifelsfall sollte man sich für eine bessere Lesbarkeit und gegen `if`-Abfragen entscheiden. Wenn Profiling-Messungen aber Probleme aufzeigen, kann man eine Einschränkung bezüglich der Lesbarkeit in Kauf nehmen.***

Verwende logische Log-Kategorien

Ein Vorteil von Logging-Frameworks besteht darin, nicht nur auf Klassenebene zu

loggen, sondern eigene Log-Kategorien nutzen zu können, die eine logische, klassenübergreifende Sicht definieren. Damit schafft man die Möglichkeit einer semantischen Trennung. Es werden dazu mehrere Logger pro Datei instanziiert. Beispielsweise kann ein Logger klassenbasierte Ausgaben vornehmen und andere Logger auf logischer Ebene arbeiten. Nehmen wir an, es wären mehrere Klassen an einer Funkkommunikation beteiligt. Man definiert dann die Kategorien `AudioIn` und `AudioOut`, um alle Aktionen im Funkeingang und -ausgang klassenunabhängig und logisch gruppiert auszugeben:

```
public final class LogCategoryExample
{
    private static final Logger logger =
                                Logger.getLogger(LogCategoryExample.class);

    private static final Logger audioInLog  = Logger.getLogger("AudioIn");
    private static final Logger audioOutLog = Logger.getLogger("AudioOut");
    private static final Logger replayLog   = Logger.getLogger("REPLAY");

    LogCategoryExample()
    {
        logger.info("LogCategoryExample created");
    }

    private void send(final byte[] msg)
    {
        logger.debug("send()");
        audioOutLog.info("Sending " + ByteUtils.byteArrayToString(msg));
        replayLog.info("Sending " + ByteUtils.byteArrayToString(msg));
    }

    private byte[] receive(final InputStream inStream)
    {
        logger.debug("receive()");

        final byte[] msg = getMsgFromStream(inStream);
        audioInLog.info("Receiving " + ByteUtils.byteArrayToString(msg));
        replayLog.info("Receiving " + ByteUtils.byteArrayToString(msg));
        return msg;
    }

    public static void main(final String[] args)
    {
        PropertyConfigurator.configureAndWatch("config/log4j.properties");

        final LogCategoryExample logExample = new LogCategoryExample();
        logExample.send("Hello".getBytes());
    }
    // …
```

Listing 6.17 *Ausführbar als* **'LOGCATEGORYEXAMPLE'**

Die jeweiligen Log-Kategorien werden mit in die Log-Ausgabe geschrieben. Wenn alle Log-Einstellungen auf `INFO` stehen, kommt es beispielsweise zu folgender Ausgabe:

```
09-28 23:39:18 INFO  [main] ch06_applikationsbausteine.LogCategoryExample:
      LogCategoryExample created
…
09-28 23:39:18 INFO  [main] AudioOut: Sending 'Hello' = '[72 101 108 108 111
```

```
]'
09-28 23:39:18 INFO  [main] REPLAY: Sending 'Hello' = '[72 101 108 108 111 ]'
```

Logging der Aufrufe an Schnittstellen zu anderen Systemen

Bei einer Zusammenarbeit mit anderen Komponenten oder Systemen kann man durch die Ausgabe der aufgerufenen öffentlichen Methoden mitsamt ihrer Parameter den Programmablauf nachvollziehbar gestalten. Auf diese Weise werden recht schnell Probleme durch inkorrekte Abläufe oder falsch übergebene Daten erkennbar.

Automatische Auswertungsmöglichkeiten

Die in den Log-Dateien protokollierten Meldungen lassen sich mit einem Texteditor anschauen und mit dessen Suchfunktion analysieren. Bei umfangreichen und feingranularen Log-Ausgaben kann dieses Vorgehen zeitaufwendig und unbequem sein.

Weitergehende Möglichkeiten der Auswertung bieten Batch-Skripte. Damit kann man Log-Dateien analysieren und daraus bestimmte Teile extrahieren, etwa auf Basis eines speziellen Log-Levels. Manchmal lässt sich dann mit etwas Erfahrung bereits anhand der Dateigröße der Extrakte erkennen, ob es sich um eine normale Situation handelt oder ob man eine detailliertere Analyse vornehmen muss. Hilfreicher als eine Analyse nach Log-Level ist es in der Regel, nach Vorkommen spezieller Log-Ausgaben in einer Log-Datei zu suchen, etwa Exceptions oder Warnmeldungen. Das `grep`-Tool[8] leistet dabei gute Dienste. Allerdings lassen sich damit nur einzeilige Meldungen besonders gut auswerten – mehrzeilige Stacktraces bereiten dagegen bereits Probleme.

```
grep "ERROR" SampleApplication.log* > All_Errors.txt
grep "WARN"  SampleApplication.log* > All_Warns.txt
grep "REPLAY" -i SampleApplication.log* > All_REPLAY.txt
grep "AudioIn" -i SampleApplication.log* > All_AudioIn.txt
grep "AudioOut" -i SampleApplication.log* > All_AudioOut.txt
```

Tipp: Auswertung von Log-Ausgaben zur Simulation

Man kann Log-Ausgaben so gestalten, dass sie später leicht maschinenlesbar sind. Dadurch lassen sich aus diesen nachträglich Informationen zum Programmablauf gewinnen. Man kann sich dann bei entsprechender Architektur eine Fernsteuerung der Software bauen, die Log-Dateien analysiert und eine Simulation des Programmablaufs basierend auf der Log-Datei ermöglicht.

Verwende mehrere unterschiedliche Konfigurationsdateien

Definiert man sich mehrere Varianten von Log-Konfigurationsdateien, so kann man ohne Programmänderungen bequem mehr oder weniger Detailinformationen abrufen. Dazu kopiert man die jeweilige Konfigurationsdatei in die tatsächlich vom

Programm verwendete `log4.properties`-Datei. Nach kurzer Zeit erkennt log4j die neue Konfiguration und passt die Log-Level für verschiedene Klassen und Pakete an.

Logging von Strings

Bei Ausgaben von textuellen Werten ist es sinnvoll, diese in einfache Anführungszeichen oder alternativ in spitze Klammern einzuschließen:

```
log.debug("Name = '" + name + "', Age = " + age);
log.debug("Name = <" + name + ">, Age = " + age);
```

Dies macht Leerzeichen vor und nach dem eigentlichen Text sichtbar und hilft dabei, tückische Fehler, bei denen Leerzeichen oder unsichtbare Sonderzeichen zu Problemen führen, leichter erkennen zu können (vgl. Abschnitt 4.1.1): Wird etwa mit einem solchen Eingabetext als Schlüssel in einer Map gesucht, so wird man dafür nichts finden.

Logging von Exceptions

Beim Auftreten von Exceptions ist es zur späteren Fehleranalyse meistens hilfreich, einen Stacktrace in die Log-Datei zu schreiben. Leider sieht man häufiger den Fehler, bei einer Log-Ausgabe wichtige Informationen abzuschneiden, etwa dadurch, dass eine Stringrepräsentation einer Exception folgendermaßen erzeugt wird:

```
try
{
    methodThrowingException();
}
catch (final IOException e)
{
    // SCHLECHT: nur String-Info ohne Stacktrace!
    log.error("An I/O error occurred! " + e);
    log.error("An I/O error occurred! " + e.getMessage());
}
```

Listing 6.18 *Ausführbar als* **'EXCEPTIONLOGGINGEXAMPLE'**

Führt man die Beispielapplikation aus, so erhält man in etwa folgende Ausgaben:

```
08-07 ERROR [main] ExceptionLoggingExample: An I/O error occurred! java.io.
      IOException: Text
08-07 ERROR [main] ExceptionLoggingExample: An I/O error occurred! Text
```

Wie man leicht sieht, wird der Stacktrace und damit der zur Fehleranalyse oftmals wichtigste Teil der Exception nicht ausgegeben. Die Ausgabemethoden von log4j besitzen jeweils eine überladene Variante, die durch die folgende kommaseparierte Angabe einer Exception zusätzlich den Stacktrace protokolliert:

```
try
{
    methodThrowingException();
}
catch (final IOException e)
```

```
{
    // String-Info MIT Stacktrace
    LOGGER.error("An I/O error occurred!", e);
}
```

Listing 6.19 *Ausführbar als* **'EXCEPTIONLOGGINGIMPROVED'**

Eine Ausgabe mit Stacktrace erleichtert ein Nachvollziehen des Wegs durch das Programm bis zu der Stelle, die den Fehler verursacht hat:

```
04-04 20:45:06 ExceptionLoggingImproved   - An I/O error occurred!
java.io.IOException: Text
    at ExceptionLoggingImproved.second(ExceptionLoggingImproved.java:42)
    at ExceptionLoggingImproved.first(ExceptionLoggingImproved.java:37)
    at
ExceptionLoggingImproved.methodThrowingException(ExceptionLoggingImproved
        .java:31)
    at ExceptionLoggingImproved.main(ExceptionLoggingImproved.java:19)
```

Stacktrace ausgeben

Manchmal ist eine Fehlersuche schwierig, etwa weil man in einigen Situationen keinen Debugger verwenden kann[9] oder Fehler extrem selten auftreten und sich dadurch nicht nachstellen lassen. Zur Fehleranalyse ist es wünschenswert, an dieser Stelle im Programm den aktuellen Stacktrace ausgeben zu können, ohne das Programmverhalten zu verändern. Die Klasse `Thread` bietet dazu die statische Methode `dumpStack()`. Die Ausgabe des Stacktrace erfolgt auf dem Error-Stream, also in der Regel auf der Konsole.

Eine Ausgabe in die Log-Datei erreicht man durch folgenden Trick: Um das Programmverhalten nicht zu verändern, wird eine Exception beliebigen Typs (bevorzugt aber `IllegalStateException`) lediglich erzeugt, aber ***nicht*** geworfen, sondern stattdessen direkt an die Log-Ausgabe-Methode übergeben. Als Folge erhält man durch den mitgelieferten Stacktrace einen Einblick in die internen Programmabläufe.

```
private static void provideStackTrace()
{
    log.info("Stacktrace: ", new IllegalStateException("Stacktrace!"));
}
```

Listing 6.20 *Ausführbar als* **'PROVIDESTACKTRACE'**

6.5 Konfigurationsparameter und -dateien

Die meisten Anwendungen bieten Möglichkeiten zur Konfiguration. Im Folgenden zeige ich verschiedene Varianten, Konfigurationsdaten zu verarbeiten, und nenne deren jeweilige Stärken und Schwächen. Abschnitt 6.5.1 beschreibt, wie man Kommandozeilenparameter entgegennehmen und auswerten kann. Eine Konfiguration mithilfe der Klasse `java.util.Properties` wird in Abschnitt 6.5.2 vorgestellt. Als Ergänzung lernen wir in Abschnitt 6.5.3 die Klasse `java.util.prefs.Preferences` kennen. Konfigurationswerte können auch in

einem kommaseparierten Format (CSV) oder in Form von XML vorliegen. In Abschnitt 6.5.4 werden einige Ideen zum Einlesen derart gespeicherter Werte vorgestellt. Weiterführende Informationen zur Verarbeitung von XML finden Sie im Buch »Java & XML« von Brett McLaughlin [64].

6.5.1 Einlesen von Kommandozeilenparametern

Eine Möglichkeit zur Konfiguration einer Java-Applikation besteht darin, dieser bei deren Start eine beliebige Anzahl an Parametern zu übergeben, etwa wie folgt:

```
java MyApp Parameter1 Parameter2 "Text mit Leerzeichen"
```

Dabei ist zu beachten, dass Leerzeichen die Kommandozeilenargumente trennen: ***Wenn ein String mit Leerzeichen übergeben werden soll, so muss dies in Anführungszeichen geschehen.*** Im Beispiel würde ansonsten der Parameterwert `Text mit Leerzeichen` als Angabe von drei Parametern ausgewertet.

Bevor ich mit Apache Commons CLI eine frei verfügbare Bibliothek zur Verarbeitung von Kommandozeilenargumenten vorstelle, zeige ich zunächst Notationsformemn für Parameter und gehe dann anschließend auf Varianten der Auswertung per Hand ein, um Sie für mögliche Schwierigkeiten zu sensibilisieren. Als Beispiel betrachten wir eine Parameterübergabe, um damit sowohl die Größe des Applikationsfensters zu ändern als auch optional spezielle Debug-Informationen zu aktivieren.

Notationsformen für Parameter

Folgende zwei Varianten bei der Parameterübergabe sind gebräuchlich:

1. **Fixe Reihenfolge** – Die Bedeutung eines übergebenen Werts wird analog zu einem Methodenaufruf anhand seiner Position ermittelt. Der Aufruf könnte etwa

   ```
   java MyApp 500 300 debug
   ```

   lauten. ***Diese starre Zuordnung ist wenig flexibel, birgt die Gefahr der Verwechselung und ist außerdem problematisch bei optionalen Parametern.*** Zudem kann nur ein optionaler Parameter am Ende der Parameterliste angegeben werden.

2. **Benannte Parameter** – Jeder Parameter wird durch einen Namen oder ein entsprechendes Kürzel eingeleitet und beschreibt so seine Intention, etwa wie folgt:

   ```
   java MyApp width 500 height 300 debug
   ```

   Diese Variante erlaubt eine flexible Reihenfolge und eine beliebige Zahl optionaler Parameter, jedoch zulasten einer komplexeren Auswertung als bei Variante 1.

Gebräuchlicher ist die zweite Variante, wobei dem Parameternamen meist ein Minuszeichen vorangestellt wird. Die Übergabe von Werten kann folgendermaßen

geschehen:

- **-<name> <Wert>**, etwa `-width 200`
- **-<name>=<Wert>**, etwa `-width=200`

Die erste Form besteht aus zwei Kommandozeilenparametern, die eine Einheit (Name und Wert) bilden. Die zweite Form entspricht einem Kommandozeilenparameter, was ein zusätzliches Parsing zur Trennung von Parameter und Wert erfordert. Boolesche Flags werden durch Parameter ohne Wertangabe, etwa `-debug`, ausgedrückt. Zusätzliche Komplexität entsteht, wenn man auch Kurzformen von Parameternamen unterstützen möchte, etwa `-h` statt `-help` oder `-d` statt `-debug`.

Auswertung von Parametern

Kommandozeilenargumente werden an die Methode `main(String[])` als `String[]` übergeben. Um daraus Daten zu extrahieren und zu speichern, wird hier für jeden Parameter eine Variable (`width`, `height`, `debug` und `showHelp`) definiert und mit einem Defaultwert belegt.

Im Beispiel nutzen wir das `String[] sampleArgs` zur Simulation von Parametern. Diese Werte werden mit einer Schleife durchlaufen und mit den erwarteten Parameterkürzeln bzw. Parameternamen verglichen:

```
public static void main(final String[] args)
{
    // Defaultwerte, wenn Wert nicht in der Kommandozeile übergeben
    int width = 700;
    int height = 200;
    boolean debug = false;
    boolean showHelp = false;

    // Test mit festdefinierten Werten
    final String[] sampleArgs = { "-h", "-w=550", "-height=550" };
    for (final String cmdArg : sampleArgs)
    {

        if (cmdArg.startsWith("-d"))          // check debug
            debug = true;
        else if (cmdArg.startsWith("-h"))     // check help
            showHelp = true;
        else if (cmdArg.startsWith("-w="))    // check width
            width = Integer.parseInt(cmdArg.substring(3));
        else if (cmdArg.startsWith("-width="))
            width = Integer.parseInt(cmdArg.substring(7));

        // check height !!! UNREACHABLE !!!
        else if (cmdArg.startsWith("-h="))
            height = Integer.parseInt(cmdArg.substring(3));
        else if (cmdArg.startsWith("-height="))
            height = Integer.parseInt(cmdArg.substring(8));
    }
    // …
```

Listing 6.21 *Ausführbar als* **'ARGSPARSINGEXAMPLENAIV'**

Die Parameterauswertung führt schnell zu umfangreichen `if-else`-Gebilden, die schon für wenige Parameter unübersichtlich sind. Das Beispiel enthält bewusst einen Flüchtigkeitsfehler: Parameter mit gleichem Startbuchstaben werden nicht unterschieden. Hier gilt das für die Abfrage `startsWith( "-h")` zur Aktivierung der Hilfe. Dadurch werden Angaben zur Höhe des Fensters mit `-h=` oder `-height=` niemals ausgewertet. Ein Start des Programms ARGSPARSINGEXAMPLENAIV zeigt das: Die Höhe des Fensters entspricht dem vordefinierten Wert von 200 Pixel und nicht dem übergebenen Wert `-height=550` (vgl. Abbildung 6-2).

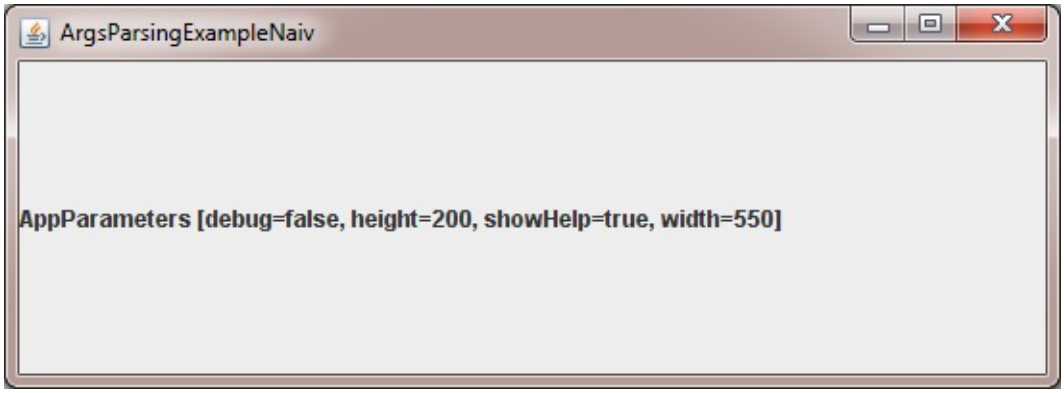


Abbildung 6-2 *Applikationsfenster, Größe basierend auf Kommandozeilenparametern*

Ein erster Ansatz zur Korrektur und damit zur Auswertung aller Parameter trotz gleicher Startzeichenfolge ist, Alternativen jeweils mit `if` statt mit `else if` auszudrücken, sodass immer alle `if`-Zweige betrachtet werden. Dabei kommt es jedoch zu anderen Fallstricken: Eine Angabe von `-h=200` setzt nicht nur den Wert für die Höhe. Es wird zusätzlich unerwartet auch der Wert für die Aktivierung der Hilfe auf `true` gesetzt. Außerdem können weitere Probleme durch die Auswertung mit `startsWith( String)` auftreten. Man kann etwa `-hello` schreiben. Das wird aber wie der Aufruf von `-h` interpretiert. Mehr noch: Das Auslesen von Werten ist durch die Abstützung auf sogenannte Magic Strings und Magic Numbers fragil.[10] Wird der Name eines Parameters umbenannt, besteht die Gefahr, dass übersehen wird, die korrespondierende Längenangabe im Aufruf von `substring( int)` anzupassen.

Verbesserungen

Ein möglicher Lösungsansatz besteht darin, Parameternamen und Kurzformen als Konstanten zu definieren. Dies lässt sich recht elegant als `enum`-Aufzählung `AppParameter` wie folgt realisieren:

```
enum AppParameter
{
    DEBUG_SHORT( "d"),
    HELP_SHORT( "?"),
    WIDTH_SHORT( "w="), WIDTH( "width="),
    HEIGHT_SHORT( "h="), HEIGHT( "height=");

    final String paramName;
    final int    valueOffset;
```

```
    AppParameter(final String name)
    {
        this.paramName = "-" + name;
        this.valueOffset = paramName.length();
    }
}
```

Diese Art der Realisierung hat den Vorteil, dass an einer zentralen Stelle übersichtlich alle erlaubten Parameter definiert werden können. Dadurch vermeidet man eine Sourcecode-Analyse tief verschachtelter `if`-Anweisungen. Zudem kann man an einer zentralen Stelle die Zeichen '-' (Präfix für alle Parameter) und ' =' (signalisiert Parameter mit nachfolgendem Wert) mit dem Parameternamen verbinden und hält den Applikationscode frei von diesen Details. Weiterhin wird die Länge des Parameterkürzels inklusive der Start- und Endemarkierungen ('-' bzw. '=') nur einmal bei der Konstruktion berechnet, sodass selbst bei Änderungen in den Parametern keine Modifikationen in einsetzenden Applikationen nötig werden.

Um die Auswertung der Parameter zu erleichtern, nachdem sie aus dem `String[]` ausgelesen worden sind, definieren wir hier folgende drei Hilfsmethoden:

```
private static boolean hasBooleanParam(final String cmdArg,
                                       final AppParameter parameter)
{
    return cmdArg.equals(parameter.paramName);
}

private static boolean hasValuedParam(final String cmdArg,
                                      final AppParameter parameter)
{
    return cmdArg.startsWith(parameter.paramName) &&
           cmdArg.length() > parameter.valueOffset;
}

private static int extractInt(final String cmdArg, final AppParameter
parameter)
{
    return Integer.parseInt(cmdArg.substring(parameter.valueOffset));
}
```

Nutzen wir die `enum`-Aufzählung und die obigen Methoden bei der Auswertung der Parameter, so ergibt sich folgender Sourcecode:

```
for (final String cmdArg : sampleArgs)
{
    if (hasBooleanParam(cmdArg, AppParameter.DEBUG_SHORT))
        debug = true;
    if (hasBooleanParam(cmdArg, AppParameter.HELP_SHORT))
        showHelp = true;

    if (hasValuedParam(cmdArg, AppParameter.WIDTH_SHORT))
        width = extractInt(cmdArg, AppParameter.WIDTH_SHORT);
    if (hasValuedParam(cmdArg, AppParameter.WIDTH))
        width = extractInt(cmdArg, AppParameter.WIDTH);

    if (hasValuedParam(cmdArg, AppParameter.HEIGHT_SHORT))
        height = extractInt(cmdArg, AppParameter.HEIGHT_SHORT);
    if (hasValuedParam(cmdArg, AppParameter.HEIGHT))
```

```
        height = extractInt(cmdArg, AppParameter.HEIGHT);
}
```

Der Sourcecode ist nun verständlicher und deutlich besser erweiterbar. Dies liegt zum einen am Einsatz des Aufzählungstyps und zum anderen an den drei eingeführten Hilfsmethoden, die zur Lesbarkeit beitragen. Wir erkennen, dass sich beispielsweise die zuvor komplizierte und fehleranfällige Abfrage mit

```
else if (cmdArg.startsWith("-width="))
    width = Integer.parseInt(cmdArg.substring(7));
```

durch den Einsatz des Aufzählungstyps und der Hilfsmethoden lesbar, verständlich und typsicher wie folgt schreiben lässt:

```
else if (hasValuedParam(cmdArg, AppParameter.WIDTH))
    width = extractInt(cmdArg, AppParameter.WIDTH);
```

Außerdem werden durch gleiche Startbuchstaben für verschiedene Parameter verursachte Probleme unwahrscheinlich. Zum einen sind alle Parameter innerhalb eines Aufzählungstyps definiert und so fallen gleiche Parameternamen schneller auf als in verschachtelten `if`-Anweisungen. Zum anderen erfolgt in den beiden Methoden `hasBooleanParam(String, AppParameter)` und `hasValuedParam(String, AppParameter)` die Abfrage auf Übereinstimmung unterschiedlich: Boolesche Parameter werden auf exakte textuelle Gleichheit geprüft, für Werteparameter wird weiterhin `startsWith(String)` genutzt.

Verbliebene Schwachstellen und Lösungen

Trotz der erzielten Verbesserungen erfordert die Unterscheidung und Auswertung von Kurz- oder Langform eines Parameters mindestens zwei `if`-Anweisungen und ist dadurch noch nicht so gut lesbar und erweiterbar, wie es wünschenswert wäre.

Weiterhin fällt beim genaueren Betrachten des Sourcecodes auf, dass an verschiedenen Stellen Zugriffe auf Variablen erfolgen. Steigt die Anzahl der auszuwertenden Parameter, so kann man für mehr Klarheit sorgen, wenn man Variablen gemäß dem Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) zusammenfasst. Im weiteren Verlauf werden wir daher folgendes Parameter Value Object `AppParameterVO` verwenden. Durch dieses erfolgt eine zentrale Definition aller durch Kommandozeilenparameter beeinflussten Variablen. Bei der Implementierung wird bewusst auf `get()` - und `set()` -Methoden verzichtet, da es sich lediglich um einen einfachen Behälter für Werte handelt, der zudem nur innerhalb des Packages selbst verwendet werden soll. Daher werden die Attribute zur Einschränkung von Zugriffsmöglichkeiten Package-private definiert:

```
class AppParameterVO
{
    // Defaultwerte, wenn Wert nicht in der Kommandozeile übergeben
    int     width    = 700;
    int     height   = 200;
    boolean debug    = false;
    boolean showHelp = false;
```

```
    @Override
    public String toString()
    {
        return "AppParameterVO [debug=" + debug + ", height=" + height + ", "
+
                               "showHelp=" + showHelp + ", width=" + width + "]";
    }
}
```

Eine weitere Verbesserung der Auswertung von Hand ist nur schwierig möglich. Es ist an der Zeit, eine Bibliothek einzusetzen, um die Auswertung weiter zu erleichtern.

Apache Commons CLI

Die Bibliothek Apache Commons CLI (`http://commons.apache.org/cli/`) ist nützlich, wenn die Anzahl zu verarbeitender Parameter zunimmt oder die Auswertung anderweitig komplex ist. Zum Einbinden der Funktionalität fügt man Folgendes in der Build-Datei hinzu:

```
compile 'commons-cli:commons-cli:1.2'
```

Definition von Parametern

Parameter werden als sogenannte Optionen repräsentiert. Die vom Programm unterstützten Parameter werden dazu in einem Container vom Typ `Options` gesammelt. Einzelne Parameter (`Option`-Objekte) werden durch Aufruf der Methode `addOption(String opt, String longOpt, boolean hasArg, String description)` dem `Options`-Container hinzugefügt.[11]

Wollten wir lediglich einen Parameter zur Anzeige von Hilfetexten erlauben, so kann man einen Container `allowedOptions` vom Typ `Options` erzeugen und die Hilfeoption wie folgt hinzufügen:

```
final Options allowedOptions = new Options();
allowedOptions.addOption("?", "help", false, "show help");
```

Die CLI-Bibliothek erwartet bei der Definition – im Gegensatz zur späteren Auswertung – für die Kurz- und Langnamen kein Minuszeichen. Über den dritten Wert, hier `false`, steuert man, ob ein Parameter Argumente hat. Zusätzlich kann eine Parameterbeschreibung übergeben werden, die beim Aufbau einer Hilfeseite nützlich ist.

Vereinfachte Definition von Parametern

Anstatt mühselig jeden Parameter in einer eigenen Programmzeile durch einen Aufruf der Methode `addOption()` hinzuzufügen, wäre eine geschicktere Realisierung wünschenswert. Tatsächlich ist dies relativ einfach durch kleinere Änderungen möglich. Dazu wird die bereits existierende `enum`-Aufzählung `AppParameter` dahingehend erweitert, dass sie alle Informationen bereitstellen kann, die die Methode `addOption(String, String, boolean, String)`

der Klasse `Options` als Eingabe benötigt. Wir realisieren die folgende `enum`-Aufzählung `AppParameterCLI`, die alle zur Konfiguration eines korrespondierenden `Option`-Objekts erforderlichen Informationen vorhält:

```
public enum AppParameterCLI
{
    WIDTH("w", "width", true, "set the width"),
    HEIGHT("h", "height", true, "set the height"),
    HELP("?", "help", false, "show help"),
    DEBUG("d", "debug", false, "activate debug mode");

    final String shortname;
    final String name;
    final boolean hasArgs;
    final String helptext;

    AppParameterCLI(final String shortname, final String name,
                    final boolean hasArgs, final String helptext)
    {
        this.shortname = shortname;
        this.name = name;
        this.hasArgs = hasArgs;
        this.helptext = helptext;
    }
}
```

Mit der Methode `values()` der `enum`-Aufzählung lässt sich eine elegante Definition aller zu unterstützenden Parameter wie folgt implementieren:

```
public final class CommandLineParsingExample
{
    public static void main(final String[] args)
    {
        final Options allowedOptions = new Options();

        // Alle Options basierend auf enum AppParameterCLI hinzufügen
        for (final AppParameterCLI parameter : AppParameterCLI.values())
        {
            allowedOptions.addOption(parameter.shortname, parameter.name,
                                     parameter.hasArgs, parameter.helptext);
        }
        //…
```

Listing 6.22 *Ausführbar als* **'COMMANDLINEPARSINGEXAMPLE'**

Auswertung von Parametern

Nachdem alle gewünschten Parameter so definiert wurden, können beim Start eines Programms Eingabewerte aus der Kommandozeile geparst werden. Dazu wird zunächst ein `CommandLineParser`-Objekt erzeugt. Mit dessen Methode `parse(String[] args)` erhält man ein `CommandLine`-Objekt. Dieses kann man über die Methode `hasOption()` befragen, ob eine gewünschte Option aktiviert wurde, d. h. als Parameter angegeben ist. Im folgenden Listing, das das obige Listing COMMANDLINEPARSINGEXAMPLE fortsetzt, ist dies gezeigt:

```
    //…
        // Test mit festdefinierten Werten
```

```
        final String[] sampleArgs = new String[] { "-d", "-?", "-w", "444",
                                                   "—height", "99" };


        final AppParameterVO parameters = new AppParameterVO();


        try
        {
            // Kommandozeilenparameter parsen
            final CommandLineParser parser = new PosixParser();
            final CommandLine line = parser.parse(allowedOptions, sampleArgs);


            // Prüfen der Optionen
            parameters.debug = hasOption(line, AppParameterCLI.DEBUG);
            parameters.showHelp = hasOption(line, AppParameterCLI.HELP);


            if (hasOption(line, AppParameterCLI.WIDTH))
                parameters.width = extractInt(line, AppParameterCLI.WIDTH);


            if (hasOption(line, AppParameterCLI.HEIGHT))
                parameters.height = extractInt(line, AppParameterCLI.HEIGHT);


            final JFrame appFrame = new AppFrame("CommandLineParsingExample",
                                                 parameters);
            appFrame.setVisible(true);
        }
        catch (final ParseException exp)
        {
            printHelp(allowedOptions);
        }
    }

    private static boolean hasOption(final CommandLine line,
                                     final AppParameterCLI parameter)
    {
        return line.hasOption(parameter.name);
    }


    private static int extractInt(final CommandLine line,
                                  final AppParameterCLI parameter)
    {
        return Integer.parseInt(line.getOptionValue(parameter.name));
    }


    private static void printHelp(final Options allowedOptions)
    {
        final HelpFormatter formatter = new HelpFormatter();
        formatter.printHelp("CommandLineParsingExample", allowedOptions);
    }
    // …
```

Wie man leicht sieht, ist diese Art der Auswertung deutlich übersichtlicher und birgt wesentlich weniger Potenzial für Fehler. Zudem ist eine Fehlerbehandlung in das Parsing der CLI-Bibliothek integriert: Wurde versehentlich eine unbekannte Option

übergeben, so wird eine `org.apache.commons.cli.ParseException` geworfen. Eine angemessene Reaktion kann beispielsweise die Ausgabe aller unterstützten Parameter sein. Wenn man dazu die Klasse `HelpFormatter` und deren Methode `printHelp()` nutzt, dann wird Folgendes aufbereitet:

```
usage: CommandLineParsingExample
-?,–help           show help
-d,–debug          activate debug mode
-h,–height <arg>   set the height
-w,–width  <arg>   set the width
```

Dies ist insofern praktisch, als eine Aufbereitung einer Hilfeseite ansonsten eine aufwendige und fehleranfällige Aufgabe ist.

Achtung: Groß-/Kleinschreibung von Parametern

Weitere Komplexität kommt ins Spiel, wenn man verschiedene Schreibweisen von Parameternamen erlauben möchte. In der Praxis werden häufig der Einfachheit halber die übergebenen Parameternamen in Kleinschreibung umgewandelt, bevor die Auswertung startet.

6.5.2 Verarbeitung von Properties

Neben der Übergabe von Kommandozeilenparametern lassen sich Einstellungen auch aus Property-Dateien oder aus Umgebungsvariablen auslesen. In beiden Fällen liegen die Informationen in Form von Schlüssel-Wert-Paaren vor, wobei sowohl Schlüssel als auch Wert vom Typ `String` sind. Wenn solche Werte aus einer Konfigurationsdatei ermittelt werden sollen, bietet sich der Einsatz der Klasse `java.util.Properties` an. Diese erledigt das Einlesen der Datei und das Parsen der Wertepaare und bietet anschließend Zugriff über die Methode `getProperty(String)`.

Erste Realisierung mit der Klasse `Properties`

Betrachten wir die Verarbeitung von Konfigurationsdaten mit der Klasse `Properties` konkret am Beispiel der folgenden Property-Datei `AppConfig.properties` aus dem Konfigurationsordner `config` unseres Projektverzeichnisses. Nehmen wir an, diese enthielte den Pfad auf ein extern zu startendes Programm sowie eine Parametrierung eines Datenbankzugriffs:

```
pdf.viewer=AcroRd32.exe

db.user=myuser
db.password=mypwd
```

Zum Einlesen von Konfigurationsdaten aus einem `InputStream` bietet die Klasse `Properties` die Methode `load(InputStream)`. Danach kann über die Methode `getProperty(String)` auf die Konfigurationswerte zugegriffen werden:

```
public static final void main(final String[] args) throws IOException
{
    final Properties props = new Properties();

    try (final InputStream in = new BufferedInputStream(
                new FileInputStream("config/AppConfig.properties")))
    {
        props.load(in);

        final String appPdfViewer = properties.getProperty("pdf.viewer");
        System.out.println("PdfViewer = '" + appPdfViewer + "'");

        final String dbUser = properties.getProperty("db.user");
        final String dbPwd = properties.getProperty("db.password");
        System.out.println("DB-USER/PWD = '" + dbUser + "'/'" + dbPwd + "'");
    }
}
```

Listing 6.23 *Ausführbar als* **'APPPROPERTIESFIRST'**

Für dieses Beispiel kommt es erwartungsgemäß zu folgender Ausgabe:

```
PdfViewer = 'AcroRd32.exe'
DB-USER/PWD = 'myuser'/'mypwd'
```

Falls eine Konfigurationsdatei tatsächlich nur einige wenige Konfigurationsparameter beherbergt, dann ist die gewählte Umsetzung in Ordnung, wenn auch nicht elegant. Man erkennt bereits jetzt die Fragilität: Durch hartcodierte Strings als Schlüsselwerte ist es möglich, eine Abfrage mit jedem beliebigen und nicht nur den definierten Namen für Konfigurationsparameter durchzuführen. Ein Tippfehler in der Konfigurationsdatei oder bei der Abfrage führt zu unerwarteten Resultaten.

Betrachten wir aber noch ein subtileres Problem, das auftreten kann, wenn man ein wenig unvorsichtig programmiert: Die Klasse `Properties` ist von der Klasse `Hashtable<Object, Object>` abgeleitet (Implementierungsvererbung) und bietet dadurch sämtliche darin definierten Methoden. Neben der typsicheren Speicherung von Schlüsseln und Werten vom Typ `String` über die Methode `setProperty(String, String)` sind demnach auch folgende direkte Aufrufe der `put(Object, Object)`-Methode der `Hashtable<K,V>` möglich, aber wenig sinnvoll:

```
public static final void main(final String[] args) throws IOException
{
    final Properties props = new Properties();

    try (final InputStream in = new BufferedInputStream(
                new FileInputStream("config/AppConfig.properties")))
    {
        props.load(in);

        // unerwartet kann man beliebige Objekte in Properties speichern
        // Einfügen eines Person-Objekts
        props.put("MIC", new Person("Micha", "Aachen", 39));

        // kein Zugriff auf Property mit getProperty()
        System.out.println("getProperty()='" + props.getProperty("MIC") +
"'");
```

```
        // aber Zugriff über die Basisklasse mit get()
        System.out.println("get()='" + props.get("MIC") + "'");
    }
}
```

Listing 6.24 *Ausführbar als* **'APPPROPERTIESPROBLEMS'**

Startet man das Programm AppPropertiesProblems, so sieht man, dass unerwarteterweise ein neues, beliebiges Property mit dem Schlüssel `MIC` und einem `Person`-Objekt als Wert eingefügt und auch ausgelesen werden kann:

```
getProperty()='null'
get()='Person: Name='Micha' City='Aachen' Age='39"
```

In der Regel ist die Anzahl der Konfigurationsparameter und die Komplexität der Zugriffe deutlich größer als in den obigen Beispielen. Dann gewinnen Themen wie eine bessere Abstraktion, ein Fehler vermeidender Zugriff und eine zentrale Fehlerbehandlung mehr an Bedeutung. Damit ergeben sich nachfolgend aufgezählte Verbesserungswünsche beim Zugriff auf Konfigurationsparameter:

- **Definition eines Aufzählungstyps** – Festlegung auf eine fest definierte Menge an Schlüsseln zum Zugriff auf die Konfigurationswerte
- **Kapselung** – Einschränkung des Zugriffs auf Aufzählungswerte und bei der Speicherung auf Werte vom Typ `String`
- **Fehlerbehandlung** – Zentrale Behandlung von Fehlern beim Dateizugriff mit der Möglichkeit, in einsetzenden Applikationen darauf reagieren zu können
- **Zentralisierung** – Zentrale Definition und Abstraktion von einer konkreten Konfigurationsdatei

Verbesserung durch Definition eines Aufzählungstyps

Das es sich bei Konfigurationsparametern meistens um eine überschaubare Anzahl an Werten handelt, bietet es sich an, eine `enum`-Aufzählung mit gültigen Property-Namen zu definieren, wodurch man Magic Strings beim Zugriff auf die Properties vermeidet.

```
enum PropertyName
{
    PDF_VIEWER("pdf.viewer"), DB_USER("db.user"), DB_PASSWORD("db.password");

    final String propertyKey;

    PropertyName(final String propertyKey)
    {
        this.propertyKey = propertyKey;
    }
}
```

Durch diese Art der Definition können mögliche Probleme durch Tippfehler (nahezu) ausgeschlossen werden. Das gilt – wie bereits bemerkt – für Tippfehler in der Applikation. Derartige Fehler in der Konfigurationsdatei erschlägt man damit nicht.

Verbesserung der Kapselung

Um die Verarbeitung von Konfigurationsdateien zu vereinfachen und um Implementierungsdetails zu verstecken, ist es sinnvoll, in eigenen Applikationen Zugriffe auf die Klasse `Properties` zu kapseln. Wir definieren dazu eine Klasse `AppProperties`. Anstelle von der Klasse `Properties` abzuleiten, nutzen wir hier Delegation. Somit können wir die Schnittstelle selbst festlegen und verhindern ungewünschte Zugriffe auf die Klasse `Properties` bzw. deren Basisklasse `Hashtable<K, V>`. Um auch Inkonsistenzen bei konkurrierenden Zugriffen bei Multithreading zu vermeiden, sind einige Methoden synchronisiert.

Durch den Einsatz von Delegation muss kein fremdes API realisiert werden. Stattdessen können wir die Schnittstelle der eigenen Klasse anforderungsgerecht zuschneiden und nur tatsächlich benötigte Methoden realisieren. Weiterhin können Zugriffsmethoden ausschließlich mit Aufzählungswerten arbeiten, sodass der Zugriff einerseits typsicher und andererseits lediglich auf vorhandene Elemente möglich ist. Anwendungsfehler durch falsche Schlüsselwerte, wie sie bei Schlüsseln vom Typ `String` auftreten können, werden somit automatisch ausgeschlossen:

```
public synchronized String getProperty(final PropertyName key)
{
    return properties.getProperty(key.propertyKey);
}

public synchronized void setProperty(final PropertyName key, final String
value)
{
    properties.setProperty(key.propertyKey, value);
}
```

Der Zugriff auf Konfigurationswerte ist nun nur noch über eine wohldefinierte Menge von Schlüsselwerten möglich. Mit den folgenden Erweiterungen steht diese Funktionalität überall im Programm zur Verfügung, ohne dass dafür eigener Sourcecode geschrieben werden müsste.

Verbesserung der Fehlerbehandlung

Die Behandlung von Fehlern wird in aufrufenden Programmteilen erleichtert, indem eine Zugriffsmethode `getPropertyFilePath()` bereitgestellt wird, die den voll qualifizierten Pfad zur Konfigurationsdatei zurückliefert:

```
public static String getPropertyFilePath()
{
    return new File(FILE_PATH).getAbsolutePath();
}
```

Verbesserung durch Zentralisierung

Zum zentralen Zugriff auf die Konfigurationsdaten wird die Klasse `AppProperties` als SINGLETON (vgl. Abschnitt 18.1.4) realisiert. Der Dateizugriff findet in der Methode `readAppProperties()` statt. Ebenso können die Werte mit einer

korrespondierenden Methode `writeAppProperties()` gesichert werden:

```
public final class AppProperties
{
    private static final String FILE_PATH = "config/AppConfig.properties";

    private static final AppProperties INSTANCE = new AppProperties();

    private final Properties properties = new Properties();

    public static final AppProperties getInstance()
    {
        return INSTANCE;
    }

    public synchronized void readAppProperties() throws IOException
    {
        try (final InputStream in = new BufferedInputStream(
                                    new FileInputStream(FILE_PATH)))
        {
            properties.load(in);
        }
    }

    public synchronized void writeAppProperties() throws IOException
    {
        try (final OutputStream out = new BufferedOutputStream(
                                      new FileOutputStream(FILE_PATH)))
        {
            properties.store(out, "");
        }
    }
    // …
```

Bei Dateizugriffen kann es immer zu Fehlern kommen. ***Es wäre schlechtes Design, eine Fehlerbehandlung in einer solchen Verwaltungsklasse durch Darstellen einer Fehlermeldungsbox zu realisieren.*** Damit höhere Schichten der Applikation auf Fehlersituationen reagieren können, werden möglicherweise auftretende `IOExceptions` propagiert. Außerdem wird der zum Einlesen verwendete Stream automatisch geschlossen, damit keine Implementierungsdetails, in diesem Fall eine Referenz auf den Stream, nach außen gegeben werden müssen.

Fazit

Mit den realisierten Erweiterungen konnten die angesprochenen Probleme gelöst werden. Weiterhin ist es mit der geänderten Realisierung sogar möglich, Verhalten zu variieren. Konfigurationsdaten können bei Bedarf mithilfe anderer Klassen verwaltet oder in einer anderen Form gespeichert werden, z. B. als XML-Datei oder in einer Datenbank. Der Zugriff auf die konkrete Form der Datenhaltung kann versteckt werden. Beachten Sie, dass wir keinen dieser Vorteile hätten, wenn wir von der Klasse `Properties` abgeleitet hätten! Wir erkennen hier, warum es häufig empfehlenswert ist, Komposition und Delegation statt Vererbung zur Erweiterung von Funktionalität zu nutzen.

Bewertung der Klasse `Properties`

Ein Einsatz der Klasse `Properties` bietet sich bei hauptsächlich lesendem Zugriff von Konfigurationswerten an. Es lassen sich so verschiedene Einstellungen eines Programms aus Dateien einlesen. Der große Vorteil ist die menschenlesbare Speicherung der Daten und die damit verbundene leichte Änderbarkeit mit einem beliebigen Texteditor.

Werden Änderungen an den Voreinstellungen aus einem Programm heraus vorgenommen und sollen diese anschließend wieder persistiert werden, so macht sich die Implementierungsvererbung von der Klasse `Hashtable<K, V>` negativ bemerkbar: Die Speicherung der Werte folgt keiner festen Ordnung – zudem gehen per Hand eingefügte Kommentare und Formatierungen als Folge einer Speicherung aus dem Programm heraus verloren. Schwerwiegender sind aber folgende weitere, konzeptuelle Unzulänglichkeiten, die unabhängig von der Nutzung einer `Hashtable<K, V>` sind:

- Es wird nicht zwischen globalen und benutzerspezifischen Einstellungen unterschieden.
- Eine hierarchische Strukturierung der Daten wird nicht unterstützt, sondern lediglich eine ungeordnete Sammlung verschiedener Schlüssel-Wert-Paare.

Beide Nachteile könnten durch Nutzung mehrerer `Properties` ausprogrammiert werden. Eine einfache Strukturierung der Konfigurationswerte wird häufig durch eine Punktnotation in der Art `Gliederungsebene1.Ebene2.Wert` nachgebildet. Dies wurde bereits für die Trennung von Parametern zur PDF-Verarbeitung mit Präfix `pdf` und zur Konfiguration einer Datenbank mit Präfix `db` eingesetzt. Die Realisierung von allgemeinen und benutzerspezifischen Einstellungen wird durch den Einsatz der Klasse `Preferences` vereinfacht. Dies beschreibt Abschnitt 6.5.3.

Tipp: System-Properties und Umgebungsvariablen

System-Properties Standardmäßig sind diverse Informationen in sogenannten System-Properties gespeichert, etwa Informationen über das Betriebssystem, die Java-Version usw. In Tabelle 6-2 sind einige nützliche System-Properties aufgelistet.

Beim Start eines Java-Programms können zusätzliche Werte per Kommandozeilenoption '`-D`' und der Syntax '`-DmyPropKey=myValue`' angegeben werden. Dabei folgt der Name des Schlüssels, hier »`myPropKey`«, direkt dem '`-D`'. Die so angegebenen Werte werden zu den normalen System-Properties hinzugefügt. Die Abfrage einzelner Werte erfolgt über die Klasse `System` und deren Methode `getProperty(String)`. ***Es empfiehlt sich jedoch, eigene Properties zu verwenden, um die System-Properties nicht zu »verschmutzen«.***

Umgebungsvariablen Über die statische Methode `getenv()` der Klasse `System`

erhält man Zugriff auf Umgebungsvariablen des Betriebssystems:

```
public static void main(final String[] args)
{
    final Map<String, String> systemEnv = System.getenv();

    for (final Map.Entry<String, String> entry : systemEnv.entrySet())
    {
        System.out.println("Key = " + entry.getKey() +
                           " / Value = " + entry.getValue());
    }
}
```

Listing 6.25 *Ausführbar als* **'SYSTEMENVEXAMPLE'**

Tabelle 6-2 *Einige nützliche System-Properties*

Property-Name	Inhalt
java.version	Java-Versionsnummer
java.home	Installationsverzeichnis des JREs
java.class.version	Versionsnummer der Java-. `class`-Dateien (JDK 5 = 49, JDK 6 = 50, JDK 7 = 51, JDK 8 = 52)
java.class.path	Aktueller `CLASSPATH`
os.name	Name des Betriebssystems (Windows 7 usw.)
user.name	Name des angemeldeten Benutzers
user.home	Home-Verzeichnis des angemeldeten Benutzers
user.dir	Aktuelles Arbeitsverzeichnis

6.5.3 Die Klasse `Preferences`

Wir haben bereits verschiedene Möglichkeiten, aber auch Schwierigkeiten und Beschränkungen der Konfigurationsverwaltung mit der Klasse `Properties` kennengelernt. Durch Einsatz der Klasse `java.util.prefs.Preferences` lassen sich einige Probleme lösen. Man kann dann Programmeinstellungen plattformunabhängig verwalten und zwischen Benutzer- und Systemeinstellungen unterscheiden. Letztere enthalten allgemeine, für alle Benutzer relevante Einstellungen. Benutzereinstellungen umfassen hingegen konkrete, individuelle

Konfigurationen für einzelne Benutzer.

Die Voreinstellungen liegen nicht in Form einer durch den Programmierer einzulesenden Datei vor, sondern werden auf Windows-Systemen in der Registry gespeichert. Unter Unix erfolgt die Ablage verteilt im Dateisystem. Diese Details sind jedoch beim Arbeiten mit der Klasse `Preferences` für den Entwickler irrelevant.

Speicherung von Konfigurationsdaten

Die Speicherung von Konfigurationsdaten erfolgt bei `Preferences` wiederum in Form von Schlüssel-Wert-Paaren. Allerdings können im Gegensatz zur Klasse `Properties` die Werte nicht nur vom Typ `String`, sondern auch von beliebigen primitiven Typen oder vom Typ `byte[]` sein. Andere Typen können nicht direkt gespeichert werden.[12]

Die maximal mögliche Länge der Informationen in Schlüsseln und Werten ist in den Konstanten `MAX_KEY_LENGTH` und `MAX_VALUE_LENGTH` als 80 bzw. 8192 Zeichen definiert. Für normale Voreinstellungen sollten diese Werte ausreichend sein – größere Datenmengen lassen sich so jedoch nicht (an einem Stück) speichern.

Hierarchische Struktur

Die Organisation von Daten mit der Klasse `Preferences` erfolgt hierarchisch in Form eines Baums. Dessen Struktur kann zwar beliebig definiert werden, folgt aber häufig der Package-Hierarchie. Jeder Knoten ist durch eine entsprechende Pfadangabe (bestehend aus allen Vorgängerknoten bis zur Wurzel) eindeutig zu identifizieren. Der Pfad besteht aus den Namen der Knoten, die durch einen Schrägstrich getrennt werden. Der Schrägstrich alleine stellt den Pfad zum Wurzelknoten der Hierarchie dar. Ein möglicher absoluter Pfad ist etwa '`/Data/General`'. Neben absoluten Pfadangaben, die immer mit einem '/' beginnen, werden auch relative Pfadangaben unterstützt.

Einstellungen ermitteln

Bei der Klasse `Preferences` erfolgt der Zugriff auf System- bzw. Benutzereinstellungen durch folgende Zugriffsmethoden:

- `systemNodeForPackage()` bzw. `userNodeForPackage()` – Gibt den Knoten in den Systemeinstellungen bzw. Benutzereinstellungen für ein Objekt zurück. Dabei wird der Package-Name in einen absoluten Pfad umgewandelt, wobei die Punkte des Package-Namens durch '/' ersetzt werden.
- `systemRoot()` bzw. `userRoot()` – Gibt die Wurzel des Baums der Systemeinstellungen bzw. der Benutzereinstellungen zurück.

Durch die ersten beiden Methoden erhält man Zugriff auf Voreinstellungen, die

einem speziellen Package zugeordnet sind. Dadurch ist deren Eindeutigkeit gewährleistet. Häufig ist eine derart feingranulare Speicherung von Voreinstellungen nicht unbedingt erforderlich. Dann sind sowohl `systemRoot()` als auch `userRoot()` nützlich, um allgemeine Einstellungen vorzunehmen. Ausgehend von diesen (aber auch von jedem beliebigen anderen) Knoten ist es mit der Methode `node(String)` möglich, Zugriff auf Einstellungen zu erhalten, die in dem durch den übergebenen String referenzierten Knoten gespeichert sind. Existiert der angegebene Pfad nicht, so wird dieser angelegt.

Einstellungswerte setzen und holen

Nachdem wir, wie zuvor beschrieben, den gewünschten Knoten, also den Speicherplatz für unsere Daten, gefunden haben, können wir dort Daten mit den diversen `get()`- und `put()`-Methoden der Klasse `Preferences` auslesen und speichern. Dazu existieren verschiedene Methoden zur Verarbeitung von primitiven Datentypen, von Strings und von Byte-Arrays. Die `get()`-Methoden besitzen alle einen zweiten Parameter, der zur Übergabe eines Defaultwerts dient:

```
final int prefWidth = prefs.getInt("WIDTH", 700 /* default-width */);
```

Für den Fall, dass kein Eintrag in dem angegebenen `Preferences`-Knoten hinterlegt ist, wird der übergebene Defaultwert zurückgeliefert. Dadurch ist es möglich, trotz gegebenenfalls fehlender Voreinstellungen, die Applikation mit einer sinnvollen Basiskonfiguration zu starten. Allerdings ist dies auch der größte Nachteil: Die Klasse `Preferences` ist eigentlich nur für solche Einstellungen zu verwenden, für die sinnvolle Defaultwerte existieren, etwa Fenstergrößen, Farben usw.

Für fixe Einstellungen, wie etwa Datenbankzugangsdaten o. Ä., sollte man besser eine Abstraktion basierend auf der Klasse `Properties` verwenden, wie wir dies in Abschnitt 6.5.2 getan haben.

6.5.4 Weitere Möglichkeiten zur Konfigurationsverwaltung

Mit CSV- bzw. XML-Dateien lassen sich auch komplexere Datenstrukturen abbilden und man ist flexibler als bei Schlüssel-Wert-Paaren, repräsentiert durch `Properties` oder `Preferences`. An einem Beispiel lernen wir nun die Verarbeitung von CSVDaten kennen. Statt trockener Anwendungsdaten nutzen wir als Eingabe eine Liste von Spielständen, die kommasepariert gespeichert werden.

Beispiel: Einlesen von Highscores aus einer CSV-Datei

Stellen wir uns eine x-beliebige Spieleapplikation vor, die es einem Spieler erlaubt, entsprechende Punktzahl vorausgesetzt, sich in einer Highscore-Liste zu verewigen.

Es wäre sehr ärgerlich, wenn die Erfolge nach jedem Programmende verloren gehen würden. Eine Speicherung in einer Datei und ein Einlesen beim Programmstart sind wünschenswert. Die Highscores werden dazu als Liste von Elementen der folgenden Immutable-Klasse `Highscore` verwaltet, die ihre Attribute Package-private definiert – hier durch `/* private */` angedeutet, was den Zugriff aus dem Package erlaubt:

```
public final class Highscore
{
    /*private*/ final String name;
    /*private*/ final int    points;
    /*private*/ final int    level;
    /*private*/ final Date   date;

    public Highscore(final String name, final int points,
                     final int level, final Date date)
    {
        this.name = name;
        this.points = points;
        this.level = level;
        // Achtung: Date ist veränderlich, Unveränderlichkeit herstellen
        this.date = new Date(date.getTime());
    }
    // …
```

Nehmen wir weiterhin an, die Spielstände wären etwa wie folgt in kommaseparierter Form in einer Datei `Highscores.csv` gespeichert. In diesem Beispiel sind bewusst auch fehlerhafte Einträge dargestellt, die dazu dienen, die Implementierung einer robusten Fehlerbehandlung zu demonstrieren und zu testen:

```
# Name, Punkte, Level, Datum
Matze,    1000,     7,   12.12.2009
Peter,     985,     6,   11.11.2009
ÄÖÜßöäü,   777,     5,   10.10.2009

# Fehlender Datumswert
Peter,     985,     6

# Falsches Format des Levels
Peter,     985,    A6,   11.11.2009

# Fehlerhaftes Datumsformat
Micha,     100,     1,   1/1/2001
```

Betrachten wir die Implementierung der Klasse `ReadHighscoresFromCsvExample`, die die dargestellte Datei mit Spielständen einliest. Weil Namen durchaus Umlaute enthalten können, verwenden wir zum Einlesen ein `FileReader`-Objekt statt der in den vorherigen Abschnitten verwendeten `FileInputStream`-Objekte. Letzterer ist bevorzugt zum Einlesen von Binärdateien gedacht und wird bei Verwendung von Zeichen (z. B. Umlauten), die nicht in einem Byte codiert werden können, bereits Probleme machen. Um performant zu lesen, ummanteln wir den `FileReader` mit einem `BufferedReader`. Dieser hat außer der Pufferung noch den Vorteil, eine `readLine()`-Methode anzubieten, mit der Daten zeilenweise gelesen werden können. Das erleichtert in diesem Fall die Arbeit ungemein. Jede einzelne Zeile soll die Daten eines `Highscore`-Objekts enthalten und wird daraufhin untersucht, ob

dies tatsächlich der Fall ist. Dazu wird in der Methode `extractHighscoreFromLine(String)` der Datenstrom mithilfe der Methode `split(String)` der Klasse `String` ausgewertet und in einzelne Tokens unterteilt. Wir stellen zuerst sicher, dass die erwartete Anzahl an Tokens vorliegt und protokollieren Verstöße. Sind die Daten vollständig, dann erfolgt eine Verarbeitung der Daten nach der hier nicht näher gezeigten Gültigkeitsprüfung durch Aufruf der Methode `validateInput()`. Liegen alle Daten im gewünschten Format vor, wird daraus ein unveränderliches `Highscore`-Objekt erstellt. Unvollständige oder ungültige Objektzustände sind somit ausgeschlossen.

Das folgende Listing zeigt die Klasse `ReadHighscoresFromCsvExample`, die die zuvor beschriebene Funktionalität implementiert:

```
public final class ReadHighscoresFromCsvExample
{
    private static final Logger log =
                         Logger.getLogger(ReadHighscoresFromCsvExample.class);

    private static final int    POS_NAME    = 0;
    private static final int    POS_POINTS  = 1;
    private static final int    POS_LEVEL   = 2;
    private static final int    POS_DATE    = 3;
    private static final int    VALUE_COUNT = 4;

    public static List<Highscore> readHighscoresFromCsv(final String fileName)
    {
        final List<Highscore> highscores = new LinkedList<Highscore>();

        try (final BufferedReader br =
                    new BufferedReader(new FileReader(fileName)))
        {
            String line = null;

            // Zeilenweises Einlesen mit readLine()
            while ((line = br.readLine()) != null)
            {
                final Highscore highscore = extractHighscoreFromLine(line);
                if (highscore != null)
                {
                    highscores.add(highscore);
                }
            }
        }
        catch (final FileNotFoundException e)
        {
            log.warn("No input file '" + fileName + "'", e);
        }
        catch (final IOException e)
        {
            log.warn("processing of file '" + fileName + "' incomplete!", e);
        }
        return highscores;
    }

    private static Highscore extractHighscoreFromLine(final String line)
    {
        // Spalte die Eingabe mit ';' oder ',' auf
```

```
        final String[] values = line.split(";|,");

        // Behandlung von Leerzeilen und Kommentaren
        if (isEmptyLineOrComment(values))
        {
            return null;
        }
        // Ignoriere fehlertoleranterweise unvollständige Einträge
        if (values.length != VALUE_COUNT)
        {
            log.warn("Wrong number of values: " + values.length + " expected:
"
                    + VALUE_COUNT + "! Skipping invalid value '" + line +
"'");
            return null;
        }

        try
        {
            // Auslesen der Werte als String + Typprüfung + Konvertierung
            final String name = values[POS_NAME].trim();
            final int points = Integer.parseInt(values[POS_POINTS].trim());
            final int level = Integer.parseInt(values[POS_LEVEL].trim());
            final String dateAsString = values[POS_DATE].trim();
            final Date date =
DateFormat.getDateInstance().parse(dateAsString);

            // Gültigkeitsprüfung
            validateInput(name, points, level, date);
            return new Highscore(name, points, level, date);
        }
        // Syntaktische Fehler: Falsches Format, keine Zahl usw.
        catch (final NumberFormatException | ParseException ex)
        {
            log.warn("Skipping invalid point, level or date in '" + line +
"'",
                  ex);
        }
        // Semantische Fehler: Ungültige Wertebereiche usw.
        catch (final IllegalArgumentException ex)
        {
            log.warn("Skipping invalid inputs from '" + line + "'", ex);
        }
        return null;
    }

    private static boolean isEmptyLineOrComment(final String[] values)
    {
        return (values.length == 1 && (values[0].trim().length() == 0) ||
                // Ignoriere Kommentare, die auch ',' oder ';' enthalten
                (values.length >= 1 && values[0].trim().startsWith("#")));
    }

    public static final void main(final String[] args)
    {
        PropertyConfigurator.configureAndWatch("config/log4j.properties",
5000);

        final String filePath = "config/Highscores.csv";
        final List<Highscore> highscores = readHighscoresFromCsv(filePath);
        new AppFrame(highscores).setVisible(true);
```

```
    }
    // …
```

Listing 6.26 *Ausführbar als* **'READHIGHSCORESFROMCSVEXAMPLE'**

Die Verarbeitung der Daten erfolgt in der obigen `main()`-Methode. Beim Start des Programms READHIGHSCORESFROMCSVEXAMPLE werden alle Werte aus der Datei `Highscores.csv` eingelesen. Alle korrekten Daten werden anschließend in Form einer Tabelle in einem Swing-GUI dargestellt. Das ist in Abbildung 6-3 gezeigt.

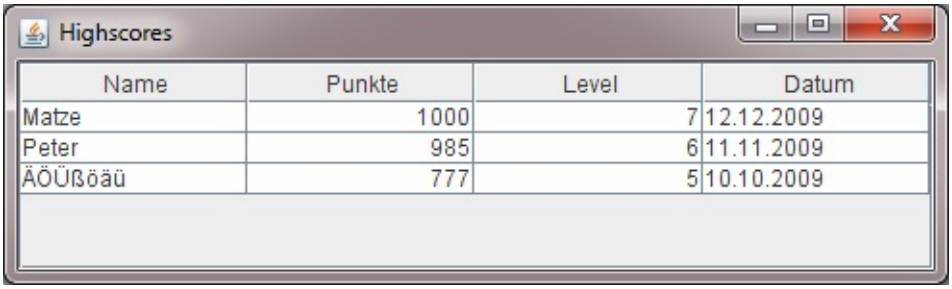


Abbildung 6-3 *Applikationsfenster der Highscore-Liste*

Achtung: Plattformabhängigkeit durch `newLine()` und `readLine()`

Die Methoden `newLine()` und `readLine()` der Klassen `BufferedWriter` bzw. `BufferedReader` sind *plattformabhängig*. Das liegt daran, dass sie beim Schreiben und Lesen spezielle betriebssystemspezifische Zeilenendezeichen verwenden. Wenn man Daten zwischen verschiedenen Systemen austauschen möchte, kann dies zu merkwürdigen und zunächst unerklärlichen Effekten führen. Ist Plattformunabhängigkeit in einem Projekt nicht erforderlich, kann man von den Vereinfachungen durch den Einsatz der beiden Methoden profitieren.

Nachdem wir – eine Ausführung des Programms vorausgesetzt – die Klasse konkret in Aktion erlebt haben, möchte ich noch auf einige Dinge etwas genauer eingehen.

Behandlung von Fehlern beim Öffnen der Datei

Im Falle einer Exception beim Öffnen der Datei `Highscores.csv` protokollieren wir ein derartiges Problem und beenden die Verarbeitung. In jedem Fall werden durch den Einsatz von ARM automatisch Aufräumarbeiten durchgeführt. Das sorgt dafür, dass allozierte Systemressourcen wieder freigegeben werden.

Auch die Applikation kann geregelt weiterarbeiten. Sie erhält in diesem Fall entweder eine leere Highscore-Liste oder eine solche, die mit den Daten gefüllt ist, die bis zum Auftreten des Problems eingelesen werden konnten.[13] Alternativ könnte man Exceptions an die Applikation durchreichen und dort eine Fehlermeldung erzeugen.

Behandlung von Leerzeilen und Kommentaren

Aus der Datei gelesene Eingaben werden mithilfe der Methode `split(String)`

der Klasse `String` gemäß einer übergebenen Trennzeichenfolge in Tokens zerlegt. Als Ergebnis erhält man ein `String[]`, das der Hilfsmethode `isEmptyLineOrComment(String[])` als Eingabe dient. Dort erfolgen verschiedene Prüfungen, die es erlauben, sowohl Leerzeilen als auch Kommentarzeilen bei der Auswertung zu überspringen.

Solche Hilfsmethoden sorgen für mehr Lesbarkeit im eigentlichen Applikationscode und halten diesen frei von Implementierungsdetails.

Ignorieren von unvollständigen Eingaben

Beim Einlesen von textuellen Daten aus CSV-Dateien kann man Inkonsistenzen in den gespeicherten Daten nicht ausschließen. So etwas sollte die Auswertung berücksichtigen und fehlertolerant darauf reagieren. In diesem Beispiel wird bei unvollständiger Angabe von Informationen eine Warnmeldung in einer Log-Datei protokolliert, die entsprechende Zeile übersprungen und das Parsing mit der nächsten Zeile der Eingabe fortgesetzt.

Auswertung der Informationen

Sind alle benötigten Daten vorhanden, d. h., liegen genau vier Tokens in der Eingabe vor, so werden zunächst führende und nachfolgende Leerzeichen der Eingabewerte durch einen Aufruf der Methode `trim()` der Klasse `String` entfernt. Dadurch ist es bei der Angabe in der CSV-Datei erlaubt, beliebig viele Leerzeichen zur Ausrichtung der Daten zu verwenden, ohne dass sich dies auf die Daten selbst auswirkt. Der textuelle Wert für den Namen wird ohne weitere Prüfung übernommen. Die beiden Zahlenwerte werden durch die statische Methode `parseInt(String)` der Klasse `Integer` in einen `int`-Wert umgewandelt. Wird in der Eingabe keine Zahl angegeben, so löst dies eine `NumberFormatException` aus und es wird kein `Highscore`-Objekt erzeugt. Die Angabe des Datums wird mithilfe der Klasse `java.util.DateFormat` (siehe Abschnitt 10.1.6) auf Gültigkeit untersucht. Neben den eher syntaktischen Prüfungen auf Zahl oder Datum wird mithilfe der Prüfmethode `validateInputs()` eine weiter gehende semantische Prüfung durchgeführt, um etwa negative Level- oder Punktzahlen oder gar Datumswerte aus der Zukunft zurückzuweisen. Darüber hinaus gibt es noch eine Obergrenze der Level usw. Sind alle Werte gültig, so werden diese dem Konstruktor der Klasse `Highscore` als Parameter zur Erzeugung eines neuen Objekts übergeben.

Im Beispiel enthalten einige Eingabedaten bewusst Fehler und sollen nicht eingelesen werden, etwa weil das Datumsformat vom erwarteten Format abweicht. Kapitel 10 zeigt wie man auch mehrere Datumsformate beim Parsing unterstützen und damit toleranter mit Benutzereingaben oder Werten aus Dateien umgehen kann.

Hinweis: Einsatz einer Bibliothek zur Konfigurationsverwaltung

Durch den Einsatz der Bibliothek Apache Commons Configuration kann die Konfiguration von eigenen Programmen vereinfacht werden. Damit lassen sich leichter verschiedenste Arten von Konfigurationen, etwa aus Properties, aus XML-Dateien oder aus Applet- bzw. Servlet-Parametern, verarbeiten. Weitere Informationen finden Sie online unter `http://commons.apache.org/configuration/`.

7 Multithreading

Das Thema Multithreading und Parallelverarbeitung mehrerer Aufgaben gewinnt durch Computer mit mehreren Prozessoren (CPUs) oder Prozessoren mit mehreren Rechenkernen (Multicores) zunehmend an Bedeutung. Ziel ist es, komplexe Aufgaben innerhalb von Programmen in voneinander unabhängige Teilaufgaben zu untergliedern, die parallel abgearbeitet werden können. Ist dies der Fall, spricht man von Nebenläufigkeit (Concurrency). Java bietet zwar einen einfachen Zugang zur Programmierung mit Threads, allerdings verleitet dies manchmal dazu, Multithreading einzusetzen, ohne die resultierenden Konsequenzen zu beachten. Dadurch kommt es in der Praxis aber immer mal wieder zu Problemen. Dieses Kapitel soll helfen, Multithreading mit seinen Möglichkeiten (aber auch Fallstricken) kennenzulernen.

Eine Einführung in das Thema Multithreading mit `Thread` und `Runnable` gibt Abschnitt 7.1. Wenn in einem Programm Aufgaben durch mehrere Threads bearbeitet werden, müssen deren Berechnungsergebnisse abgeglichen oder zusammengeführt werden. Das ist Thema von Abschnitt 7.2. Auch die Kommunikation zwischen Threads hat Einfluss auf eine erfolgreiche Zusammenarbeit. Darauf gehe ich in Abschnitt 7.3 ein. Ebenso spielt das Java-Memory-Modell eine wichtige Rolle. Einige Details dazu liefert Abschnitt 7.4. Das Themengebiet Multithreading wird in Abschnitt 7.5 mit der Vorstellung verschiedener Besonderheiten beim Einsatz von Threads abgerundet. Im Speziellen widmen wir uns dem Beenden von Threads sowie der zeitgesteuerten Ausführung von Aufgaben mithilfe der Klassen `Timer` und `TimerTask`. Mit den in JDK 5 eingeführten Concurrency Utilities werden viele Aufgaben im Bereich von Multithreading und Nebenläufigkeit deutlich erleichtert. Abschnitt 7.6 geht darauf ein. Auch die mit JDK 7 eingeführten Erweiterungen werden dort kurz behandelt.

Grundlagen zu Parallelität und Nebenläufigkeit

Moderne Betriebssysteme beherrschen sogenanntes ***Multitasking*** und führen mehrere Programme gleichzeitig aus oder vermitteln dem Benutzer zumindest die Illusion, dass verschiedene Programme gleichzeitig ausgeführt würden. Wie eingangs erwähnt, besitzen modernere Computer zum Teil mehrere Prozessoren oder Prozessoren mit mehreren Kernen und erlauben so, dass das Betriebssystem verschiedene Programme direkt auf alle verfügbaren Prozessoren bzw. Kerne aufteilen kann. Allerdings kann zu jedem Zeitpunkt jede Recheneinheit (Prozessor bzw. Kern) nur genau ein Programm ausführen. Sollen mehrere Programme parallel ausgeführt werden, müssen diese intelligent auf die jeweiligen Recheneinheiten aufgeteilt werden. Bei nur einer Recheneinheit ist dies bereits bei zwei Programmen der Fall. Ganz allgemein gilt für eine *n*-ProzessorMaschine[1], dass eine Verteilung bei mehr als *n* Programmen notwendig wird. Ein sogenannter ***Scheduler*** bestimmt

anhand verschiedener Kriterien, welches Programm auf welcher Recheneinheit ausgeführt wird, und führt bei Bedarf eine Umschaltung des gerade ausgeführten Programms auf ein anderes durch. Etwas später wird das zuvor unterbrochene Programm an gleicher Stelle fortgesetzt. Diese Vorgehensweise sorgt dafür, dass die Ausführung jeweils in kleinen Schritten erfolgt, und erlaubt es, andere Programme minimal zeitlich versetzt und damit scheinbar parallel ausführen zu können.

Laufende Programme können wiederum aus »schwergewichtigen« ***Prozessen*** und »leichtgewichtigen« ***Threads*** bestehen. Prozesse sind im Gegensatz zu Threads bezüglich des von ihnen verwendeten Speichers voneinander abgeschottet und belegen unterschiedliche Bereiche. Dadurch beeinflusst ein abstürzender Prozess andere Prozesse (meistens) nicht. Allerdings erschwert diese Trennung auch die Zusammenarbeit und Kommunikation der Prozesse untereinander. Dazu wird eine spezielle Form der Kommunikation, die ***Interprozesskommunikation***, notwendig. Diese kann nicht durch normale Methodenaufrufe erfolgen, sondern man nutzt eine spezielle ***Middleware***, die den Transport von Daten zwischen Prozessen oder gar Anwendungen, das sogenannte ***Messaging***, realisiert. Dazu setzt man z. B. RMI (Remote Method Invocation), JMS (Java Messaging Service) oder CORBA (Common Object Request Broker Architecture) ein. Alternativen aus dem Webservices-Umfeld sind etwa SOAP (Simple Object Access Protocol) oder das leichtgewichtigere REST (REpresentational State Transfer).

Ein Java-Programm wird im Betriebssystem durch einen Prozess einer JVM ausgeführt. Ein solcher Prozess kann wiederum verschiedene eigene Threads starten. Alle Threads in einer JVM teilen sich dann den gleichen Adressraum und Speicher. Änderungen an Variablen sind dadurch theoretisch für alle Threads sichtbar. In der Realität ist es durch das Java-Memory-Modell jedoch etwas komplizierter. Abschnitt 7.4 geht detailliert darauf ein. Damit sich zwei Threads beim Zugriff auf dieselben Variablen oder Ressourcen nicht gegenseitig stören, müssen sich diese abstimmen, wenn sie zugreifen wollen. Dazu kann man sogenannte ***kritische Bereiche*** definieren, die zu einer Zeit exklusiv immer nur von einem Thread betreten werden können. Damit vermeidet man konkurrierende Zugriffe auf gemeinsame Daten durch verschiedene Threads und erreicht so eine ***Synchronisierung***. Allerdings birgt dies die Gefahr, dass sich Threads gegenseitig behindern oder sogar blockieren. Eine solche Situation nennt man ***Verklemmung*** (oder ***Deadlock***) und muss vom Entwickler selbst verhindert werden. Strategien dazu stellt Abschnitt 7.2.1 vor.

Beginnen wir nun mit dem Einstieg in das Multithreading mit Java und schauen uns Threads und deren Kommunikation relativ gründlich an, da diese Themen elementare Grundlagen darstellen. In der Praxis bietet sich zur Vereinfachung oftmals der Einsatz der Concurrency Utilities an, weil damit Abläufe weniger Details zeigen und auf eher logischer Ebene formuliert werden können (vgl. Abschnitt 7.6).

7.1 Threads und Runnables

Jeder Thread entspricht in Java einer Instanz der Klasse `java.lang.Thread` oder einer davon abgeleiteten Klasse. Dabei ist es wichtig, zwei Dinge zu unterscheiden:

1. Den tatsächlichen Thread, der eine Aufgabe ausführt und vom Betriebssystem erzeugt und verwaltet wird.
2. Das zugehörige `Thread`-Objekt, das den zuvor genannten Thread des Betriebssystems repräsentiert und Steuerungsmöglichkeiten auf diesen bietet.

Die JVM erzeugt und startet automatisch einen speziellen Thread, den man `main`-Thread nennt, weil dieser die `main()`-Methode der Applikation ausführt. Ausgehend von diesem Thread können weitere Threads erzeugt und gestartet werden.

Betrachten wir zunächst, wie wir eine auszuführende Aufgabe beschreiben, bevor wir uns dem Ausführen von Threads zuwenden.

7.1.1 Definition der auszuführenden Aufgabe

Zur Beschreibung der auszuführenden Aufgaben eines Threads nutzt man die Klasse `Thread` selbst oder das Interface `java.lang.Runnable`. Beide bieten eine `run()`-Methode, die mit »Leben gefüllt« werden muss. Das kann auf zwei Arten geschehen:

1. Ableiten von der Klasse `Thread` und Überschreiben der `run()`-Methode
2. Implementieren des Interface `Runnable` und der `run()`-Methode

Betrachten wir beide Varianten anhand zweier einfacher Klassen `CountingThread` und `DatePrinter`. Abbildung 7-1 zeigt das zugehörige Klassendiagramm.

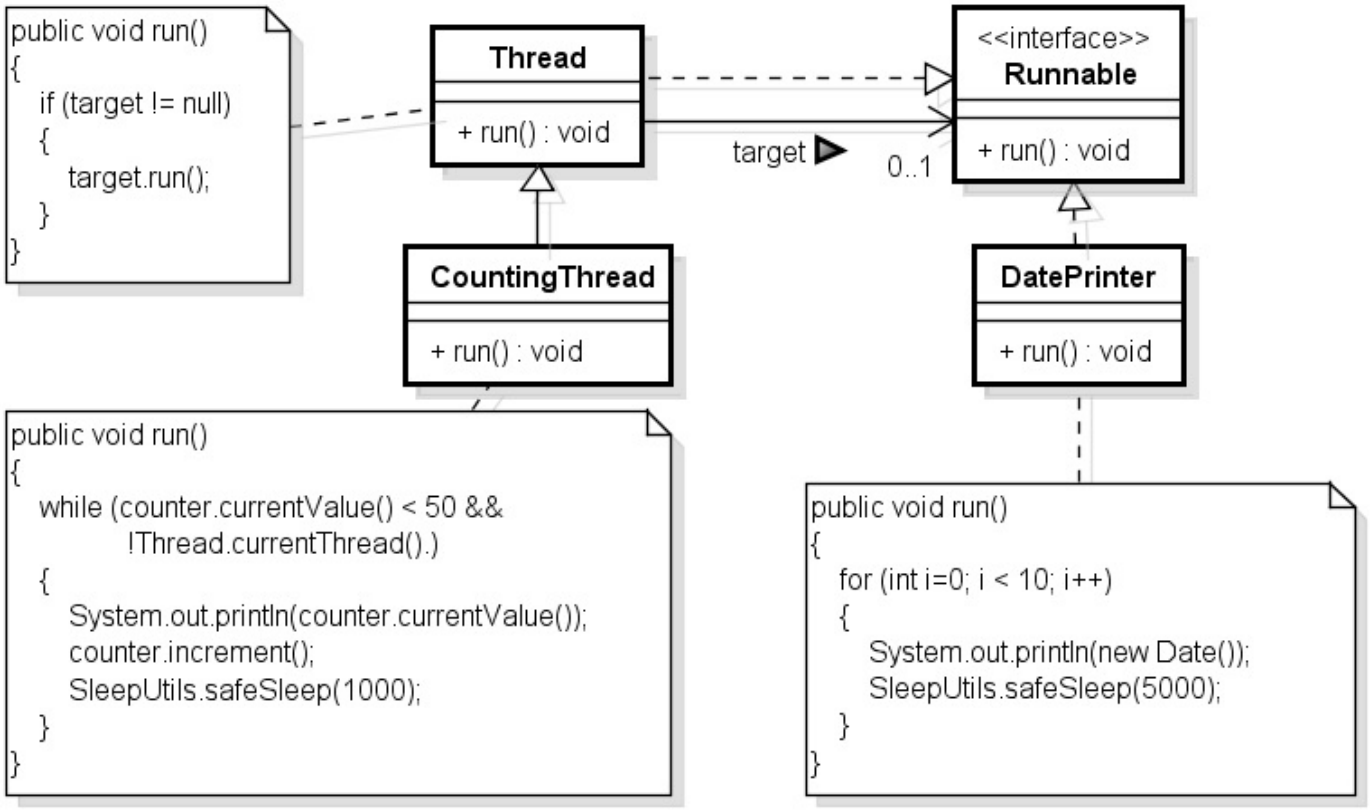


Abbildung 7-1 *Thread und Runnable*

Die Klasse `CountingThread` erweitert die Klasse `Thread`. Die Implementierung der `run()`-Methode erhöht im Abstand von einer Sekunde einen Zähler und gibt dessen Wert anschließend auf der Konsole aus. Die Klasse `DatePrinter` implementiert das Interface `Runnable` und gibt im Takt von fünf Sekunden das aktuelle Datum aus. Beide Klassen verwenden zum Warten eine Utility-Klasse `SleepUtils`, die wir später in Abschnitt 7.1.3 kennenlernen werden. Nur im `CountingThread` wird auf die Aufforderung zum Abbruch per `interrupt()` geprüft, wodurch nach 10 Sekunden die Ausgabe beendet wird.

In beiden Realisierungen wird zunächst die `run()`-Methode des Threads ausgeführt. Wird einem `Thread`-Objekt ein `Runnable`-Objekt übergeben, so delegiert die `run()`-Methode des `Thread`-Objekts die Ausführung an die `run()`-Methode des `Runnable`-Objekts. Die Variante basierend auf `Runnable` ist zu bevorzugen, da sie OO-technisch sauberer ist: Es findet keine Ableitung von einer Utility-Klasse statt und die Funktionalität ist als Einheit gekapselt. Zur nebenläufigen Ausführung eines `Runnable`-Objekts muss dann ein `Thread`-Objekt als Laufzeitcontainer entweder neu erzeugt oder bereitgestellt werden.

Hinweis: Beschränkungen von `run()`

Wie gesehen, lässt sich die abzuarbeitende Aufgabe beschreiben, indem man die `run()`-Methode implementiert. Diese besitzt durch das Interface `Runnable` eine vordefinierte Signatur ohne Aufrufparameter. Oftmals benötigt man zur Abarbeitung einer Aufgabe aber gewisse Kontextinformationen, die man jedoch nicht an `run()` übergeben kann. In eigenen Realisierungen von `Runnable` bzw. `Thread` nutzt man als Abhilfe zur Parametrierung Attribute.

Eine weitere Einschränkung der `run()`-Methode besteht darin, dass diese keinen Rückgabewert definiert. Wenn diese Anforderung besteht, sollte man mittlerweile das mit JDK 5 eingeführte und später in Abschnitt 7.6.2 beschriebene Executor-Framework und seine Bestandteile `Callable<V>` und `Future<V>` aus dem Package `java.util.concurrent` nutzen. Damit wird eine Realisierung auf logischer statt eher technischer Ebene möglich. Bis JDK 1.4 war man in eigenen Implementierungen dazu gezwungen, zur Rückgabe von Berechnungsergebnissen ein oder mehrere Attribute sowie zugehörige Accessor-Methoden vorzusehen, damit Aufrufer dann auf das Ergebnis der Verarbeitung per `get()`-Methode zugreifen konnten. Insgesamt wird der Austausch von Berechnungsergebnissen zwischen Threads so recht schnell mühsam, sodass sich die neueren Varianten anbieten.

7.1.2 Start, Ausführung und Ende von Threads

Starten von Threads

Die Konstruktion eines neuen `Thread`-Objekts führt nicht direkt dazu, dass ein neuer Ausführungspfad abgespalten wird. Erst durch Aufruf der `start()`-Methode entsteht ein eigenständiger Thread: Das Programm arbeitet nach diesem Aufruf weiter, ohne blockierend auf das Ende des gestarteten Threads zu warten. In folgendem Beispiel werden die zwei vorgestellten unterschiedlichen Realisierungen von Threads jeweils erzeugt und gestartet:

```
public static void main(final String[] args)
{
    final Thread derivedThread = new CountingThread();
    derivedThread.start();

    final Thread threadWithRunnable = new Thread(new DatePrinter());
    threadWithRunnable.start();
}
```

Listing 7.1 *Ausführbar als* **'THREADEXAMPLE'**

Ausführen von Threads

Nach Aufruf von `start()` kommt es zu einer Verarbeitung und zur Ausführung des Threads durch den Thread-Scheduler. In der Folgezeit wird die `run()`-Methode so lange ausgeführt, bis deren letztes Statement erreicht ist (oder ein Abbruch erfolgt, etwa durch Auslösen einer Exception). Bis dahin wird der Thread normalerweise bei der Abarbeitung der in der `run()`-Methode aufgeführten Anweisungen einige Male vom Thread-Scheduler unterbrochen, um andere Threads auszuführen. Über die Methode `isAlive()` kann man ein `Thread`-Objekt befragen, ob es noch aktiv ist und vom Thread-Scheduler verwaltet wird. Die Methode liefert dann den Wert `true`.

Ein `Thread`-Objekt führt seine durch die `run()`-Methode beschriebene Aufgabe nur genau einmal nebenläufig aus. Eine Wiederholung ist nicht möglich: Ein erneuter Aufruf von `start()` löst eine `java.lang.IllegalThreadStateException` aus.

Achtung: Versehentlicher direkter Aufruf von `run()`

Manchmal sieht man Sourcecode, der die Methode `run()` direkt aufruft, um einen neuen Thread zu erzeugen und die Aufgabe parallel auszuführen. ***Das funktioniert jedoch nicht korrekt!*** Stattdessen werden die Anweisungen der Methode `run()` einfach synchron im momentan aktiven Thread wie jede normale andere Methode ausgeführt. Dieser Fehler ist relativ schwierig zu finden, da die gewünschten Aktionen ausgeführt werden — in diesem Fall allerdings nicht nebenläufig.

Ende der Ausführung von Threads

Nachdem das letzte Statement der `run()`-Methode ausgeführt wurde, endet auch der Thread. Das zugehörige `Thread`-Objekt bleibt jedoch weiter erhalten, stellt aber

nur noch ein Objekt wie jedes andere dar. Man kann weiterhin auf Attribute des Threads zugreifen, um beispielsweise die Ergebnisse einer Berechnung zu ermitteln. Ein Aufruf von `isAlive()` liefert dann den Wert `false`.

Zum Teil sollen Bearbeitungen, d. h. die Ausführung der `run()`-Methode, zu einem beliebigen Zeitpunkt abgebrochen werden. Leider lassen sich Threads nicht so einfach beenden wie starten. Es gibt zwar im API eine zu `start()` korrespondierende Methode `stop()`, doch diese ist als `deprecated` markiert und sollte nicht (mehr) verwendet werden, da sie verschiedene Probleme auslösen kann. Zum Verständnis der Details müssen wir zunächst einige andere Themen besprechen. Abschnitt 7.5.3 greift das Thema »Beenden von Threads« erneut auf und geht detaillierter auf zugrunde liegende Probleme und mögliche Lösungen ein.

Thread-Prioritäten

Jeder Thread besitzt eine Ausführungspriorität als Zahlenwert im Bereich von 1 bis 10, entsprechend den Konstanten `Thread.MIN_PRIORITY` und `Thread.MAX_PRIORITY`. Diese Priorität beeinflusst, wie der Thread-Scheduler zu aktivierende Threads auswählt. Threads höherer Priorität werden normalerweise bevorzugt, allerdings wird dies nicht garantiert, und jede JVM oder das Betriebssystem können dies anders handhaben. Dadurch können beispielsweise Threads mit der Priorität n und $n + 1$ durch den Thread-Scheduler ohne Unterschied zur Ausführung ausgewählt werden.

Beim Erzeugen übernimmt ein Thread die Priorität des erzeugenden Threads, die in der Regel dem Wert `Thread.NORM_PRIORITY` (5) entspricht. Diese kann nachträglich über die Methoden `getPriority()` und `setPriority(int)` abgefragt und verändert werden. Dabei sind allerdings nur Werte aus dem durch die obigen Konstanten definierten Wertebereich erlaubt.[2] In der Regel muss die Priorität nicht angepasst werden. Hierbei gibt es zwei Ausnahmen: Für im Hintergrund zu erledigende, nicht zeitkritische Aufgaben kann z. B. der Wert 3 verwendet werden. Für zeitkritische Berechnungen bietet sich eine Priorität nahe `MAX_PRIORITY` an, etwa der Wert 8. Doug Lea gibt in seinem Buch »Concurrent Programming in Java« [59] folgende Empfehlungen:

Tabelle 7-1 *Empfehlungen für Thread-Prioritäten*

Priorität	Verwendungszweck
10	Krisenmanagement
7 – 9	Interaktive, ereignisgesteuerte Aufgaben
4 – 6	Normalfall und I/O-abhängige Aufgaben
2 – 3	Berechnungen im Hintergrund
1	Unwichtige Aufgaben

Threads und Thread-Gruppen

Eingangs erwähnte ich, dass beim Programmstart von der JVM automatisch der `main()`-Thread erzeugt und gestartet wird. Dabei wird auch eine sogenannte Thread-Gruppe erzeugt, in der verschiedene Threads zusammengefasst werden können und die im JDK einem `java.lang.ThreadGroup`-Objekt entspricht. Ein Thread ist immer genau einer Thread-Gruppe zugeordnet. Thread-Gruppen wiederum können hierarchisch kombiniert werden. Bei der Zugehörigkeit zu einer Thread-Gruppe verhält es sich wie bei der Priorität: Ein Thread wird automatisch in der Gruppe des erzeugenden Threads angelegt, sofern dies nicht bei der Konstruktion anders spezifiziert wird.

In der Praxis werden Sie eher selten mit Thread-Gruppen arbeiten – trotzdem kann deren Kenntnis und Einsatz nützlich sein, etwa um die aktiven Threads und ihre Thread-Gruppen ausgeben zu können. Das werden wir später beim Verständnis der Ereignisbehandlung in Swing in Abschnitt 9.1.4 gut gebrauchen können. Hier entwickeln wir die benötigte Funktionalität in Form einer Hilfsmethode `dumpThreads()` in einer Utility-Klasse `ThreadUtils`. In der Hilfsmethode ermitteln wir die Thread-Gruppe des aktuellen Threads durch Aufruf von `getThreadGroup()`. Für diese bestimmen wir dann mithilfe der Methode `activeCount()` die Anzahl der laufenden Threads der Gruppe und füllen anschließend durch Aufruf von `enumerate(Thread[])` ein `Thread`-Array mit den momentan aktiven Threads:

```
public final class ThreadUtils
{
   public static void dumpThreads()
   {
      final ThreadGroup group = Thread.currentThread().getThreadGroup();
      final int activeCount = group.activeCount();
      final Thread[] threads = new Thread[activeCount];
      group.enumerate(threads);
```

```
        System.out.println("Thread-Group " + group + " contains " + activeCount
+
                                                    " threads");
        for (final Thread thread : threads)
        {
            System.out.println("Thread" + thread);
        }
    }
}
```

Verdeutlichen wir uns den Einsatz der erstellten Methode am bereits kennengelernten Beispiel THREADEXAMPLE, das wir wie folgt erweitern:

```
public static void main(final String[] args)
{
    final Thread derivedThread = new CountingThread();
    // Thread-Name setzen
    derivedThread.setName("CountingThread");
    derivedThread.start();

    final Thread threadWithRunnable = new Thread(new DatePrinter());
    // threadWithRunnable.setName("DatePrinter");
    threadWithRunnable.start();

    // Ausgabe der Threads
    ThreadUtils.dumpThreads();
}
```

Listing 7.2 *Ausführbar als* **'DUMPTHREADSEXAMPLE'**

Führen wir das Programm DUMPTHREADSEXAMPLE aus, so sehen wir, dass die beiden Threads parallel zum `main()`-Thread in der Thread-Gruppe `main` gestartet werden. Es kommt in etwa zu folgender Ausgabe, wobei die Angaben in eckigen Klammern die Informationen Thread-Name, Priorität und Parent-Thread darstellen:

```
Thread-Group java.lang.ThreadGroup[name=main,maxpri=10] contains 3 threads
Thread Thread[main,5,main]
Thread Thread[CountingThread,5,main] // Thread Thread[Thread-0,5,main]
Thread Thread[Thread-1,5,main]       // Thread Thread[DatePrinter,5,main]
```

Anhand des Sourcecodes und der Ausgabe des Programms erkennen wir noch einen Trick aus der Praxis: Geben Sie Ihren Threads durch Aufruf der Methode `setName(String)` der Klasse `Thread` sprechende Namen. Ansonsten wird der Name des Threads einfach durchnummeriert. Wenn Sie diesen Hinweis befolgen, erleichtern Sie sich eine Fehlersuche enorm. Im obigen Konsolenauszug sieht man den Unterschied in der Verständlichkeit zwischen `CountingThread` und `Thread-1` (für den `DatePrinter`-Thread) bereits sehr deutlich.

7.1.3 Lebenszyklus von Threads und Thread-Zustände

Threads haben einen definierten Lebenszyklus, der durch eine festgelegte Menge an Zuständen beschrieben wird. Gültige Zustände sind als innere `enum`-Aufzählung `State` in der Klasse `Thread` definiert. Durch Aufruf der Methode `getState()`

kann man den momentanen Thread-Zustand ermitteln. Mögliche Rückgabewerte sind in Tabelle 7-2 aufgelistet und kurz beschrieben.

Tabelle 7-2 *Thread-Zustände*

Thread-Zustand	Beschreibung
`NEW`	Der Thread wurde erzeugt, ist aber noch nicht gestartet.
`RUNNABLE`	Der Thread ist lauffähig oder wird gerade ausgeführt.
`BLOCKED`	Der Thread ist in seiner Ausführung blockiert und wartet auf den Zutritt in einen kritischen Bereich (vgl. Abschnitt 7.2.1).
`WAITING`	Der Thread wartet unbestimmte Zeit auf eine Benachrichtigung durch einen anderen Thread (vgl. Abschnitt 7.3).
`TIMED_WAITING`	Identisch mit `WAITING`, allerdings wird maximal eine angegebene Zeitdauer auf eine Benachrichtigung gewartet.
`TERMINATED`	Der Thread ist beendet.

Für das Verständnis der Zusammenarbeit von Threads ist es wichtig, den Lebenszyklus von Threads und die dabei eingenommenen Thread-Zustände zu kennen. Im Folgenden stelle ich daher die Bedingungen und Auslöser für Zustandswechsel vor.

Nachdem ein Thread erzeugt wurde, wechselt dieser durch den Aufruf seiner `start()`-Methode in den Zustand `RUNNABLE`. Meistens gibt es mehrere Threads in diesem Zustand. Da auf einer 1-Prozessor-Maschine jedoch jeweils nur ein Thread ausgeführt werden kann, ist es Aufgabe des ***Thread-Schedulers***, alle Threads zu überwachen und den nächsten auszuführenden zu bestimmen, indem ein Thread aus allen denjenigen ausgewählt wird, die sich im Zustand `RUNNABLE` befinden. Der gewählte Thread wird dann aktiv (dies wird nicht durch einen eigenen Zustand beschrieben, man kann sich allerdings einen »künstlichen« Unterzustand `ACTIVE` im Zustand `RUNNABLE` vorstellen). Auf einer 1-Prozessor-Maschine existiert immer genau ein aktiver Thread, auf einer Mehrprozessormaschine sind es in der Regel mehrere. Unabhängig davon kann es aber beliebig viele Threads geben, die zur Ausführung bereit sind (Zustand `RUNNABLE`). Abbildung 7-2 stellt die Thread-Zustände und mögliche Zustandswechsel dar.

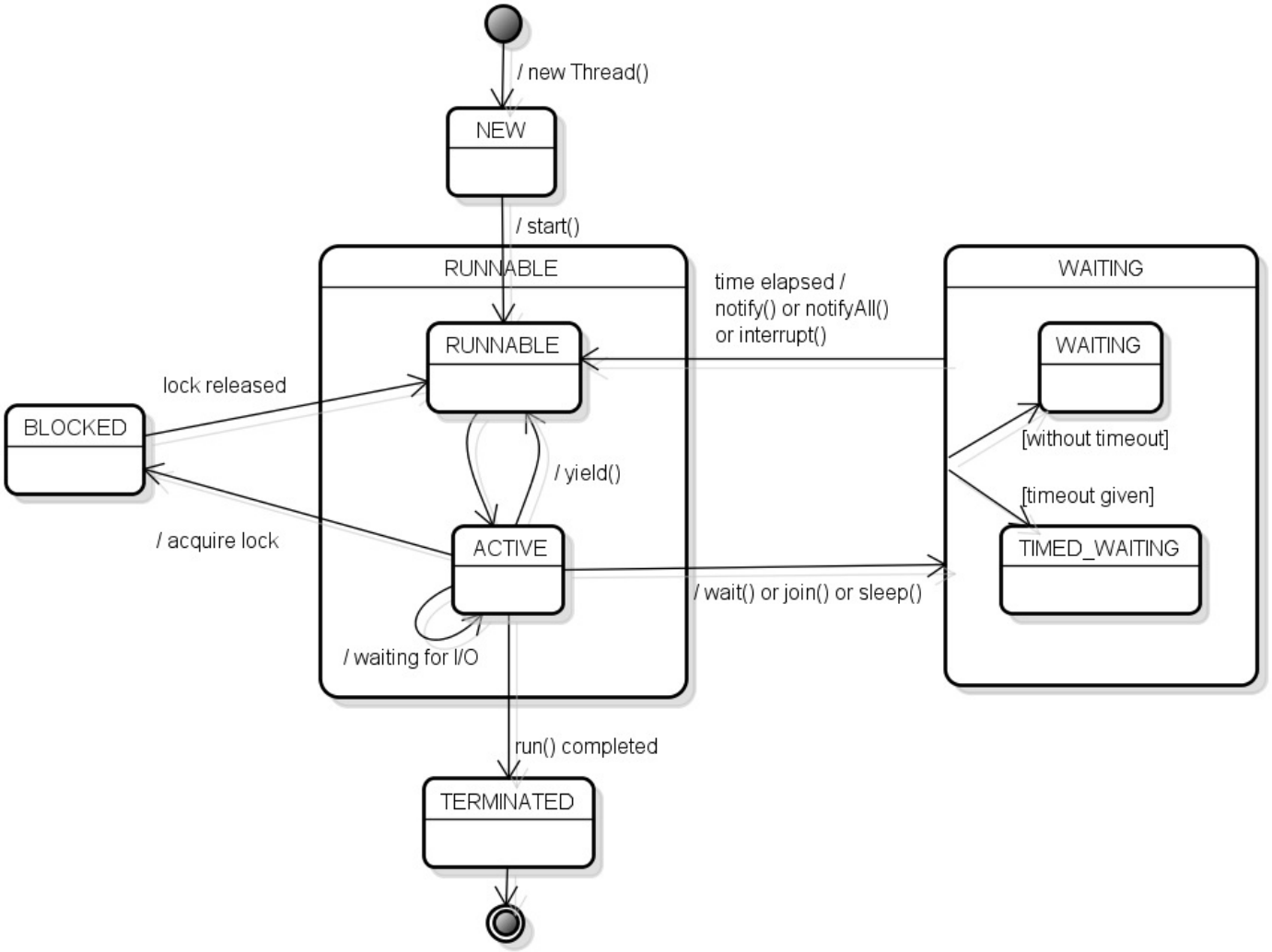


Abbildung 7-2 *Thread-Zustände*

Nicht jeder Thread im Zustand `RUNNABLE` ist tatsächlich immer lauffähig. Er kann in einem speziellen »blockiert«-Zustand verharren, wenn er zwar läuft, aber gerade auf eine Ressource wartet, etwa einen Dateizugriff. Dadurch wird der Thread am Weiterarbeiten gehindert und kann in seiner Ausführung blockiert sein. Der Zustand `RUNNABLE` wird dabei jedoch nicht verlassen. Diese Situation ist nicht mit derjenigen beim Zustand `BLOCKED` zu verwechseln, der dem Warten auf die Zuteilung des Zutritts zu einem momentan durch einen anderen Thread belegten kritischen Bereich entspricht. Im Zustand `BLOCKED` erhält ein Thread keine Rechenzeit und konkurriert dadurch nicht mit anderen Threads um Zuteilung des Prozessors. Gleiches gilt für die beiden Wartezustände `WAITING` und `TIMED_WAITING`.

Zustandswechsel ausführen

Nehmen wir eine normale, nicht blockierende Abarbeitung der Kommandos des aktiven Threads an, so können durch Aufruf der nun vorgestellten Methoden spezielle Zustandswechsel selbst initiiert werden.

yield()

Der aktive Thread kann über den Aufruf der statischen Methode `yield()` pausiert werden, verbleibt aber im Zustand `RUNNABLE`. Dadurch wird anderen Threads die Gelegenheit zur Ausführung gegeben. Besonders für lang dauernde Berechnungen scheint es sinnvoll, diese Methode von Zeit zu Zeit aufzurufen. Die Wahl des nächsten auszuführenden Threads erfolgt durch den Thread-Scheduler prioritätsorientiert, aber zufällig. Daher kann möglicherweise der gerade angehaltene Thread sofort wieder gewählt werden, weshalb der Aufruf der im Folgenden beschriebenen Methode `sleep()` zu bevorzugen ist.

sleep()

Durch Aufruf der statischen Methode `sleep(long)` wird der momentan aktive Thread für eine angegebene Zeitdauer »schlafen« gelegt. Der Thread wechselt in den Zustand `TIMED_WAITING` und verbraucht dadurch keine CPU-Zeit mehr, was anderen Threads ihre Ausführung erlaubt. Nachdem die angegebene Wartezeit abgelaufen ist, wechselt der ruhende Thread automatisch in den Zustand `RUNNABLE`, was jedoch nicht zum sofortigen Wiederaufnehmen seiner Ausführung führt. Gegebenenfalls werden zunächst noch andere Threads im Zustand `RUNNABLE` ausgeführt.

wait()

Bei der Zusammenarbeit mehrerer Threads kann einer von diesen seine Ausführung unterbrechen wollen, bis eine spezielle Bedingung erfüllt ist. Dies ist beispielsweise der Fall, wenn Berechnungsergebnisse anderer Threads zur sinnvollen Weiterarbeit benötigt werden. Durch einen Aufruf der Objektmethode `wait()` erfolgt dann ein Wechsel in den Zustand `WAITING` bzw. `TIMED_WAITING`, je nachdem, ob eine maximale Wartezeit übergeben wurde oder nicht. In diesem Zustand verweilt der Thread, bis entweder ein anderer Thread die Methode `notify()` bzw. `notifyAll()` aufruft, um das Eintreten einer Bedingung zu signalisieren, oder eine optional angegebene Wartezeit verstrichen ist.

***Die in diesem Absatz genannten Methoden* `wait()`, `notify()` *bzw.* `notifyAll()` *zur Kommunikation von Threads stammen nicht, wie man zunächst vermuten könnte, aus der Klasse* `Thread`, *sondern aus der Klasse* `Object`.** Dies ist dadurch begründet, dass *mehrere* Threads auf zu schützenden Daten *eines* Objekts arbeiten und die Threads untereinander abgestimmt werden müssen (vgl. Abschnitt 7.2).

7.1.4 Unterbrechungswünsche durch Aufruf von `interrupt()`

Bei der Kommunikation von Threads, auf die ich in Abschnitt 7.3 detaillierter

eingehe, kann ein Thread einen anderen Thread in seiner Abarbeitung unterbrechen bzw. beenden wollen. Dazu dient die Methode `interrupt()` der Klasse `Thread`. Diese hat jedoch keine unmittelbare unterbrechende Wirkung, sondern ist lediglich als Aufforderung oder Hinweis zu verstehen. Durch die JVM wird beim empfangenden Thread ein spezielles Interrupted-Flag gesetzt.

Ein Empfänger dieser Aufforderung sollte geeignet darauf reagieren. Einem gerade aktiven Thread ist dies unmittelbar möglich, indem er über einen Aufruf der Methode `isInterrupted()` der Klasse `Thread` prüft, ob er zwischenzeitlich eine solche Aufforderung empfangen hat, d. h., ob das Flag gesetzt ist. In diesem Fall sollte die Abarbeitung der Anweisungen in der `run()`-Methode beendet werden. Die dazu notwendige Prüfung ist gegebenenfalls mehrfach innerhalb der `run()`-Methode auszuführen, um eine zeitnahe Reaktion auf einen Unterbrechungswunsch zu garantieren. Nachfolgend ist dies symbolisch für drei Arbeitsschritte `doSomeWork1/2/3()` gezeigt:

```
while (!Thread.currentThread().isInterrupted())
{
    doSomeWork1();

    if (!Thread.currentThread().isInterrupted())
    {
        doSomeWork2();
    }

    if (!Thread.currentThread().isInterrupted())
    {
        doSomeWork3();
    }
}
```

Behandlung von `interrupt()` und `InterruptedExceptions`

Komplizierter wird die Abfrage, wenn ein Thread momentan nicht aktiv ist, weil er eine bestimmte Zeitspanne pausiert (`sleep(long)`) oder aber auf ein bestimmtes Ereignis wartet (`wait()`). Kommt es während dieser Zeit zu einem Aufruf von `interrupt()` durch einen anderen Thread, so wird zwar in beiden Fällen das Flag gesetzt, eine Reaktion ist jedoch nicht sofort möglich, da der Thread noch nicht aktiv ist. Daher wird von der JVM das Warten abgebrochen und der Thread wechselt in den Zustand `RUNNABLE`. Bei einer anschließenden Aktivierung durch den Thread-Scheduler wird von der JVM eine `java.lang.InterruptedException` ausgelöst.[3] Da es sich um eine Checked Exception handelt, muss diese propagiert oder mit einem `catch`-Block behandelt werden (vgl. Abschnitt 4.7). Dabei ist ein Detail zu beachten, auf das ich nun eingehe.

Existiert ein `catch (InterruptedException)`-Block, so werden seine Anweisungen ausgeführt. Leider sieht man dort häufig in etwa folgende »Behandlung«:

```
try
{
    Thread.sleep(duration);
}
catch (final InterruptedException e)
{
    // ACHTUNG: unzureichende Behandlung
    e.printStackTrace();
}
```

Dieses Vorgehen ist nahezu immer problematisch. Die JVM löscht nämlich bei Eintritt in den `catch`-Block das zuvor gesetzte Interrupted-Flag! Dadurch führt eine mit einem »leeren« `catch`-Block abgefangene `InterruptedException` dazu, dass ein Thread einen Unterbrechungswunsch nicht mehr erkennen kann und daher auch nicht terminiert.[4]

Wie geht man also mit der Situation um? Eine konsistente Abfrage des Interrupted-Flags mit der Methode `isInterrupted()` wird ermöglicht, wenn man im `catch`-Block über einen expliziten Aufruf der Methode `interrupt()` dieses Flag erneut setzt:

```
while (!Thread.currentThread().isInterrupted())
{
    doSomething();

    try
    {
        Thread.sleep(duration);
    }
    catch (final InterruptedException e)
    {
        // Flag erneut setzen und dadurch Abbruch ermöglichen
        Thread.currentThread().interrupt();
    }
}
```

Die gezeigte Schleifenkonstruktion kann man dazu einsetzen, Threads gezielt zu beenden. Abschnitt 7.5.3 beschreibt gebräuchliche Alternativen.

Entwurf einer Utility-Klasse

Da Aufrufe von `Thread.sleep(long)` in der Praxis immer wieder zum Verzögern oder Warten eingesetzt werden und dabei fälschlicherweise häufig der `catch`-Block leer implementiert wird, bietet sich der Entwurf einer Utility-Klasse `SleepUtils` an. Diese stellt zwei überladene Methoden `safeSleep()` bereit, die für eine korrekte Verarbeitung der Exception sorgen und das Interrupted-Flag erneut setzen:

```
public final class SleepUtils
{
    public static void safeSleep(final TimeUnit timeUnit, final long duration)
    {
        safeSleep(timeUnit.toMillis(duration));
    }

    public static void safeSleep(final long durationInMilliSecs)
```

```
    {
        try
        {
            Thread.sleep(durationInMilliSecs);
        }
        catch (final InterruptedException e)
        {
            Thread.currentThread().interrupt();
        }
    }

    private SleepUtils()
    {}
}
```

Hinweis: Leere `catch`-Blöcke für `InterruptedException`

Findet keine Kommunikation mit anderen Threads statt und ruft man die statische Methode `Thread.sleep(long)` auf, um die Ausführung des eigenen Threads kurzfristig zu unterbrechen, so kann der `catch`-Block in diesem Spezialfall leer implementiert werden. Der Grund ist einfach: Kennen sich Threads nicht, können sich diese auch nicht per `interrupt()` gegenseitig aufrufen. Eine `InterruptedException` kann nicht auftreten, muss aber bekanntermaßen behandelt oder propagiert werden. Kommunizieren Threads miteinander, deutet eine `InterruptedException` jedoch darauf hin, dass ein Thread einen anderen unterbrechen und eventuell sogar stoppen möchte, und darf somit natürlich nicht ignoriert werden. Dazu kann man generell die zuvor gezeigte Lösung verwenden. Eine Abfrage des Interrupted-Flags ist dadurch jederzeit bei Bedarf konsistent möglich.

7.2 Zusammenarbeit von Threads

Bei der Zusammenarbeit von Threads und dem Zugriff auf gemeinsam benutzte Daten müssen einige Dinge beachtet werden. Wir betrachten zunächst Situationen, in denen es zu Zugriffsproblemen und Zweideutigkeiten kommt. Zu deren Vermeidung werden anschließend Locks, Monitore und kritische Bereiche vorgestellt.

7.2.1 Konkurrierende Datenzugriffe

Der Zugriff auf gemeinsame Daten muss beim Einsatz von Multithreading immer aufeinander abgestimmt erfolgen, um Konsistenzprobleme zu vermeiden. ***Ein einziger unsynchronisierter, konkurrierender Datenzugriff kann bereits Probleme auslösen.*** Meistens sind solche Situationen schwierig zu reproduzieren, wodurch eine Fehlersuche sehr mühselig wird. Zudem können bereits minimale Änderungen im Zusammenspiel von Threads zu einem Verschwinden der Probleme oder zu ihrem

erstmaligen Auftreten führen. Bei derartigen Situationen spricht man auch von ***Race Conditions***: Das Ergebnis einer Berechnung ist dann nicht deterministisch und durch die Reihenfolge oder die zeitlichen Abläufe bei der Ausführung einzelner Anweisungen geprägt.

Betrachten wir als Beispiel den Start einer Applikation, die aus mehreren Komponenten besteht. Diese greifen während ihrer Startphase auf eine Klasse, realisiert gemäß dem SINGLETON-Muster (vgl. Abschnitt 18.1.4), zu. Der Zugriff geschieht mithilfe der folgenden problematischen, aber doch mitunter selbst in Produktivcode zu findenden `getInstance()`-Methode:

```
// ACHTUNG: SCHLECHT !!!
public static BadSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BadSingleton();
    }
    return INSTANCE;
}
```

Untersuchen wir nun, was daran problematisch ist. Nehmen wir dazu vereinfachend an, während des Starts der Applikation würden zwei Komponenten (Thread 1 und Thread 2) auf diese `getInstance()`-Methode zugreifen. Erfolgt der Aufruf nahezu zeitgleich, so ist es möglich, dass einer der aufrufenden Threads direkt nach der `null`-Prüfung durch den Thread-Scheduler unterbrochen und anschließend der andere Thread aktiviert wird. Beide Threads »sehen« damit einen `null`-Wert für die Variable `INSTANCE`, weil ja noch keine Konstruktion und Zuweisung erfolgt ist. Somit wird der Konstruktoraufruf und die Zuweisung für jeden der beiden Threads ausgeführt. Somit existieren also zwei Instanzen eines Singletons: Beide Threads besitzen ihre eigene Instanz! Dies widerspricht natürlich vollständig dem Gedanken des Singletons.

Die Wahrscheinlichkeit für eine solche Race Condition durch einen gemeinsamen Zugriff erhöht sich mit der Verweildauer eines Threads im Konstruktor der Klasse `BadSingleton`, beispielsweise, wenn dort aufwendige Initialisierungen stattfinden. In diesem Szenario ruft Thread 2 die Methode `getInstance()` auf, während Thread 1 noch im Konstruktor beschäftigt ist: Die Variable `INSTANCE` ist dann aber noch nicht zugewiesen und hat damit immer noch den Wert `null`.

Durch das Java-Memory-Modell (JMM), dessen Implikationen wir in Abschnitt 7.4 genauer betrachten werden, ist es jedoch prinzipiell egal, ob Thread 1 bereits einen Wert `INSTANCE != null` produziert hat. Selbst wenn Thread 1 bereits einige Zeit mit einer solchen Referenz arbeitet, ist nicht sichergestellt, dass Thread 2 die Änderungen von Thread 1 an der Variablen `INSTANCE` mitbekommt. Das liegt daran, dass Threads zum Teil einige Variablen in Thread-lokalen Caches zwischenspeichern und die Werte nur zu definierten Zeitpunkten mit dem Hauptspeicher abgleichen. Da für Thread 2 ohne das Schlüsselwort `synchronized` kein Zwang besteht, sich beim Zugriff auf `get-Instance()` neu mit dem

Hauptspeicher zu synchronisieren, ist es durchaus möglich, dass er mit einem alten Wert der Variablen `INSTANCE` arbeitet.

Anhand dieses Beispiels sehen wir, wie wichtig es ist, den Zugriff auf von mehreren Threads gemeinsam benutzte Variablen abzustimmen. Dies ist durch die Definition sogenannter ***kritischer Bereiche*** möglich, zu denen jeweils nur ein Thread zur gleichen Zeit »Zutritt« hat. Im einfachsten Fall kann man dazu das Schlüsselwort `synchronized` auf Methodenebene wie folgt nutzen:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

7.2.2 Locks, Monitore und kritische Bereiche

Um konkurrierende Zugriffe mehrerer Threads für kritische Bereiche zu verhindern, existiert eine Zugangskontrolle, auch ***Monitor*** genannt. Ein solcher Monitor sorgt dafür, dass der Zutritt zu einem kritischen Bereich immer nur einem einzelnen Thread gewährt wird. Dazu wird eine spezielle Sperre (***Lock***) genutzt, die durch das Schlüsselwort `synchronized` gesteuert wird. In Java verwaltet jedes Objekt zwei Sperren: Eine davon bezieht sich auf die Objektreferenz (`this`) und dient dazu, Objektmethoden zu synchronisieren. Die andere wirkt auf die Klassenreferenz (`class`-Referenz), womit die Synchronisation für statische Methoden möglich ist.

Tipp: Locks und die Analogie zum Wartezimmer

Das Vorgehen beim Synchronisieren mit Locks und Monitoren kann man sich mithilfe der Analogie eines Wartezimmers beim Arzt klarmachen. Der Arzt stellt die durch den Lock geschützte Ressource dar, die exklusiv von einem Patienten (Thread) belegt wird. Kommen weitere Patienten zum Arzt, so müssen diese zunächst im Wartezimmer (Warteliste) Platz nehmen und gehen dort in einen Wartezustand, einen Zustand der Blockierung für andere Dinge. Die Arzthelferin (Monitor) gewährt den Zutritt zum Doktor, sobald dieser wieder Zeit hat, also der vorherige Patient den Lock zurückgegeben hat.

Vor Eintritt in einen mit `synchronized` markierten kritischen Bereich wird von der JVM automatisch geprüft, ob der angeforderte Lock verfügbar ist. In diesem Fall wird der Lock an den anfragenden Thread vergeben und der Thread erhält Zutritt zu dem kritischen Bereich. Nach Abarbeitung des `synchronized`-Blocks wird der Lock ebenfalls automatisch wieder freigegeben. Kann dagegen ein Lock nicht akquiriert werden, so muss der anfragende, momentan aktive Thread darauf warten, wird inaktiv und in eine spezielle Warteliste des Locks eingetragen. Das geschilderte Vorgehen führt zu einem gegenseitigen Ausschluss: ***Ein durch einen kritischen***

Bereich geschützter Zugriff auf eine gemeinsame Ressource serialisiert die Abarbeitung mehrerer Threads. Statt einer parallelen Abarbeitung kommt es zu einer zeitlich versetzten Bearbeitung. Je mehr Threads um den Zugriff auf einen Lock konkurrieren, desto mehr leidet die Nebenläufigkeit. Abbildung 7-3 zeigt dies symbolisch für drei Threads T_1, T_2 und T_3, die eine ungeschützte und eine durch `synchronized` geschützte Methode ausführen.

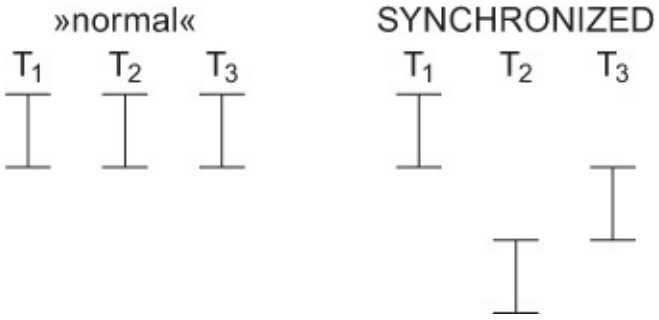


Abbildung 7-3 *Serialisierung der Abarbeitung durch Locks*

Im ersten Fall kann die Abarbeitung parallel erfolgen (oder zumindest quasi parallel durch mehrfache Thread-Wechsel). Im zweiten Fall müssen Threads warten und werden sequenziell abgearbeitet (aber nicht unbedingt in der Reihenfolge des versuchten Eintritts in den kritischen Bereich).

Ein Monitor sorgt sowohl dafür, dass ein verfügbarer Lock vergeben wird, als auch dafür, dass ein anfragender Thread bei Nichtverfügbarkeit in die Liste der auf diesen Lock wartenden Threads aufgenommen wird. Wird ein Lock freigegeben, erhält ihn ein beliebiger Thread aus der Warteliste und kann daraufhin seine Ausführung fortsetzen. Diese Aktionen laufen für den Entwickler unsichtbar automatisch durch die JVM ab.

Kritische Bereiche und das Schlüsselwort `synchronized`

Zur Definition kritischer Bereiche gibt es in Java das Schlüsselwort `synchronized`. Dabei existieren zwei Varianten des Einsatzes:

- `synchronized`-Methode – Bei dieser gebräuchlichen Variante wird eine Methode mit dem Schlüsselwort `synchronized` gekennzeichnet.
- Synchronisationsobjekt – Etwas seltener sieht man, dass ein Block von Anweisungen durch einen `synchronized`-Block ummantelt wird.

`synchronized`-Methode

Die bekannteste und bereits beschriebene Variante zur Definition eines kritischen Bereichs ist das ***Synchronisieren der gesamten Methode*** durch Erweitern der Signatur um das Schlüsselwort `synchronized`:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
```

```
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

Zum Ausführen einer synchronisierten Methode muss ein Thread zunächst den entsprechenden Lock zugeteilt bekommen, ansonsten wird seine Abarbeitung angehalten und der Thread wartet auf die Freigabe des benötigten Locks, um diesen zunächst zu akquirieren und anschließend mit der Berechnung fortfahren zu können.

Greifen weitere Methoden auf eine zu schützende Ressource bzw. ein Attribut zu, so müssen alle diese Methoden synchronisiert werden. Aufgrund der Arbeitsweise der Locks wissen wir, dass dadurch alle `synchronized`-Methoden des Objekts für Zugriffe durch andere Threads gesperrt sind. ***Diese Art der Synchronisierung erschwert somit die Nebenläufigkeit***. Die parallele Ausführung weiterer als `synchronized` deklarierter Methoden eines Objekts ist nicht möglich.

Was kann an dem `synchronized` für eine Methode außerdem problematisch sein? Ist beispielsweise die Ausführung einer geschützten Methode zeitintensiv, so müssen andere Threads lange warten, um den Lock zu erhalten. Damit ist klar, dass ein Synchronisieren über Methoden bei dem Wunsch nach viel Parallelität nicht optimal ist. Es gibt aber noch einen weiteren Fallstrick: Das synchronisierte Verhalten betrifft nur die deklarierende Klasse, sodass in Subklassen die Synchronisierung durch Überschreiben verloren geht! Betrachten wir nun eine andere, elegantere Synchronisierungsvariante.

Synchronisationsobjekt

Die unbekanntere, aber oft sinnvollere Möglichkeit zur Definition eines kritischen Bereichs ist das ***Synchronisieren eines Abschnitts innerhalb von Methoden***. Man nutzt dazu ein sogenanntes ***Synchronisationsobjekt***. Prinzipiell kann jedes beliebige Objekt dazu verwendet werden, seinen Lock zur Verfügung zu stellen. Um Anwendungsfehler zu vermeiden und die Semantik eines solchen Synchronisationsobjekts besonders herauszustellen, sollte jedoch bevorzugt eine finale Objektreferenz vom Typ `Object` genutzt werden, im folgenden Beispiel `LOCK` genannt. Zum Eintritt in einen durch `synchronized (LOCK)` geschützten kritischen Bereich muss ein Thread den Lock dieses speziellen Objekts erhalten.

Synchronisationsobjekte nutzt man, um feingranulare Sperren zu realisieren, d. h., um kleine Blöcke innerhalb von Methoden zu schützen. Am Beispiel der folgenden Methoden `calcValue1()` und `calcValue2()` zeige ich dies anhand von Berechnungen, die sich zu verschiedenen Zeitpunkten in ihrer Abarbeitung gegenseitig ausschließen sollen. Statt aber beide Methoden über `synchronized` exklusiv auszuführen und damit Nebenläufigkeit zu verhindern, werden in diesen Methoden nur die wirklich kritischen Abschnitte über `synchronized (LOCK)` vor einer gleichzeitigen Ausführung geschützt:

```
// Gemeinsame genutzte und daher zu schützende Daten
private final List<Integer> sharedData = new ArrayList<Integer>();

// Synchronisationsobjekt
private final Object        LOCK       = new Object();

public int calcValue1()
{
    // Blockiert calcValue2()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    // Aktionen ohne Lock, calcValue2() kann ausgeführt werden

    return 1;
}

public int calcValue2()
{
    // Blockiert calcValue1()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    // Aktionen ohne Lock, calcValue1() kann ausgeführt werden

    // Blockiert calcValue1()
    synchronized (LOCK)
    {
        // Kritischer Bereich, Zugriff auf sharedData
    }

    return 2;
}
```

Mehr Nebenläufigkeit kann man erreichen, wenn man mehrere solcher Lock-Objekte für diejenigen Ressourcen und Attribute einführt, die geschützt werden sollen. Dadurch werden konkurrierende Zugriffe auf andere gemeinsame Attribute nicht mehr blockiert und jeweils immer nur der benötigte Zugriff exklusiv ausgeführt. Zum Teil lässt sich das zu schützende Objekt selbst als Synchronisationsobjekt verwenden. Allerdings ist dies nur dann möglich, wenn sichergestellt ist, dass sich die Referenz darauf nicht ändert, diese also `final` ist. Ansonsten kann es zu einer `java.lang.IllegalMonitorStateException` kommen. Darauf gehe ich in Abschnitt 7.3 genauer ein.

Analogie von Synchronisationsobjekt und `synchronized`-Methode

Der Einsatz von Synchronisationsobjekten bietet mehr Möglichkeiten als die Nutzung von `synchronized`-Methoden. Insbesondere kann man das Verhalten von `synchronized`-Methoden durch den Einsatz von Synchronisationsobjekten

abbilden, die die `this`-Referenz verwenden, und den synchronisierten Abschnitt auf die komplette Methode ausdehnen. Eine Synchronisierung auf `this` innerhalb einer Methode wie folgt

```
void method()
{
    synchronized (this)
    {
        // …
    }
}
```

entspricht dem Verhalten des Schlüsselworts `synchronized` auf Methodenebene:[5]

```
synchronized void method()
{
    // …
}
```

Eine analoge Aussage gilt für statische Methoden. Dort wird der Lock durch das Synchronisieren auf die `class`-Variable beschrieben. Das folgende

```
static void staticMethod()
{
    synchronized (SynchronizationExample.class)
    {
        // …
    }
}
```

ist mit der synchronisierten Methode äquivalent:

```
static synchronized void staticMethod()
{
    // …
}
```

Tipp: Fallstricke beim Synchronisieren

Ein Schutz vor konkurrierenden Zugriffen muss immer vollständig erfolgen. ***Ein einziger nicht synchronisierter Zugriff auf ein zu schützendes Attribut reicht aus, um Multithreading-Probleme zu verursachen.*** Dies kann schnell geschehen, wenn verschiedene Synchronisationsvarianten nicht konsequent, sondern beispielsweise gemischt verwendet werden. Somit hält man potenziell den Lock auf das »falsche« Synchronisationsobjekt. Lassen Sie mich – weil es wichtig ist – nochmals auf Folgendes hinweisen: ***Werden nicht alle Zugriffe über denselben Lock geschützt, kann es leicht zu Inkonsistenzen oder Deadlocks kommen***.

Mächtigkeit von `synchronized`-Blöcken: Synchronisations-Proxy

Gerade haben wir gesehen, dass die Möglichkeiten mit `synchronized`-Blöcken vielseitiger sind als mit dem Schlüsselwort `synchronized`. Was man mit `synchronized`-Blöcken darüber hinaus noch machen kann, betrachten wir am

Beispiel der Klasse `ThreadUnsafeClass`. Als ergänzende Anforderung wird für diese Thread-Sicherheit benötigt. Als zusätzliche Randbedingung gilt, dass wir den Sourcecode der Klasse nicht besitzen oder ändern dürfen.

Wenn wir diese Klasse um Thread-Sicherheit oder genauer synchronisierte Zugriffe erweitern wollen, dann müssen wir dafür sorgen, dass alle Methoden in einem geschützten Bereich ausgeführt werden. Dazu können wir eine sogenannte Stellvertreter-Klasse definieren, die alle öffentlichen Methoden der zu schützenden Klasse implementiert (also deren öffentliche Schnittstelle erfüllt) und dort die Methodenaufrufe an die jeweiligen korrespondierenden Methoden der Originalklasse weiter delegiert. Außerdem muss jede derartige Methodendelegation in einen `synchronized(this)`-Block eingeschlossen werden. Die Zugriffe auf das Objekt vom Typ `ThreadUnsafeClass` mit dem Interface `OriginalIF` werden dann durch Locks von der dekorierenden Klasse geschützt. Folgendes Listing deutet diese Form der Realisierung an:

```
public class SynchronizationProxy implements OriginalIF
{
    private final ThreadUnsafeClass original;

    public void doSomething()
    {
        synchronized(this)
        {
            original.doSomething();
        }
    }
}
```

Ähnlich zu dieser Umsetzung sind auch die `synchronized`-Wrapper des Collections-Frameworks realisiert. Schauen wir kurz auf einige Besonderheiten bei der Synchronisierung für Collections.

Thread-Sicherheit und Parallelität mit »normalen« Collections

Die `synchronized`-Wrapper des Collections-Frameworks ermöglichen einen Thread-sicheren Zugriff durch Synchronisierung aller Methoden. Betrachten wir folgende synchronisierte Liste von `Person`-Objekten als Ausgangsbasis unserer Diskussion:

```
final List<Person> syncPersons = Collections.synchronizedList(personList);
```

Bei parallelen Zugriffen auf diese Liste können sich Threads gegenseitig blockieren und stören. Die Synchronisierung stellt einen Engpass dar und serialisiert die Zugriffe: Es kommt dadurch zu (stark) eingeschränkter Parallelität. Eine derartige Ummantelung schützt zudem nicht vor möglichen Inkonsistenzen: Wenn man mehrere für sich Thread-sichere Methoden hintereinander aufruft, ist dadurch eine atomare Ausführung als kritischer Bereich nicht garantiert. Dies habe ich bereits bei der Beschreibung der `synchronized`-Wrapper in Abschnitt 5.3.2 und bei der Darstellung eines Singletons und dessen `getInstance()`-Methode in Abschnitt

7.2.1 diskutiert.

Jeder einzelne Methodenaufruf einer synchronisierten Collection ist für sich gesehen Thread-sicher. Damit ist gemeint, dass Zugriffe mehrerer Threads keine Inkonsistenzen innerhalb der Collection selbst verursachen. Für eine nutzende Komponente sind solche feingranularen Sperren aber häufig nicht ausreichend. Vielmehr sollen Operationen mit mehreren Schritten atomar und Thread-sicher ausgeführt werden. Solche Mehrschrittoperationen sind etwa das Iterieren oder Methoden wie »`testAndGet()`«, die zunächst prüfen, ob ein gewisses Element enthalten ist, und nur dann einen Zugriff bzw. eine Modifikation ausführen. Ohne viel nachzudenken, könnte man auf die etwas naive Idee kommen, die Mehrschrittoperationen durch den Einsatz einzelner Thread-sicherer Methoden wie folgt zu realisieren:

```
// ACHTUNG: nicht Thread-sicher
public Person testAndGet(final int index)
{
    if (index < syncPersons.size())
    {
        // index < size gilt evtl. nicht mehr
        return syncPersons.get(index);
    }
    return null;
}
```

Dieser Ansatz ist – wie schon erwähnt – nicht Thread-sicher: Zur Erinnerung sei nochmals erwähnt, dass ***Thread-sichere Methode in ihrer Kombination nicht Thread-sicher sind, da nach jedem geschützten Methodenaufruf ein Thread-Wechsel möglich ist.*** Erfolgt die Unterbrechung im obigen Beispiel etwa direkt nach Ausführung der `if`-Bedingungen und verändert ein aktivierter Thread die Datenstruktur, so kann dies verheerende Auswirkungen bei Wiederaufnahme eines unterbrochenen Threads haben: Bei nachfolgenden Zugriffen kommt es entweder zu Inkonsistenzen oder Exceptions. Daher müssen zusätzliche Synchronisierungsschritte ausgeführt werden. Man kann dazu ein Synchronisationsobjekt verwenden. In diesem Fall bietet sich die Collection selbst, genauer die finale Referenz `syncPersons`, an. Eine atomar ausgeführte Version der `testAndGet(int)`-Methode kann wie folgt implementiert werden:[6]

```
public Person testAndGet(final int index)
{
    // Kritischer Bereich für Mehrschrittoperationen
    synchronized (syncPersons)
    {
        if (index < syncPersons.size())
        {
            return syncPersons.get(index);
        }
    }
    return null;
}
```

Tipp: Regeln zur Verwendung von `synchronized`

Um die im Anschluss beschriebenen Deadlocks und andere Probleme möglichst auszuschließen, sollte man Folgendes beachten:

- Ein Thread kann den Lock von einem oder die Locks von verschiedenen Objekten besitzen. Er erhält Zutritt zu den derart geschützten kritischen Bereichen.
- Ein geschützter Bereich sollte möglichst kurz sein. Ideal ist es, wenn nur die wirklich zu schützenden Operationen synchronisiert werden. Nicht zu schützende Teile sollten in andere Methoden verlagert oder außerhalb des geschützten Bereichs durchgeführt werden.
- Aus einem geschützten Bereich sollten keine blockierenden Aufrufe (z. B. `read()` eines `InputStream`) erfolgen, da der Thread den Lock während des Wartens nicht abgibt. Auch Aufrufe von `Thread.sleep(long)` sind zu vermeiden, da auch hier keine Freigabe des Locks erfolgt. Andere Threads werden ansonsten möglicherweise länger blockiert.
- Aus dem vorherigen Punkt ergibt sich indirekt folgender Tipp: ***Wenn man einen Lock hält, sollte man möglichst keine Methoden anderer Objekte aufrufen (oder dabei zumindest große Vorsicht walten lassen)***. Dadurch vermeidet man Aufrufe, die zu Deadlocks führen können.
- Ein Objekt kann mehrere Methoden `synchronized` definieren. Besitzt ein Thread den Lock, können diese Methoden sich gegenseitig aufrufen, ohne dass bei jedem Methodenaufruf erneut versucht wird, den Lock zu bekommen. Wäre dies nicht so, würde sich ein Thread selbst blockieren, wenn er versuchen würde, andere `synchronized`-Methoden aufzurufen. Locks nennt man daher eintrittsinvariant oder auch ***reentrant*** [a]. Basierend darauf können `synchronized`-Methoden sich gegenseitig oder gar rekursiv aufrufen. Nur deshalb kam es zu keiner Blockierung beim obigen Beispiel mit der synchronisierten Liste!
- Nicht durch `synchronized` geschützte Methoden können jederzeit von jedem Thread aufgerufen werden.

[a] Man bezeichnet eine Methode als reentrant, wenn sie von mehreren Threads problemlos ohne gegenseitige Beeinflussung gleichzeitig ausgeführt werden kann.

7.2.3 Deadlocks und Starvation

Wie bereits eingangs diesen Kapitels erwähnt, spricht man von ***Deadlocks***, wenn Threads sich gegenseitig blockieren. Als ***Starvation*** bezeichnet man Situationen, in denen es für einen oder mehrere Threads kein Vorankommen im Programmfluss mehr gibt. Beim Einsatz von Multithreading kann beides leicht auftreten.

Deadlocks

Wenn ein beliebiger Thread 1 den Lock auf Objekt A belegt und versucht, den Lock auf Objekt B zu erhalten, kann es zu einem sogenannten ***Deadlock*** kommen, wenn ein anderer Thread 2 bereits den Lock auf Objekt B hält und seinerseits wiederum versucht, den Lock auf Objekt A zu bekommen. Beide warten daraufhin endlos. In der Informatik ist die exklusive Belegung von Ressourcen auch als das ***Philosophenproblem*** bekannt. Stark vereinfacht treffen sich zwei Philosophen P1 und P2 zum Essen an einem Tisch, allerdings haben sie nur eine Gabel und ein Messer. Dies zeigt Abbildung 7-4.

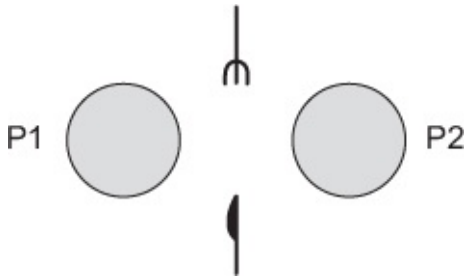


Abbildung 7-4 *Ressourcenproblem*

Wählt Philosoph P1 immer zuerst die Gabel und nimmt Philosoph P2 immer zunächst das Messer, so kann es bei ungünstiger zeitlicher Abfolge (nahezu zeitgleichem Zugriff auf die Besteck-Ressource) dazu kommen, dass beide niemals etwas essen, da ihnen die jeweils andere Ressource fehlt. Halten sich beide an eine identische Reihenfolge erst Gabel, dann Messer (oder umgekehrt), so kann es niemals zu einer gegenseitigen Blockierung kommen.

Hinweis: Vermeidung von Deadlocks

Deadlocks lassen sich vermeiden, indem man Locks und Ressourcen immer in der gleichen Reihenfolge belegt, etwa gemäß dem folgenden Schema: Lock A, Lock B, Unlock B, Unlock A. Hält man sich an diese Regel, so vermeidet man Situationen, in denen ein Thread auf einen anderen wartet und keiner von beiden ausgeführt werden kann, weil beide jeweils auf eine bereits belegte Ressource zugreifen wollen. Bei komplexeren Abläufen ist jedoch nicht immer sofort klar, wer wann welche Ressource belegt und wieder freigibt bzw. sogar, ob dies überhaupt geschieht.

Starvation

Wenn der momentane Besitzer einer Ressource diese nach getaner Arbeit nicht wieder freigibt, obwohl er sie nicht weiter benötigt, kann es zu dem ***Starvation*** genannten Phänomen kommen: Andere auf diese Ressource wartende Threads sind endlos blockiert, da sie vergeblich auf den Erhalt der benötigten Ressource warten. Verhalten sich alle Beteiligten allerdings fair, so muss zwar manchmal einer auf den anderen warten (wenn dieser gerade die benötigten Ressourcen besitzt), aber alle können ihre Aufgabe auf jeden Fall nacheinander bzw. abwechselnd ausführen.

7.2.4 Kritische Bereiche und das Interface `Lock`

Java bietet mit dem Schlüsselwort `synchronized` leider keine Unterbrechbarkeit oder Time-outs beim Zugriff auf Locks. Als Abhilfe gibt es seit JDK 5 zur Definition kritischer Bereiche ergänzend zu den impliziten Sperren über `synchronized` explizite Sperrvarianten. Die für diese Funktionalität wichtigsten Interfaces sind `Lock`, `ReadWriteLock` und `Condition` sowie die beiden Klassen `ReentrantLock` und `ReentrantReadWriteLock` aus dem Package `java.util.concurrent.locks`. Durch deren Einsatz können einige Schwächen des in die Sprache integrierten Lock- und Synchronisationsmechanismus umgangen bzw. behoben werden. Die wichtigsten Möglichkeiten sind in folgender Aufzählung genannt:

- Ein Thread kann unterbrechbar versuchen, auf einen Lock zuzugreifen.
- Ein Thread kann mit einer maximalen Wartezeit und nicht nur unbeschränkt lange auf den Zugriff eines Locks warten, wie dies bei `synchronized` geschieht.
- Es werden unterschiedliche Arten von Locks unterstützt. Beispielsweise sind dies sogenannte Read-Write-Locks, die mehrere parallele Lesezugriffe erlauben, solange der zugehörige Write-Lock nicht vergeben wurde.
- `Lock`-Objekte können nun nicht nur einen gewissen Block aufspannen, sondern beliebige Bereiche. Dadurch lassen sie sich im Gegensatz zu `synchronized` in verschiedenen Klassen über Block- und Methodengrenzen hinweg einsetzen.

Grundlagen von Locks

`Lock`-Objekte stellen eine Erweiterung von Synchronisationsobjekten dar. Letztere verwendet man bekanntermaßen wie folgt:

```
synchronized (lockObj)
{
    // Kritischer Bereich
}
```

Da `Lock`-Objekte nicht auf den in die JVM integrierten Mechanismen der Monitore beruhen, muss die Definition eines kritischen Bereichs explizit über Methodenaufrufe programmiert werden. Zum Sperren dient die Methode `lock()`. Diese wird auf einem `Lock`-Objekt aufgerufen. Soll der kritische Bereich beendet werden, muss explizit ein Methodenaufruf von `unlock()` erfolgen:

```
final Lock lockObj = new ReentrantLock();
lockObj.lock();       // Explizit Lock anfordern
try
{
    // Kritischer Bereich
}
finally
{
```

```
    lockObj.unlock(); // Explizit Lock freigeben
}
```

Das Listing zeigt den Einsatz des Interface `Lock` und dessen konkreter Realisierung `ReentrantLock`, um ein zu `synchronized` kompatibles Verhalten nachzubilden, das sowohl Eintrittsinvarianz bietet als auch rekursive Methodenaufrufe erlaubt. Die Implementierungen von Locks garantieren zudem ein bezüglich der Sichtbarkeit von Änderungen zu `synchronized` kompatibles Verhalten. Die Hintergründe beschreibt der folgende Praxistipp.

Info: Konsistenz von Locks und `synchronized`

Es mag zunächst verwundern, dass Locks die Sichtbarkeit von Änderungen wie `synchronized` herstellen, da keine explizite Synchronisierung zu sehen ist. Diese sorgt neben dem gegenseitigen Ausschluss von Threads auch für Konsistenz von Wertänderungen an Attributen, indem die sogenannte ***Happens-before***-Ordnung des Java-Memory-Modells (vgl. Abschnitt 7.4) eingehalten wird. Dadurch sind Änderungen innerhalb eines `synchronized`-Blocks anschließend für andere lesend zugreifende Threads sichtbar. Locks nutzen `volatile`-Attribute, die ebenfalls eine Happens-before-Ordnung sicherstellen. Damit erreicht man auch die Konsistenz von Änderungen.

Parallelität durch den Einsatz von Read-Write-Locks

Für viele Anwendungsfälle erfolgen deutlich mehr Lese- als Schreibzugriffe. Dann bietet sich der Einsatz sogenannter Read-Write-Locks an, die beide Arten von Zugriffen schützen können. Mehrere Threads können Read-Locks halten, um parallele Lesezugriffe zu erlauben, solange kein Thread Schreibzugriffe ausführt bzw. genauer: das Write-Lock akquiriert hat. Das beschriebene Verhalten wird durch die Klasse `ReentrantReadWriteLock` realisiert, die das Interface `ReadWriteLock` implementiert. Dieses bietet Zugriff auf zwei spezielle Locks und ist folgendermaßen definiert:

```
public interface ReadWriteLock
{
    public Lock readLock();
    public Lock writeLock();
}
```

Während ein Thread das Write-Lock besitzt, können andere Threads das Read-Lock nicht akquirieren. Finden andererseits noch Leseaktionen statt, während ein Thread Zugriff auf das Write-Lock bekommen möchte, muss dieser warten, bis alle lesenden Threads ihre Read-Locks freigegeben haben. Das kann jedoch dazu führen, dass Schreibzugriffe stark verzögert werden, wenn es sehr viele Lesezugriffe gibt. Ist ein solches Verhalten nicht gewünscht, so kann man bei der Konstruktion der Klasse `ReentrantReadWriteLock` einen Parameter übergeben, der dafür sorgt, dass die Klasse sich »fair« verhält. In diesem Modus werden Threads daran gehindert, das Read-Lock zu akquirieren, solange es noch Threads gibt, die auf das Write-Lock

warten. Dadurch sollen Schreibzugriffe selbst bei höherem Leseaufkommen möglich sein und Starvation der Schreibzugriffe vermieden werden.

Beispiel: `synchronized` durch Locks ersetzen

In diesem Abschnitt erläutere ich, wie man einen Benachrichtigungsmechanismus gemäß dem BEOBACHTER-Muster (vgl. Abschnitt 18.3.7) mit gegenseitigem Ausschluss per `synchronized` auf die Möglichkeiten der Lock-Klassen umstellen kann.

Ausgangslage mit `synchronized`

Nehmen wir an, dass sich verschiedene Beobachter vom Typ `ChangeListener` für Änderungsmitteilungen an- bzw. abmelden können. Die entsprechenden `add/removeChangeListener(ChangeListener)`-Methoden sind jeweils `synchronized` definiert. Zur Darstellung von Veränderungen am Objektzustand ist hier exemplarisch lediglich eine Methode `changeState(int)` definiert. In dieser wird, durch `synchronized` geschützt, der Objektzustand modifiziert. Danach werden dann alle angemeldeten Beobachter benachrichtigt. Dazu dient die synchronisierte Methode `notifyChangeListeners()`.

Im Folgenden ist eine gebräuchliche, Thread-sichere und auf den konsequenten Einsatz von `synchronized` beruhende, aber bezüglich Nebenläufigkeit nicht optimale Realisierung eines Benachrichtigungsmechanismus gezeigt. Die Beobachter sind hier in einer `ArrayList<ChangeListener>` gespeichert:

```
private final List<ChangeListener> listeners = new ArrayList<>();

public synchronized void addChangeListener(final ChangeListener listener)
{
    listeners.add(listener);
}

public synchronized void removeChangeListener(final ChangeListener listener)
{
    listeners.remove(listener);
}

private synchronized void notifyChangeListeners()
{
    for (final ChangeListener currentListener : listeners)
    {
        // Benachrichtigung der Listener: Aufruf von Callback update()
        currentListener.update();
    }
}

public synchronized void changeState(final int newValue)
{
    state = newValue;
    notifyChangeListeners();
}
```

Durch die Synchronisierung auf Methodenebene erfolgt der gegenseitige Ausschluss für Lese- und Schreiboperationen gleichermaßen. Änderungen in der Liste der Beobachter, also Schreibzugriffe auf das Attribut `listeners`, sind nach einer initialen Registrierungsphase in der Praxis eher selten. Mitteilungen über Zustandsänderungen an angemeldete Beobachter sind dagegen die Regel. Sie erfordern lediglich lesenden Zugriff auf die Liste der Beobachter. Obwohl diese Lesezugriffe voneinander unabhängig ausgeführt werden könnten, blockieren sie sich bei dieser Art der Umsetzung aber gegenseitig. Zusätzlich ist zu bedenken, dass es dabei zur Störung von Nebenläufigkeit kommen kann, wenn die Abarbeitung der Callback-Methode `update()` innerhalb einzelner Beobachter aufwendig ist. ***Als goldene Regel gilt allgemein: Callback-Methoden sollten schnell abgearbeitet werden können.***

Unzulänglichkeiten von `synchronized`

Beim Einsatz von `synchronized` sind generell alle auf diesen Lock wartenden Aktionen blockiert, bis die momentan ausgeführte Aktion beendet ist und den gehaltenen Lock freigibt. Oftmals ist jedoch eine solche exklusive Sperre für parallele Zugriffe gar nicht erforderlich. Dies gilt insbesondere, wenn viele Lesezugriffe und nur wenige Schreibzugriffe erfolgen sollen.

Bessere Nebenläufigkeit bei der Listener-Verwaltung

Zur Entkopplung der Vorgänge »Benachrichtigung« sowie »An- und Abmeldung« von Beobachtern kann man die Methode `notifyChangeListeners()` leicht modifizieren: Statt direkt auf der Liste der Beobachter zu arbeiten, legt man eine lokale Kopie dieser Liste an. Die Benachrichtigung der Listener erfolgt anschließend mithilfe einer Iteration über diese lokale Kopie der Liste. Das sorgt für mehr Nebenläufigkeit:

```
private void notifyChangeListeners()
{
    final List<ChangeListener> copyOfListeners;
    synchronized(this)
    {
        copyOfListeners = new LinkedList<ChangeListener>(listeners);
    }
    for (final ChangeListener currentListener : copyOfListeners)
    {
        currentListener.update();
    }
}
```

Diese Art der Realisierung mit Zugriffen auf lokale Daten ist automatisch Thread-sicher und sorgt zudem für mehr Nebenläufigkeit, da An- oder Abmeldevorgänge anderer Beobachter nur für den Zeitraum der Listenkopie ausgeschlossen sind. Es bleibt allerdings das Problem einer lang andauernden Verarbeitung der Callback-Methode eines Listeners, was die weitere Abarbeitung (stark) verzögern kann. Im Gegensatz zu der vorherigen Lösung wird nur der aktuelle Thread verzögert, nicht

jedoch andere Threads, die auf synchronisierte Methoden dieses Objekts zugreifen.

Umsetzung mit Locks

Für eine gute Nebenläufigkeit ist es wünschenswert, wenn Lesezugriffe parallel erfolgen. Ein Schreibzugriff muss dagegen exklusiv ausgeführt werden: Gleichzeitig dürfen weder andere Schreib- noch Lesezugriffe erfolgen. Allerdings lässt sich diese Anforderung kaum mit `synchronized` umsetzen. Verwenden wir stattdessen die Klasse `ReentrantReadWriteLock`, so wird die Realisierung der Aufgabe relativ einfach: Über die Methode `readLock()` kann ein Lock angefordert werden, der parallele Lesezugriffe erlaubt und lediglich blockiert, wenn parallel ein Lock zum Schreiben angefordert wurde. Einen solchen erhält man mit der Methode `writeLock()` und besitzt dadurch exklusiven Schreib- und Lesezugriff.

Für die Realisierung des Benachrichtigungsmechanismus ergeben sich die in den folgenden Listings gezeigten Änderungen. Erstens werden Schreibzugriffe auf das Attribut `listeners` durch einen speziellen Lock (durch Aufruf der Methode `writeLock()`) sowohl beim Hinzufügen als auch beim Löschen von Beobachtern gesichert:

```
private final List<ChangeListener> listeners = new ArrayList<>();

private final ReadWriteLock        lockObj   = new ReentrantReadWriteLock();
private final Lock                 readLock  = lockObj.readLock();
private final Lock                 writeLock = lockObj.writeLock();

public void addChangeListener(final ChangeListener listener)
{
    writeLock.lock();
    try
    {
        listeners.add(listener);
    }
    finally
    {
        writeLock.unlock();
    }
}

public void removeChangeListener(final ChangeListener listener)
{
    writeLock.lock();
    try
    {
        listeners.remove(listener);
    }
    finally
    {
        writeLock.unlock();
    }
}
```

Zweitens benötigt eine Benachrichtigung mit `notifyChangeListeners()` nur lesenden Zugriff. Daher nutzt man hier die Methode `readLock()` zum Zugriff auf ein `Lock`-Objekt und man kann auf die Kopie der Liste der Beobachter verzichten:

```
private void notifyChangeListeners()
{
    readLock.lock();
    try
    {
        for (final ChangeListener currentListener : listeners)
        {
            currentListener.update();
        }
    }
    finally
    {
        readLock.unlock();
    }
}
```

Schließlich verbleibt die Methode `changeState(int)`, die keine Zugriffe auf das Attribut `listeners` ausführt. Sie wird weiterhin über das Lock des Objekts synchronisiert, wodurch man Inkonsistenzen durch parallele Zugriffe im eigentlichen Objektzustand verhindert. Es wird hier jedoch nur noch der wirklich kritische Bereich der Zuweisung über einen `synchronized`-Block geschützt:

```
public void changeState(final int newValue)
{
    synchronized(this)
    {
        state = newValue;
    }
    notifyChangeListeners();
}
```

Wir sollten kurz innehalten und rekapitulieren: Was bringen uns diese Änderungen? Die Umsetzung erlaubt zwar mehr Nebenläufigkeit, erfordert aber den Einsatz mehrerer Locks. Das ist noch der kleinste Nachteil: Wie man leicht sieht, entsteht viel Sourcecode und die eigentliche Programmlogik ist schlecht nachzuvollziehen. Anstatt also den Schutz der Listener – wie hier zur Demonstration der Arbeitsweise von Locks geschehen – selbst zu implementieren, sollte man auf vorgefertigte Bausteine wie die später in Abschnitt 7.6.1 beschriebenen Klassen `CopyOnWriteArrayList<E>` und `CopyOnWriteArraySet<E>` zurückzugreifen. Damit vermeidet man, dass eigene Klassen mit Utility-Funktionalität aufgebläht werden und ihren Fokus verlieren. Die Verwaltung der Listener ließe sich folgendermaßen deutlich klarer umgestalten:

```
final List<ChangeListener> listeners = new CopyOnWriteArrayList<>();

private void addChangeListeners(final ChangeListener listenerToAdd)
{
    listeners.add(listenerToAdd}
}

private void removeChangeListeners(final ChangeListener listenerToRemove)
{
    listeners.remove(listenerToRemove}
}

private void notifyChangeListeners()
{
```

```
    for (final ChangeListener currentListener : lListeners)
    {
        currentListener.update();
    }
}
```

7.3 Kommunikation von Threads

Bis hierher haben wir Möglichkeiten kennengelernt, Thread-sicher auf gemeinsam benutzte Daten zuzugreifen. In der Zusammenarbeit von Threads benötigt man aber häufig außerdem Möglichkeiten zur Abstimmung zwischen Threads – genauer: zu deren Kommunikation. Ein Beispiel dafür ist das sogenannte ***Producer-Consumer-Problem***. Hierbei geht es darum, dass ein Erzeuger (Producer) gewisse Daten herstellt und ein Konsument (Consumer) diese verbraucht und beide über eine gemeinsame Datenstruktur interagieren, wie es symbolisch in Abbildung 7-5 dargestellt ist.

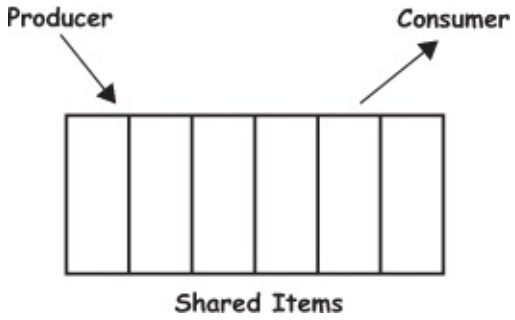


Abbildung 7-5 *Kommunikation von Producer und Consumer*

Im Folgenden stelle ich verschiedene Formen der Kommunikation von `Producer`- und `Consumer`-Klassen vor, damit Sie dabei lauernde Probleme nachvollziehen können.

7.3.1 Kommunikation mit Synchronisation

Als Erstes betrachten wir, wie eine Kommunikation über eine gemeinsame Datenstruktur und den Einsatz von Synchronisation gelöst werden kann. Producer und Consumer implementieren wir als Spezialisierungen von `Runnable`.

Realisierung des Producers

In diesem Beispiel soll der Producer periodisch im Takt einer vorgegebenen Wartezeit `sleepTime` Daten vom Typ `Item` erzeugen und sie in der gemeinsam genutzten Datenstruktur `sharedItems` vom Typ `List<Item>` ablegen. Damit es dabei nicht zu Zugriffskonflikten kommt, muss das Ganze innerhalb eines kritischen Bereichs exklusiv erfolgen, hier über `synchronized (sharedItems)` realisiert:

```
public static class Producer implements Runnable
{
```

```
    private final List<Item> sharedItems;
    private final long       sleepTime;

    public Producer(final List<Item> items, final long sleepTime)
    {
        this.sharedItems = items;
        this.sleepTime = sleepTime;
    }

    public void run()
    {
        int counter = 0;

        while (!Thread.currentThread().isInterrupted())
        {
            // Ereugen eines Items
            final Item newItem = new Item("Item" + counter);
            System.out.println("Producing …" + newItem);
            SleepUtils.safeSleep(sleepTime);

            // Lock akquirieren, dann exklusiv zugreifen und Item hinzufügen
            synchronized (sharedItems)
            {
                sharedItems.add(newItem);
                System.out.println("Produced" + newItem);
            } // Lock wird automatisch freigegeben

            counter++;
        }
    }
}
```

Realisierung des Consumers

Der Consumer schaut zyklisch nach, ob bereits Daten für ihn vorliegen, und liest diese dann aus der Liste `sharedItems`. Ebenso wie der Producer muss auch der Consumer seinen Zugriff exklusiv in einem kritischen Bereich erledigen. Dabei ist zu beachten, dass nur der Zugriff, nicht aber das Warten mit `Thread.sleep(long)` im kritischen Bereich erfolgt, weil der Lock des `sharedItems`-Objekts so nicht freigegeben würde und folglich der Producer niemals auf die gemeinsame Liste zugreifen könnte, um Daten abzulegen. Der Consumer würde endlos und vergeblich warten.

```
public static class Consumer implements Runnable
{
    private final List<Item> sharedItems;
    private final long       sleepTime;

    public Consumer(final List<Item> items, final long sleepTime)
    {
        this.sharedItems = items;
        this.sleepTime = sleepTime;
    }

    public void run()
    {
        while (!Thread.currentThread().isInterrupted())
```

```
        {
            // Status-Flag ist als lokale Variable immer Thread-sicher
            boolean noItems = true;
            while (noItems)
            {
                // Lock akquirieren, dann exklusiv zugreifen und Item auslesen
                synchronized (sharedItems)
                {
                    noItems = (sharedItems.size() == 0);
                    if (noItems)
                       System.out.println("Consumer waiting for items …");
                    else
                       System.out.println("Consuming" +
sharedItems.remove(0));
                } // Lock wird automatisch freigegeben

                // Achtung: sleep() nicht in synchronized aufrufen
                SleepUtils.safeSleep(sleepTime);
            }
        }
    }
}
```

Das hier vom Consumer umgesetzte Verfahren des aktiven Wartens wird auch ***Busy Waiting*** genannt und ist in der Regel zu vermeiden, weil es Rechenzeit kostet und es elegantere Alternativen gibt. Wird allerdings häufig genug `Thread.sleep(long)` aufgerufen, ist der Einsatz nicht ganz so tragisch. Geht die Wartezeit jedoch gegen 0, so erhöht sich die Prozessorlast deutlich.

Beispiel der Kommunikation

Wir schauen nun, wie die beiden Basisbausteine zusammenarbeiten. Zur Kommunikation wird jeweils eine Instanz eines `Producers` und eines `Consumers` erzeugt. Danach legt der Producer jede Sekunde etwas in die gemeinsame Liste und der Consumer schaut alle 500 ms nach, ob etwas zu konsumieren ist:

```
public static void main(final String[] args)
{
    final List<Item> sharedItems = new LinkedList<>();

    new Thread(new Producer(sharedItems, 1000)).start();
    new Thread(new Consumer(sharedItems, 500)).start();
}
```

Listing 7.3 *Ausführbar als* **'PRODUCERCONSUMERSYNCHRONISATIONEXAMPLE'**

Betrachten wir die gekürzte Ausgabe, um das Programm zu analysieren:

```
Producing … [Item] Item 0
Consumer waiting for items …
Consumer waiting for items …
Produced [Item] Item 0
Producing … [Item] Item 1
Consuming [Item] Item 0
Consumer waiting for items …
Consumer waiting for items …
Produced [Item] Item 1
Producing … [Item] Item 2
```

```
Consuming [Item] Item 1
```

Anhand dieser Ausgabe erkennen wir folgende Dinge:

1. Producer und Consumer laufen zwar parallel, aber hier synchron ab und der produzierte Gegenstand wird immer sofort konsumiert, sobald er verfügbar ist.
2. Das vom Consumer genutzte aktive Warten führt zu ständigen Prüfungen, kostet etwas Rechenzeit und sollte möglichst vermieden werden.
3. Es sind auch andere Wartezeiten für Producer und Consumer denkbar. Würde man etwa die Wartezeiten vertauschen, so würden in einer Sekunde zwei Elemente produziert, der Consumer würde aber nur ein Element konsumieren. Dadurch würde der Zwischenspeicher irgendwann überlaufen. Deshalb sollte die Größe beschränkt werden. Der Producer sollte die Arbeit einstellen, wenn der maximale Füllgrad erreicht ist, und ein Consumer nur so lange aktiv werden, wie Elemente vorhanden sind. Dies lässt sich mit aktivem Warten kaum adäquat ausdrücken.

Beginnen wir mit einer alternativen Realisierung, die das aktive Warten adressiert. Später schauen wir dann auf größenbeschränkte Datenstrukturen, die dabei helfen, die im dritten Aufzählungspunkt erwähnten Nachteile zu behandeln.

7.3.2 Kommunikation über die Methoden `wait()`, `notify()` und `notifyAll()`

Die Kommunikation zwischen Producer und Consumer lässt sich statt mit Busy Waiting eleganter durch die im Folgenden vorgestellten Methoden `wait()`, `notify()` und `notifyAll()` lösen. Wie beim Schlüsselwort `synchronized` werden auch hier die Verwaltung des Locks und die Warte- und Aufweckarbeiten automatisch durch die JVM erledigt.

- `wait()` – Versetzt den aktiven Thread in den Zustand `WAITING` und der belegte Lock wird freigegeben. Um ein endloses Warten auf ein möglicherweise nicht eintretendes Ereignis zu verhindern, kann beim Aufruf von `wait()` eine maximale Wartezeit mitgegeben werden: Der Thread wechselt dadurch in den Zustand `TIMED_WAITING`.
- `notify()` – Informiert einen beliebigen wartenden Thread und versetzt diesen in den Zustand `RUNNABLE`. Unschönerweise kann man nicht kontrollieren, welcher Thread benachrichtigt wird. Daher sollte man bei der Kommunikation mehrerer Threads bevorzugt das nachfolgend beschriebene `notifyAll()` verwenden.
- `notifyAll()` – Informiert alle auf den Lock des Objekts wartenden Threads und versetzt diese in den Zustand `RUNNABLE`.

Die genannten Methoden werden nicht auf Threads, sondern auf Objekten aufgerufen. In diesem Fall kontrolliert das Objekt über seinen Lock indirekt den

Thread. Wenn ein Thread *A* während seiner Bearbeitung Daten benötigt oder auf das Eintreten einer speziellen Bedingung warten möchte, so kann er die Methode `wait()` eines Objekts `obj` aufrufen. Ein anderer Thread *B* kann dadurch bei Bedarf den benötigten Lock erhalten, Berechnungen durchführen und Ergebnisse produzieren. Danach informiert er einen oder mehrere auf den Lock dieses Objekts `obj` wartende Threads durch Aufruf der Methode `notify()` oder bevorzugt `notifyAll()` auf der Objektreferenz `obj`. Einer dieser Threads erhält dann den Lock des Objekts `obj`: Dies kann der wartende Thread *A* oder ein beliebiger anderer sein. Was dies im Einzelnen bedeutet, zeigen die folgenden Abschnitte.

Producer-Consumer mit `wait()`, `notify()` und `notifyAll()`

Nachdem wir die Methoden zur Steuerung der Kommunikation von Threads kennengelernt haben, nutzen wir sie, um das Producer-Consumer-Beispiel zu vereinfachen und das Busy Waiting zu vermeiden.

Producer

Damit der im Anschluss realisierte Consumer nach Aufrufen von `wait()` nicht endlos wartet, muss der `Producer` einen Aufruf der Methode `notify()` bzw. `notifyAll()` ausführen, der zum Aufwachen des `Consumers` führt. Die `run()`-Methode im `Producer` wird somit wie folgt modifiziert:

```
public void run()
{
    int counter = 0;

    while (!Thread.currentThread().isInterrupted())
    {
        final Item newItem = new Item("Item" + counter);
        System.out.println("Producing …" + newItem);

        SleepUtils.safeSleep(sleepTime);

        synchronized (sharedItems)
        {
            sharedItems.add(newItem);
            System.out.println("Produced" + newItem);
            // Informiere wartende Threads
            sharedItems.notifyAll();
        }
        counter++;
    }
}
```

Listing 7.4 *Ausführbar als* **'PRODUCERCONSUMEREXAMPLE'**

Hintergrundwissen: Gleichwertige Varianten bei `notify()`

Bei der Kommunikation von Threads über die Methoden `wait()` und `notify()`

bzw. `notifyAll()` sieht man beim Benachrichtigen folgende zwei Varianten:

```
Variante 1                          Variante 2
synchronized (lock)                 synchronized (lock)
{                                   {
     modCount++;                         lock.notify();
     lock.notify();                      modCount++;
}                                   }
```

Obwohl es zunächst nicht intuitiv ist, sind beide Lösungen aber gleichwertig. Auf den ersten Blick könnte man meinen, dass in Variante 2 das Hochzählen nach dem Benachrichtigen eines möglicherweise wartenden Threads geschieht und dieser einen falschen Wert ausliest. Tatsächlich erhält ein wartender Thread den Lock jedoch erst beim Verlassen des `synchronized`-Blocks – das vorzeitige Benachrichtigen ändert durch das Halten des Locks nichts am Ablauf.

Dennoch ist Variante 2 verwirrend und sollte nicht verwendet werden, um erst gar keine Fragen oder Unsicherheiten aufkommen zu lassen.

Consumer

Die `run()`-Methode der `Consumer`-Klasse wird so modifiziert, dass vor dem Zugriff auf die Daten in `sharedItems` ein Aufruf von `wait()` in einem kritischen Bereich ausgeführt wird.

```
public void run()
{
     while (!Thread.currentThread().isInterrupted())
     {
          synchronized (sharedItems)
          {
               try
               {
                    System.out.println("Consumer waiting …");
                    sharedItems.wait();
                    // ACHTUNG: Potenziell unsicherer Zugriff
                    final Item item = sharedItems.remove(0);
                    System.out.println("Consuming" + item);
               }
               catch (final InterruptedException e)
               {
                    Thread.currentThread().interrupt();
               }
          }
          SleepUtils.safeSleep(sleepTime);
     }
}
```

Diese Art der Kommunikation ist klarer und erfordert keine Tricks wie Busy Waiting. Allerdings habe ich die Methode `wait()` bewusst etwas naiv eingesetzt, um auf Probleme beim Aufruf ohne vorherige und/oder nachfolgende Prüfung zeigen zu können.

IDIOM: WARTEN AUF EINE BEDINGUNG

Grundsätzlich sollte ein wartender Thread, nachdem er aufgeweckt wurde, prüfen, ob tatsächlich die erwartete Situation eingetreten ist. Wenn man `wait()` nutzt, so benötigt man sogar eigentlich zwei Prüfungen, nämlich eine vor und eine nach dem Aufruf: Zunächst prüft man für den Fall, dass eine Benachrichtigung vor dem Warten durch Aufruf von `wait()` versendet wird. Nach dem Aufwachen prüft man erneut, um sicherzustellen, dass die erwartete Bedingung immer noch gilt, bevor man dann eine Aktion, nachfolgend `doWork()` ausführt. Das Warten auf produzierte Elemente sollte damit folgendermaßen realisiert werden:

```
if (sharedItems.size() == 0)
    wait();

if (sharedItems.size() > 0)
    doWork();
```

Das Ganze lässt sich eleganter schreiben, doch schauen wir uns zunächst detailliertere Begründungen für die Prüfungen an.

Prüfung vor `wait()`

Vor dem Aufruf von `wait()` sollte die gewünschte Bedingung geprüft werden. Das scheint zunächst unlogisch, doch nehmen wir einmal an, der Producer würde deutlich vor dem Consumer gestartet. Er produziert ein Element, legt es in der gemeinsamen Liste ab und ruft `notifyAll()` auf. Zu dem Zeitpunkt wartet aber noch kein Consumer darauf. Die Benachrichtigung geht ins Leere. Später wird dann `wait()` vom Consumer aufgerufen. Er wartet nun bis zum Sankt Nimmerleinstag auf eine Benachrichtigung, die er nie erhält. Aber auch für den Fall, dass der Producer (versehentlich) keine Benachrichtigung versendet, ist die Prüfung hilfreich. Es können trotzdem Elemente in der Datenstruktur vorhanden sein, die zu konsumieren sind.

Prüfung nach `wait()`

Der Wartezustand durch Aufruf von `wait()` wird nur dann verlassen, wenn eine Benachrichtigung per `notify()` bzw. `notifyAll()` erfolgt. Es können aber auch andere Threads zwischendrin geweckt worden sein und bei ihrer Ausführung eventuell Daten geändert haben. Daher gilt die erwartete Bedingung möglicherweise nicht mehr, aufgrund derer die Benachrichtigung erfolgte.

Ich erweitere das vorherige Beispiel auf mehrere unabhängige Consumer, um auf die Wichtigkeit der nachfolgenden Prüfung und ein ansonsten auftretendes Problem bei Zugriffen auf gemeinsame Daten einzugehen. Dazu schreiben wir folgende `main()`-Methode, die einen Producer mit Sekundentakt und drei Consumer erzeugt, die nach dem Konsumieren 100 ms Pause einlegen:

```
public static void main(final String[] args)
{
    final List<Item> sharedItems = new LinkedList<>();
```

```
    new Thread(new Producer(sharedItems, 1000)).start();
    new Thread(new Consumer(sharedItems, 100, "Consumer 1")).start();
    new Thread(new Consumer(sharedItems, 100, "Consumer 2")).start();
    new Thread(new Consumer(sharedItems, 100, "Consumer 3")).start();
}
```

Listing 7.5 *Ausführbar als* **'PRODUCERMULTICONSUMERWRONG1EXAMPLE'**

Startet man das Programm PRODUCERMULTICONSUMERWRONG1EXAMPLE, treten schnell zwei `IndexOutOfBoundsExceptions` auf. Die Ursache dafür ist, dass durch den Producer und dessen Aufruf von `notifyAll()` alle Consumer aufgeweckt werden. Alle konkurrieren dann um den einen Lock des `sharedItems`-Objekts, den sie beim Aufruf von `wait()` abgegeben haben. Der erste Consumer, der den Lock erhält, führt den Zugriff auf die gemeinsamen Daten mit der Zeile

```
final Item item = sharedItems.remove(0);
```

aus. Die beiden anderen Consumer führen diese Zeile später bei Erhalt des Locks auch aus. Da durch den ersten Aufruf die gespeicherten Daten entnommen wurden, schlägt jeder weitere Aufruf von `sharedItems.remove(0)` mit einer Exception fehl, wodurch zwei der drei Consumer beendet werden und nur noch ein Consumer aktiv ist.

Dieses Beispiel verdeutlicht die Problematik, dass beim Warten und bei Benachrichtigen keine Bedingungen angegeben werden können. Folglich kann auch nicht garantiert werden, dass die Bedingung, auf die mit `wait()` gewartet wurde, nach dem Aufwecken tatsächlich eingetreten ist. Aber selbst wenn die Bedingung, auf die gewartet wurde, eingetreten ist, so kann man nicht sicher sein, dass beim Erhalt des Locks die Bedingung noch gilt. Wenn der Producer `notifyAll()` aufruft, dann sind auf jeden Fall Daten in der Datenstruktur `sharedItems` vorhanden. Dies gilt aber nicht mehr, wenn ein beliebiger Consumer aktiviert wird und vor ihm ein anderer an der Reihe war. Allgemeiner gilt also: Andere Threads können den Lock bereits erhalten und Modifikationen an den Daten durchgeführt haben. Dadurch ist möglicherweise die erwartete Bedingung nicht mehr erfüllt. Zur Abhilfe könnte die erste Lösungsidee sein, die erwartete Bedingung nach dem Aufwachen wie folgt zu prüfen:

```
System.out.println(consumerName + "waiting …");

sharedItems.wait();

if (sharedItems.size() > 0)
    System.out.println(consumerName + " consuming " + sharedItems.remove(0));
else
    System.out.println(consumerName + " - item already consumed by " +
                                      "other consumer! ");

SleepUtils.safeSleep(sleepTime);
```

Das funktioniert zwar prinzipiell. Allerdings werden durch diese Art der Realisierung immer alle Consumer ausgewertet und in der Abarbeitung fortgesetzt. Dadurch muss zusätzlich eine Fehlerbehandlung in den Sourcecode integriert werden, die sich negativ auf die Verständlichkeit auswirkt.

Verbesserung der Prüfung

Um die genannten Probleme zu adressieren, sollte man `wait()` besser innerhalb einer `while`-Schleife ausführen und dort erneut die Bedingung prüfen. Dabei nutzt man folgendes Idiom:

```
while (!condition)
    wait();
```

Doug Lea schlägt in seinem Buch »Concurrent Programming in Java« [59] vor, die Prüfung von Bedingungen in eigene Methoden auszulagern. Dieses Vorgehen ist sehr empfehlenswert und trägt deutlich zur Lesbarkeit bei. Um beispielsweise im Consumer zu prüfen, ob Daten zur Verarbeitung bereitgestellt wurden, kann folgende Methode `waitForItemsAvailable(List<Item>)` wie folgt realisiert werden:

```
private static void waitForItemsAvailable(final List<Item> items) throws
      InterruptedException
{
    while (items.size() == 0)
        items.wait();
}
```

Neben der besseren Lesbarkeit gibt es noch einen triftigen Grund für diese Art der Umsetzung: Erfolgt eine Benachrichtigung mit `notify()`, bevor darauf mit `wait()` gewartet wird, so wird diese Benachrichtigung nicht wahrgenommen und ein Thread wird dann eventuell bis zum Programmende vergeblich auf das Eintreffen einer Benachrichtigung warten. Deshalb muss zusätzlich die Bedingung geprüft werden. Wie gezeigt könnte man das zwar über eine `if`-Abfrage vor dem `wait()` lösen. Eleganter als die eingangs gezeigten zwei Prüfungen über `if`-Abfragen ist aber die gerade vorgestellte Umsetzung der Prüfung in einer Schleife.

Korrektur

Mit dem bis hierher erlangten Verständnis für das Warten mit `wait()` wollen wir die `run()`-Methode des Consumers korrigieren: Dazu nutzen wir die zuvor erstellte Methode `waitForItemsAvailable(List<Item>)` wie folgt:

```
System.out.println(consumerName + "waiting …");

waitForItemsAvailable(sharedItems);
// Nachfolgende Zugriffe auf items durch waitForItemsAvailable() immer sicher
final Item item = sharedItems.remove(0);
System.out.println(consumerName + "consuming" + item);

SleepUtils.safeSleep(sleepTime);
```

Listing 7.6 *Ausführbar als* **'PRODUCERMULTICONSUMEREXAMPLE'**

Als Folge dieser Korrektur warten anfangs alle Consumer auf das Eintreffen einer Benachrichtigung. Es werden zwar alle Consumer geweckt, allerdings erhält nur einer von diesen zunächst den Lock und konsumiert. Erhalten in der Folgezeit die anderen Consumer den Lock, prüfen diese zunächst wieder die Bedingung, so ist diese

eventuell nicht mehr gültig, nämlich dann, wenn der Consumer zuvor alle Elemente konsumiert hat. Dann legen sich die anderen Consumer schlafen und warten sofort wieder auf das Eintreten der Bedingung.

Erweiterung auf eine Größenbeschränkung

Bisher haben wir lediglich den Fall betrachtet, dass die Consumer schneller verbrauchen als der Producer Dinge herstellen kann. Für den Fall, dass der Producer allerdings wesentlich schneller ist, kämen die Consumer nicht mehr hinterher und die gemeinsame Datenstruktur würde immer voller, bildlich gesprochen: Sie würde überlaufen.

Daher bietet es sich zum Datenaustausch an, einen Zwischenspeicher begrenzter Kapazität zu nutzen. Wird dessen Kapazität erreicht, so muss der Producer mit dem Fortsetzen des Produzierens darauf warten, dass der oder die Consumer wieder Platz geschaffen haben. Umgekehrt gilt: Sind die Consumer schneller als der Producer, und gibt es keine zu verarbeitenden Dinge mehr, so sollten die Consumer so lange warten, bis wieder Dinge zu konsumieren sind.

Wir erweitern das Beispiel. Wenig objektorientiert wäre es, die Anforderungen an die Verwaltung gemeinsamer Daten in den Klassen des Producers und Consumers zu realisieren. Stattdessen definieren wir eine typsichere Containerklasse, deren Anforderungen durch folgendes Interface beschrieben sind (dadurch sind wir in der Lage, Realisierungen auszutauschen oder miteinander zu vergleichen):

```
public interface FixedSizeContainer<T>
{
    /**
     * put the passed item into this container, blocks if container reached
     * its capacity (the queue is full)
     */
    void putItem(final T item) throws InterruptedException;

    /**
     * returns the next item from this container, blocks if there are no
     * items available (the queue is empty)
     */
    T takeItem() throws InterruptedException;
}
```

Als erste Realisierung erstellen wir die Klasse `FixedSizeListContainer<T>`, die mit einer Liste arbeitet:

```
public static class FixedSizeListContainer<T> implements FixedSizeContainer<T>
{
    private final List<T> queuedItems;
    private final int     maxSize;

    public FixedSizeListContainer(final int maxSize)
    {
        this.queuedItems = new LinkedList<T>();
        this.maxSize = maxSize;
    }
```

```
    public synchronized void putItem(final T item) throws InterruptedException
    {
        waitWhileQueueFull();
        // Aufwecken von waitWhileQueueEmpty()
        notifyAll();

        queuedItems.add(item);
    }

    public synchronized T takeItem() throws InterruptedException
    {
        waitWhileQueueEmpty();
        // Aufwecken von waitWhileQueueFull()
        notifyAll();

        return queuedItems.remove(0);
    }

    private void waitWhileQueueFull() throws InterruptedException
    {
        while (queuedItems.size() == maxSize)
            wait();
    }

    private void waitWhileQueueEmpty() throws InterruptedException
    {
        while (queuedItems.size() == 0)
            wait();
    }
}
```

Die beiden Methoden `putItem(T)` und `takeItem()` zum Einfügen und zum Abholen müssen synchronisiert sein, um gleichzeitige Zugriffe durch Consumer und Producer konfliktfrei bearbeiten zu können. Wenn es keinen freien Platz mehr für ein neu produziertes Element gibt, wartet der Producer mit `wait()`, bis dies wieder der Fall ist. Ein Consumer ruft beim Abholen `notifyAll()` auf, sodass der Producer informiert wird und prüfen kann, ob wieder Platz vorhanden ist. Beide Wartebedingungen werden, wie zuvor vorgeschlagen, über die Methoden `waitWhileQueueFull()` bzw. `waitWhileQueueEmpty()` sichergestellt.

In folgendem Beispiel erzeugen wir wieder drei Consumer, die von einem Producer versorgt werden. Wenn wir die neue größenbeschränkte Datenstruktur hier mit maximal sieben Elementen zur Kommunikation (Programm PRODUCERMULTICONSUMERSIZEDEXAMPLE) nutzen, so erhält man tatsächlich genau die gleiche Ausgabe wie zuvor, weil immer jeweils das produzierte Element direkt konsumiert wird. Wir drehen nun etwas an den Stellschrauben und lassen den Producer schneller arbeiten. Damit wir die Größenbeschränkung nachvollziehen können, wird hier zudem eine künstliche Pause von zwei Sekunden nach Erzeugung des Producers eingelegt:

```
public static void main(final String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final FixedSizeContainer<Item> sharedItems =
                                    new FixedSizeListContainer<>
```

```
(MAX_QUEUE_SIZE);

    new Thread(new Producer(sharedItems, 100)).start();

    // Warte 2 Sekunden, dadurch sieht man die Größenbeschränkung
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    new Thread(new Consumer(sharedItems, 1000, "Consumer 1")).start();
    new Thread(new Consumer(sharedItems, 1000, "Consumer 2")).start();
    new Thread(new Consumer(sharedItems, 1000, "Consumer 3")).start();
}
```

***Listing 7.7** Ausführbar als* **'PRODUCERMULTICONSUMERSIZEDEXAMPLE2'**

Wenn man das Programm PRODUCERMULTICONSUMERSIZEDEXAMPLE2 startet, so erkennt man, dass zunächst sieben Elemente produziert werden. Der Producer wartet dann, dass einige davon konsumiert werden, bis erneut immer bis zu einem Füllgrad von sieben Elemente nachproduziert werden, wie es folgender Ausschnitt der Programmausgaben andeutet:

```
Producing … [Item] Item 6
Produced [Item] Item 6
Producing … [Item] Item 7
Consumer 1 waiting …
Consumer 1 consuming [Item] Item 0
Consumer 2 waiting …
Consumer 2 consuming [Item] Item 1
Produced [Item] Item 7
Producing … [Item] Item 8
Consumer 3 waiting …
Consumer 3 consuming [Item] Item 2
Produced [Item] Item 8
```

Bedingungen und das Interface `Condition`

Wir haben zum Ersatz von `synchronized` bereits das Interface `Lock` kennengelernt. Dieses erlaubt durch den Einsatz von `Condition`-Objekten, Bedingungen zu formulieren. Die zuvor vorgestellte größenbeschränkte Containerklasse schreiben wir derart um, dass die `synchronized`-Blöcke durch Aufrufe von `lock()` und `unlock()` realisiert werden. Bislang wurden Bedingungen ausschließlich anhand von Abfragen an die Daten speichernde Liste ermittelt. Wir nutzen in diesem Beispiel zusätzlich zwei `Condition`-Objekte `notFull` bzw. `notEmpty`. Über deren Methoden `await()` und `signal()` bzw. `signalAll()` kann man das Warten auf sowie das Eintreten von Bedingungen klarer formulieren als lediglich über Aufrufe der Objektmethoden `wait()`, `notify()` und `notifyAll()`:

```
public final class BlockingFixedSizeBuffer<T> implements FixedSizeContainer<T>
{
    private final Lock      lock     = new ReentrantLock();
    private final Condition notFull  = lock.newCondition();
    private final Condition notEmpty = lock.newCondition();

    private final List<T>   queuedItems;
    private final int       maxSize;
```

```
    public BlockingFixedSizeBuffer(final int maxSize)
    {
        this.queuedItems = new LinkedList<T>();
        this.maxSize = maxSize;
    }

    public void putItem(final T item) throws InterruptedException
    {
        lock.lock();
        try
        {
            waitWhileQueueFull();
            notEmpty.signal();

            queuedItems.add(item);
        }
        finally
        {
            lock.unlock();
        }
     }

    public T takeItem() throws InterruptedException
    {
        lock.lock();
        try
        {
            waitWhileQueueEmpty();
            notFull.signal();

            return queuedItems.remove(0);
        }
        finally
        {
            lock.unlock();
        }
    }

    private void waitWhileQueueFull() throws InterruptedException
    {
        while (queuedItems.size() == maxSize)
            notFull.await();
    }

    private void waitWhileQueueEmpty() throws InterruptedException
    {
        while (queuedItems.size() == 0)
            notEmpty.await();
    }
}
```

Auch wenn diese Umsetzung durch Verwendung von `Condition`-Objekten eine deutliche Verbesserung zu den vorherigen Versionen darstellt, so ist sie doch ziemlich unübersichtlich und umständlich, da die Implementierung auf einem relativ tiefen Abstraktionsniveau mit vielen Details geschieht. Beim Einsatz von Multithreading sind gemeinsam genutzte, größenbeschränkte Datenstrukturen elementar, um die Kommunikation verschiedener Threads zu steuern. Praktischerweise gibt es seit JDK 5 entsprechende Containerklassen. In Abschnitt 7.6.1 betrachten wir exemplarisch das

Interface `java.util.concurrent.BlockingQueue<E>` und einige Realisierungen, die in ihrer Funktionalität ähnlich zu der Klasse `BlockingFixedSizeBuffer<T>` sind.

Kommen wir zur eben entwickelten Klasse `BlockingFixedSizeBuffer<T>` zurück und schauen uns deren Einsatz im Programm PRODUCERMULTICONSUMERSIZEDEXAMPLE3 an. Führt man es aus, so sieht man, dass diese Realisierung vollständig kompatibel zur Klasse `FixedSizeListContainer<T>` ist.

```
public static void main(final String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final FixedSizeContainer<Item> sharedItems =
                                    new BlockingFixedSizeBuffer<>
(MAX_QUEUE_SIZE);

    new Thread(new Producer(sharedItems, 100)).start();

    // Warte 2 Sekunden, dadurch sieht man die Größenbeschränkung
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    new Thread(new Consumer(sharedItems, 1000, "Consumer 1")).start();
    new Thread(new Consumer(sharedItems, 1000, "Consumer 2")).start();
    new Thread(new Consumer(sharedItems, 1000, "Consumer 3")).start();
}
```

Listing 7.8 *Ausführbar als* **'PRODUCERMULTICONSUMERSIZEDEXAMPLE3'**

7.3.3 Abstimmung von Threads

Bei der Zusammenarbeit mehrerer Threads müssen mitunter Aufgaben in parallele Teile aufgespalten und später an Synchronisationspunkten wieder zusammengeführt werden. Dies ist mithilfe der Methode `join()` der Klasse `Thread` möglich.

Die Methoden `isAlive()` und `join()`

Bekanntermaßen kann man mit der Methode `isAlive()` ein `Thread`-Objekt fragen, ob dieses durch einen Aufruf an `start()` als neuer Ausführungspfad gestartet wurde und die Methode `run()` abgearbeitet wird. Vor und nach der Abarbeitung von `run()` stellt ein Thread nur ein ganz normales Objekt dar. Ein Aufruf von `isAlive()` liefert dann den Wert `false`.

Startet man Threads zur Erledigung bestimmter Aufgaben und möchte man auf deren Beendigung und die Berechnungsergebnisse warten, so ist dies durch Aufruf der Methode `join()` möglich. Repräsentiert beispielsweise die Referenzvariable `workerThread` einen Thread, so wartet der momentan aktive Thread durch den Aufruf

```
workerThread.join();
```

synchron, d. h. blockierend, auf die Beendigung des Threads `workerThread` und

wird erst danach fortgesetzt. Um nicht endlos zu warten, falls der andere Thread niemals endet, kann eine Time-out-Zeit angegeben werden, nach der das Warten auf das Ende des Threads abgebrochen wird.

Verdeutlichen wir uns die Arbeitsweise der beiden Methoden durch folgendes Programm PRODUCERJOINEXAMPLE:

```
public static void main(final String[] args)
{
    final List<Item> items = new LinkedList<>();
    final Thread producerThread = new Thread(new Producer(items, 1000));
    producerThread.start();

    try
    {
        // Aktueller Thread wird für 5 Sekunden angehalten
        producerThread.join(TimeUnit.SECONDS.toMillis(5));
        System.out.println("after join(): producer is alive?" +
                                          producerThread.isAlive());
        // 1000 ms Produktionszeit und 5000 ms Wartezeit => ca. 5 Items
        System.out.println("Item-Count after join(): " + items.size());
        // Der Producer arbeitet noch 2 Sekunden weiter …
        SleepUtils.safeSleep(TimeUnit.SECONDS, 2);
    }
    catch (final InterruptedException e)
    {
        Thread.currentThread().interrupt();
    }

    // Der Producer wird aufgefordert, nun anzuhalten …
    producerThread.interrupt();

    // 1000 ms Produktionszeit und 7 s Wartezeit => ca. 7 Items
    System.out.println("Item-Count after interrupt(): " + items.size());
    System.out.println("after interrupt(): producer is alive? " +
                                           producerThread.isAlive());
}
```

Listing 7.9 *Ausführbar als* **'PRODUCERJOINEXAMPLE'**

In diesem Beispiel wartet der aktuelle Thread (hier: der `main`-Thread) durch Aufruf von `join()` fünf Sekunden auf das Ende des `Producer`-Threads. Dieser ist dann aber noch nicht terminiert. Die Ausgabe der Anzahl der produzierten Elemente stellt daher nur einen Zwischenstand dar. Der `Producer`-Thread setzt seine Arbeit fort und wird nach weiteren zwei Sekunden über einen Aufruf von `interrupt()` zum Anhalten aufgefordert und, weil er darauf adäquat reagiert, auch beendet.

Starten wir das Programm PRODUCERJOINEXAMPLE, so kommt es zu folgender Ausgabe (hier auf das Wesentliche gekürzt), die das zuvor Gesagte bestätigen:

```
[ …]
Producing … [Item] Item 4
Produced [Item] Item 4
Producing … [Item] Item 5
after join(): producer is alive? true
Item-Count after join(): 5
Produced [Item] Item 5
Producing … [Item] Item 6
Produced [Item] Item 6
```

```
Item-Count after interrupt(): 7
after interrupt(): producer is alive? false
```

Achtung: Verwirrende Syntax von `join()`

Die Schreibweise beim Aufruf von `join()` ist verwirrend, aber trotzdem korrekt. Besser verständlich und objektorientierter wäre in etwa folgende Schreibweise gewesen: `currentThread.waitForTermination(workerThread);`

Besonderheiten bei `join()`

Zwei Dinge sind zu beachten: Zum einen sollte ein Aufruf von `join()` nur erfolgen, wenn der Thread, auf den gewartet werden soll, bereits gestartet wurde. Ansonsten würde dadurch nicht gewartet. Zum anderen kehrt ein Aufruf von `join()` sofort zurück, falls der zu überprüfende Thread bereits beendet ist. Der aufrufende Thread wird augenblicklich fortgesetzt.

Kommunikation mehrerer Threads mit `join()`

Zum Teil soll ein Thread auf das Ende mehrerer Threads warten. Aufgrund der Tatsache, dass `join()` sofort zurückkehrt, wenn ein zu überprüfender Thread beendet ist, kann man einfach mehrere `join()`-Aufrufe in beliebiger Reihenfolge hintereinander ausführen. Die Reihenfolge ist unbedeutend, denn die maximale Wartezeit ist sowieso durch den Thread mit der längsten Ausführungszeit bestimmt.

Die Abstimmung der Ablaufreihenfolge mehrerer Threads lässt sich zwar über Aufrufe von `join()` realisieren, etwa indem abhängige Threads erst nach den Threads gestartet werden, auf deren Beendigung sie warten sollen. Sinnvoller ist es aber, dafür spezielle Klassen zu verwenden, etwa einen Semaphor.

Kommunikation über einen Semaphor

Oftmals ist bei Multithreading eine begrenzte Anzahl an Ressourcen auf eine größere Anzahl parallel arbeitender Threads zu verteilen. Eine Möglichkeit, diese Funktionalität bereitzustellen, besteht darin, einen sogenannten ***Semaphor*** zu nutzen. Dieser verwaltet einen Zähler, der mit der Anzahl zur Verfügung stehender Ressourcen initialisiert wird und die momentan verfügbare Anzahl an Ressourcen beschreibt. Benötigt ein Thread Zugriff auf eine Ressource, so befragt er den Semaphor und ruft dazu die Methode `acquire()` auf, die den Zähler um eins reduziert, sofern noch Ressourcen verfügbar sind. Ist dies nicht der Fall, so werden diese und alle folgenden Anfragen so lange blockiert, bis wieder mindestens eine Ressource bereitgestellt werden kann. Nach der Bearbeitung sollte ein Thread durch Aufruf der Methode `release()` seine Ressource wieder freigeben, wodurch der Zähler um eins erhöht wird. Die zuvor beschriebenen Ideen kann man als Klasse `SimpleSemaphore` wie folgt implementieren:

```
public final class SimpleSemaphore
{
    private int count;

    public SimpleSemaphore(final int n)
    {
        this.count = n;
    }

    public synchronized void acquire() throws InterruptedException
    {
        waitWhileNoResources();
        count–;
    }

    public synchronized void release()
    {
        count++;
        notifyAll();
    }

    private void waitWhileNoResources() throws InterruptedException
    {
        while (count == 0)
            wait();
    }
}
```

Eine funktionale Erweiterung dieser einfachen Implementierung stellt die Klasse `java.util.concurrent.Semaphore` aus den Concurrency Utilities (vgl. Abschnitt 7.6) dar. Dort finden sich auch Klassen wie `CyclicBarrier`, `CountDownLatch` usw. Diese liegen nicht im Fokus dieses Buchs und werden u. a. im Buch »Java Concurrency in Practice« von Brian Goetz [31] behandelt. Eine kurze informelle Einführung in deren Arbeitsweise gibt jedoch der folgende Praxistipp.

Tipp: Weitere Synchronizer des Packages `java.util.concurrent`

Barrieren Müssen sich mehrere Threads abstimmen, so kann man dies durch sogenannte Barrieren realisieren. Diese kann man sich wie Treffpunkte bei einer Radtour vorstellen, die in verschiedene Etappen eingeteilt ist. Jedes Etappenziel besitzt einen Sammelpunkt. Verlieren sich Tourteilnehmer, so treffen sie sich alle wieder an diesen Sammelpunkten und starten gemeinsam von dort die neue Etappe. Mit der Klasse `CyclicBarrier` kann man derartige Treffpunkte mit der zum Weiterfahren erforderlichen Anzahl von Teilnehmern definieren.

Latches Latches lassen sich mit folgender Analogie beschreiben: Bei einem Marathon treffen sich Teilnehmer am Start und warten dort auf den Startschuss, der nach einem Countdown erfolgt. Mit der Klasse `CountDownLatch` kann man dieses Verhalten nachbilden: Ein oder mehrere Threads können sich durch Aufruf einer `await()`-Methode in einen Wartezustand versetzen. Bei der Konstruktion eines `CountDownLatch`-Objekts gibt man die Anzahl der Schritte bis zum Startschuss an. Durch Aufruf der Methode `countDown()` wird der verwendete Zähler

schrittweise bis auf den Wert null heruntergezählt. Dann erfolgt der Startschuss und alle darauf wartenden Threads können weiterarbeiten.

Exchanger Zum Teil besteht die Kommunikation zweier Threads nur daraus, Daten miteinander auszutauschen. Erst wenn beide Threads am Austauschpunkt »anwesend« sind, kann der Tausch stattfinden. Jeder Thread bietet jeweils ein Element an und nutzt dazu die Klasse `Exchanger<V>` und die Methode `exchange()`. Haben beide Threads ihre Elemente angeboten, so erhalten sie jeweils das Gegenstück des anderen Threads als Tauschobjekt. Erscheint (zunächst) nur ein Thread am Austauschpunkt, wartet dieser, bis der andere Thread dort auftaucht und `exchange()` aufruft. Um ein endloses Warten zu verhindern, kann eine maximale Wartezeit spezifiziert werden.

7.3.4 Unerwartete IllegalMonitorStateExceptionS

Beim Einsatz von Multithreading und der Kommunikation von Threads treten zum Teil unerwartet `java.lang.IllegalMonitorStateExceptions` auf. Manche Entwickler sind dann ratlos. Wie kann es dazu kommen?

Eine solche Exception wird dadurch ausgelöst, dass der Thread, der eine der Methoden `wait()`, `notify()` bzw. `notifyAll()` aufruft, zu der Zeit nicht den Lock des zugehörigen Objekts besitzt. Diese Tatsache kann zur Kompilierzeit nicht geprüft werden und führt erst zur Laufzeit zu der genannten Exception.

Ein kurzes Beispiel hilft dabei, diesen Sachverhalt zu verstehen. In nachfolgendem Listing wird über das Objekt `lock` synchronisiert und auch der Aufruf von `notifyAll()` geschieht augenscheinlich auf diesem Objekt:

```
static Integer lock = new Integer(1);

public static void main(final String[] args)
{
    synchronized (lock)
    {
        System.out.println("lock is" + lock);
        lock++;
        System.out.println("lock is" + lock);
        lock.notifyAll();
    }
    System.out.println("notWorking");
}
```

Listing 7.10 *Ausführbar als* **'ILLEGALMONITORSTATEEXAMPLE'**

Beim Ausführen des Programms ILLEGALMONITORSTATEEXAMPLE kommt es zu einer `IllegalMonitorStateException` und man erhält in etwa folgende Ausgabe:

```
lock is 1
lock is 2
Exception in thread "main" java.lang.IllegalMonitorStateException
    at java.lang.Object.notifyAll(Native Method)
    at
multithreading.IllegalMonitorStateExample.main(IllegalMonitorStateExample
        .java:14)
```

Tritt eine solche `IllegalMonitorStateException` auf, kann man folgendermaßen vorgehen, um die Ursache zu finden:

1. Prüfe, ob die Aufrufe an `wait()`, `notify()` bzw. `notifyAll()` innerhalb eines `synchronized`-Blocks ausgeführt werden.

   (a) Ist dies der Fall, weiter mit Punkt 2.

   (b) Erfolgt ein Aufruf ohne `synchronized`, so muss die Aufrufhierarchie der Methode verfolgt werden. Falls dort ein `synchronized` gefunden wird,

geht es weiter mit Punkt 2. Ansonsten ist an geeigneter Stelle eine Synchronisierung einzufügen.

2. Prüfe, ob das korrekte Objekt zum Synchronisieren verwendet wird.

Für das Beispiel ist der erste Punkt offensichtlich gegeben. Anschließend prüft man, ob auch das korrekte Synchronisationsobjekt verwendet wird. Auf den ersten Blick scheint dies ebenfalls zuzutreffen. Doch der Operator '++' und das Auto-Boxing führen hier dazu, dass wir nicht auf derselben Instanz des `Integer`-Objekts `lock` synchronisieren, auf der wir auch `notifyAll()` aufrufen. Somit kommt es zur `IllegalMonitorStateException`.

Tipp: Regeln zu Synchronisationsobjekten

Beim Einsatz von Synchronisationsobjekten helfen folgende Hinweise:

1. Verwende als Synchronisationsobjekt möglichst die Basisklasse `Object`.
2. Synchronisiere immer auf unveränderlichen Referenzen.
3. Vermeide den Einsatz von Objekten vom Typ `Integer`, `Long` und `String` als Synchronisationsobjekte. Das verhindert Effekte durch Auto-Boxing oder Stringmanipulationen.

`Lock`-Klassen und `IllegalMonitorStateException`s

Beim Einsatz der bereits vorgestellten `Lock`-Klassen kann es gleichermaßen zu `IllegalMonitorStateException`s kommen. Man muss dann sicherstellen, dass `await()`, `signal()` und `signalAll()` innerhalb eines durch einen Lock geschützten Bereichs aufgerufen werden. Zudem muss man dann prüfen, ob das korrekte `Condition`-Objekt des `Lock`-Objekts verwendet wird.

7.4 Das Java-Memory-Modell

Bis hierher haben wir bereits einige Fallstricke beim Zugriff auf gemeinsam verwendete Datenstrukturen beim Einsatz von Multithreading kennengelernt. Das Verständnis des Java-Memory-Modells (JMM) hilft, verlässlichere Multithreading-Applikationen zu schreiben. Es legt fest, wie Programme, im Speziellen Threads, Daten in den Hauptspeicher schreiben und wieder daraus lesen. Im Folgenden gehe ich auf die wichtigsten Punkte ein, die Kapitel 17 der JLS [34] im Detail dargestellt sind. Ich versuche, diese hier möglichst anschaulich und weniger theoretisch als in der JLS zu vermitteln.

Das JMM regelt die Ausführungsreihenfolge und Unterbrechbarkeit von Operationen sowie den Zugriff auf den Speicher und bestimmt damit die Sichtbarkeit von Wertänderungen gemeinsamer Variablen verschiedener Threads. Dabei müssen drei Dinge beachtet werden:

1. **Sichtbarkeit** – Variablen können in Thread-lokalen Caches zwischengespeichert werden, wodurch ihre aktuellen Werte für andere Threads nicht sichtbar sind. Diese Speicherung erfolgt jedoch nicht immer für alle Variablen.
2. **Atomarität** – Lese- und Schreibzugriffe auf Variablen werden für die 64-BitDatentypen `long` und `double` ***nicht atomar*** in einem »Rutsch« ausgeführt, sondern durch Abarbeitung mehrerer Bytecode-Anweisungen, die daher unterbrochen werden können.
3. **Reorderings** – Befehle werden gegebenenfalls in einer von der statischen Reihenfolge im Sourcecode abweichenden Reihenfolge ausgeführt. Dies ist immer dann der Fall, wenn der Compiler einige Optimierungen durchgeführt hat.

7.4.1 Sichtbarkeit

Konzeptionell laufen alle Threads einer JVM parallel und greifen dabei auf gemeinsam verwendete Variablen im Hauptspeicher zu. Aufgrund der Zwischenspeicherung von Werten in einem eigenen Cache eines Threads ist nicht in jedem Fall eine konsistente Sicht aller Threads auf diese Variablen gegeben. Findet kein Abgleich der gecachten Daten mit dem Hauptspeicher statt, so sind dadurch Änderungen eines Threads an einer Variablen für andere Threads nicht sichtbar.[7]

Das JMM garantiert einen solchen Abgleich von Daten mit dem Hauptspeicher lediglich zu folgenden Zeitpunkten:

- **Thread-Start** – Beim Start eines Threads erfolgt das initiale Einlesen der verwendeten Variablen aus dem Hauptspeicher, d. h., der lokale Cache eines Threads wird mit deren Werten belegt.
- **Thread-Ende** – Erst beim Ende der Ausführung eines Threads erfolgt in jedem Fall ein Abgleich des Cache mit dem Hauptspeicher. Dieser Vorgang ist zwingend notwendig, damit modifizierte Daten für andere Threads sichtbar werden.

Während der Abarbeitung eines Threads erfolgen in der Regel keine Hauptspeicherzugriffe mehr, sondern es wird mit den Daten des Cache gearbeitet. Der Thread »lebt isoliert« in seiner eigenen Welt. Ein expliziter Abgleich von Daten kann über die Definition eines kritischen Bereichs mit `synchronized` sowie über das Schlüsselwort `volatile` durchgeführt werden. Die Verwendung von `volatile` garantiert, dass sowohl der lesende als auch der schreibende Datenzugriff direkt auf dem Hauptspeicher erfolgen. Dadurch ist garantiert, dass ein `volatile`-Attribut nie einen veralteten Wert aufweist. Beim Einsatz muss man allerdings beachten, dass die im Folgenden beschriebene Atomarität sichergestellt wird.

7.4.2 Atomarität

Wie bereits erwähnt, erfolgt ein Zugriff auf Variablen der 64-Bit-Datentypen `long` und `double` nicht atomar. Für Multithreading ist dies insbesondere für nicht gecachte Variablen zu beachten, da folgendes Szenario entstehen kann: Nachdem die ersten 32 Bit zugewiesen wurden, kann ein anderer Thread aktiviert werden und »sieht« dann möglicherweise einen ungültigen Zwischenzustand der Variablen, hier am Beispiel der Zuweisung eines Werts an die Variable `val` und den Zeitpunkten *t*1 und *t*2 illustriert:

```
   long val = 0x6789ABCD;

t1: long val = 0x0000ABCD;
t2: long val = 0x6789ABCD:
```

Will man den Zugriff atomar ausführen, kann man das Schlüsselwort `volatile` nutzen. Damit ist zudem die Sichtbarkeit in anderen Threads gegeben. Man könnte daher auf die Idee kommen, nur noch `volatile`-Attribute zum Datenaustausch bei Multithreading einzusetzen. Zur Vermeidung von Synchronisation könnte man beispielsweise die `increment()` -Methode eines Zählers wie folgt schreiben:

```
private volatile long counter = 0;

public void increment()
{
    // Achtung: ++ ist nicht atomar
    counter++;
}
```

Warum ist das weder sinnvoll noch ausreichend? Die Antwort ist überraschend einfach: Über `volatile` wird *kein* kritischer Abschnitt definiert, sondern lediglich *ein* Schreiboder Lesevorgang atomar ausgeführt. Bereits das Post-Increment `counter++` besteht tatsächlich aus folgenden Einzelschritten (vgl. Abschnitt 3.1.2):

```
final long temp = counter;
counter = counter + 1;
return temp;
```

Somit ist es möglich, dass andere Threads zu einem beliebigen Zeitpunkt der Ausführung der obigen Anweisungsfolge aktiviert werden. Dadurch ist keine Atomarität und keine Thread-Sicherheit mehr gegeben. Zum Schutz vor Race Conditions und den im nächsten Abschnitt detailliert vorgestellten möglichen Folgen von Reorderings muss daher jede Folge von Anweisungen, die exklusiv durch einen Thread ausgeführt werden soll, als kritischer Abschnitt geschützt werden.

Atomare Variablen als Lösung?

Mit JDK 5 wurden die Klassen `AtomicInteger` und `AtomicLong` eingeführt, die atomare Read-Modify-Write-Sequenzen, wie `counter++`, ermöglichen – für Gleitkommatypen gibt es keine Unterstützung. Die Atomic-Klassen nutzen eine sogenannte ***Compare-and-Swap-Operation*** (CAS). Die Besonderheit ist, dass diese zunächst einen Wert aus dem Speicher lesen und mit einem erwarteten Wert vergleichen, bevor sie eine Zuweisung mit einem übergebenen Wert durchführen.

Dies geschieht allerdings nur, wenn ein erwarteter Wert gespeichert ist. Übertragen auf das vorherige Beispiel des Zählers würde dies unter Verwendung der Klasse `AtomicLong` wie folgt aussehen:

```
public long incrementAndGetUsingCAS()
{
    long oldValue = counter.get();
    // Unterbrechung möglich
    while (!atomicLongCounter.compareAndSet(oldValue, oldValue + 1))
    {
        // Unterbrechung möglich
        oldValue = atomicLongCounter.get();
        // Unterbrechung möglich
    }
    return oldValue + 1;
}
```

Wie man sieht, ist selbst das einfache Hochzählen eines Werts deutlich komplizierter als ein Einsatz von `synchronized`. Das liegt vor allem daran, dass in einer Schleife geprüft werden muss, ob das Setzen des neuen Werts korrekt erfolgt ist. Diese Komplexität entsteht dadurch, dass zwischen dem Lesen des Werts und dem Setzen ein anderer Thread den Wert verändert haben könnte. In einem solchen Fall wird die Schleife so lange wiederholt, bis eine Inkrementierung erfolgreich ist.

Auf einer solch niedrigen Abstraktionsebene möchte man in der Regel nicht arbeiten. Zur Vereinfachung werden die Details durch verschiedene Methoden der atomaren Klassen gekapselt: Ein sicheres Inkrementieren erfolgt beispielsweise mit der Methode `incrementAndGet()`. Weitere Details zu den atomaren Klassen finden Sie in den Büchern »Java Concurrency in Practice« von Brian Goetz [31] und »Java Threads« von Scott Oaks und Henry Wong [71].

Wie bei `volatile` gilt, dass sobald mehrere Attribute konsistent zueinander geändert werden sollen, eine Definition eines kritischen Abschnitts über `synchronized` oder Locks zwingend notwendig wird.

7.4.3 Reorderings

Die JVM darf gemäß der JLS zur Optimierung beliebige Anweisungen in ihrer Ausführungsreihenfolge umordnen, wenn dabei die Semantik des Programms nicht verändert wird. Bei der Entwicklung von Singlethreading-Anwendungen muss man den sogenannten ***Reorderings*** keine Beachtung schenken. Für Multithreading gibt es jedoch einige Dinge zu berücksichtigen. Betrachten wir zur Verdeutlichung eine Klasse `ReorderingExample` mit den Attributen `x1` und `x2` und folgenden Anweisungen:

```
public class ReorderingExample
{
    int x1 = 0;
    int x2 = 0;

    void method()
```

```
    {
        // Thread 1
        x1 = 1;                                                    // #1
        x2 = 2;                                                    // #2
        System.out.println( "x1 = " + x1 + " / x2 = " + x2 ); // #3
    }
}
```

Werden die obigen Anweisungen ausgeführt, so kann der Compiler die Anweisungen 1 und 2 der Methode `method()` vertauschen, ohne dass dies Auswirkungen auf die nachfolgende Anweisung 3, hier die Ausgabe, hat. Anweisung 3 kann jedoch nicht mit Anweisung 1 oder 2 getauscht werden, da die Ausgabe lesend auf die zuvor geschriebenen Variablen zugreift. Reorderings sind demnach nur dann erlaubt, wenn sichergestellt werden kann, dass es keine Änderungen an den Wertzuweisungen der sequenziellen Abarbeitung der Befehle gibt. Es kommt also immer zu der folgenden Ausgabe: '`x1 = 1 / x2 = 2`'.

Für Multithreading wird die Situation komplizierter: Würde parallel zu den obigen Anweisungen in einem separaten Thread etwa folgende Methode `otherMethod()` dieser Klasse abgearbeitet, so kann es zu unerwarteten Ergebnissen kommen.

```
void otherMethod()
{
    // Thread 2
    int y1 = x2;                                                  // #1
    int y2 = x1;                                                  // #2
    System.out.println( "x1 = " + x1 + " / x2 = " + x2 );  // #3
    System.out.println( "y1 = " + y1 + " / y2 = " + y2 );  // #4
}
```

Betrachten wir zunächst den einfachen Fall ohne Reorderings. Wird Thread 1 vor Thread 2 ausgeführt, also `method()` komplett vor `otherMethod()` abgearbeitet, dann gilt offensichtlich $x1 = 1 = y2$ und $x2 = 2 = y1$. Dies ergibt sich daraus, dass alle Schreiboperationen in Thread 1 bereits ausgeführt wurden, bevor die Leseoperationen in Thread 2 durchgeführt werden. Werden die beiden Threads allerdings abwechselnd, beliebig ineinander verwoben, ausgeführt, so kann es zu merkwürdigen Ausgaben kommen. Denkbar ist etwa eine Situation, in der zwar wie vermutet $x1 = 1$ und $x2 = 2$ gilt, aber auch $y1 = 0$ und $y2 = 1$. Dies ist der Fall, wenn zunächst die Ausführung von Thread 1 (`method()`) begonnen und nach der Zuweisung $x1 = 1$ unterbrochen wird und dann Thread 2 ausgeführt wird, wie dies im Folgenden beispielhaft dargestellt ist:

```
int x1 = 0;
int x2 = 0;
// Thread 1
                                    // Thread 2
x1 = 1;
                                    y1 = x2 = 0;
                                    y2 = x1 = 1;
                                    // System.out: x1 = 1 / x2 = 0
                                    // System.out: y1 = 0 / y2 = 1
x2 = 2;
// System.out: x1 = 1 / x2 = 2
```

Wie man leicht sieht, kann es bereits beim abwechselnden Ausführen (***Interleaving***) von Threads zu Inkonsistenzen kommen. War obige Ausgabe noch mit etwas Nachdenken intuitiv verständlich, ist jedoch auch folgende, unerwartete Ausführungsreihenfolge möglich, wenn die Anweisungen umgeordnet werden:

```
int x1 = 0;
int x2 = 0;
// Thread 1
                                  // Thread 2
                                  y1 = x2 = 0;
x2 = 2;
                                  y2 = x1 = 0;
                                  // System.out: x1 = 0 / x2 = 2
                                  // System.out: y1 = 0 / y2 = 0
x1 = 1;
// System.out: x1 = 1 / x2 = 2
```

Man erkennt einige Besonderheiten beim Zusammenspiel von Threads und gemeinsamen Variablen: Es kann zu Inkonsistenzen durch Reorderings und Interleaving kommen. Für Singlethreading analysiert und ordnet die JVM die Folge von Schreib- und Lesezugriffen automatisch, sodass diese Probleme gar nicht erst entstehen können. Bei Multithreading ist die zeitliche Reihenfolge der Abarbeitung nicht im Vorhinein bekannt, sodass keine definierte Reihenfolge bezüglich des Schreibens und Lesens durch verschiedene Threads gegeben ist und die JVM keine Konsistenz sicherstellen kann. ***Die Folge ist, dass ein Programm zufällig (häufig sogar korrekte) Resultate liefert, aber im schlimmsten Fall unbrauchbar wird***. Bei Multithreading ist es daher Aufgabe des Entwicklers, der JVM Hinweise zu geben, in welcher Reihenfolge die Abarbeitungen erfolgen sollen bzw. welche Bereiche kritisch sind und zu welchen Zeitpunkten keine Reorderings stattfinden dürfen.

Für Multithreading ist dazu eine spezielle Ordnung ***Happens-before*** (*hb*) definiert, die für zwei Anweisungen *A* und *B* beliebiger Threads besagt, dass für *B* alle Änderungen von Variablen einer Anweisung *A* sichtbar sind, wenn *hb*(*A*, *B*) gilt. Durch diese Ordnung wird indirekt festgelegt, in welchen Rahmen Reorderings stattfinden dürfen, da durch *hb* (*A*, *B*) »Abstimmungspunkte« oder »Synchronisationspunkte« von Multithreading-Applikationen definiert werden. Zwischen diesen Abstimmungspunkten sind Reorderings allerdings möglich. An solchen Abstimmungspunkten im Programm weiß man dann jedoch sicher, dass alle Änderungen stattgefunden haben und diese für andere Threads sichtbar sind. Beispielsweise kann über Synchronisation eine gewisse Abarbeitungsreihenfolge erreicht werden: Zwei über dasselbe Lock synchronisierte kritische Abschnitte schließen sich gegenseitig aus. Laut *hb* (*A*, *B*) gilt, dass nachdem einer von beiden abgearbeitet wurde, alle Modifikationen für den nachfolgenden sichtbar sind. Über die Reihenfolge der Ausführung innerhalb eines `synchronized`-Blocks kann allerdings keine Aussage getroffen werden. ***Besteht die Happens-before-Ordnung zwischen zwei Anweisungen jedoch nicht, so darf die JVM den Ablauf von Befehlen beliebig umordnen***.

Hintergrundinformationen zur Ordnung *hb*

Zum besseren Verständnis des Ablaufs bei Multithreading ist es hilfreich zu wissen, für welche Anweisungen eine Happens-before-Ordnung gilt:

- Zwei Anweisungen *A* und *B* erfüllen *hb* (*A*, *B*), wenn *B nach A* im selben Thread in der sogenannten Program Order steht. Vereinfacht bedeutet dies, dass jeder Schreibzugriff für folgende Lesezugriffe auf ein Attribut sichtbar ist. Erfolgen Lese- und Schreibzugriffe nicht auf gleiche Attribute, stehen Anweisungen also nicht in einer Lese-Schreib-Beziehung, dürfen diese vom Compiler beliebig umgeordnet werden. Bei Singlethreading macht sich dies nicht bemerkbar, da trotz Reorderings die Bedeutung nicht verändert wird.
- Es gilt *hb* (*A*, *B*), wenn zwei Anweisungen *A* und *B* über denselben Lock synchronisiert sind.
- Für einen Schreibzugriff *A* auf ein `volatile`-Attribut und einen späteren Lesezugriff *B* gilt *hb* (*A*, *B*).
- Für den Aufruf von `start()` (*A*) eines Threads und alle Anweisungen *B*, die in diesem Thread ausgeführt werden, gilt *hb* (*A*, *B*).
- Für alle Anweisungen *A* eines Threads vor dessen Ende *B* gilt *hb* (*A*, *B*), d. h., alle nachfolgenden Threads können diese Änderungen sehen – aber nur, wenn diese über den gleichen Lock synchronisiert sind oder `volatile`-Attribute verwenden.

In Multithreading-Programmen muss man daher als Programmierer dafür sorgen, eine Happens-before-Ordnung herzustellen, um mögliche Fehler durch Reorderings zu verhindern. Auch ohne sämtliche Details dieser Ordnung zu verstehen, kann man Thread-sichere Programme schreiben, wenn man sich an folgende Grundregeln hält:

1. **Korrekte, minimale, aber vollständige Synchronisierung** – Greifen mehrere Threads auf ein gemeinsam benutztes Attribut zu, so müssen immer *alle* Zugriffe über *dasselbe* Lock-Objekt synchronisiert werden. Für semantische Einheiten von Attributen kann auch ein und dasselbe Lock-Objekt verwendet werden.
2. **Beachtung von Atomarität** – Zuweisungen erfolgen in der Regel atomar. Für die 64-Bit-Datentypen muss dies explizit über das Schlüsselwort `volatile` sichergestellt werden. Mehrschrittoperationen (beispielsweise `i++`) können so nicht geschützt werden und müssen zwingend synchronisiert werden.

Tipp: Eigenschaften der Ordnung *hb*

Die Ordnung *hb* besitzt für beliebige Anweisungen *A* und *B* folgende Eigenschaften:

- **Irreflexiv** – !*hb*(*A*, *A*) – Keine Anweisung *A* »sieht« ihre eigenen Änderungen.
- **Transitiv** – Wenn *hb*(*A*, *B*) und *hb* (*B*, *C*) => *hb* (*A*, *C*) – Die Transitivität sorgt für die Sichtbarkeit gemäß der Reihenfolge der Anweisungen im Sourcecode.

- **Antisymmetrisch** – Wenn *hb*(*A*, *B*) und *hb*(*B*, *A*) => *A* = *B* oder hier intuitiver: Wenn *hb*(*A*, *B*) und *A* ≠ *B* => ! *hb*(*B*, *A*) – Die Antisymmetrie stellt sicher, dass eine Anweisung *A* nicht die Modifikationen einer Nachfolgeanweisung *B* »sieht«. Tatsächlich ist dies schon durch die Irreflexivität ausgeschlossen.

7.5 Besonderheiten bei Threads

Nachdem wir ausführlich den Lebenszyklus und die Zusammenarbeit von Threads kennengelernt haben, geht dieser Abschnitt auf Besonderheiten dabei ein. Im Folgenden stelle ich zunächst verschiedene Arten von Threads vor. Anschließend betrachten wir die Auswirkungen von Exceptions in Threads. Danach greife ich das Thema »Beenden von Threads« erneut auf. Abschließend stelle ich Möglichkeiten zur zeitgesteuerten Ausführung von Aufgaben vor.

7.5.1 Verschiedene Arten von Threads

Bei der Arbeit mit Threads und Nebenläufigkeit gibt es noch ein bisher nicht erwähntes Detail zu beachten: In Java unterscheidet man zwischen User- und Daemon-Threads.

`main`-Thread und User-Threads

Wie bereits erwähnt, erzeugt die JVM beim Start einen speziellen Thread, den man `main`-Thread nennt, weil dieser statt der `run()`-Methode die `main()`-Methode des Programms ausführt. Nehmen wir an, eine Applikation würde mehrere Threads aus dem `main`-Thread erzeugen und starten. Diese Threads nennt man ***User-Threads***. Die JVM bleibt selbst nach Ausführung des `main`-Threads bzw. der letzten Anweisung der `main()`-Methode aktiv, solange noch vom `main`-Thread abgespaltene User-Threads existieren und deren `run()`-Methode ausgeführt wird.

Daemon-Threads

Manchmal sollen Aufgaben im Hintergrund ablaufen und die Terminierung der JVM von solchen Threads unabhängig sein. Werden aber lediglich User-Threads gestartet, so wird die JVM nicht beendet, solange noch einer von diesen läuft. Als Abhilfe gibt es sogenannte ***Daemon-Threads***. Durch Aufruf der Methode `setDaemon(boolean)` kann man einen beliebigen Thread *vor* seiner Ausführung zu einem Daemon-Thread erklären. Das ist praktisch, wenn die Hintergrundaufgaben nicht wirklich zur eigentlichen Programmfunktionalität beitragen. Das gängigste Beispiel ist der Garbage Collector, der im Hintergrund Speicher aufräumt (vgl. Abschnitt 8.5).

Den Unterschied zwischen User- und Daemon-Threads erkennt man, wenn ein

Programm bzw. die zugehörige JVM terminieren soll. Bekanntermaßen geschieht dies nur, nachdem alle User-Threads beendet sind. Dann noch aktive Daemon-Threads werden abrupt beendet, d. h. irgendwo in der Abarbeitung ihrer `run()`-Methode. Daher muss man sorgsam sein, wenn Daemon-Threads Ressourcen belegen. Eine Variante zu deren Freigabe stellt eine eigene `finalize()`-Methode (vgl. Abschnitt 8.5.6) dar. Weil deren Abarbeitung laut JLS jedoch nicht garantiert ist, bietet sich ein Shut-down-Hook (vgl. Abschnitt 16.1.13) an.

7.5.2 Exceptions in Threads

In diesem Abschnitt wollen wir uns damit beschäftigen, was passiert, wenn während der Abarbeitung von Threads Exceptions auftreten. Nehmen wir dazu an, eine Applikation würde mehrere Threads aus dem `main`-Thread starten und in irgendeinem der Threads würde eine Exception ausgelöst. Aus Abschnitt 4.7.1 wissen wir, dass eine Exception mit einem `catch`-Block behandelt werden muss oder weiter entlang der Methodenaufrufkette gereicht wird, bis ein passender `catch`-Block gefunden wird oder man schließlich die `run()`- bzw. `main()`-Methode erreicht. Deren Ausführung bricht ab und führt zu einer Beendigung des ausführenden Threads. Abschließend wird die vom Programm unbehandelte Exception über den Error-Stream `System.err` inklusive des kompletten Stacktrace ausgegeben. Zum Nachvollziehen des zuvor beschriebenen Verhaltens können Sie folgendes Programm EXCEPTIONINTHREADSEXAMPLE ausführen:

```
// Achtung: Nur zur Demonstration des Exception Handlings
public static void main(final String[] args) throws InterruptedException
{
    exceptionInMethod();
}

static void exceptionInMethod() throws InterruptedException
{
    final Thread exceptional = new Thread()
    {
        public void run()
        {
            throw new IllegalStateException("run() failed");
        }
    };

    exceptional.start();
    Thread.sleep(1000);
}
```

Listing 7.11 *Ausführbar als* **'EXCEPTIONINTHREADSEXAMPLE'**

Es erscheint folgende Ausgabe (gekürzt) auf der Konsole:

```
Exception in thread "Thread-0" java.lang.IllegalStateException: run() failed
```

Wird das abrupte Ende eines Threads lediglich auf der Konsole ausgegeben, so wird eine spätere Fehlersuche sehr erschwert. Zum Nachvollziehen ist es nützlicher, Exceptions in eine Log-Datei zu schreiben. Dazu wollen wir eine kleine Utility-

Klasse erstellen und greifen auf das mit Java 5 eingeführte innere Interface `UncaughtExceptionHandler` der Klasse `Thread` zurück. Dessen Implementierung erlaubt es, von `catch`-Blöcken unbehandelte Exceptions behandeln zu können, indem man in der Methode `uncaughtException(Thread, Throwable)` das gewünschte Verhalten realisiert.

Die folgende Klasse `LoggingUncaughtExceptionHandler` implementiert beispielsweise eine Ausgabe in eine Log-Datei. Im Listing ist eine `main()`-Methode gezeigt, die zu Demonstrationszwecken Exceptions provoziert:

```
public final class LoggingUncaughtExceptionHandler implements Thread.
      UncaughtExceptionHandler
{
    private static final Logger log = Logger.getLogger("UncaughtExceptions");

    @Override
    public void uncaughtException(final Thread thread, final Throwable
throwable)
    {
        log.error("Unexpected exception occured: ", throwable);
    }
}
```

In diesem Beispiel wird ein mögliches Problem zwar nicht weiter behandelt, aber zumindest in einer Log-Datei protokolliert, was eine spätere Analyse erleichtert.

Ein solcher `UncaughtExceptionHandler` kann für alle Threads durch Aufruf von `Thread.setDefaultUncaughtExceptionHandler(Thread.UncaughtExce` global gesetzt werden. Bei Bedarf kann dies für jeden Thread einzeln durch Aufruf von `setUncaughtExceptionHandler(Thread.Uncaught-ExceptionHandler)` erfolgen.

7.5.3 Sicheres Beenden von Threads

Wie bereits erwähnt, lassen sich Threads leider nicht so einfach beenden wie starten. Die Methode `stop()` der Klasse `Thread` ist als `@deprecated` markiert. In verschiedenen Quellen wird als Grund dafür genannt, dass beim Beenden eines Threads nicht alle Locks freigegeben werden. Das ist ein verbreiteter Irrtum. Definitiv werden beim Auftreten von Exceptions alle gehaltenen Locks durch die JVM zurückgegeben – ansonsten wäre kein sinnvolles Programmverhalten mehr möglich und Aufrufe an diesen Lock betreffende `synchronized`-Methoden für die Laufzeit der JVM blockiert. Der Grund für die Markierung als `@deprecated` ist vielmehr folgender: Wenn ein Thread durch Aufruf von `stop()` beendet wird, können dadurch die Daten, auf denen der Thread gerade gearbeitet hat, in einen inkonsistenten Zustand gebracht werden. Insbesondere gilt dies, wenn der Thread innerhalb einer eigentlich atomaren Anweisungsfolge unterbrochen wird: Wird `stop()` durch einen anderen Thread mitten während der Ausführung eines über

`synchronized` definierten kritischen Bereichs aufgerufen, so wird dieser irgendwo unterbrochen und nicht mehr atomar ausgeführt. Eine mögliche Inkonsistenz im Objektzustand ist die Folge.

Um Konsistenz zu wahren, müssen wir andere Wege finden, einen Thread korrekt zu beenden. Zwei Möglichkeiten, dies sauber zu lösen, sind die folgenden:

1. Einführen einer Hilfsklasse, die die Funktionalität bereitstellt
2. Beenden durch Aufruf der Methode `interrupt()`

Hilfsklasse zum Beenden

Eine mögliche Lösung zum Beenden von Threads besteht darin, eine Hilfsklasse, bestehend aus einem Attribut und ein paar Zugriffsmethoden, zu implementieren und dort periodisch das Flag-Attribut `shouldStop` abzufragen.

Schauen wir zunächst auf eine denkbare, aber falsche Umsetzung, wie man sie in der Praxis und in manchem Buch findet:

```
// Achtung: Nicht Thread-sicher
public class BaseStoppableThread extends Thread
{
    private boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public void shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // Kein Aufruf von requestStop() und
            // keine Schreibzugriffe auf shouldStop
        }
    }
}
```

In der Praxis wird das Flag häufig `stopped` genannt. Zudem heißen die Zugriffsmethoden auf das Flag etwa `setStopped(boolean)` und `isStopped()`. Dies entspricht zwar der Intention des Beendens, allerdings wird hier eher ein Stoppwunsch geäußert. Daher werden die Methoden von mir `requestStop()` und `shouldStop()` genannt.

Auf den ersten Blick ist im Listing – abgesehen von der ungeschickten Ableitung von der Utility-Klasse `Thread` (vgl. folgenden Hinweis »Ableitung von der Klasse Thread«) – kein Fehler zu erkennen. Wieso ist diese Umsetzung trotzdem problematisch? Nach Lektüre von Abschnitt 7.4 über das Java-Memory-Modell sind

wir bereits etwas sensibilisiert: Die JVM darf zur Optimierung Reorderings durchführen, sofern die Happens-before-Ordnung eingehalten wird. Ohne diese Ordnung beachtet der Compiler bei der Optimierung keine Multithreading-Aspekte: Er kann beispielsweise wiederholte Lesezugriffe auf sich nicht ändernde Variablen zu vermeiden versuchen.

Innerhalb der `run()`-Methode finden wir keinen Aufruf von `requestStop()` oder einen sonstigen Schreibzugriff auf das Attribut `shouldStop`. Für Singlethreading ergibt sich daraus, dass sich das Attribut `shouldStop` in der Schleife nicht mehr ändert. Damit ist das Ergebnis der Bedingung konstant. Um den Methodenaufruf und die wiederholte Auswertung der Bedingung `!shouldStop()` einzusparen, kann der Sourcecode durch den Compiler und die JVM wie folgt optimiert und umgewandelt werden:

```
public void run()
{
    if (!shouldStop())
    {
        while (true)
        {
            // …
```

Beim Einsatz von Multithreading und Reorderings ist zum korrekten Ablauf dieses Beispiels durch den Entwickler die Happens-before-Ordnung sicherzustellen. Das kann in diesem Fall entweder über die Deklaration des Attributs `shouldStop` als `volatile` oder einen synchronisierten Zugriff geschehen. Bei der Korrektur vermeiden wir außerdem, von der Klasse `Thread` abzuleiten, und implementieren folgende Lösung, die auf dem Interface `Runnable` basiert und die Happens-before-Ordnung sicherstellt, wodurch keine Reorderings und keine Optimierung der Schleifenabfrage erfolgen:

```
abstract class AbstractStoppableRunnable implements Runnable
{
    private volatile boolean shouldStop = false;

    public void requestStop()
    {
        shouldStop = true;
    }

    public boolean shouldStop()
    {
        return shouldStop;
    }

    public void run()
    {
        while (!shouldStop())
        {
            // …
        }
    }
}
```

Hinweis: Ableitung von der Klasse `Thread`

Ableitungen von Utility-Klassen sind in der Regel ungünstig. ***Leider sieht man aber genau dies häufig beim Einsatz der Klasse*** **`Thread`**. Schauen wir uns an, wieso es zu Problemen kommen kann. Nehmen wir dazu an, eine Klasse `AbstractStoppableThread` sei durch Ableitung realisiert und die restliche Realisierung entspräche der zuvor vorgestellten Klasse `AbstractStoppableRunnable`:

```
abstract class AbstractStoppableThread extends Thread
{
    private volatile boolean shouldStop = false;

    // …

    public void run()
    {
        while (!shouldStop())
        {
            // …
        }
    }
}
```

Durch diese Implementierung wird das Substitutionsprinzip und somit auch die »isa«-Eigenschaft (vgl. Kapitel 3) verletzt. Der Grund ist folgender: Mit dieser von `Thread` abgeleiteten Klasse `AbstractStoppableThread` können keine Implementierungen von `Runnable` ausgeführt werden. Es entsteht eine Inkompatibilität:

Die `run()`-Methode der Basisklasse `Thread` wird durch eine eigene Implementierung überschrieben, die keine Ausführung von `Runnables` erlaubt. Die obige Umsetzung ist somit OO-technisch unsauber, weil sich die eigene Klasse nicht so verwenden lässt wie die Klasse `Thread`.

Beenden mit `interrupt()`

Statt einen Thread durch die Verwendung eigener Mechanismen, etwa den Einsatz eines Stop-Flags, zu beenden, kann man über die Methode `interrupt()` der Klasse `Thread` die Beendigung der Ausführung eines anderen Threads anregen. Wie bereits bekannt, ist ein Aufruf der Methode `interrupt()` jedoch nur als Aufforderung zu sehen, sie besitzt keine unterbrechende Wirkung: Es wird lediglich ein Flag gesetzt. Die Bearbeitung und Auswertung dieses Flags mithilfe der Methode `isInterrupted()` ist Aufgabe des Entwicklers der `run()`-Methode und kann in etwa so geschehen:

```
public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        // …
    }
```

```
}
```

Achtung: `Thread.interrupted()` vs. `isInterrupted()`

Bei der Abfrage des Flags muss man etwas Vorsicht walten lassen. Die Klasse `Thread` bietet die statische Methode `interrupted()`, die zwar das Flag prüft, dieses allerdings auch zurücksetzt. In der Regel soll eine Prüfung ohne Seiteneffekt erfolgen. Dazu ist immer die Objektmethode `isInterrupted()` zu verwenden.

Fazit

Beide gezeigten Lösungen zum Beenden von Threads sind funktional nahezu gleichwertig. Damit länger andauernde Aktionen in der `run()`-Methode tatsächlich abgebrochen werden können, müssen dort gegebenenfalls weitere Abfragen und Aufrufe von `shouldStop()` bzw. `isInterrupted()` erfolgen, um auch zwischen den ausgeführten Arbeitsschritten eine Reaktion auf Stoppwünsche zu ermöglichen. Das haben wir bereits in Abschnitt 7.1.4 diskutiert.

7.5.4 Zeitgesteuerte Ausführung

Möchte man gewisse Aufgaben zu einem speziellen Zeitpunkt oder periodisch ausführen, ist dies mit `Thread`-Objekten und deren `sleep(long)`-Methode zwar möglich, aber umständlich zu realisieren.

Eine einmalige Ausführung zu einem Zeitpunkt in der Zukunft kann man durch den Aufruf von `Thread.sleep(long)` vor der eigentlichen Funktionalität erreichen. Dazu berechnet man die Differenz von dem aktuellen Zeitpunkt bis zu dem gewünschten Zeitpunkt der Ausführung und wartet vor Ausführung der Aufgabe die zuvor berechnete Zeitdauer. Eine periodische Ausführung erreicht man, wenn die eben beschriebene Logik wiederholt innerhalb einer Schleife ausgeführt wird. Kompliziert wird dies, wenn mehrere Aufgaben zeitgesteuert abgearbeitet werden sollen.

Statt dies umständlich von Hand zu programmieren, ist es sinnvoller, die Utility-Klassen `Timer` und `TimerTask` zu verwenden. Darauf gehe ich im Folgenden ein.

Ausführung einer Aufgabe mit den Klassen `Timer` und `TimerTask`

Die Klasse `Timer` ist für die Verwaltung und Ausführung von `TimerTask`-Objekten zuständig, die durchzuführende Aufgaben implementieren. Man erzeugt zunächst ein `Timer`-Objekt und übergibt diesem ein oder mehrere `TimerTask`-Objekte. Deren Realisierung erfolgt durch Ableitung von der abstrakten Klasse `TimerTask`. Diese

erfüllt das Runnable-Interface und bietet folgende Methoden:

- void run() – Hier wird analog zu Thread und Runnable die Implementierung der auszuführenden Aufgabe vorgenommen.
- boolean cancel() – Beendet den TimerTask. Weitere Ausführungen werden verhindert.
- long scheduledExecutionTime() – Liefert den Zeitpunkt der letzten Ausführung und kann z. B. zur Kontrolle der Ausführungsdauer genutzt werden.

Zur Demonstration eines TimerTasks und der Verarbeitung mit einem Timer definieren wir folgende Klasse SampleTimerTask, die lediglich einen im Konstruktor übergebenen Text ausgibt:

```
public static class SampleTimerTask extends TimerTask
{
    private final String message;

    SampleTimerTask(final String message)
    {
        this.message = message;
    }

    public void run()
    {
        System.out.println(message);
    }
}
```

Einmalige Ausführung

Zur Ausführung werden TimerTask-Objekte an einen Timer übergeben und dort eingeplant. Der Zeitpunkt der Ausführung kann relativ in Millisekunden bis zur Ausführung oder absolut in Form eines Date angegeben werden. Das folgende Beispiel zeigt die Einplanung von SampleTimerTasks per schedule(TimerTask, long) einmalig sofort und zu einem gewissen Zeitpunkt in Form der Ausgaben "OnceImmediately" sowie "OnceAfter5s". Die Ausgabe von "OnceAfter1min" zeigt die Variante von schedule(TimerTask, Date), die den Ausführungszeitpunkt als Date-Objekt übergeben bekommt und die Ausführung nach einer Minute startet. Dieser Zeitpunkt wird über eine eigene Hilfsmethode oneMinuteFromNow() berechnet:

```
public static void main(final String[] args)
{
    final Timer timer = new Timer();

    // Sofortige Ausführung
    final long NO_DELAY = 0;
    timer.schedule(new SampleTimerTask("OnceImmediately"), NO_DELAY);

    // Ausführung nach fünf Sekunden
    final long INITIAL_DELAY_FIVE_SEC = 5000;
    timer.schedule(new SampleTimerTask("OnceAfter5s"),
```

```
INITIAL_DELAY_FIVE_SEC);

    // Ausführung nach einer Minute
    final Date ONE_MINUTE_AS_DATE = oneMinuteFromNow();
    timer.schedule(new SampleTimerTask("OnceAfter1min"), ONE_MINUTE_AS_DATE);
}
```

Listing 7.12 *Ausführbar als* **'TIMERTASKEXAMPLE'**

Periodische Ausführung

Möchte man eine Aufgabe periodisch ausführen, bietet die Klasse `Timer` hierfür zwei Varianten an: Zum einen kann dies mit festem Intervall durch einen Aufruf von `schedule(TimerTask, long, long)` und zum anderen mit festem Takt durch einen Aufruf der Methode `scheduleAtFixedRate(TimerTask, long, long)` geschehen. Optional kann eine Verzögerung bis zur ersten Ausführung angegeben werden. Liegt der Ausführungszeitpunkt in der Vergangenheit oder wird eine Verzögerung von 0 angegeben, so startet die Ausführung sofort.

Intuitiv könnte man denken, bei festem Intervall würde immer eine gleich lange Pause zwischen zwei Ausführungen entstehen und bei festem Takt entspräche der Abstand der Startzeitpunkte der angegebenen Taktrate. Tatsächlich ist nur Letzteres in etwa korrekt, nämlich solange es nur einen Task auszuführen gibt und der Takt größer als die Ausführungsdauer ist. Wenn beides gilt, dann gibt es keinen Unterschied zwischen `schedule()` und `scheduleAtFixedRate()`. Erst wenn man den Takt erhöht bzw. die Ausführungsdauer verlängert, sieht man Unterschiede in den Varianten. `schedule()` produziert dann gleichmäßigere Ausführungen, wohingegen diejenigen durch `scheduleAtFixedRate()` recht schnell etwas unüberschaubar werden – insbesondere dann, wenn mehrere Tasks mit unterschiedlichem Takt eingeplant werden.

Damit Sie sich ein besseres Bild davon machen können, sollten Sie folgendes Programm ausführen und ein wenig mit den Wartezeiten sowie der Anzahl an Tasks experimentieren. Die Klasse `DurationTimerTask` bildet dazu eine gute Ausgangsbasis.

```
public static void main(final String[] args) throws InterruptedException
{
    final Timer timer = new Timer();
    timer.schedule(new DurationTimerTask("FixedDelay1"), 0, 4000);
    // timer.schedule(new DurationTimerTask("FixedDelay2"), 0, 2000);
    stopTimer(timer);

    final Timer timer2 = new Timer();
    timer2.scheduleAtFixedRate(new DurationTimerTask("FixedRate1"), 0, 4000);
    // timer2.scheduleAtFixedRate(new DurationTimerTask("FixedRate2"), 0,
2000);
    stopTimer(timer2);
}

private static void stopTimer(final Timer timer) throws InterruptedException
{
    Thread.sleep(30* 1000);
```

```
    timer.cancel();
}

public static class DurationTimerTask extends TimerTask
{
    private static final long[] sleepTimesSecs = { 1, 2, 4 };
    private int index = 0;
    private final String info;

    public DurationTimerTask(final String info)
    {
        this.info = info;
    }

    @Override
    public void run()
    {
        final long sleepTimeSecs = sleepTimesSecs[index];
        index = (index + 1) % sleepTimesSecs.length;

        System.out.println(info + " – at " + new Date() +
                           " sleeping " + (sleepTimeSecs) + " secs");
        try
        {
            Thread.sleep(sleepTimeSecs* 1000);
        }
        catch (final InterruptedException e)
        {
            // ignore here
        }
    }
}
```

Listing 7.13 *Ausführbar als* **'TIMERTASKSCHEDULINGEXAMPLE'**

Achtung: Fallstricke beim Einsatz von `Timer` und `TimerTask`

Ausführung als User-Thread Ein `Timer` besitzt zur Ausführung *genau einen* Thread. Dieser läuft in der Regel als User-Thread. Bekanntermaßen wird eine JVM jedoch nicht terminiert, solange noch mindestens ein User-Thread läuft. ***Daher muss bei Programmende jeder*** **`Timer`** ***explizit mit der Methode*** **`cancel()`** ***angehalten werden, um eine Terminierung der JVM zu erlauben.*** Um sich nicht um solche Details kümmern zu müssen, ist es sinnvoller, einen `Timer` und dessen `Thread` als Daemon-Thread wie folgt zu starten:

```
final Timer daemonTimer = new Timer("DaemonTimer", true);
```

Gegenseitige Beeinflussung von `TimerTasks` Die Genauigkeit der angegebenen Ausführungszeiten kann gewissen Schwankungen unterliegen, da alle `TimerTasks` in einem gemeinsamen Thread des `Timers` ausgeführt werden. Weiterhin beendet eine nicht gefangene oder explizit ausgelöste Exception sowohl den `Timer` als auch alle enthaltenen `TimerTasks`.

Sollen sich Ausführungszeiten oder Fehler in einzelnen `TimerTasks` nicht auf die Ausführung anderer `TimerTasks` auswirken, so lässt sich dies durch die

Ausführung in eigenen `Timer`-Objekten und damit separaten Threads lösen. Es bietet sich alternativ der Einsatz sogenannter Thread-Pools an. Eine derartige Ausführung durch mehrere Threads sollte man nicht selbst realisieren, sondern die in Abschnitt 7.6.2 beschriebenen Möglichkeiten des mit JDK 5 eingeführten Executor-Frameworks nutzen.

Fazit

Wie man sieht, handelt es sich bei den bisher vorgestellten Techniken vorwiegend um (komplizierte) Konstrukte mit einem niedrigen Abstraktionsgrad, die einiges an Komplexität in eine Applikation einführen und in der Praxis eher umständlich zu nutzen sind. Mit dieser etwas schwierigeren Kost haben wir nun die elementaren Grundlagen für das Schreiben sicherer Multithreading-Anwendungen kennengelernt.

Ein Ansatz, die Probleme beim Multithreading eleganter zu lösen, bestand lange Zeit darin, die Klassenbibliothek `util.concurrent` von Doug Lea zu verwenden. Sie wurde durch den JSR 166 in Form der sogenannten »Concurrency Utilities« ins JDK 5 aufgenommen. Der folgende Abschnitt stellt diese vor.

7.6 Die Concurrency Utilities

Die bis einschließlich JDK 1.4 vorhandenen und zuvor vorgestellten Sprachmittel, wie die Schlüsselwörter `volatile` und `synchronized` sowie die Methoden `wait()`, `notify()` und `notifyAll()`, erlauben es zwar, ein Programm auf mehrere Threads aufzuteilen und zu synchronisieren, allerdings lassen sich viele Aufgabenstellungen so nur relativ umständlich realisieren.

Die mit JDK 5 eingeführten Concurrency Utilities erleichtern die Entwicklung von Multithreading-Anwendungen, da sie für viele zuvor nur mühsam zu lösende Probleme bereits fertige Bausteine bereitstellen. Durch deren Einsatz wird Komplexität aus der Anwendung in das Framework verlagert, was die Lesbarkeit und Verständlichkeit des Applikationscodes deutlich erhöht: Für Locks, Conditions und atomare Variablen haben wir bereits gesehen, dass sich Ideen und Konzepte klarer ausdrücken lassen.

Im Package `java.util.concurrent` befinden sich die Bausteine (Klassen und Interfaces) der Concurrency Utilities. Es gibt folgende Kernfunktionalitäten:

- **Concurrent Collections** – Die Concurrent Collections enthalten auf Parallelität spezialisierte Implementierungen der Interfaces `List<E>`, `Set<E>`, `Map<K, V>`, `Queue<E>` und `Deque<E>`, wodurch ein einfaches Ersetzen »normaler« Datenstrukturen durch deren Concurrent-Pendants wesentlich erleichtert wird, falls Nebenläufigkeit erforderlich ist. Abschnitt 7.6.1 geht darauf ein.
- **Executor-Framework** – Das Executor-Framework unterstützt bei der

Ausführung asynchroner Aufgaben durch Bereitstellung von Thread-Pools und ermöglicht es, verschiedene Abarbeitungsstrategien zu verwenden sowie die Bearbeitung abzubrechen und Ergebnisse abzufragen. Das wird in Abschnitt 7.6.2 behandelt.

- **Locks und Conditions** – Verschiedene Ausprägungen von Locks und Conditions vereinfachen die zum Teil umständliche Kommunikation und Koordination mit den Methoden `wait()`, `notify()` und `notifyAll()` sowie den Schutz kritischer Bereiche durch `synchronized`. Eine kurze Einführung in die Thematik habe ich bereits als Abschluss der Diskussion über kritische Bereiche und das Schlüsselwort `synchronized` in Abschnitt 7.2.2 gegeben.
- **Atomare Variablen** – Atomare Aktionen auf Attributen waren vor JDK 5 nur über Hilfsmittel, etwa die Schlüsselwörter `volatile` und `synchronized`, zu erreichen. Die mit JDK 5 neu eingeführten atomaren Variablen haben wir bereits bei der Vorstellung der Atomarität des JMM in Abschnitt 7.4.2 kennengelernt. Sie erlauben eine atomare Veränderung ohne Einsatz von Synchronisation.
- **Synchronizer** – Manchmal sollen Aktivitäten an speziellen Stellen in parallele Teile aufgespalten und später an Synchronisationspunkten wieder zusammengeführt werden. Lange Zeit war die Koordination mehrerer Threads lediglich auf unterster Sprachebene unter Verwendung der Methode `join()` möglich. Mit Java 5 wurden unter anderem die Klassen `CountDownLatch`, `CyclicBarrier`, `Exchanger` und `Semaphore` im Package `java.util.concurrent` eingeführt, die dabei helfen, Synchronisationspunkte zu realisieren. Eine kurze Vorstellung der Arbeitsweise wurde bereits in Abschnitt 7.3 gegeben.

7.6.1 Concurrent Collections

Wenn mehrere Threads parallel verändernd auf eine Datenstruktur zugreifen, kann es leicht zu Inkonsistenzen kommen. Dies gilt insbesondere, da die meisten Containerklassen des Collections-Frameworks nicht Thread-sicher sind. Die Zusammenarbeit und Kommunikation von Threads haben wir recht ausführlich in den Abschnitten 7.2 und 7.3 kennengelernt. Rekapitulieren wir in diesem Abschnitt zunächst kurz typische Probleme in Multithreading-Anwendungen beim Einsatz dieser »normalen« Collections. Anschließend werden wir zur Lösung dieser Probleme die Concurrent Collections nutzen. Diese können bei Bedarf nach Parallelität stellvertretend für die »Original«-Container eingesetzt werden. Die Grundlage für diese Austauschbarkeit bilden gemeinsame Interfaces (z. B. `List<E>`, `Set<E>` und `Map<K, V>`).

Wenn tatsächlich Nebenläufigkeit und viele parallele Zugriffe unterstützt werden müssen, können die Concurrent Collections ihre Stärken ausspielen. In

Singlethreading-Umgebungen oder bei sehr wenigen konkurrierenden Zugriffen ist ihr Einsatz gut abzuwägen, da in den Containern selbst einiges an Aufwand betrieben wird, um sowohl Thread-Sicherheit als auch Parallelität zu gewährleisten.

Thread-Sicherheit und Parallelität mit »normalen« Collections

Die `synchronized`-Wrapper des Collections-Frameworks ermöglichen einen Thread-sicheren Zugriff durch Synchronisierung aller Methoden (vgl. Abschnitt 5.3.2). Betrachten wir folgende synchronisierte Map als Ausgangsbasis unserer Diskussion:

```
final Map<String, Person> syncMap = Collections.synchronizedMap(personsMap);
```

Eine derartige Ummantelung führt zu einer (stark) eingeschränkten Parallelität, weil die Synchronisierung die Zugriffe serialisiert. Zudem schützt die Ummantelung nicht vor möglichen Inkonsistenzen, wenn man mehrere für sich Thread-sichere Methoden hintereinander aufruft. Das habe ich bereits bei der Beschreibung der `synchronized`-Wrapper in Abschnitt 5.3.2 diskutiert und greife es hier kurz auf, um die Arbeitsweise und Vorteile der Concurrent Collections zu motivieren.

Datenzugriff

Jeder einzelne Methodenaufruf einer synchronisierten Collection ist für sich gesehen Thread-sicher. Für eine benutzende Komponente sind solche feingranularen Sperren aber häufig uninteressant. Vielmehr sollen Operationen mit mehreren Schritten atomar und Thread-sicher ausgeführt werden. Solche Mehrschrittoperationen sind etwa »`testAndGet()`« oder »`putIfAbsent()`«, die zunächst prüfen, ob ein gewisses Element enthalten ist, und nur dann einen Zugriff bzw. eine Modifikation ausführen. Funktional würde man Folgendes schreiben:

```
// ACHTUNG: nicht Thread-sicher
public Person putIfAbsent(final String key, final Person newPerson)
{
    if (!syncMap.containsKey(key))
    {
        return syncMap.put(key, newPerson);
    }
    return syncMap.get(key);
}
```

Bekanntermaßen sind für sich Thread-sicherer Methoden in ihrer Kombination nicht Thread-sicher. Abhilfe schafft eine Synchronisierung der gesamten Aktionen. Mithilfe eines Synchronisationsobjekts implementiert man eine korrekt synchronisierte Version der `putIfAbsent(String, Person)`-Methode etwa wie folgt:

```
public Person putIfAbsent((final String key, final Person newPerson)
{
    synchronized (syncMap)  // Kritischer Bereich für Mehrschrittoperationen
    {
        if (!syncMap.containsKey(key))
```

```
        {
            return syncMap.put(key, newPerson);
        }
        return syncMap.get(key);
    }
}
```

Es wird keine gute Nebenläufigkeit erreicht, da andere Zugriffe blockiert werden.

Iteration

Eine weitere sehr gebräuchliche Mehrschrittoperation ist das Iterieren über eine Datenstruktur. Die Iteratoren der »normalen« Container sind »fail-fast«, d. h., sie prüfen sehr streng, ob möglicherweise während einer Iteration eine Veränderung an der Datenstruktur vorgenommen wurde, und reagieren darauf mit einer `ConcurrentModificationException`.

Zur Thread-sicheren Iteration über die Einträge einer synchronisierten Liste ist dieser Vorgang zusätzlich durch einen `synchronized`-Block zu schützen:

```
// Blockiert andere Zugriffe auf syncPersons während der Iteration
synchronized (syncPersons)
{
    for (final Person person : syncPersons)
    {
        person.doSomething();
    }
}
```

Auf diese Weise ist zwar eine korrekt synchronisierte Iteration möglich, allerdings auf Kosten von Nebenläufigkeit: Durch den `synchronized`-Block wird ein kritischer Bereich definiert und ***die Liste* `syncPersons` *ist während des Iterierens für mögliche andere Zugriffe blockiert. Nebenläufigkeit wird dadurch stark behindert.***

Thread-Sicherheit und Parallelität mit den Concurrent Collections

Wenn mehrere Threads gemeinsam lesend auf einer Collection arbeiten, muss nicht immer die gesamte Datenstruktur gesperrt und dadurch der Zugriff für andere Threads blockiert werden. Mehr noch: Lesezugriffe sollten sich gegenseitig nicht blockieren. Lese- und Schreibzugriffe müssen dagegen aufeinander abgestimmt werden. Dazu existieren verschiedene Verfahren. In den Concurrent Collections werden die zwei aufgelisteten Techniken eingesetzt, um neben Thread-Sicherheit auch für mehr Parallelität als bei den `synchronized`-Wrappern zu sorgen.

- **Kopieren beim Schreiben** – Die dahinterliegende Idee ist, vor jedem Schreibzugriff die Datenstruktur zu kopieren und dann das Element hinzuzufügen. Andere Threads können dadurch lesend zugreifen, ohne durch das Schreiben gestört zu werden. Diese Variante realisieren die Klassen `CopyOnWriteArrayList<E>` und `CopyOnWriteArraySet<E>` für Listen

sowie Sets.

- **Lock-Striping / Lock-Splitting** – Hierbei werden verschiedene Teile eines Objekts oder einer Datenstruktur mithilfe mehrerer Locks geschützt. Dadurch sinkt die Wahrscheinlichkeit für gleichzeitige Zugriffe auf jeden einzelnen Lock und Threads werden seltener durch das Warten auf Locks am Weiterarbeiten gehindert. Als Folge davon steigt die Parallelität. Zur Realisierung der parallelen `ConcurrentHashMap<K, V>` wird genau dieses Verfahren angewendet, das auf die spezielle Arbeitsweise von Hashcontainern mit Buckets abgestimmt ist und statt der gesamten `HashMap<K, V>` nur jeweils Teile davon schützt.

Datenzugriff in den Klassen `CopyOnWriteArrayList/-Set`

Die Implementierung der Klasse `CopyOnWriteArraySet<E>` nutzt die Klasse `CopyOnWriteArrayList<E>`. Diese wiederum verwendet ein Array zur Speicherung von Elementen und passt dieses bei Schreibvorgängen ähnlich der in Abschnitt 5.1.2 beschriebenen Größenänderung von Arrays an. Jede Änderung der Daten erzeugt eine Kopie des zugrunde liegenden Arrays. Dadurch können Threads ungestört parallel lesen, sehen eventuell jedoch nicht die aktuellen Änderungen durch andere Threads.

Allerdings sollte man folgende zwei Dinge beim Einsatz bedenken: Für kleinere Datenmengen (< 1.000 Elemente) ist das Kopieren und die mehrfache Datenhaltung in der Regel vernachlässigbar. Der negative Einfluss steigt mit der Anzahl zu speichernder Elemente linear an. Aufgrund der gewählten Strategie des Kopierens bieten sich diese Datenstrukturen daher vor allem dann an, wenn deutlich mehr Lese- als Schreibzugriffe erfolgen und die zu speichernden Datenvolumina nicht zu groß sind. Das haben wir bereits für eine Vereinfachung der Listener-Verwaltung in Abschnitt 7.2.4 eingesetzt.

Datenzugriff in der Klasse `ConcurrentHashMap`

Betrachten wir die Klasse `ConcurrentHashMap<K, V>`, die eine Realisierung einer `Map<K, V>` ist und mehreren Threads paralleles Lesen und Schreiben ermöglicht. Die Realisierung garantiert, dass sich Lesezugriffe nicht gegenseitig blockieren. Für Schreibzugriffe kann man zum Konstruktionszeitpunkt bestimmen, wie viel Nebenläufigkeit unterstützt werden soll, indem man die Anzahl der Schreibsperren festlegt.

Streng genommen ist die Klasse `ConcurrentHashMap<K, V>` lediglich ein Ersatz für die Klasse `Hashtable<K, V>` und nicht für die Klasse `HashMap<K, V>`, wie es der Name andeutet. Das liegt daran, dass die Klasse `ConcurrentHashMap<K, V>` im Gegensatz zur Klasse `HashMap<K, V>` keine `null`-Werte für Schlüssel und Werte unterstützt. Der Wert `null` drückt stattdessen

aus, dass ein gesuchter Eintrag fehlt. In vielen Fällen wird diese Unterstützung für `null` nicht benötigt. Dann kann man die Klasse `ConcurrentHashMap<K, V>` problemlos anstelle einer `HashMap<K, V>` verwenden.

Das Interface `ConcurrentMap<K, V>` deklariert vier Mehrschrittoperationen. Das Besondere daran ist, dass diese von den konkreten Realisierungen `ConcurrentHashMap<K, V>` und `ConcurrentSkipListMap<K, V>` atomar ausgeführt werden müssen. Die folgende Aufzählung nennt die Methoden und zeigt zur Verdeutlichung der Funktionalität eine schematische Pseudoimplementierung. Die tatsächliche Implementierung ist wesentlich komplizierter, da sie Parallelität ohne Blockierung gewährleistet.

- `V putIfAbsent(K key, V value)` – Erzeugt einen neuen Eintrag zu diesem Schlüssel und dem übergebenen Wert, falls kein Eintrag zu dem Schlüssel existiert. Liefert den zuvor gespeicherten Wert zurück, falls bereits ein Eintrag zu dem Schlüssel vorhanden ist, oder `null`, wenn dies nicht der Fall war.

```
if (!map.containsKey(key))
{
    return map.put(key, value);
}
return map.get(key);
```

- `boolean remove(Object key, Object value)` – Entfernt den Eintrag zu diesem Schlüssel, falls der gespeicherte Wert mit dem übergebenen Wert übereinstimmt. Liefert `true`, falls es einen solchen Eintrag gab, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(oldValue))
    {
        map.remove(key);
        return true;
    }
}
return false;
```

- `V replace(K key, V value)` – Ersetzt zu diesem Schlüssel den gespeicherten durch den übergebenen Wert. Liefert den zuvor mit dem Schlüssel assoziierten Wert zurück oder `null`, wenn es keinen Eintrag zu dem Schlüssel gab:

```
if (map.containsKey(key))
{
    return map.put(key, newValue);
}
return null;
```

- `boolean replace(K key, V oldValue, V newValue)` – Ersetzt den Eintrag zu diesem Schlüssel mit dem neuen Wert, falls der gespeicherte Wert mit dem alten Wert übereinstimmt. Liefert `true`, falls dem so ist, ansonsten `false`.

```
if (map.containsKey(key))
{
    if (map.get(key).equals(oldValue))
    {
        map.put(key, newValue);
        return true;
    }
}
return false;
```

Die gezeigten Aufrufe von `map.containsKey(key)` sind notwendig, um zu verhindern, dass bei einem fehlenden Eintrag eine `NullPointerException` ausgelöst wird.

Iteration

Um eine parallele Verarbeitung zu unterstützen, wurde das Verhalten der Iteratoren so angepasst, dass diese weder `ConcurrentModificationExceptions` auslösen noch eine Synchronisierung benötigen. Allerdings sind die Iteratoren auch nur »schwach« konsistent oder »weakly consistent«, d. h., eine Iteration liefert nicht immer die aktuellste Zusammensetzung der Datenstruktur, sondern zumindest den Zustand zum Zeitpunkt der Erzeugung des Iterators. Nachfolgende Änderungen an der Zusammensetzung können bei der Iteration berücksichtigt werden, müssen es aber nicht. Das erlaubt es, Iterationsvorgänge parallel zu Veränderungen durchzuführen.

Blockierende Warteschlangen und das Interface `BlockingQueue`

Zum Austausch von Daten oder Nachrichten zwischen unterschiedlichen Programmkomponenten können Implementierungen des Interface `Queue<E>` (vgl. Abschnitt 5.5) verwendet werden. Das Interface `BlockingQueue<E>` erweitert sein Basisinterface `Queue<E>` um folgende Methoden:

- `boolean offer(E element, long time, TimeUnit unit)` – Fügt ein Element in die Queue ein, falls keine Größenbeschränkung existiert oder die Queue nicht voll ist. Ansonsten wartet der Aufruf blockierend maximal die angegebene Zeitspanne, bis eine andere Programmkomponente ein Element entfernt, sodass als Folge ein Einfügen möglich wird. Liefert `true`, wenn das Einfügen erfolgreich war, ansonsten `false`.
- `E poll(long time, TimeUnit unit)` – Gibt das erste Element zurück und entfernt es aus der Queue. Wenn die Queue leer ist, wird maximal die angegebene Zeitspanne auf das Einfügen eines Elements durch eine andere Programmkomponente gewartet und nach einem erfolglosen Warten `null` zurückgegeben.
- `void put(E element)` – Fügt ein Element in die Queue ein. Dieser Aufruf

erfolgt blockierend. Das bedeutet, dass gewartet werden muss, wenn die Queue eine Größenbeschränkung besitzt und zum Zeitpunkt des Einfügens voll ist.

- `E take()` – Gibt das erste Element zurück und entfernt es aus der Queue. Der Aufruf blockiert, solange die Queue leer ist.

Wie schon angedeutet, blockieren diese Methoden beim Schreiben in eine volle Queue bzw. beim Lesen aus einer leeren Queue.[8] Das erleichtert die Kommunikation zwischen Threads, da auf eine fehleranfällige Synchronisierung und Benachrichtigung über die Methoden `wait()` und `notify()` bzw. `notifyAll()` verzichtet werden kann.

Die durch das Interface `BlockingQueue<E>` beschriebene Schnittstelle erinnert an das in Abschnitt 7.3 entwickelte Interface `FixedSizeContainer<T>` und die implementierende Klasse `BlockingFixedSizeBuffer<T>`. Ähnliche Funktionalität wird durch folgende Klassen der Concurrent Collections realisiert:

- `ArrayBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff, besitzt eine Größenbeschränkung und verwendet ein Array zur Speicherung.
- `LinkedBlockingQueue<E>` – Diese Realisierung bietet einen FIFO-Zugriff und nutzt eine `LinkedList<E>`, wodurch keine Größenbeschränkung gegeben ist.
- `PriorityBlockingQueue<E>` – Diese Realisierung nutzt ein Sortierkriterium, um die Reihenfolge der Elemente innerhalb der Queue zu bestimmen. Elemente mit der höchsten Priorität stehen am Anfang und werden somit bei Lesezugriffen zuerst zurückgeliefert.

Zusätzlich gibt es noch zwei besondere Implementierungen:

- `SynchronousQueue<E>` – Diese Queue ist ein Spezialfall einer größenbeschränkten Queue, allerdings mit der Größe 0. Zunächst klingt dies unsinnig, ist aber für solche Anwendungsfälle geeignet, die erfordern, dass zwei Threads unmittelbar aufeinander warten. Bezogen auf das Producer-Consumer-Beispiel bedeutet dies, dass ein Producer erst sein Produkt »speichern« kann, wenn ein Consumer dieses direkt abholt. Andersherum gilt: Möchte ein Consumer Daten aus der Queue entnehmen, muss er warten, bis ein Producer Daten ablegt.
- `DelayQueue<E>` – Bei dieser Art von Queue beschreiben zu speichernde Elemente ihre Sortierung, indem sie das Interface `java.util.concurrent.Delayed` implementieren, Das Interface `Delayed` erweitert `Comparable<Delayed>` und beschreibt somit eine Ordnung: In einer solchen Queue sind die Elemente nach ihrem »Verfallsdatum« geordnet. Dazu kann ein Ablaufzeitpunkt über die Methode `getDelay(java.util.concurrent.TimeUnit unit)` angeben werden.

Beispiel: Producer-Consumer mit `BlockingQueue`

Das in Abschnitt 7.3 vorgestellte und dort ständig weiterentwickelte Beispiel des Producer-Consumer-Problems kann durch den Einsatz des Interface `BlockingQueue<E>` leicht auf die Concurrent Collections umgestellt werden. Dieser Umstieg ist möglich, da bereits eine konzeptionell ähnliche Realisierung existiert. Es ist lediglich ein Implementierungsdetail anzupassen (Einsatz eines anderen Interface).

```
public static void main(final String[] args)
{
    final int MAX_QUEUE_SIZE = 7;
    final BlockingQueue<Item> items = new LinkedBlockingQueue<>
(MAX_QUEUE_SIZE);

    new Thread(new Producer(items, 100)).start();

    // warte 2 Sekunden, dadurch sieht man die Größenbeschränkung
    SleepUtils.safeSleep(TimeUnit.SECONDS, 2);

    new Thread(new Consumer(items, 1000, "Consumer 1")).start();
    new Thread(new Consumer(items, 1000, "Consumer 2")).start();
    new Thread(new Consumer(items, 1000, "Consumer 3")).start();
}
```

Listing 7.14 *Ausführbar als* **'PRODUCERCONSUMERBLOCKINGQUEUEEXAMPLE3'**

Info: Erweiterungen in `java.util.concurrent` mit JDK 6 und 7

In JDK 6 und JDK 7 wurden einige Verbesserungen und Erweiterungen in den Concurrency-bezogenen Klassen des Collections-Frameworks vorgenommen.

Erweiterungen in JDK 6 Im JDK 6 sind folgende Interfaces hinzugekommen:

- `BlockingDeque<E>` – Dieses Interface ist eine Spezialisierung einer `Deque<E>` und bietet Methoden, die beim Holen bzw. beim Speichern eines Elements so lange warten, bis die gewünschte Aktion möglich ist. Dieses Interface erweitert die Interfaces `Deque<E>` und `BlockingQueue<E>`.
- `ConcurrentNavigableMap<K, V>` – Dieses Interface ist eine Kombination aus den Interfaces `ConcurrentMap<K, V>` und `NavigableMap<K, V>`. Letzteres ist eine Erweiterung der `SortedMap<K, V>` um die Möglichkeit, passende Elemente für übergebene Schlüsselwerte zu finden. Die Klasse `ConcurrentSkipListMap<K, V>` implementiert das Interface `ConcurrentNavigableMap<K, V>` und stellt einen auf Parallelität der Zugriffe ausgelegten Ersatz für die Klasse `TreeMap<K, V>` dar.

Erweiterungen in JDK 7 In JDK 7 wurde das Interface `TransferQueue<E>` sowie dessen Realisierung `LinkedTransferQueue<E>` im Package `java.util.concurrent` hinzugefügt. Das Interface `TransferQueue` ist eine Erweiterung des Interface `BlockingQueue<E>`, das bekanntermaßen selbst wieder das Interface `Queue<E>` erweitert. Eine `BlockingQueue<E>` eignet sich etwa zur Anwendung im Producer-Consumer-Muster und vermeidet dort explizite

Synchronisierung: Ein Producer wird durch die Datenstruktur so lange blockiert, bis ein Einfügen möglich ist. Außerdem wird ein Consumer beim Auslesen von Daten blockiert, wenn keine Daten verfügbar sind. Das wurde bereits zuvor im Text erläutert. Mit dem Interface `TransferQueue<E>` wird die Abstimmung von Producer und Consumer fortgeführt: Ein weiteres Einfügen wird so lange blockiert, bis die Daten tatsächlich vom Consumer verarbeitet wurden, und nicht wie bisher nur bis zum Zeitpunkt des Auslesens aus der Queue. Dieses neue Verhalten wird durch die Methode `transfer(E)` realisiert. Deren Name ist wörtlich zu nehmen: Die weitere Abarbeitung wird so lange blockiert, bis die Übergabe der Daten von einem Thread zu einem anderen erfolgt ist. Man transferiert tatsächlich Daten zwischen Threads in einer Art und Weise, die die Happens-before-Ordnung herstellt.

Zudem stehen Methoden wie `tryTransfer(E e, long timeout, TimeUnit unit)` bereit, die einen Einfügevorgang entweder nicht blockierend ausführen oder maximal eine angegebene Wartezeit versuchen, Daten einzufügen. Darüber hinaus existieren Hilfsmethoden, die Auskunft über möglicherweise wartende Consumer und deren Anzahl geben (`hasWaitingConsumers()` bzw. `getWaitingConsumerCount()`).

7.6.2 Das Executor-Framework

Das Executor-Framework vereinfacht den Umgang mit Threads und die Verarbeitung von Ergebnissen asynchroner Aufgaben, sogenannter ***Tasks***. Es erfolgt eine Trennung der Beschreibung eines Tasks und dessen tatsächlicher Ausführung. Dabei wird vollständig von den Details abstrahiert, etwa wie und wann eine Aufgabe von welchem Thread ausgeführt wird. Die Kernidee ist, Tasks an sogenannte Executoren zur Ausführung zu übergeben.

Zunächst stelle ich die wichtigsten Interfaces und Klassen vor, bevor diese in einigen kleineren Beispielanwendungen genutzt werden.

Das Interface `Executor`

Tasks (Aufgaben bzw. Arbeitsblöcke), die das Interface `Runnable` implementieren, könnten im einfachsten Fall durch den Einsatz der Klasse `Thread` ausgeführt werden. Bekanntermaßen schreibt man dazu Folgendes:

```
new Thread(runnableTask).start();
```

Statt wie derart explizit mit Threads zu arbeiten, kann man alternativ eine Realisierung des Interface `java.util.concurrent.Executor` nutzen, um einen Tasks auszuführen. Dadurch wird von den konkreten Aktionen »Erzeugen« und »Starten« abstrahiert. Das Interface `Executor` ist dazu im JDK wie folgt definiert:

```
public interface Executor
{
```

```
    void execute(final Runnable runnableTask);
}
```

Damit lässt sich die obige Ausführung folgendermaßen schreiben:

```
executor.execute(runnableTask);
```

Sollen nur einige wenige Tasks ausgeführt werden, so ergibt sich kaum ein Vorteil durch den Einsatz eines Executors gegenüber dem Einsatz von Threads. Für MultithreadingAnwendungen, die aus diversen (unabhängigen) Tasks bestehen, kann eine derartige Abstraktion allerdings sehr vorteilhaft sein, wie wir dies nun beleuchten.

Implementierung des Interface `Executor`

Die zuvor gezeigte Abstraktion von Threads und Implementierungsdetails wirkt zunächst wenig spektakulär. Warum das Ganze dennoch hilfreich ist, werden wir sukzessive kennenlernen. Ein erster Vorteil besteht darin, dass das Interface `Executor` keine konkrete Art der Ausführung vorgibt. Die Details werden durch die jeweilige `Executor`-Implementierung spezifiziert, beispielsweise wie viele Aufgaben parallel ausgeführt werden und wann bzw. in welcher Reihenfolge dies erfolgt.

Betrachten wir zunächst zwei extreme Arten der Ausführung: alle Tasks synchron hintereinander und alle Tasks vollständig asynchron. Für Ersteres könnte man eine Klasse `SynchronousExecutor` wie folgt implementieren:

```
public class SynchronousExecutor implements Executor
{
    public void execute(final Runnable runnable)
    {
        runnable.run();
    }
}
```

In der Regel sollen Tasks nicht synchron im aufrufenden Thread, sondern parallel dazu ausgeführt werden. Eine Klasse `AsyncExecutor`, die für jeden Task einen eigenen Thread startet, könnte folgendermaßen realisiert werden:

```
public class AsyncExecutor implements Executor
{
    public void execute(final Runnable runnableTask)
    {
        new Thread(runnable).start();
    }
}
```

Hinweis: `Executor` selbst implementieren?

Wie bereits gesehen, ist es zwar möglich, eigene Implementierungen des Interface `Executor` zu schreiben, normalerweise empfiehlt es sich jedoch, die vordefinierten Klassen des Executor-Frameworks zu verwenden, die wir im Verlauf dieses Abschnitts kennenlernen werden.

Motivation für Thread-Pools

Nehmen wir an, es seien viele Aufgaben zu erledigen. Die zuvor vorgestellten Extreme bei der Umsetzung, d. h. die streng sequenzielle Ausführung aller Aufgaben durch einen einzigen Thread bzw. die maximale Parallelisierung durch Abspalten eines eigenen Threads pro Aufgabe, besitzen beide unterschiedliche Nachteile. Der erste Ansatz führt durch die Hintereinanderausführung zu langen Wartezeiten und schlechten Antwortzeiten. Es kommt zu einem schlechten Durchsatz. Der zweite Ansatz parallelisiert zwar die Ausführung und erreicht dadurch einen besseren Durchsatz. Demnach könnte man etwas naiv auf die Idee kommen, einfach so viele Threads zu erzeugen, wie Aufgaben existieren, um dadurch die Parallelität der Abarbeitung zu maximieren und für gute Antwortzeiten zu sorgen. Allerdings skaliert dieses Vorgehen nur begrenzt: Steigt die Anzahl der genutzten Threads deutlich über die Anzahl der verfügbaren CPUs bzw. Rechenkerne, so sinkt der Leistungszuwachs. Dies ist dadurch bedingt, dass maximal so viele Threads gleichzeitig aktiv sein können, wie es CPUs bzw. Rechenkerne gibt. Zudem steigt der Aufwand zur Verwaltung und Abstimmung der Threads und für Thread-Wechsel.

Weiterhin ist die Ausführung von Aufgaben durch Threads mit einem gewissen Overhead verbunden: Zum einen kostet das Erzeugen und Starten deutlich mehr Rechenzeit als ein Methodenaufruf. Zum anderen belegen Threads sowohl Betriebssystemressourcen als auch Speicher innerhalb sowie außerhalb der JVM. Sie können daher nur in begrenzter Anzahl[9] erzeugt werden.

Einen sinnvollen Kompromiss erreicht man durch den Einsatz sogenannter ***Thread-Pools***. Die Idee ist dabei, eine bestimmte Anzahl lauffähiger, aber pausierter Threads vorrätig zu halten und bei Bedarf zur Ausführung von Tasks zu aktivieren. Dadurch spart man sich die aufwendige Konstruktion neuer Threads. Thread-Pools ermöglichen zudem ein klares Programmdesign, da Verwaltungsaufgaben, d. h. das gesamte Thread-Management (Erzeugung, Zuteilung, Fehlerbehandlung usw.), aus der Applikation in den Thread-Pool verlagert wird. Ein solcher Thread-Pool kann auch dazu dienen, bei sehr hoher Last momentan nicht verarbeitbare Anfragen zu speichern und im Nachhinein zu bearbeiten. Alternativ kann man mit einem Thread-Pool beim Auftreten von Belastungsspitzen dynamisch versuchen, den Durchsatz zu erhöhen, indem temporär die Anzahl der Bearbeitungs-Threads (auch Worker-Threads genannt) erhöht wird.

Achtung: Nachteile von Thread-Pools

Thread-Pools bieten sich dann an, wenn die auszuführenden Tasks möglichst unabhängig voneinander sind. Doch auch schon dabei kann eine ungünstige Mischung von langlaufenden und kurzen Tasks zu Problemen führen, wenn die »Langläufer« sich vor den »Kurzläufern« registriert haben und letztere so blockiert werden. Die Ausführung kurzer Tasks wird dann eventuell unerwartet stark verzögert.

Wenn übergebene Tasks voneinander abhängig sind, kann der Einsatz eines Thread-Pools ungeeignet sein, da keine Abarbeitungsreihenfolge garantiert ist. Im Extremfall muss ein auszuführender Task auf das Ergebnis eines noch nicht durch den Thread-Pool ausgeführten Vorgänger-Tasks warten. Dadurch ist keine weitere Abarbeitung dieses Tasks mehr möglich.

Das Interface `ExecutorService`

Im Executor-Framework lassen sich Thread-Pools über FABRIKMETHODEN (vgl. Abschnitt 18.1.2) der Utility-Klasse `java.util.concurrent.Executors` erzeugen. Diese Methoden geben Objekte vom Typ `ExecutorService` zurück. Dieser erweitert das Interface `Executor` um die Möglichkeit, bereits laufende Aufgaben abbrechen zu können.

Mit folgenden Fabrikmethoden können spezielle Realisierungen des Interface `java.util.concurrent.ExecutorService` erzeugt werden:

- `newFixedThreadPool(int poolSize)` – Erzeugt einen Thread-Pool fester Größe.
- `newCachedThreadPool()` – Erzeugt einen Thread-Pool unbegrenzter Größe, der nach Bedarf wachsen und schrumpfen kann. Auf diese Weise kann dynamisch auf Belastungsspitzen reagiert werden.
- `newSingleThreadExecutor()` – Erzeugt einen Thread-Pool, der lediglich einen Thread verwaltet und folglich übergebene Aufgaben sequenziell abarbeitet (dies jedoch parallel zur eigentlichen Applikation).
- `newScheduledThreadPool(int poolSize)` – Erzeugt einen Thread-Pool der angegebenen Größe, der eine zeitgesteuerte Ausführung unterstützt und als Ersatz für die Klassen `Timer` und `TimerTask` (vgl. Abschnitt 7.5.4) dienen kann.

Wenn gegen das Interface `ExecutorService` programmiert wird, können die verwendeten, konkreten Realisierungen ausgetauscht und auf den jeweiligen Einsatz abgestimmt gewählt werden. Ein weiterer Vorteil der eben genannten Thread-Pool-Implementierungen ist, dass diese eine Fehlerbehandlung durchführen: Kommt es bei der Abarbeitung eines Tasks zu einer Exception, so führt dies zum Ende des ausführenden Worker-Threads. Durch den Thread-Pool werden »sterbende« Worker-Threads automatisch ersetzt, sodass für weitere Tasks immer eine Abarbeitung garantiert wird. Dies ist ein entscheidender Vorteil. Beim Einsatz eines `Timers` führte eine Exception zum Abbruch aller angemeldeten `TimerTasks`.

Die erzeugten Thread-Pools stellen spezielle Ausprägungen der folgenden beiden Klassen dar:

- `ThreadPoolExecutor` – Diese Klasse nutzt einen Thread-Pool. Neu eintreffende Aufgaben werden von bereitstehenden Threads bearbeitet, d. h. von

solchen, die momentan kein `Runnable` ausführen. Gibt es keinen freien Thread, so werden die Tasks in einer Warteliste zur Abarbeitung vorgemerkt.

- `ScheduledThreadPoolExecutor` – Diese Klasse erlaubt es, Tasks zeitgesteuert und insbesondere auch periodisch auszuführen und ähnelt damit der Klasse `Timer`.

Bevor ich auf die Details der Ausführung eingehe, stelle ich im Folgenden zwei Arten der Realisierung von Tasks vor.

`Callables` und `Runnables`

Bei der Abarbeitung nebenläufiger Tasks gibt es gewisse Anforderungen, die mithilfe von `Runnables` nur schwierig oder umständlich zu realisieren sind. Daher betrachten wir hier das Interface `java.util.concurrent.Callable`, nachdem ich gewisse Beschränkungen des Interface `Runnable` vorgestellt habe.

Um die Ergebnisse nebenläufiger Berechnungen wieder an den startenden Thread kommunizieren zu können, muss man sich beim Einsatz von `Runnables` gewisser Tricks bedienen. Ursache dafür ist, dass die `run()`-Methode weder einen Ergebniswert zurückgeben noch eine Checked Exception werfen kann. Ein Austausch von Ergebnissen mit einem Aufrufer ist daher lediglich dadurch möglich, dass diese in gemeinsam benutzten Datenstrukturen abgelegt werden. Alternativ dazu kann man in eigenen Realisierungen von `Runnables` spezielle Attribute zur Speicherung von Rückgabewerten definieren, die nach Abschluss der Berechnung von Aufrufern ausgelesen werden können.

Idiom: Ein- und Ausgabe mit Runnable

Die Methode `run()` aus dem Interface `Runnable` ist für Berechnungen ohne Abhängigkeiten recht gut geeignet. Des Öfteren benötigt man Ein- und Ausgaben und eine Möglichkeit, wie man das `Runnable` mit Informationen versorgen bzw. parametrieren und nach der Berechnung auch wieder auf das Ergebnis zugreifen kann. Das Ganze wird nun an einem Beispiel verdeutlicht. Zunächst implementieren wir das Interface `Runnable` in Form der Klasse `InOutputRunnableExample` und bieten einen Konstruktor mit Eingabeparametern, um Informationen zu übergeben. Ein Aufrufer kann auf das Berechnungsergebnis über die Methode `getResult()` zugreifen. Ob ein solches bereits vorliegt, ermittelt man durch Abfrage von `isCalculationFinished()`.

```
public class InOutputRunnableExample implements Runnable
{
    private final Object input1;
    private final Object input2;

    private Object resultValue = null;

    private volatile boolean calculationFinished = false;
```

```
    public InOutputRunnableExample(final Object input1, final Object input2)
    {
        this.input1 = input1;
        this.input2 = input2;
    }

    public void run()
    {
        calculationFinished = false;

        // calculate return value depending on input1 and input2
        returnValue = …;

        calculationFinished = true;
    }

    public Object getResult()
    {
        return resultValue;
    }

    public boolean isCalculationFinished()
    {
        return calculationFinished;
    }
}
```

Der hier erforderliche recht hohe Implementierungsaufwand ist durch die Beschränkungen des Interface `Runnable` begründet. Es ist für Aufrufer darüber hinaus nicht immer einfach, ohne Polling-Aufruf von `isCalculationFinished()` das Ende einer Berechnung zu erkennen.

Das Interface `Callable`

Aufgrund dieser Einschränkungen bei der Ausführung nebenläufiger Aktivitäten und deren Überwachung wurde das Interface `Callable<V>` eingeführt, womit man einem Aufrufer einen Rückgabewert übermitteln und außergewöhnliche Situationen über Exceptions kommunizieren kann. Dieses Interface ist im JDK folgendermaßen definiert:

```
public interface Callable<V>
{
    V call() throws Exception;
}
```

Betrachten wir nun, wie sich das zuvor dargestellte Idiom zum Datenaustausch mit einem Aufrufer durch den Einsatz eines `Callable<V>` klarer gestalten lässt. Auch hier haben wir zwei Eingabeparameter und berechnen zu Demonstrationszwecken lediglich eine Wartezeit in Millisekunden, während der wir dann auch die Ausführung pausieren lassen. Das generische Interface `Callable<V>` erlaubt beliebige nicht primitive Rückgabetypen. Für dieses einfache Beispiel nutzen wir den Typ `Long`:

```
public class CalcDurationInMs implements Callable<Long>
{
    private final TimeUnit timeUnit;
    private final long     duration;
```

```
    CalcDurationInMs(final TimeUnit timeUnit, final long duration)
    {
        this.timeUnit = timeUnit;
        this.duration = duration;
    }

    @Override
    public Long call() throws Exception
    {
        timeUnit.sleep(duration);

        return timeUnit.toMillis(duration);
    }
}
```

Wie man sieht, ist diese Realisierung kürzer und benötigt keine Hilfsvariable `result` zur Zwischenspeicherung der Berechnungsergebnisse mehr.

Ausführen von `Runnable` und `Callable`: Das `Future`-Interface

Nachdem wir nun wissen, wie Tasks definiert werden, wenden wir uns deren asynchroner Ausführung durch einen `ExecutorService` zu. Dieser kann Tasks ausführen, die durch Implementierungen der Interfaces `Runnable` sowie `Callable<V>` realisiert sind. Dazu werden folgende überladene `submit()`-Methoden genutzt:

- `<V> Future<V> submit(Callable<V> task)` – Es erfolgt eine Abarbeitung des `Callable<T>` mit anschließender Möglichkeit, das Ergebnis auszuwerten.
- `Future<?> submit(Runnable task)` – Das übergebene `Runnable` wird von einem Thread des `ExecutorService` abgearbeitet. Aufgrund der zuvor beschriebenen Einschränkungen von `Runnables` ist es im Gegensatz zur Ausführung von `Callable<V>` nur möglich, abzufragen, ob die Abarbeitung bereits beendet ist. Es wird jedoch kein Rückgabewert über `get()` geliefert, sondern immer `null`.
- `<T> Future<T> submit(Runnable task, T result)` – Erweitert den vorherigen Aufruf um die Möglichkeit, mit `get()` ein Ergebnis abfragen zu können. Nach der Abarbeitung des übergebenen Tasks wird der als Referenz übergebene Wert `result` vom Typ `T` zurückgegeben. Dies kann man jedoch nur dann sinnvoll nutzen, wenn dieser Typ veränderlich ist: Der ausführende Task ändert den Inhalt als Folge seiner Berechnungen. Das ist zwar ein Seiteneffekt, dieser ist aber lokal begrenzt und daher vertretbar.

`Runnable` mit Ergebnis

Die Möglichkeit zur Ausführung eines `Runnable` mit Rückgabe möchte ich kurz

aufgreifen. Wir betrachten hier einen Task, der eine `List<Integer>` als Ergebnisdatenstruktur erhält und diese modifiziert. Die Methode `submitRunnable()` übergibt die Verarbeitung an einen `ExecutorService`. In der Methode `accessResult()` ist gezeigt, wie man auf das Vorhandensein eines Berechnungsergebnisses prüft bzw. blockierend mit `get()` darauf wartet.

Folgende Zeilen deuten die Abarbeitung für ein Ergebnis vom Typ `List<Integer>` und einen Task `ModifyingTask` an:

```
public final class FutureExampleWithRunnableAndResult
{
    public static final class ModifyingTask implements Runnable
    {
        private final List<Integer> result;

        ModifyingTask(final List<Integer> result)
        {
            this.result = result;
        }

        @Override
        public void run()
        {
            result.add(Integer.valueOf(4711));
        }

        public List<Integer> getResult()
        {
            return result;
        }
    }

    public static void main(final String[] args)
    {
        final int POOL_SIZE = 3;
        final ExecutorService executorService =
                              Executors.newFixedThreadPool(POOL_SIZE);

        final Future<List<Integer≫ future = submitRunnable(executorService);
        accessResult(future);

        executorService.shutdown();
    }

    private static Future<List<Integer≫ submitRunnable(final ExecutorService
                                                       executorService)
    {
        final List<Integer> result = new ArrayList<>();
        final Future<List<Integer≫ future = executorService.submit(
                                            new ModifyingTask(result),
result);
        return future;
    }

    private static void accessResult(final Future<List<Integer≫ future)
    {
        try
        {
            System.out.println("isDone? " + future.isDone());
```

```
            System.out.println("Job finished with result: " + future.get());
        }
        catch (final InterruptedException e)
        {
            // Kann in diesem Beispiel nicht auftreten
        }
        catch (final ExecutionException e)
        {
            // Kann in diesem Beispiel nicht auftreten, wird geworfen wenn
            // versucht wird, auf ein Ergebnis eines Tasks zuzugreifen, der
            // mit einer Exception beendet wurde
        }
    }
}
```

Listing 7.15 *Ausführbar als* **'FUTUREEXAMPLEWITHRUNNABLEANDRESULT'**

Die von den Tasks gelieferten Ergebnisse werden asynchron berechnet und über das bisher kurz beschriebene Interface `java.util.concurrent.Future<V>` bereitgestellt. Da weder der genaue Ausführungszeitpunkt noch die Zeitdauer der einzelnen Tasks bekannt sind, ist es als Konsequenz unmöglich, direkt das Ende einer Berechnung festzustellen, um auf das Ergebnis zuzugreifen. Zum Verfolgen des Ausführungsfortschritts eines übergebenen Tasks erfolgt die Rückgabe eines `Future<T>`-Interface, das sowohl das Ergebnis einer Berechnung als auch deren Asynchronität kapselt. Mithilfe dieses Interface kann man unter anderem den Lebenszyklus von Tasks abfragen. Überladene `get()`-Methoden ermitteln den Ergebniswert und blockieren so lange, bis das Ergebnis verfügbar ist oder eine angegebene Time-out-Zeit überschritten wurde. Das Interface `Future<V>` ist im JDK folgendermaßen definiert:

```
public interface Future<V>
{
    V get() throws InterruptedException, ExecutionException;
    V get(long timeout, TimeUnit unit) throws
        InterruptedException, ExecutionException, TimeoutException;
    boolean cancel(boolean mayInterruptIfRunning);
    boolean isCancelled();
    boolean isDone();
}
```

Parallele Abarbeitung im `ExecutorService`

Nachdem die Verarbeitung über eine der überladenen `submit()`-Methoden angestoßen wurde, können parallel weitere Aufgaben gestartet werden. Jede einzelne kann man über das zurückgelieferte `Future<V>`-Interface mit der Methode `isDone()` befragen, ob deren Berechnung beendet ist. Folgendes Listing zeigt dies exemplarisch für zwei Berechnungen mit der bekannten Task `CalcDurationInMs`. Der erste Task wartet 5, der zweite 10 Sekunden:

```
public static void main(final String[] args)
{
    final int POOL_SIZE = 3;
    final ExecutorService executorService =
```

```
                            Executors.newFixedThreadPool(POOL_SIZE);

    // Definition und Start zweier Tasks
    final Future<Long> future1 = executorService.submit(
                                 new CalcDurationInMs(TimeUnit.SECONDS, 5));
    final Future<Long> future2 = executorService.submit(
                                 new CalcDurationInMs(TimeUnit.SECONDS, 10));

    System.out.println("Start: " + new Date());
    try
    {
        // synchron auf das Ende von Task 1 warten
        final Long result1 = future1.get();
        System.out.println("After Job 1: " + new Date());
        System.out.println(result1);

        // Zugriff nach 5s sollte false liefern
        System.out.println("isDone? Job 2: " + future2.isDone());

        // synchron auf das Ende von Task 2 warten
        final Long result2 = future2.get();
        System.out.println("After Job 2: " + new Date());
        System.out.println(result2);
    }
    catch (final InterruptedException | ExecutionException ex)
    {
        // Kann in diesem Beispiel nicht auftreten, s. o.
    }

    // Aufruf, um Thread-Pool zu beenden und JVM-Terminierung zu ermöglichen
    executorService.shutdown();
}
```

Listing 7.16 *Ausführbar als* **'EXECUTORSERVICEEXAMPLE'**

Mehrere Tasks abarbeiten

Die Methode `submit()` vom `ExecutorService` nimmt zu einem Zeitpunkt genau ein `Callable<V>` an und führt es aus. Häufig möchte man jedoch mehrere Tasks parallel ausführen. Dazu kann man sukzessiv `submit()` aufrufen. Eine Alternative stellen die – allerdings blockierenden – Methoden des `ExecutorService` dar:

- `invokeAll(Collection<Callable<V≫ tasks)` – Führt alle Aufgaben aus und liefert eine Liste von `Future<T>`-Objekten zum Zugriff auf die Ergebnisse.
- `invokeAny(Collection<Callable<V≫ tasks)` – Führt alle Aufgaben aus und liefert nur das Ergebnis des Tasks, der als Erster fertig ist.

Für beide Methoden existiert eine überladene Version, der man eine maximale Ausführungszeit mitgeben kann. Das berechnete Ergebnis wird nur geliefert, wenn es zu keiner Zeitüberschreitung kommt.

`ScheduledExecutorService` vs. `Timer`

Das Interface `ScheduledExecutorService` bietet Methoden, um Aufgaben zu bestimmten Zeiten bzw. wiederholt auszuführen, und wird von der Klasse `ScheduledThreadPoolExecutor` implementiert. Die Intention ist vergleichbar mit derjenigen der Klasse `Timer`. Im Gegensatz zu dieser werden jedoch mehrere Threads aus einem Pool zur Ausführung benutzt. Zudem können Ausführungszeiten nicht nur in Millisekunden angegeben werden, sondern mithilfe der Klasse `TimeUnit` in beliebigen Zeiteinheiten. Das trägt deutlich zur Lesbarkeit bei.

Kommen wir auf das Beispiel aus Abschnitt 7.5.4 zurück, das zur Demonstration der Arbeitsweise der Klassen `Timer` und `TimerTask` gedient hat. Statt eines `TimerTasks` zur Ausgabe eines Textes auf der Konsole verwende ich hier folgende simple Implementierung eines `Runnables` mit gleicher Aufgabe:

```
public static class SampleMessageTask implements Runnable
{
    private final String message;

    SampleMessageTask(final String message)
    {
        this.message = message;
    }

    public void run()
    {
        System.out.println(message);
    }
}
```

Listing 7.17 *Ausführbar als* **'SCHEDULEDEXECUTOREXAMPLE'**

Diese Klasse wird in der nachfolgenden `main()`-Methode genutzt, um die zeitgesteuerte Verarbeitung zu demonstrieren:

```
public static void main(final String[] args)
{
    final int POOL_SIZE = 3;
    final ScheduledExecutorService executorService =
                                    Executors.newScheduledThreadPool(POOL_SIZE);

    // Sofortige Ausführung
    executorService.schedule(new SampleMessageTask("OnceImmediately"), 0,
                             TimeUnit.SECONDS);

    // Ausführung nach fünf Sekunden
    executorService.schedule(new SampleMessageTask("OnceAfter5s"), 5,
                             TimeUnit.SECONDS);

    // Ausführung nach einer Minute
    executorService.schedule(new SampleMessageTask("OnceAfter1min"), 1,
                             TimeUnit.MINUTES);
}
```

Listing 7.18 *Ausführbar als* **'SCHEDULEDEXECUTOREXAMPLE'**

Durch die Verwendung der Klasse `TimeUnit` lassen sich Zeitangaben gut lesbar notieren. Ansonsten existiert kaum ein Unterschied zu den Methoden der Klasse `Timer`. Allerdings kann, im Gegensatz zu dieser, keine Einplanung über die Angabe eines Zeitpunkts in Form der Klasse `Date` erfolgen. Dafür lässt sich die periodische

Ausführung noch klarer als bei der Klasse `Timer` ausdrücken.

Die Methoden zur Ausführung mit festem Takt und festgelegter Verzögerung zwischen den Ausführungen besitzen sprechende Namen: `scheduleAtFixedRate()` und `scheduleWithFixedDelay()`. Letztere Ausführungsart wurde im `Timer` durch die Methode `schedule()` angestoßen. Besser lesbar ist der Methodenaufruf `scheduleWithFixedDelay()`. Folgendes Beispiel zeigt beide Varianten der wiederholten Ausführung:

```
public static void main(final String[] args)
{
    final int POOL_SIZE = 3;
    final ScheduledExecutorService executorService =
                                    Executors.newScheduledThreadPool(POOL_SIZE);

    // Eingeplante Ausführung mit INITIAL_DELAY und Verzögerung DELAY
    final long INITIAL_DELAY_ONE_SEC = 1;
    final long DELAY_30_SECS = 30;
    executorService.scheduleWithFixedDelay(new SampleMessageTask(
                "PeriodicRefreshing"), INITIAL_DELAY_ONE_SEC, DELAY_30_SECS,
                TimeUnit.SECONDS);

    // Eingeplante Ausführung mit INITIAL_DELAY und Takt RATE
    final long RATE_10_SECS = 10;
    executorService.scheduleAtFixedRate(new SampleMessageTask(
                "10s FixedRateExecution"), INITIAL_DELAY_ONE_SEC,
RATE_10_SECS,
                TimeUnit.SECONDS);
}
```

Listing 7.19 *Ausführbar als* **'SCHEDULEDEXECUTORSERVICEEXAMPLE'**

Fazit

Dieser Abschnitt hat einen Einstieg in die Verarbeitung von Tasks mit dem Executor-Framework gegeben. Ein großer Vorteil besteht darin, dass man von vielen Details der Thread-Verwaltung abstrahieren kann. Dadurch gibt es nahezu keinen Grund mehr, direkt mit Threads zu arbeiten. Wenn viele Tasks auszuführen sind, gelten diese Aussagen umso mehr. ***In neuen Projekten sollten* `Executor`*-Realisierungen bevorzugt werden, da sie eine bessere Abstraktion als die Klassen* `Thread` *und* `Timer` *bieten***. Dies empfiehlt auch Joshua Bloch in seinem Buch »Effective Java« [9] in Item 68.

7.6.3 Das Fork-Join-Framework

Mit JDK 7 wurde das Executor-Framework um das sogenannte Fork-Join-Framework erweitert, das die Parallelverarbeitung von Aufgaben gut unterstützt. Die dazu eingeführten `ForkJoinTasks` bilden eine plattformunabhängige Möglichkeit, rechenintensive Operationen tatsächlich parallel auf Mehrprozessormaschinen auszuführen. Diese Art der Verarbeitung lässt sich auf viele Algorithmen der Kategorie »teile und herrsche« (***divide and conquer***) anwenden. Die gewählte

Lösungsstrategie besteht darin, ein größeres Problem in diverse kleinere zu lösende Probleme aufzuteilen und diesen Vorgang so lange rekursiv fortzusetzen, bis eine einfache (häufig nur aus wenigen Anweisungen bestehende) Lösung möglich ist. Man spricht hier vom rekursiven Abstieg mit Abbruchkriterium. Darunter versteht man, dass die Berechnung der Lösung zunächst in immer kleinere Problemstellungen zerlegt wird, dann eine einfache Berechnung erfolgt und anschließend die Ergebnisse schrittweise zusammengeführt werden. Das Besondere an den Problemstellungen ist, dass diese voneinander unabhängig sind und daher parallel gelöst werden können.

Ich zeige den Einsatz von `ForkJoinTasks` am Beispiel der Berechnung der Fibonacci-Zahlen *fib* (*n*).[10] Diese sind rekursiv wie folgt definiert:

$fib(0) = 0$

$fib(1) = 1$

$fib(n) = fib(n-1) + fib(n-2), \forall n \geq 2$

Nehmen wir dazu an, es soll eine beliebige Fibonacci-Zahl berechnet werden. Basierend auf der rekursiven mathematischen Definition könnte man eine direkte Abbildung in eine rekursive programmatische Berechnung vornehmen. Diese wird hier lediglich zu Demonstrationszwecken der Funktionalität des Fork-Join-Frameworks gewählt. In der Praxis ist dieser Ansatz schlecht, da ein einfacherer Algorithmus zur Berechnung der Fibonacci-Zahlen mit linearer Komplexität existiert, dessen Laufzeitverhalten von der zu berechnenden Fibonacci-Zahl abhängt und damit $O(n)$ entspricht (vgl. Abschnitt 22.1.5). Die im folgenden Beispiel gewählte Unterteilung in einzelne Berechnungsschritte ist zudem ungünstig, da jeder dieser Schritte zu einfach und schnell zu berechnen ist, als dass sich eine Aufteilung lohnen würde. Wir nehmen die genannten Nachteile in Kauf, um das Prinzip verdeutlichen zu können.

Grundlage für Berechnungen bildet die generische abstrakte Klasse `ForkJoinTask<V>`, die das Interface `Future<V>` implementiert. Eine Instanz davon kann über den Aufruf der Methode `fork()` einen neuen Worker-Thread abspalten und mit `join()` die Berechnungen wieder zusammenführen. Zur rekursiven Berechnung ist die generische abstrakte Klasse `RecursiveTask<V>` als Erweiterung der Klasse `ForkJoinTask<V>` definiert. Dort wird eine abstrakte Methode `compute()` mit dem Rückgabetyp `V` angeboten, in der in eigenen Realisierungen die eigentlichen Berechnungen stattfinden sollen.

Nutzen wir das gewonnene Wissen zur Implementierung einer Klasse `FibonacciTask` wie folgt:

```
public final class FibonacciTask extends RecursiveTask<Integer>
{
    final int n;

    public FibonacciTask( final int n)
    {
        if ( n < 0)
            throw new IllegalArgumentException( "parameter n must be
```

```
positive");

        this.n = n;
    }

    protected Integer compute()
    {
        // Ende des rekursiven Abstiegs
        if (n == 0)
            return 0;

        if (n == 1)
            return 1;

        // Rekursiver Abstieg für fib(n-1)
        final FibonacciTask fibTask1 = new FibonacciTask(n - 1);
        fibTask1.fork();

        // Rekursiver Abstieg für fib(n-2)
        final FibonacciTask fibTask2 = new FibonacciTask(n - 2);

        // Zusammenführung fib(n-2) mit fib(n-1)
        return fibTask2.compute() + fibTask1.join();
    }

    public static void main(final String[] args)
    {
        // Definition eines Pools von Worker-Threads
        final int WORKER_THREAD_COUNT = 4;
        final ForkJoinPool forkJoinPool = new
ForkJoinPool(WORKER_THREAD_COUNT);

        // Berechnung von fib(42)
        System.out.println(forkJoinPool.invoke(new FibonacciTask(42)));
    }
}
```

Listing 7.20 *Ausführbar als* **'FIBONACCITASK'**

Zur Ausführung derartiger Tasks muss man sich einen Pool an Worker-Threads in Form der Klasse `ForkJoinPool` anlegen. Anschließend wird durch den Aufruf der Methode `invoke(ForkJoinTask<T>)` die Berechnung gestartet.

Hinweise

Bei den `ForkJoinTasks` sind zwei Aspekte zu beachten: Zum einen ist dies die Komplexität des zu lösenden Problems und zum anderen der Verwaltungsoverhead, der durch das Aufteilen in kleinere Teilaufgaben entsteht. Dieser führt dazu, dass ab einer gewissen Granularität (Anzahl an Anweisungen) die sequenzielle Berechnung schneller ist als die parallele.

Weiterführende Informationen finden Sie in Doug Leas Buch »Concurrent Programming in Java« [59]. Das dort beschriebene eigenständiges Framework bildet die Grundlage für das in JDK 7 eingeführte Fork-Join-Framework.

Tipp: Aufspalten von Teilaufgaben

Es gibt eine Grenze für eine minimale Granularität bzw. Anzahl an Berechnungsschritten, die als Einheit innerhalb eines Subtasks ausgeführt werden sollten. Das sequenzielle Berechnen oder Lösen eines Problems ist ab dieser Granularität effizienter als das weitere Aufteilen. Es bedarf allerdings einiges an Erfahrung und Fingerspitzengefühl sowie Messungen, um diese Grenze für die Aufspaltung passend wählen zu können.

7.7 Weiterführende Literatur

Da die Themen Multithreading und Nebenläufigkeit recht komplex und vielschichtig sind, empfiehlt es sich, weitere Quellen zu konsultieren. Weiterführende Informationen finden Sie in folgenden Büchern:

- »**Parallele und verteilte Anwendungen in Java**« von Rainer Oechsle [72]

  Dieses Buch gibt einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5.

- »**Java Threads**« von Scott Oaks und Henry Wong [71]

  Ebenso wie das Buch von Rainer Oechsle bietet dieses Buch einen sehr lesbaren und verständlichen Einstieg in das Thema Multithreading mit Java 5. Durch den Fokus auf Threads erklärt es einige Dinge noch gründlicher.

- »**Java Concurrency in Practice**« von Brian Goetz et al. [31]

  Dieses Buch gibt eine sehr umfassende und fundierte Beschreibung zum Thema Multithreading. Dort werden auch die Erweiterungen der Concurrent Collections in JDK 6 behandelt.

- »**Concurrent Programming in Java**« von Doug Lea [59]

  Dieser Klassiker, der bereits einige Jahre auf dem Buckel hat, stammt vom Entwickler der Concurrency Utilities selbst. Viele der im Buch beschriebenen Klassen haben später Einzug in die Java-Klassenbibliothek gehalten. Es ist allerdings keine leichte Lektüre.

- »**Taming Java Threads**« von Allen Holub [41]

  Dieses Buch gibt einen fundierten Einstieg und beschreibt sehr genau die Details von Multithreading inklusive diverser Fallstricke. Es werden auch fortgeschrittenere Themen wie Multithreading und Swing, Thread-Pools und blockierende Warteschlangen behandelt. Da dieses Buch noch auf Java 1.4 basiert, werden hier ähnliche Ideen und Klassen entwickelt wie bei Doug Lea.

- »**Fortgeschrittene Programmierung mit Java 5**« von Johannes Nowak [70]

  Dieses Buch beschäftigt sich intensiv mit den in JDK 5 hinzugekommenen Sprachfeatures Generics und den Concurrency Utilities. Es geht nicht so detailliert wie die zuvor genannten Bücher auf die Thematik ein, gibt aber einen

guten Uberblick.

- »**Java 7 Concurrency Cookbook**« von Javier Fernández González [32]

  Dieses Buch liefert um die 60 kleine Beispiele rund um Multithreading mit Java 7. Dabei geht es fundiert auf die dortigen Neuerungen ein. Erwähnenswert ist auch ein Kapitel zum anspruchsvollen Thema des Testens von Multithreading.

8 Fortgeschrittene Java-Themen

In diesem Kapitel werden einige fortgeschrittene Java-Themen vorgestellt. Abschnitt 8.1 beginnt mit einem in die Sprache integrierten, ***Reflection*** genannten Mechanismus. Dieser ermöglicht es, zur Laufzeit Informationen zu Klassen, Objekten usw. zu ermitteln. Es ist sogar möglich, neue Instanzen von zum Kompilierzeitpunkt unbekannten Klassen zu erzeugen und deren Methoden aufzurufen. An die Besprechung von Reflection schließt sich eine Darstellung der mit JDK 5 neu eingeführten ***Annotations*** zur Angabe von Zusatzinformationen im Sourcecode an. Abschnitt 8.2 gibt sowohl eine Einführung in die Annotations des JDKs als auch eine Beschreibung, wie man eigene Annotations definieren und auswerten kann.

Mithilfe zweier spezieller Marker-Interfaces des JDKs können Klassen indirekt um Funktionalität erweitert werden: Eine markierte Klasse aktiviert in die JVM integrierte Automatiken. Abschnitt 8.3 beschreibt die Möglichkeiten zur Konvertierung von Objekten in eine Folge von Bytes und zurück. Dies wird ***Serialisierung*** genannt und über das Interface `java.io.Serializable` als Funktionalität aktiviert. Den Automatismus zum Kopieren von Objekten kann man über das Marker-Interface `java.lang.Cloneable` anschalten. Allerdings sind im Gegensatz zur Serialisierung einige weitere Schritte vom Entwickler selbst zu programmieren. Abschnitt 8.4 geht darauf genauer ein.

In Java muss man sich als Entwickler kaum oder gar nicht um das Speichermanagement und insbesondere die Freigabe von Objekten bzw. genauer des durch sie belegten Speichers kümmern. Abschnitt 8.5 erläutert das in die JVM integrierte, automatische Speichermanagement, ***Garbage Collection*** genannt.

8.1 Crashkurs Reflection

Die Technik ***Reflection*** erlaubt es einem Programm, verschiedene Informationen über sich selbst herauszufinden und zur Laufzeit Instanzen von Klassen zu erzeugen. Bei Reflection handelt sich es um eine fortgeschrittene Technik, die man bei der normalen Anwendungsentwicklung seltener verwendet. Es gibt jedoch einige Anwendungsfälle, die nur durch den Einsatz von Reflection umzusetzen sind. Die Java-Seiten von Oracle[1] nennen unter anderem folgende Einsatzgebiete:

- **Erweiterbarkeit** – Eine Applikation kann mit Reflection sogar solche Klassen instanziieren, die zum Zeitpunkt der Kompilierung unbekannt sind. Zur Laufzeit wird nur ihr voll qualifizierter Name benötigt. Ausgehend davon können alle möglichen Bestandteile von Klassen, also Methoden, Attribute usw., abgefragt werden.
- **Class-Browser, Debugger und Tools zum Testen** – Mithilfe von Reflection erhält ein Programm Zugriff auf alle Bestandteile einer Klasse. Selbst der Zugriff

auf private Attribute und Methoden ist möglich. Beispielsweise nutzt JUnit 3.x Reflection, um die auszuführenden Testmethoden zu ermitteln.

Neben den obigen Einsatzgebieten sollte Reflection nur in speziellen Situationen und für den Fall, dass es wirklich notwendig und sinnvoll ist, verwendet werden, etwa wenn zur Laufzeit Informationen über das Programm selbst benötigt werden oder dessen Verhalten geändert werden soll. Denken Sie an folgenden Hinweis: ***Gibt es eine Lösung ohne Reflection, so wähle diese***, weil sie in der Regel klarer und verständlicher ist.

Hinweise zum Einsatz

Vor einem möglichen Einsatz sollte man die im Folgenden genannten Besonderheiten beachten, die sich nachteilig auswirken können:

- Bei der Verwendung von Reflection werden Informationen erst zur Laufzeit ermittelt. Das verursacht etwas Overhead, wodurch Methodenaufrufe, Attributzugriffe usw. etwas langsamer als »normale« Aufrufe ausgeführt werden.
- Aufrufe per Reflection werden möglicherweise durch nicht ausreichende Berechtigungen verhindert. Grundsätzlich ist es korrekt, eine nicht berechtigte Ausführung zu unterbinden. Die dazu notwendigen Berechtigungen werden über eine Instanz der Klasse `java.lang.SecurityManager`[2] kontrolliert. Zur Laufzeit können unterschiedliche Instanzen der Klasse `SecurityManager` aktiv sein und somit das Programmverhalten unerwartet verändern.
- Ein Nachteil beim Einsatz von Reflection ist, dass man sowohl Kapselung als auch einiges von der durch den Compiler garantierten Sicherheit aufgibt. Insbesondere kann der Sichtbarkeitsschutz privater Bestandteile von Klassen ausgehebelt werden, wodurch Interna sichtbar werden. Außerdem sind Abhängigkeiten zwischen Klassen schwieriger zu ermitteln, da diese textuell als Klassen-, Methoden- oder Attributnamen beschrieben sind.

Mit diesen Hinweisen im Hinterkopf wollen wir Reflection zumindest so weit kennenlernen, dass ein grundlegendes Verständnis vorhanden ist. Es gibt wenig Literatur, die sich explizit mit diesem Thema auseinandersetzt. Eine Ausnahme bildet das Buch »Java Reflection in Action« von Ira R. und Nate Forman [26].

8.1.1 Grundlagen

Damit eine Untersuchung eines Programms überhaupt möglich ist, müssen spezielle Informationen über Klassen und Objekte existieren. Diese nennt man ***Metadaten*** oder ***Metainformationen***, sie sind im JDK durch Klassen modelliert, z. B. folgende:

- `java.lang.Class` – Metadaten für Klassen, etwa Klassenname, implementierte Interfaces, Methoden, Attribute usw.

- `java.lang.reflect.Field` – Metadaten für Attribute, etwa Typ, Name, Sichtbarkeit usw.
- `java.lang.reflect.Method` – Metadaten für Methoden, etwa Name, Parameter, Sichtbarkeit usw.

Den Startpunkt für die Programmierung mit Reflection bildet immer eine Instanz der Klasse `java.lang.Class`, über die man andere Metadaten beziehen kann. Beschäftigen wir uns also zunächst mit folgender Frage: Wie erhält man Zugriff auf die Metainformationen einer Klasse, also das zugehörige `Class`-Objekt? Es gibt drei verschiedene Möglichkeiten, die Metadaten einer Klasse zu ermitteln:

1. Besitzt man einen voll qualifizierten Klassennamen, so erhält man über einen Aufruf der statischen Methode `Class.forName(String)` ein `Class`-Objekt:

```
// kein Import und kein Class-Loading vor dem ersten Zugriff
final Class<?> clazz = Class.forName("packagePath.ClassName");
```

   Der Name der gewünschten Klasse wird dabei in Form eines Strings übergeben. Dies kann man beispielsweise nutzen, wenn der konkrete Typ zur Kompilierzeit noch unbekannt ist. Dadurch können zur Laufzeit Klassennamen etwa aus einer Konfigurationsdatei eingelesen und verwendet werden. Im einsetzenden Sourcecode existieren keine `import`-Anweisungen und keine direkten Verweise auf die entsprechenden Klassen. Dies vermeidet wiederum ein Laden der Klassenbeschreibung, wie dies bei einem `import` implizit ausgelöst wird. ***Negativ an der Übergabe eines Klassennamens als String ist, dass es leicht zu Fehlern kommt***. Dies können sowohl Tippfehler im Programm selbst oder aber in den Konfigurationsdaten sein.

2. Durch Aufruf der Methode `getClass()` kann die Klasseninformation zu einem bestehenden Objekt wie folgt ermittelt werden:

```
// Zugriff auf ein Class-Objekt über ein Objekt
final Class<?> clazz = obj.getClass();
```

3. Wenn man den konkreten Typ kennt, erhält man über das statische Attribut `.class` Zugriff auf das `Class`-Objekt:

```
// Es wird ein Import nötig und die Klasse wird gegebenenfalls geladen.
Class<?> clazz = MyClass.class;
```

Hinweis: Notation von Arrays in `Class.forName()`

Zum Ermitteln der Klassenbeschreibung für Arrays ist eine etwas kryptische Notation zu verwenden. Im folgenden Beispiel wird das für die Konstruktion eines zweidimensionalen `int`-Arrays gezeigt. Die führenden '`[`'-Zeichen beschreiben die Anzahl der Array-Dimensionen. Für den Typ `int` wird das Typkürzel '`I`' verwendet. Für Referenzvariablen ist dabei zusätzlich Folgendes zu beachten: Das Typkürzel ist ein '`L`' und erfordert einen voll qualifizierten Klassennamen, der durch ein Semikolon abgeschlossen wird – nachfolgend für die Klasse `java.lang.String` gezeigt:

```
// Zugriff auf den Typ „zweidimensionales int-Array": [[I
final Class<?> intintArrayClazz = Class.forName("[[I");
System.out.println(int[][].class);
System.out.println(intintArrayClazz == int[][].class);

// Zugriff auf den Typ „eindimensionales String-Array": [Ljava.lang.String;
final Class<?> stringArrayClazz = Class.forName("[Ljava.lang.String;");
System.out.println(String[].class);
System.out.println(stringArrayClazz == String[].class);
```

Listing 8.1 *Ausführbar als* **'REFLECTIONARRAYCLASSESEXAMPLE'**

Typkürzel Die Typkürzel werden nicht nur für Angaben in `Class.forName()` verwendet, sondern auch bei Ausgaben von Arrays mit `toString()` erzeugt. Bei der Notation repräsentiert die Anzahl der `'['` die Array-Dimensionen. Anschließend wird der Typ der gespeicherten Elemente durch ein Buchstabenkürzel gekennzeichnet. Das große `'L'` steht für eine Referenzvariable (Klasse oder Interface). Viele weitere Kürzel sind intuitiv und folgen den Anfangsbuchstaben der Typen, etwa `'B'` für `byte`, `'I'` für `int` usw. Allerdings wird für den Typ `long` aufgrund des Konflikts zur Abkürzung für Referenzen der Buchstabe `'J'` verwendet. Auch für den Typ `boolean` musste mit `'Z'` wegen der Kollision mit `'B'` für `byte` ein anderer Buchstabe genutzt werden. Details finden Sie unter `http://download.oracle.com/javase/8/docs/api/java/lang/Class`.

Der erste Methodenaufruf per Reflection

Beginnen wir die Entdeckungsreise von Reflection mit einem Beispiel. Nehmen wir an, wir wollen einen Methodenaufruf von `equals(Object)` – das ist normalerweise nicht sinnvoll und geschieht hier nur zur Demonstration – dynamisch mit Reflection ausführen. Für Methodenaufrufe sind generell immer folgende Schritte notwendig:

1. Ermitteln der benötigten Informationen zur bereitstellenden Klasse
2. Ermitteln der benötigten Informationen zur gewünschten Methode
3. Dynamischer Aufruf der gewünschten Methode
4. Behandlung von Fehlersituationen

Der durch die obigen Schritte beschriebene Aufrufmechanismus wird in der folgenden Methode `callEquals(Person, Object)` gekapselt:

```
private static boolean callEquals(final Person person, final Object
otherObject)
{
    final String methodName = "equals";
    final Class<?>[] parameterTypes = new Class<?>[] { Object.class };
    final String methodInfo = createMethodInfo(methodName, parameterTypes);

    // Schritt 1: Ermitteln der Klasseninformation
    final Class<?> clazz = person.getClass();
    try
    {
        // Schritt 2: Ermitteln der Methode
```

```
        final Method equalsMethod = clazz.getMethod( methodName,
parameterTypes);

        // Schritt 3: Aufruf der Methode
        final Object[] parameters = new Object[] { otherObject };
        final Object result = equalsMethod.invoke( person, parameters);
        return Boolean.valueOf( result.toString());
    }
    // Schritt 4: Behandlung sämtlicher durch Reflection möglicher Exceptions
    catch ( final NoSuchMethodException e)
    {
        // Es gibt keine solche Methode
        throw new IllegalStateException( clazz.getName() + " does not support "
+
                                         methodInfo);
    }
    catch ( final SecurityException e)
    {
        // Keine Erlaubnis, auf die Methode zuzugreifen
        throw new IllegalStateException( clazz.getName() + " insufficent " +
                              " security rights to access " + methodInfo);
    }
    catch ( final IllegalAccessException e)
    {
        // Kein Zugriff auf die Definition der Methode (im .class-File)
        throw new IllegalStateException( clazz.getName() + " can't access " +
                                         methodInfo);
    }
    catch ( final IllegalArgumentException e)
    {
        // Ungültige Parameter beim Aufruf
        throw new IllegalStateException( clazz.getName() + " parameters are " +
                    "invalid " + methodInfo);
    }
    catch ( final InvocationTargetException e)
    {
        // Ausführung der Methode löst eine Exception aus
        throw new IllegalStateException( clazz.getName() + " exception in " +
                                         methodInfo);
    }
}

private static String createMethodInfo( final String methodName,
                                        final Class<?>[] parameterTypes)
{
    return "method: " + methodName + "( " + Arrays.toString( parameterTypes)
+" ) ";
}
```

Dieses Beispiel verdeutlicht zwei wesentliche Nachteile beim Einsatz von Reflection:

1. **Es entsteht viel Sourcecode:** Hier werden ca. 50 Zeilen benötigt, um lediglich *einen* Methodenaufruf zu kapseln.
2. **Es sind viele unterschiedliche Fehlersituationen zu behandeln:** Bereits in diesem Beispiel wird der Aufwand für die Fehlerbehandlung deutlich, selbst wenn diese wie hier nur rudimentär erfolgt: Exceptions werden lediglich mit Informationen angereichert und als `IllegalStateException` weiter propagiert.

Hinweis: Varianten der Fehlerbehandlung bei Reflection

Oftmals wird man diese unterschiedlichen Fehler nicht für sich einzeln behandeln wollen. Ein `catch (Exception)` ist keine geeignete Alternative. Details dazu beschreibt BAD SMELL: FANGEN DER ALLGEMEINSTEN EXCEPTION in Abschnitt 16.3.4. Mit JDK 7 kann man zwei andere Varianten nutzen. Zum einen gibt es nun eine spezielle Basisklasse `java.lang.ReflectiveOperationException` für alle möglichen Exceptions im Zusammenhang mit Reflection. Zum anderen kann man das Sprachfeature Multi Catch nutzen, wodurch man eine Menge von Exceptions gleichartig behandeln kann. Im Allgemeinen sollte man im Produktivcode auf einzelne Fehler adäquat reagieren – für Reflection gilt dies nur eingeschränkt. Allerdings folgt daraus auch gegenüber diesem Beispiel ein nochmals höherer Aufwand. Eine interessante Diskussion findet man im Internet unter `http://www.javaworld.com/article/2074084/core-java/jdk-7-reflection-exception-handling-with-reflectiveoperation-exception-and-multi-catch.html`.

8.1.2 Zugriff auf Methoden und Attribute

Hat man ein `Class`-Objekt ermittelt, so kann man davon ausgehend weitere Metaobjekte ermitteln, die wiederum Zugriff auf Attribute, Methoden, Konstruktoren, Basisklassen und implementierte Interfaces bieten. Außerdem erhält man Zugriff auf Annotations. Auf diese gehe ich näher im nachfolgenden Abschnitt 8.2 ein.

Methoden ermitteln

Im einführenden Beispiel haben wir ein `Method`-Objekt durch Aufruf der Methode `getMethod()` erhalten. Diese Methode besitzt eine kleine Einschränkung. Sie erlaubt es lediglich, direkt in dieser Klasse definierte, öffentliche Methoden zu ermitteln. Zum Teil sollen aber Methoden anderer, d. h. nicht öffentlicher Sichtbarkeiten oder auch solche von Basisklassen ermittelt und aufgerufen werden. Für beide Fälle ist etwas mehr Programmieraufwand nötig: Sind alle in einer Klasse definierten Methoden unabhängig von der Sichtbarkeit ausfindig zu machen, bietet das `Class`-Objekt die Methode `getDeclaredMethod(Class<?>)`. Sollen alle verfügbaren Methoden aller Klassenbestandteile (also inklusive aller Basisklassen) ermittelt werden, so muss die Methode `getDeclaredMethod(Class<?>)` so lange für die von `getSuperclass()` zurückgelieferte Superklasse aufgerufen werden, bis entweder ein passendes `Method`-Objekt gefunden wurde oder aber die Suche erfolglos verläuft (was spätestens dann der Fall ist, wenn die Klasse `Object` erreicht wurde und dort keine passende Methode existiert, denn dann gibt es keine Superklasse mehr, die durchsucht werden könnte).

Anstatt diese Suchfunktionalität in der eigentlichen Applikation selbst zu

programmieren, sollte man besser eine Hilfsmethode `findMethod(Class<?>, String, Class<?>…)` in einer Utility-Klasse `ReflectionUtils` wie folgt implementieren:

```
public static Method findMethod(final Class<?> clazz, final String methodName,
                                final Class<?>… parameterTypes)
{
    Objects.requireNonNull(methodName, "methodName must not be null");
    Objects.requireNonNull(parameterTypes, "parameterTypes must not be null");

    // Abbruch der Rekursion
    if (clazz == null)
        return null;

    try
    {
        return clazz.getDeclaredMethod(methodName, parameterTypes);
    }
    catch(final NoSuchMethodException ex)
    {
        // rekursive Suche in Superklasse
        return findMethod(clazz.getSuperclass(), methodName, parameterTypes);
    }
}
```

Für den Einsatz in eigenen Applikationen ist die Definition einer weiteren Hilfsmethode `getAllMethods(Class<?>)` praktisch, die Zugriff auf alle in der Klasse und in Superklassen definierten Methoden beliebiger Sichtbarkeiten bietet. Diese nutzt die Methode `getDeclaredMethods()` der Klasse `Class`, um alle verfügbaren Methoden innerhalb eines Klassenbestandteils ausfindig zu machen. Durch den rekursiven Aufruf für alle Superklassen lassen sich dann alle Methoden einer Klasse ermitteln:

```
public static Method[] getAllMethods(final Class<?> clazz)
{
    Objects.requireNonNull(clazz, "class must not be null");

    final List<Method> methods = new ArrayList<>();
    methods.addAll(Arrays.asList(clazz.getDeclaredMethods()));

    if (clazz.getSuperclass() != null)
    {
        // rekursive Suche in Superklasse
        methods.addAll(Arrays.asList(getAllMethods(clazz.getSuperclass())));
    }

    return methods.toArray(new Method[0]);
}
```

Achtung: Fallstrick von `getMethod()`

Gibt man beim Aufruf von `getMethod(String)` versehentlich den Namen der zu ermittelnden Methode mit Klammern an, so kann die Methode nicht gefunden werden! Es wird `null` anstelle eines `Method`-Objekts zurückgeliefert.

Attribute ermitteln

Benötigt man zur Laufzeit Zugriff auf die Attribute einer Klasse, so erhält man über den Namen eines Attributs durch Aufruf der Methode `getField(String)` Zugriff auf ein zugehöriges `Field`-Objekt. Ebenso wie beim Zugriff auf Methoden erfolgt hier eine Einschränkung auf öffentliche Attribute, die in der Klasse selbst definiert sind. Analog kann man mit der Methode `getDeclaredField(String)` Attribute beliebiger Sichtbarkeit bestimmen, die innerhalb der Klasse definiert sind. Eine Suche nach Attributen in Basisklassen kann, wie für Methoden dargestellt, rekursiv programmiert werden. Dies wird hier jedoch nicht nochmals gezeigt.

Hinweis: Nicht intuitive Namensgebung

Die Methoden `getMethod(String)` und `getField(String)` bieten Zugriff nur auf öffentliche Methoden und Attribute. Demnach wären `getPublicMethod()` und `getPublicField()` gelungenere Namen gewesen. Gleiches gilt auch für die Namensgebung der Bulk-Methoden `getMethods()` und `getFields()`, die Zugriff auf alle öffentlichen Methoden bzw. Attribute bieten.

Hilfsfunktionalität zur Ausgabe von Informationen zu Klassen

Nachdem wir nun ein Basiswissen zu Reflection aufgebaut und bereits einige Hilfsmethoden geschrieben haben, wollen wir ein Programm erstellen, das beliebige Klassen untersuchen kann. Es soll Informationen zu Superklassen, implementierten Interfaces, Konstruktoren, Attributen und Methoden sowie die Annotations für diese ausgeben.

Dazu implementieren wir zunächst fünf weitere Hilfsmethoden in unserer Utility-Klasse `ReflectionUtils`:

```
public static void printCtorInfos(final Constructor<?> ctor)
{
    Objects.requireNonNull(ctor, "ctor must not be null");

    System.out.println(Modifier.toString(ctor.getModifiers()) + " " +
                       ctor.getName() +
                       buildParameterTypeString(ctor.getParameterTypes()));
    printAnnotations(ctor.getAnnotations());
}

public static void printMethodInfos(final Method method)
{
    Objects.requireNonNull(method, "method must not be null");

    System.out.println(Modifier.toString(method.getModifiers()) + " " +
                       method.getReturnType() + " " + method.getName() +
                       buildParameterTypeString(method.getParameterTypes()));
    printAnnotations(method.getAnnotations());
}

public static void printFieldInfos(final Field field)
{
```

```
    Objects.requireNonNull(field, "method must not be null");

    System.out.println(Modifier.toString(field.getModifiers()) + " " +
                       field.getType() + " " + field.getName());
    printAnnotations(field.getAnnotations());
}

public static String buildParameterTypeString(final Class<?>[] parameterTypes)
{
    Objects.requireNonNull(parameterTypes, "parameterTypes must not be null");

    if (parameterTypes.length > 0)
        return "(" + Arrays.toString(parameterTypes) + ")";

    return "()";
}

public static void printAnnotations(final Annotation[] annotations)
{
    Objects.requireNonNull(annotations, "annotations must not be null");

    if (annotations.length > 0)
        System.out.println("Annotations: " + Arrays.toString(annotations));
}
```

Beispiel: Untersuchung einer Klasse

Betrachten wir die obigen Hilfsmethoden nun im Einsatz und analysieren exemplarisch die Klasse `String` des JDKs:

```
public static void main(final String[] args)
{
    inspectClass(String.class);
}
private static void inspectClass(final Class<?> clazz)
{
    System.out.println("Untersuchte Klasse: " + clazz.getCanonicalName());
    System.out.println("Superklasse:        " + clazz.getSuperclass());
    System.out.println("Interfaces:         " + Arrays.toString(clazz.
          getInterfaces()));

    // Zugriff und Ausgabe der Konstruktoren
    final Constructor<?>[] ctors = clazz.getDeclaredConstructors();
    System.out.println("\nKonstruktoren: ");
    for (final Constructor<?> ctor : ctors)
        ReflectionUtils.printCtorInfos(ctor);

    // Zugriff und Ausgabe der Attribute
    System.out.println("\nAttribute: ");
    for (final Field field : clazz.getDeclaredFields())
        ReflectionUtils.printFieldInfos(field);

    // Zugriff und Ausgabe aller Methoden
    System.out.println("\nAlle Methoden: ");
    for (final Method method : ReflectionUtils.getAllMethods(clazz))
        ReflectionUtils.printMethodInfos(method);
}
```

Listing 8.2 *Ausführbar als* **'INSPECTIONEXAMPLE'**

Startet man das Programm INSPECTIONEXAMPLE, kommt es zu folgender Ausgabe (gekürzt):

```
Untersuchte Klasse: java.lang.String
Superklasse:        class java.lang.Object
Interfaces:         [interface java.io.Serializable, interface java.lang.
      Comparable, interface java.lang.CharSequence]

Konstruktoren:
public java.lang.String()
public java.lang.String([class [C, int, int])
// …
Attribute:
private final class [C value
private final int offset
private final int count
// …
Alle Methoden:
public boolean equals([class java.lang.Object])
public class java.lang.String toString()
// …
```

Eigenschaften ermitteln

Für Objekte vom Typ `Method` bzw. `Field` liefert die Methode `getModifiers()` Informationen in Form eines `int`-Werts. Bei der leichteren Handhabung dieses Werts hilft die Klasse `java.lang.reflect.Modifier`, die etwa eine menschenlesbare Ausgabe durch Aufruf der statischen Hilfsmethode `toString(int)` erstellt. Die Klasse bietet zudem statische Methoden zum Auslesen einzelner Eigenschaften, etwa der Sichtbarkeit. Ebenso sind Eigenschaften wie statisch oder final oder abstrakt von Interesse. Auch für Multithreading relevante Angaben können ermittelt werden.

- `isPublic(int)`, `isProtected(int)` und `isPrivate(int)`
- `isStatic(int)`, `isFinal(int)` und `isAbstract(int)`
- `isSynchronized(int)` bzw. `isVolatile(int)`

Hinweis: Google Guava als Hilfestellung

Exemplarisch für die Ermittlung von Eigenschaften möchte ich darauf hinweisen, dass Google Guava hier mit der Klasse `Invokable` eine Erleichterung der Lesbarkeit bietet, und dies sowohl für Methoden als auch Konstruktoren, was bei Nutzung der JDK-Möglichkeiten meistens zu Code-Duplikation führt. Das Beispiel entstammt der Guava-Dokumentation: Es soll per Reflection festgestellt werden, ob eine Methode überschreibbar ist. Das Ganze erfordert folgenden Sourcecode mit dem JDK:

```
final boolean isOverridable = !((Modifier.isFinal(method.getModifiers())
                              || Modifiers.isPrivate(method.getModifiers())
                              || Modifiers.isStatic(method.getModifiers())
            || Modifiers.isFinal(method.getDeclaringClass().getModifiers()))
```

Nutzt man dagegen `Invokable` wird daraus kurz und knackig:

```
final boolean isOverridable = Invokale.from(method).isOverridable();
```

8.1.3 Spezialfälle

In diesem Abschnitt werden einige Besonderheiten beim Aufruf von Konstruktoren und bei der Übergabe und Verwendung primitiver Datentypen mit Reflection vorgestellt.

Aufruf von Konstruktoren

Sollen Objektkonstruktionen per Reflection erfolgen, so werden die notwendigen Konstruktoraufrufe durch die Methode `newInstance()` möglich. Dabei sind Aufrufe des Defaultkonstruktors von Aufrufen anderer Konstruktoren zu unterscheiden: Zum Aufruf des Defaultkonstruktors erfolgt ein Aufruf von `newInstance()` ohne Übergabeparameter direkt an das `Class`-Objekt. Der Aufruf eines speziellen Konstruktors ist etwas komplizierter und im Ablauf ähnlich zu einem Methodenaufruf per Reflection. Zunächst wird über die Methode `getDeclaredConstructor(Class<?>...)` der gewünschte Konstruktor mit passender Signatur als `Constructor`-Objekt ermittelt. Anschließend kann dann ein Aufruf an `newInstance(Object...)` dieses `Constructor`-Objekts erfolgen.

Übergabe von primitiven Datentypen an Konstruktoren oder Methoden

Bisher haben wir beim Aufruf von Konstruktoren bzw. Methoden außer Acht gelassen, dass dort häufig primitive Datentypen und nicht nur Referenzvariablen genutzt werden. Zum Auffinden der passenden Konstruktoren bzw. Methoden müssen die Typen der Parameter jedoch als `Class`-Objekt übergeben werden. Zur Beschreibung der passenden Signatur existieren für jeden primitiven Datentyp korrespondierende `Class`-Objekte, etwa `int.class`, `long.class` oder für Arrays z. B. `int[].class`. Variable Argumentlisten, Varargs, werden dabei wie Arrays angegeben.

Beim Aufruf von Konstruktoren oder Methoden können Werte primitiver Typen per Reflection nicht direkt übergeben werden, da die Parameterübergabe in Form von `Object`-Referenzen erfolgt. Daher müssen primitive Datentypen in entsprechende Wrapper-Klassen umgewandelt werden. Das geschieht hier durch Auto-Boxing. Im folgenden Abschnitt werden wir dies auch beim Aufruf von Konstruktoren per Reflection nutzen.

Objektkonstruktion und primitive Typen am Beispiel

Wir wollen das erworbene Wissen nutzen, um verschiedene Konstruktoraufrufe für die Klasse `String` des JDKs auszuführen. Zunächst wird das `Class`-Objekt ermittelt und dessen Defaultkonstruktor aufgerufen. Es soll außerdem der Konstruktor `String(char value[], int offset, int count)` aufgerufen werden.

Zunächst muss ein `Constructor`-Objekt mit passender Signatur gefunden werden. Zur Demonstration erfolgt hier ein Aufruf mit den Übergabewerten `"a Test"` als `char[]` und zwei `int`-Werten. Diese entsprechen dem Startindex 2 sowie der Länge 4. Dadurch wird der Text `"Test"` aus dem `char[]` extrahiert:

```
public static void main(final String[] args)
{
   try
   {
      final Class<?> stringClass = Class.forName("java.lang.String");

      // Aufruf des Defaultkonstruktors
      final String stringInstance1 = (String) stringClass.newInstance();

      // Suche den Konstruktor String(char[], int, int)
      final Class<?>[] parameterTypes = { char[].class, int.class, int.class
};
      final Constructor<?> ctor = stringClass.getDeclaredConstructor(
                                                      parameterTypes);

      // Aufruf des Konstruktors String(char[], int, int)
      final char[] input = { 'a', ' ', 'T', 'e', 's', 't' };
      final String stringInstance2 = (String) ctor.newInstance(input, 2, 4);

      System.out.println("String 1 = '" + stringInstance1 + "'");
      System.out.println("String 2 = '" + stringInstance2 + "'");
   }
   // Behandlung sämtlicher durch Reflection möglicher Exceptions
   catch (final ReflectiveOperationException e)
   {
      throw new IllegalStateException("can't execute constructor: ", e);
   }
}
```

Listing 8.3 *Ausführbar als* **'REFLECTIONCTOREXAMPLE'**

Normalerweise führt der Einsatz von Reflection zu einem erhöhten Aufwand bei der Fehlerbehandlung. In diesem Fall wären sieben verschiedene Typen von Exceptions zu behandeln. Seit JDK 7 ist die `ReflectiveOperationException` verfügbar, die eine One-for-All-Behandlung erlaubt. Das ist aus Gründen der Übersichtlichkeit praktisch und weil man die Fehlersituation für Reflection in der Regel nicht unterscheiden möchte. Für den seltenen Fall, dass man einzelne Exceptions doch spezifisch behandeln möchte, verbleiben mehrere `catch`-Blocke.

Zugriff auf private Dinge und Auslesen von Werten von Attributen

Mit Reflection werden im Normalfall die Sichtbarkeiten beachtet, wie sie im Sourcecode definiert sind. Ist ein Zugriff auf private Attribute oder Methoden anderer Klassen gewünscht, so kann der Sichtbarkeitsschutz durch einen Aufruf von `setAccessible(true)` umgangen werden. Ansonsten führen Zugriffe auf private Attribute oder Methoden zu einer `java.lang.IllegalAccessException`.

Den Aufruf von Konstruktoren und Methoden haben wir bereits kennengelernt. Gleiches gilt auch für den Zugriff auf Attribute. Nun zeige ich, wie man per Reflection auf deren Wertebelegung zugreifen kann. Zunächst erhält man durch einen Aufruf von `getField(String)` Zugriff auf ein `Field`-Objekt, das das namentlich übergebene Attribut repräsentiert. Um den Wert eines Attributs eines speziellen Objekts auszulesen, bietet die Klasse `Field` die Methode `get(Object)`. Als Parameter ist die Referenz des Objekts zu übergeben, dessen Attributwert man ermitteln möchte. Für statische Attribute wird statt einer konkreten Objektreferenz der Wert `null` übergeben.

Folgendes Beispiel zeigt den Zugriff auf ein öffentliches, ein statisches und ein privates Attribut. Zur Demonstration der Abfrage von Eigenschaften und Annotations ist das Attribut `value` als `volatile` und `@Deprecated` gekennzeichnet:

```
public final class AttributeAccessExample
{
    public static long instanceCounter = 0;
    @Deprecated public volatile int value;
    private String description = "Hello World";

    public static void main(final String[] args)
    {
        try
        {
            final AttributeAccessExample obj = new AttributeAccessExample();
            final Class<?> clazz = obj.getClass();

            // Zugriff auf das Attribut 'value'
            final Field field = clazz.getField("value");

            // Zugriff auf den Wert des Attributs 'value'
            final Object attributeValue = field.get(obj);
            System.out.println("value = " + attributeValue);

            // Zugriff auf die Annotation des Attributs 'value'
            ReflectionUtils.printAnnotations(field.getAnnotations());

            // Zugriff auf das statische Attribut 'instanceCounter'
            final Field staticfield = clazz.getField("instanceCounter");

            // Zugriff auf den Wert des statischen Attributs 'instanceCounter'
            final Object staticvalue = staticfield.get(null);
            System.out.println("instanceCounter = " + staticvalue);

            // Zugriff auf das private Attribut 'description'
            final Field field2 = clazz.getDeclaredField("description");

            // Zugriff ermöglichen
```

```
            field2.setAccessible(true);
            // Wertänderung des finalen Attributs 'description'
            field2.set(obj, "Changed FINAL attribute");
            final Object attributeValue2 = field2.get(obj);
            System.out.println("description = " + attributeValue2);
        }
        catch (final ReflectiveOperationException e)
        {
            // Behandlung sämtlicher durch Reflection möglicher Exceptions
            throw new IllegalStateException("can't access field!", e);
        }
        catch (final SecurityException e)
        {
            // Keine Erlaubnis, auf das Attribut zuzugreifen
            throw new IllegalStateException("insufficent security rights to "
+
                                            "access field!", e);
        }
    }
}
```

Listing 8.4 *Ausführbar als* **'ATTRIBUTEACCESSEXAMPLE'**

8.1.4 Type Erasure und Typinformationen bei Generics

Im Internet und auch in Büchern liest man immer wieder mal Aussagen, dass durch die Type Erasure alle Typangaben zur Laufzeit verloren gehen und man darauf keinen Zugriff hat. Richtig ist, dass zur Laufzeit eine Vielzahl an Typangaben nicht mehr zugreifbar sind, so sind tatsächlich keine Typangaben für Containerklassen vorhanden: Eine Liste »weiß« dann nicht mehr, auf welchen Typ sie noch beim Kompilieren eingeschränkt war. Allerdings ist es sowohl für Deklarationen von Variablen als auch für Parameter möglich, deren generische Typen zu ermitteln. Dazu dienen Methoden wie `getTypeParameters()` zum Ermitteln der generischen Parameter einer Klasse oder `getGenericType()` zur Bestimmung des Typs eines Attributs sowie `getGenericReturnType()` und `getGenericParameterTypes()`, die Rückgabeund generische Parametertypen liefern.

Wir werden uns nachfolgend eine Klasse `TypeErasureAndTypeInfoExample` ansehen, die zur Veranschaulichung zwei Attribute und zwei Methoden mit generischen Typenangaben definiert:

```
public class TypeErasureAndTypeInfoExample<A, B, C>
{
   private final String[] infoArray = {"Reflection", "and", "generic",
"types"};
   protected List<String> info = Arrays.asList(infoArray);
   protected Set<SimplePerson> persons =
                               Collections.singleton(new SimplePerson("Tim"));

   // Generische Parameter nur zur Demonstration, hier funktional nutzlos
   public List<String> getInfo(final Map<Integer, SimplePerson> mapping)
```

```
    {
        return this.info;
    }

    // Generische Parameter nur zur Demonstration, hier funktional nutzlos
    public Set<SimplePerson> getPersons(final List<Long> values)
    {
        return this.persons;
    }

    public static void main(final String[] args)
    {
        final Class<?> clazz = TypeErasureAndTypeInfoExample.class;

        // Zugriff auf die Typparameter der Klasse
        System.out.println("getTypeParameters(): " +
                           Arrays.asList(clazz.getTypeParameters()));
        System.out.println();

        System.out.println("Fields:");
        final Field[] fields = clazz.getDeclaredFields();
        for (final Field field : fields)
        {
            System.out.println("getName():         " + field.getName());
            System.out.println("getType():         " + field.getType());
            // Zugriff auf generischen Typ des Attributs
            System.out.println("getGenericType(): " + field.getGenericType());
            System.out.println();
        }

        System.out.println("Methods:");

        final Method[] methods = clazz.getDeclaredMethods();
        for (final Method method : methods)
        {
            System.out.println("getName():                    " +
                               method.getName());
            System.out.println("getReturnType():              " +
                               method.getReturnType());
            // Zugriff auf generischen Rückgabetyp
            System.out.println("getGenericReturnType():       " +
                               method.getGenericReturnType());
            System.out.println("getParameterTypes():          " +
                               Arrays.asList(method.getParameterTypes()));
            // Zugriff auf generische Parametertypen
            System.out.println("getGenericParameterTypes(): " +
                               Arrays.asList(method.getGenericParameterTypes()));
            System.out.println();
        }
    }
}
```

Listing 8.5 *Ausführbar als* **'TYPEERASUREANDTYPEINFOEXAMPLE'**

Führt man das obige Programm TYPEERASUREANDTYPEINFOEXAMPLE aus, so kommt es zu folgenden Ausgaben, die zeigen, dass zur Laufzeit trotz Type Erasure sehr wohl Zugriff auf einige generische Typangaben besteht, und zwar für die Deklaration von Attributen und Parametern:

```
getTypeParameters():    [A, B, C]
```

```
Fields:
getName():         infoArray
getType():         class [Ljava.lang.String;
getGenericType():  class [Ljava.lang.String;

getName():         info
getType():         interface java.util.List
getGenericType():  java.util.List<java.lang.String>

getName():         persons
getType():         interface java.util.Set
getGenericType():  java.util.Set<ch08_advancedjava.reflection.SimplePerson>

Methods:
getName():                    main
// … Infos zu main() ausgelassen …

getName():                    getInfo
getReturnType():              interface java.util.List
getGenericReturnType():       java.util.List<java.lang.String>
getParameterTypes():          [interface java.util.Map]
getGenericParameterTypes(): [java.util.Map<java.lang.Integer,
                              ch08_advancedjava.reflection.SimplePerson>]

getName():                    getPersons
getReturnType():              interface java.util.Set
getGenericReturnType():       java.util.Set<ch08_advancedjava.
                                            reflection.SimplePerson>
getParameterTypes():          [interface java.util.List]
getGenericParameterTypes(): [java.util.List<java.lang.Long>]
```

Fazit

Reflection bietet zusätzliche Möglichkeiten bei der Programmierung. Mit Bedacht eingesetzt können elegante Lösungen entstehen. ***Ist ein direkter Zugriff auf Klassen möglich, sollte dieser jedoch dem Einsatz von Reflection vorgezogen werden***. Dadurch erhöhen sich die Lesbarkeit und die Nachvollziehbarkeit.

Achtung: Typsicherheit und Refactorings

Durch Reflection verliert man Typsicherheit, da diverse Prüfungen nicht mehr zur Kompilierzeit, sondern erst zur Laufzeit erfolgen. Werden `Class`-Referenzen über voll qualifizierte Namen mithilfe der Methode `Class.forName(String)` ermittelt, so kann dies zur Laufzeit Probleme auslösen, wenn Klassen umbenannt oder in andere Packages verschoben werden. Als Strings definierte Klassennamen werden von den Refactoring-Automatiken der IDEs in der Regel nicht erkannt und daher auch nicht angepasst.

8.2 Annotations

Manchmal ist es wünschenswert, den Sourcecode mit Zusatzinformationen, sogenannten Metainformationen, also ergänzenden Informationen zum Sourcecode

selbst, zu versehen. Eine mögliche Form besteht darin, Kommentare im Sourcecode zu hinterlegen, die gewisse Abläufe, Methoden, Parameter usw. beschreiben. Folgen diese Kommentare dem Javadoc-Stil, so kann man daraus mithilfe des Javadoc-Tools des JDKs eine Programmdokumentation erstellen. Dazu werden spezielle Teile und relevante Informationen aus den Kommentaren extrahiert.

Natürlich ist man bei der Kommentierung nicht auf die reine Beschreibung des Sourcecodes beschränkt, sondern man kann in Kommentaren auch Angaben etwa zu Konfigurationseinstellungen oder Datenbankverbindungsparametern hinterlegen. Genau dies wurde im Laufe der Zeit erkannt und vermehrt eingesetzt, nachdem neben dem Javadoc-Tool weitere Tools entwickelt wurden, die die Auswertung von in Kommentaren angegebenen Informationen erlaubten. Beispielsweise ist es mit dem Tool XDoclet möglich, spezielle Angaben in Kommentaren zur Sourcecode-Generierung zu nutzen. Dies war besonders für die EJB-2-Spezifikation hilfreich. Diese schreibt nämlich vor, dass sogenannte Session Beans verschiedene Klassen und Interfaces erfüllen müssen, die jeweils einem gleichartigen Aufbau folgen. Diese Artefakte waren mühselig und fehleranfällig von Hand zu erstellen. Durch den Einsatz von XDoclet konnte dieser Prozess automatisiert werden.

Neben der Generierung von Sourcecode oder anderer Dateien sind aber weitere Einsatzzwecke denkbar. Man kann auch Parametrierungen oder Konfigurationseinstellungen aus den Angaben im Sourcecode auslesen. Diese Variante ist oftmals einfach praktischer, als separate Metainformationsdateien zu verwenden (z. B. unübersichtliche XML-Konfigurationsdateien).

Die ganze Sache hat aber einen Haken, da beim Einsatz von Metainformationen in Kommentaren folgendes Problem existiert: Die Angaben gehen beim Kompilieren verloren und stehen zur Laufzeit (in der `.class`-Datei) nicht mehr zur Verfügung. Insbesondere für Tools und Laufzeitumgebungen, wie z. B. Applikationsserver, ist es aber wichtig, auf diese Zusatzinformationen auch zur Laufzeit zugreifen zu können. Genau das wird durch die mit JDK 5 neu eingeführten Annotations möglich. Mit diesen lassen sich die Metainformationen direkt mit dem Sourcecode (und nicht separat) pflegen und somit häufig Konfigurationsaufwände reduzieren.

8.2.1 Einführung in Annotations

Seit JDK 5 existieren Annotations als neue Java-Sprachelemente. Eine Annotation beginnt immer mit einem `'@'`-Zeichen und wird vor demjenigen Programmelement (z. B. Klasse, Methode, Attribut) notiert, auf das sie sich bezieht. Im folgenden Beispiel wird die Annotation `@CreationInfo` an der Klasse `MyClass` sowie die Annotation `@Override` an der Methode `checkValues(String, int)` genutzt. Für die Annotation `@CreationInfo` sehen wir, dass Informationen als benannte Parameter (hier `author`, `description`) in Form einer kommaseparierten Angabe von Schlüssel-Wert-Paaren bereitgestellt werden können:

```
@CreationInfo( author = "Michael Inden",
```

```
                description = "This class is responsible for XYZ"
)
public class MyClass
{
    void checkValues(final String name, final int age)
    {
        …
    }
}
```

Anhand der einleitenden Beschreibung und des obigen Beispiels kann man sich verschiedene Einsatzgebiete für Annotations vorstellen:

1. **Informationsbereitstellung für Compiler** – Im JDK sind diverse Annotations definiert. Einige davon erlauben es uns Entwicklern, dem Compiler Hinweise über das Programm selbst bereitzustellen. So kann z. B. mithilfe der Annotation `@Override` ausgedrückt werden, dass eine Methode einer Basisklasse überschrieben werden soll. Der Compiler kann diesen Hinweis nutzen, um mögliche Verstöße beim Überschreiben, etwa durch einen Tippfehler im Methodennamen, erkennen zu können.
2. **Informationsbereitstellung für Tools** – Die zuvor im Listing genutzte Annotation `@CreationInfo` kann beispielsweise von einem speziellen Tool, einem sogenannten ***Annotation Processor***, ausgewertet werden. Diese besitzen die Basisklasse `javax.annotation.processing.AbstractProcessor` und können beim Kompilieren angegeben werden. Auf diese Weise kann man die in der Annotation hinterlegten Angaben zum Autor (`author`) und zur Beschreibung (`description`) auslesen und zu einer Implementierungsdokumentation zusammentragen. Sinnvollerweise würde man dies über Javadoc regeln, hier soll das Beispiel lediglich als Idee einen möglichen Anwendungsfall verdeutlichen. Normalerweise dienen Annotations als Eingabe für Tools, die daraus beispielsweise Konfigurationsdateien, Datenbankabfragen oder sogar Sourcecode generieren können.
3. **Informationsbereitstellung zur Laufzeit** – Annotations und zugehörige Werte von Parametern kann man zur Laufzeit über Reflection (vgl. Abschnitt 8.1) auslesen. Das kann z. B. zur Umsetzung von Cross Cutting Concerns, wie Transaktionssteuerung, Logging usw., genutzt werden. Annotations können zudem als Hilfsmittel dienen, um Abhängigkeiten aufzulösen und Referenzen an annotierte Attribute zuzuweisen. Für beide Anwendungsfälle müssen Annotations von der jeweiligen Laufzeitumgebung (Webcontainer, Applikationsserver etc.) ausgelesen und verarbeitet werden. Mit der Annotation `@Inject` wird etwa eine Instanz einer Enterprise Java Bean (kurz EJB) in eine Referenzvariable einer Java-Klasse injiziert.

Nicht alle Annotations sind per se für jeden der oben genannten Anwendungsfälle nutzbar, da Annotations verschiedene Lebensdauern haben. Einige sind nur während des Kompilierens verfügbar, andere sogar noch zur Laufzeit des Programms: Abhängig von ihrer Definition verwirft oder überträgt der Java-Compiler die Annotations bei der Übersetzung der Source-Datei in die `.class`-Datei. Beim

Laden der Klassen in die JVM werden annotierte Informationen nur dann geladen, wenn die Lebensdauer in der Definition entsprechend festgelegt wurde. Darauf gehe ich später bei der Beschreibung der Definition der selbst definierten Annotation `@CreationInfo` ein.

8.2.2 Standard-Annotations des JDKs

Wie eingangs erwähnt, ist ein Anwendungsfall von Annotations die Bereitstellung von Informationen für den Compiler. Die Annotations `@Deprecated`, `@Override` und `@SuppressWarnings` sind im Package `java.lang` definiert und werden während der Kompilierung ausgewertet. Diese Annotations haben wir bereits verwendet, ohne uns dabei viele Gedanken zu machen. Rekapitulieren wir deren Einsatzzweck:

@Deprecated

Diese Annotation zeigt an, dass ein markiertes Element (Klasse, Methode, …) veraltet ist und nicht mehr benutzt werden sollte. Wird eine derart markierte Methode eingesetzt, so kommt es beim Kompilieren zu Warnungen. Zusätzlich zu dieser Annotation sollte der Hinweis `@deprecated` im Javadoc-Kommentar genutzt werden. Ergänzend kann dort auf mögliche alternativ zu nutzende Klassen oder Methoden eingegangen werden:

```
/**
* @deprecated Diese Methode führt zu Problemen, Bitte stattdessen
*             {@link #newMethod()} nutzen.
*/
@Deprecated
public void oldMethod(int someValue)
```

@Override

Diese Annotation drückt aus, dass eine damit annotierte Methode eine gleichnamige Methode einer Basisklasse überschreibt bzw. eines Interface implementiert. Obwohl sich dieser Einsatz womöglich nicht sinnvoll anhören mag, gibt es doch gute Gründe dafür. Durch Angabe dieser Annotation gleicht der Compiler die Signatur einer Methode mit derjenigen einer Basisklasse ab und warnt, wenn kein Überschreiben der gekennzeichneten Methode vorliegt. Auf diese Weise kann man Fehler finden, wenn man sich beim Methodennamen vertippt. Statt – wie gewollt – die Methode der Basisklasse zu überschreiben, würde man jedoch eine neue Methode definieren. Dieser Fehler ist ohne den Einsatz der Annotation `@Override` schwieriger zu entdecken und könnte möglicherweise zu ungewünschtem Programmverhalten führen.

Der Einsatz der Annotation `@Override` ist beispielsweise bei der Definition einer `equals()` -Methode in eigenen Klassen nützlich, um zu verhindern, dass deren Parameter fälschlicherweise nicht vom Typ `Object` ist und die Methode dann

überladen anstatt überschrieben wird:

```
@Override // => Compile-Error
public void equals(final Person person)
{
    …
}
```

@SuppressWarnings

Diese Annotation erlaubt es, Compiler-Warnungen zu unterdrücken. Der zu unterdrückende Typ von Warnung wird als Parameter übergeben, etwa `@SuppressWarnings("unchecked")`. Um Fehler zu entdecken und nicht zu verstecken, sollte man nur ausnahmsweise und im kleinstmöglichen Scope Gebrauch von `@SuppressWarnings` machen. Wichtige vordefinierte Parameterwerte sind:

- `boxing` – Es werden keine Warnungen für Typumwandlungen mit Auto-Boxing und Auto-Unboxing erzeugt.
- `deprecation` – Beim Einsatz veralteter Methoden oder Klassen kommt es nicht zu Warnungen.
- `unused` – Unbenutzte Variablen oder Methoden führen nicht zu Warnungen. Das kann beim Einsatz externer Bibliotheken hilfreich sein, um deren Warnungen auszublenden.
- `unchecked` – Wenn Generics und untypisierte Klassen kombiniert eingesetzt werden und somit durch den Compiler keine Typsicherheit mehr garantiert werden kann, wird beim Einsatz dieser Annotation für möglicherweise problematische Zugriffe keine Warnung erzeugt. Das sollte man aber mit großer Vorsicht und nur nach genauer Prüfung der Zugriffe verwenden.

Das folgende (Negativ-)Beispiel zeigt einige Annotations für eine Methode `getPersons()`, die noch keine Generics nutzt, deren Rückgabe jedoch für eine typsichere Iteration in der `main()`-Methode eingesetzt werden soll. Durch die Nutzung der zuvor genannten Annotations kompiliert folgender Sourcecode ohne Warnungen:

```
@SuppressWarnings("unchecked")
public static void main(String[] args)
{
    // Achtung: nur zur Demonstration der Möglichkeiten von Annotations
    @SuppressWarnings(value={"unchecked", "deprecation"})
    final List<Person> persons = getPersons();
    for (final Person currentPerson : persons)
    {
        doSomethingWithPerson(currentPerson);
    }
}

@Deprecated
public static List getPersons()
{
```

```
    return new ArrayList();
}
```

Beachten Sie bitte, dass es in diesem Beispiel wirklich nur um die Darstellung der Nutzung von Annotations geht. In der Praxis sollte man besser die Methode `getPersons()` überarbeiten, statt die Fehler mithilfe von Annotations auszublenden.

In diesem Listing lernen wir ein weiteres Feature kennen: Es kann nicht nur ein zu unterdrückender Warnungstyp, sondern auch eine Aufzählung zu unterdrückender Warnungstypen angegeben werden. Dazu besitzt die Annotation einen impliziten logischen Parameter namens `value`. Diesem kann – eine entsprechende Definition der Annotation vorausgesetzt – auch eine kommaseparierte Folge von Strings in der Notation `value={"Wert1", ..., "Wert n"}` übergeben werden. Heißt das Attribut `value` und ist es das einzige, braucht man es nicht anzugeben. Man notiert kürzer Folgendes: `@SuppressWarnings("unchecked", "deprecation")`.

8.2.3 Definition eigener Annotations

Im Programmiereralltag setzt man die zuvor beschriebenen vordefinierten Annotations ein, ohne sich (viele) Gedanken darüber zu machen, wo diese Annotations definiert sind und vor allem, wie sie ausgewertet werden. Tatsächlich ist diese Spezialkenntnis auch eher selten wichtig.

Zum besseren Verständnis der Arbeitsweise von Annotations lohnt sich allerdings ein Blick hinter die Kulissen. Annotations werden in eigenen Dateien analog zu normalen Klassen und Interfaces definiert. Statt des Schlüsselworts `class` bzw. `interface` wird hier `@interface` verwendet. Das sorgt automatisch dafür, dass die Annotation den Basistyp `Annotation` aus dem Package `java.lang.annotation` besitzt.

Betrachten wir die Realisierung unserer im einführenden Beispiel kennengelernten Annotation `@CreationInfo`. Dieser wurden verschiedene Informationen in Form von Parametern übergeben. In der Annotation-Definition ist für jeden Parameter eine parameterlose Methode gleichen Namens zu erstellen:

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

// Meta-Annotations
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)

// Annotation-Definition
public @interface CreationInfo
{
    // Methoden zur Übergabe von Informationen durch Parameter
    String author() default "Michael Inden";
```

```
    String description();
    Class<?>  baseclass() default java.lang.Object.class;
    Class<?>[] interfaces() default {};
    String[] tags() default {};
}
```

Das ist tatsächlich alles an Sourcecode zur Definition dieser eigenen Annotation! Allerdings sehen wir hier einige Schreibweisen und Elemente, die uns in dieser Form oder aus der bisher vorgestellten Java-Syntax (noch) nicht geläufig sind. Das sind vor allem das Schlüsselwort `default` sowie die beiden Annotations `@Retention` und `@Target`: Mit `@Retention(RetentionPolicy.RUNTIME)` wird für Annotations festgelegt, dass sie später zur Laufzeit verfügbar sein sollen. Über die Angabe von `@Target(ElementType.TYPE)` wird festgelegt, dass beliebige Typen, d. h. Klassen, Interfaces, Annotations und Enums, markiert werden können.

Konzentrieren wir uns hier zunächst auf die Methoden, die in der Annotation-Definition aufgeführt sind. Durch diese werden die Parameter einer Annotation festgelegt, die bei Verwendung der Annotation im Sourcecode angegeben werden müssen bzw. können (im Falle von Defaultparametern mit dem Schlüsselwort `default`), um Informationen zu übermitteln. Der Datentyp der Parameter wird durch deren Rückgabetyp bestimmt. Dabei dürfen als Einschränkung nur primitive Typen, Strings, Klassentypen, Enums und Annotation-Typen bzw. eindimensionale Arrays der zuvor genannten Typen als Rückgabewert genutzt werden. Achten Sie zudem darauf, dass die Methoden immer parameterlos anzugeben sind.

Zur einfacheren Handhabung beim späteren Einsatz dieser Annotation im Sourcecode wurden hier durch Angabe des Schlüsselworts `default` und eine korrespondierende Wertangabe einige Defaultparameter definiert. Bei der Verwendung der Annotation können derartige Parameter einen Übergabewert erhalten, müssen es aber nicht. Fehlt bei der Angabe im Sourcecode der Wert eines solchen Parameters, so wird der in der Definition der Annotation hinterlegte Defaultwert genommen. Alle anderen Parameter sind erforderlich und müssen angegeben werden. Das bedeutet insbesondere, dass eine fehlende Angabe eines Parameters ohne Defaultwert im Sourcecode zu Kompilierfehlern führt.

Um Fehler zu vermeiden und vollständige Angaben zu den Parametern einer Annotation beim Editieren des Sourcecodes machen zu können, ist die Auto-Complete-Funktionalität in Eclipse sehr hilfreich. Diese gibt kontextbezogene Hinweise. Das ist in Abbildung 8-1 für unsere selbst definierte Annotation `@CreationInfo` dargestellt.

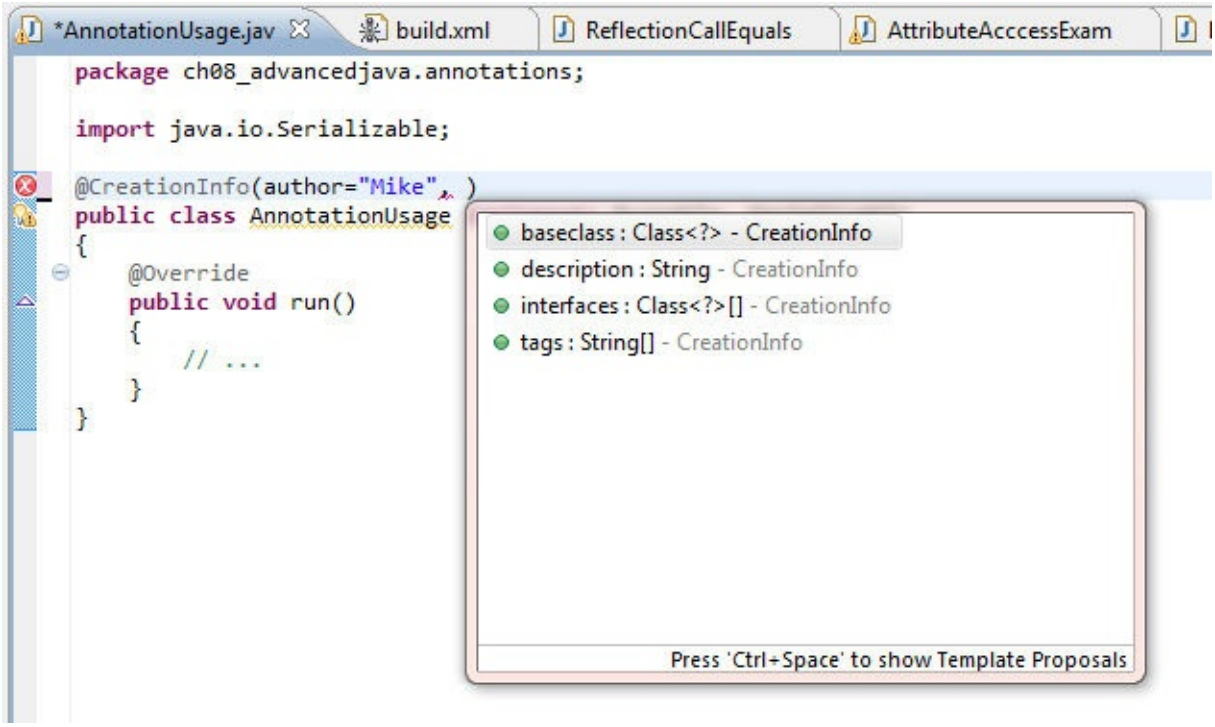


Abbildung 8-1 *Auto-Complete-Funktionalität für Annotations in Eclipse*

Meta-Annotations (`@Retention` und `@Target`)

Bei der obigen Definition der Annotation `@CreationInfo` haben wir weitere Elemente kennengelernt: Meta-Annotations, die vor der eigentlichen Definition der Annotation `@CreationInfo` notiert sind. Über die Meta-Annotations `@Retention` und `@Target` werden Informationen bzw. Eigenschaften der hier definierten Annotation `@CreationInfo` festgelegt. Dabei kann man folgende Angaben machen:

- **`@Retention`** – Legt die Lebensdauer einer Annotation fest. Diese wird als Parameter durch eine sogenannte `RetentionPolicy` spezifiziert:
  - `Source`: Die Annotation ist *nur* im Sourcecode vorhanden und wird *nicht* in den Bytecode, die `.class`-Datei, übernommen. Nach der Kompilierung stehen die Annotation-Informationen demnach nicht mehr zur Verfügung.
  - `Class`: Die Annotation wird in die generierte `.class`-Datei übernommen. Zur Laufzeit stehen die Informationen aber nicht mehr zur Verfügung. ***Das ist die Defaulteinstellung***.
  - `Runtime`: Die Annotation ist während der Laufzeit verfügbar. Auf sie kann dann per Reflection zugegriffen werden. Der folgende Abschnitt 8.2.4 geht darauf genauer ein.
- **`@Target`** – Über diese Meta-Annotation wird geregelt, welche Elemente mit der Annotation markiert werden dürfen. Das wird über ein Element vom Typ `ElementType` genauer spezifiziert. Folgende selbsterklärende Werte sind erlaubt: `ANNOTATION_TYPE`, `TYPE`, `CONSTRUCTOR`, `METHOD`, `FIELD`, `PARAMETER` und `PACKAGE`. Für `TYPE` sei angemerkt, dass dieser für beliebige

Typen, also für Klassen, Interfaces, Annotations sowie Enums, steht. Mit `ANNOTATION_TYPE` beschreibt man, dass die Annotation nur für Annotations genutzt werden darf.

8.2.4 Annotation zur Laufzeit auslesen

Die Definition der Annotation `@CreationInfo` ist nun bekannt. Jetzt wollen wir die Annotation im Einsatz betrachten. Als Beispiel dient eine einfache Klasse `AnnotationUsage`, die die Interfaces `Runnable` und `Serializable` implementiert und eine direkte Subklasse von `Object` ist. Diese Informationen wollen wir mit unserer Annotation `@CreationInfo` beschreiben und kommen zu folgender Realisierung:

```
@CreationInfo(author="Mike",
              description="Demonstration einer eigenen Annotation",
              baseclass=java.lang.Object.class,
              interfaces={java.lang.Runnable.class,
                          java.io.Serializable.class},
              tags={"Annotation", "Definition", "Advanced Java"})
public class AnnotationUsage implements Runnable, Serializable
{
    @Override
    public void run()
    {
        // …
    }
}
```

Interessant ist die Syntax zur Angabe der Array-Parameter `interfaces` und `tags`. Diese erfolgt in geschweiften Klammern als kommaseparierte Liste, die Objekte vom Typ `Class<?>` bzw. im zweiten Fall Objekte vom Typ `String` enthält.

Die Annotation `@CreationInfo` haben wir zur Beschreibung von Metainformationen über die Klasse `AnnotationUsage` eingesetzt. Nun wollen wir die dort hinterlegten Informationen mithilfe eines Java-Programms auslesen. Dazu nutzen wir hier Reflection – im Speziellen die Methode `getAnnotation(Class<A extends Annotation>)`, um eine Instanz unserer Annotation `@CreationInfo` zu erhalten und deren Methoden aufzurufen. Wie im vorherigen Abschnitt über Reflection bereits beschrieben, könnte man viele der Informationen auch ohne die Annotation auslesen, z. B. diejenigen über implementierte Interfaces – zudem wären die Angaben auch verlässlicher, falls mal eine Änderung nicht in der Annotation nachgezogen wird. Im folgenden Listing betrachten wir nun trotzdem eine programmatische Umsetzung mithilfe von Reflection:

```
public static void main(final String[] args) throws Exception
{
   // Auslesen des selbst definierten Annontationstyps CreationInfo
   final CreationInfo creationInfo = AnnotationUsage.class.
                                          getAnnotation(CreationInfo.class);

   if (creationInfo != null)
```

```
    {
        printCreationInfo(creationInfo);
    }
    else
    {
        System.out.println("No " + CreationInfo.class.getSimpleName() +
                           " annotation present!");
    }
}
private static void printCreationInfo(final CreationInfo creationInfo)
{
    System.out.println("author():       " + creationInfo.author());
    System.out.println("description():  " + creationInfo.description());
    System.out.println("baseclass():    " + creationInfo.baseclass());
    System.out.println("interfaces():   " + Arrays.toString(creationInfo.
          interfaces()));
    System.out.println("tags():         " +
Arrays.toString(creationInfo.tags()));
}
```

Listing 8.6 *Ausführbar als* **'ANNOTATIONREADEREXAMPLE'**

Führen Sie das Programm ANNOTATIONREADEREXAMPLE aus, so wird in etwa folgende Ausgabe auf der Konsole erscheinen:

```
author():       Mike
description():  Demonstration einer eigenen Annotation
baseclass():    class java.lang.Object
interfaces():   [interface java.lang.Runnable, interface java.io.Serializable]
tags():         [Annotation, Definition, Advanced Java]
```

Fazit

Dieser Abschnitt hat eine kurze Einführung in die Definition und Verarbeitung von Annotations gegeben. Das erlangte Wissen ist für den normalen Programmiereralltag vollkommen ausreichend. Der Gebrauch eigener Annotations ist eher unüblich – für Tool-Hersteller oder fortgeschrittene Entwickler kann diese Thematik jedoch von großem Interesse sein. Am Kapitelende finden Sie Hinweise auf weiterführende Literatur.

8.3 Serialisierung

Java bietet einen Automatismus, der Objekte in eine Folge von Bytes umwandeln bzw. Objekte aus einer solchen erzeugen kann. Diese Vorgänge nennt man in Java ***Serialisierung*** bzw. ***Deserialisierung***. Vereinfachend wird im Folgenden für beides der Begriff Serialisierung verwendet. Über das Marker-Interface `java.io.Serializable` werden Klassen als serialisierbar markiert. Damit wird es der JVM möglich, Objekte in einem beliebigen Ausgabestream zu speichern und aus einem Eingabestream später wieder einzulesen. Die Serialisierung ist vom Betriebssystem unabhängig und ermöglicht dadurch den Austausch von Daten bzw. Objekten zwischen JVMs auf verschiedenen Rechnern.

Aufgrund der genannten Eigenschaften kann man Serialisierung dazu nutzen, um

Objekte über ein Netzwerk zu transportieren. Die Technik eignet sich auch dazu, den momentanen Objektzustand in einer Datei persistent zu speichern. Dieser kann dann später wieder eingelesen werden und lässt sich so z. B. für Undo-Operationen nutzen.

8.3.1 Grundlagen der Serialisierung

Betrachten wir zunächst ein einfaches Beispiel. Dort wird das Speichern und Laden von `Person`-Objekten in einer Datei durch den Einsatz von Serialisierung realisiert. Dazu verwenden wir die Klassen `ObjectInputStream` und `ObjectOutputStream`. Diese bieten folgende Methoden:

- `writeObject(Object)` – Schreibt ein Objekt in einen `ObjectOutputStream`.
- `Object readObject()` – Liest ein Objekt aus einem `ObjectInputStream`

Doch damit diese Methoden korrekt arbeiten können, müssen die zu speichernden bzw. lesenden Klassen das Marker-Interface `Serializable` implementieren.

Klassen als serialisierbar markieren

Gemäß der Einleitung muss die bereits bekannte Klasse `Person` nur derart erweitert werden, dass sie das Interface `Serializable` implementiert:

```
public final class Person implements Serializable
{
    private final String name;
    private final String city;
    private final Date birthday;

    public Person(final String name, final String city, final Date birthday)
    {
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.city = Objects.requireNonNull(city, "city must not be null");
        this.birthday = Objects.requireNonNull(birthday, "birthday must " +
                                                         "not be null");
    }

    // …
```

Die Klassen `ObjectInputStream` und `ObjectOutputStream`

Nehmen wir an, wir wollten ein `Person`-Objekt in einer Datei `Test.ser` speichern. Die Ausgabe in die Datei nutzt einen `FileOutputStream`. Die Konvertierung in eine Folge von Bytes und das Speichern des Objekts in diesen Stream wird durch die Klasse `ObjectOutputStream` und dessen Methode `writeObject(Object)` durchgeführt:

```
public static void main(final String[] args) throws IOException
```

```
{
   // Nutze ARM, um lesbare Ressourcenzugriffe zu schreiben
   try (final ObjectOutputStream objectOutStream = new ObjectOutputStream(
               new BufferedOutputStream(new FileOutputStream("Test.ser"))))
   {
      // Schreibe Objekt in die Datei
      final Person person = new Person("Test", "TestCity", new Date());
      objectOutStream.writeObject(person);
      System.out.println("Wrote to stream: " + person);
   }
}
```

Listing 8.7 *Ausführbar als* **'SERIALIZATIONEXAMPLE'**

An diesem Beispiel erkennt man gut, wie wenig Aufwand die Speicherung auf Seite der Applikation erfordert – insbesondere, wenn man, wie im Listing gezeigt, das mit JDK 7 eingeführte Sprachfeature ARM zum automatischen Schließen von Ressourcen nutzt. Tatsächlich wird die gesamte Funktionalität in dem Methodenaufruf `writeObject(person)` gekapselt. Auf diese Weise Daten in den Stream zu schreiben, stellt gegenüber dem Einsatz von Methoden der Klasse `DataOutputStream` einen enormen Fortschritt dar. Jene Art der Verwendung haben wir bereits in Abschnitt 4.6.2 besprochen und dort erfahren, dass für jedes Attribut eine typabhängige `write()`-Methode, etwa `writeLong(long)`, aufgerufen werden muss, um die Daten in den Stream zu schreiben. Allerdings stehen nicht für jeden Datentyp passende `write()`-Methoden bereit. Dies erfordert unter Umständen eine vom Entwickler implementierte Umwandlung der Attribute in eine Folge von Bytes.

Als Nächstes betrachten wir das Deserialisieren zum Einlesen der gespeicherten Daten aus der gerade geschriebenen Datei `Test.ser`. Wir nutzen einen `FileInputStream` zum Dateizugriff und die Methode `readObject()` der Klasse `ObjectInputStream` zur Rekonstruktion eines Objekts. Diese Methode liest alle dazu benötigten Informationen aus einem beliebigen Stream ein. Da die Methode nur eine Referenz vom Typ `Object` zurückliefert, ist für ein sinnvolles Weiterarbeiten mit dem Objekt in der Regel ein anschließender Cast auf den erwarteten Typ notwendig – vorausgesetzt, man kennt den Typ des eingelesenen Objekts:

```
public static void main(final String[] args) throws Exception
{
   try (final ObjectInputStream objectInStream = new ObjectInputStream(
               new BufferedInputStream(new FileInputStream("Test.ser"))))
   {
      // Rücklesen des Objekts, ohne Konstruktoraufruf
      final Person personFromStream = (Person) objectInStream.readObject();
      System.out.println("Back from stream: " + personFromStream);
   }
}
```

Listing 8.8 *Ausführbar als* **'DESERIALIZATIONEXAMPLE'**

So einfach die Serialisierung in diesem Beispiel scheint, so komplex sind teilweise die Zusammenhänge und Abläufe im Hintergrund. Im Datenformat der Serialisierung müssen alle diejenigen Informationen zu Typen und Werten enthalten sein, die ein späteres Wiederherstellen eines Objekts ermöglichen. Beim späteren Einlesen muss

ein Objekt mit seinen Attributen aus den gespeicherten Informationen rekonstruiert werden – mit gleichen Inhalten, allerdings anderen Referenzen. Wenn man die beiden Programme SERIALIZATIONEXAMPLE und DESERIALIZATIONEXAMPLE nacheinander ausführt, erkennt man dies anhand der unterschiedlichen Referenzen, die für die beiden `Person`-Objekte auf der Konsole beim Speichern und Einlesen ausgegeben werden:

```
Person: [name=Test, …] / ch08_advancedjava.serializable.Person@8b2fd8f
Person: [name=Test, …] / ch08_advancedjava.serializable.Person@2626d4f1
```

Den Serialisierungsvorgang, d. h. die Konvertierung in eine und aus einer Folge von Bytes sowie den Transfer in und aus Streams, selbst zu programmieren wäre fehleranfällig und aufwendig. Die Serialisierungsautomatik nimmt einem Programmierer dabei viel Arbeit ab. Wie bereits bekannt, muss man lediglich das Interface `Serializable` implementieren. Diese Vereinfachung hat allerdings ihren Preis – das verwendete Datenformat ist recht »gesprächig«: Tatsächlich werden in den Datenstrom die Klassennamen und die Typen in Form voll qualifizierter Angaben gespeichert. Über die reinen Nutzinformationen hinaus entsteht ein gewisser Overhead: Die Datei `Test.ser` wird beispielsweise 188 Bytes groß. Das erscheint doch relativ viel, wenn man bedenkt, dass lediglich zwei kurze Texte und ein Datumswert gespeichert werden. Bei komplexeren Objekten entsteht ein erheblicher Overhead. Das kann negative Auswirkungen auf die Performance haben. Bevor wir einige Optimierungsmöglichkeiten bei der Speicherung kennenlernen, betrachten wir zunächst weitere Grundlagen zur Serialisierung.

Der Serialisierungsvorgang im Detail

Bei der Serialisierung eines Objekts werden alle nicht statischen[3] Attribute der Klasse (*einschließlich der privaten*) verarbeitet. Allerdings werden explizit mit dem Schlüsselwort `transient` gekennzeichnete Attribute nicht berücksichtigt. Der Automatismus der Serialisierung kann »von Hause aus« alle primitiven Datentypen in einen Stream schreiben. Damit Objektreferenzen verarbeitet werden können, müssen diese ebenfalls das Interface `Serializable` implementieren. Dies ist für viele Klassen des JDKs der Fall. Für Arrays gilt Ähnliches: Wenn ein Array serialisierbar sein soll, müssen auch die gespeicherten Elemente serialisierbar sein. Ansonsten kommt es während des Serialisierungsvorgangs zu einer `java.io.NotSerializableException`. Im einführenden Beispiel haben wir diese Eigenschaft der Klassen `String` und `Date` genutzt, ohne diese Details zu kennen.

Nicht serialisierbare Attribute und `transient`

Eben erwähnte ich kurz, dass man den expliziten Ausschluss eines Attributs von der automatischen Verarbeitung durch die Angabe des Schlüsselworts `transient` erreicht. Jedes derart gekennzeichnete Attribut wird nicht in den Datenstrom

geschrieben und auch bei einem späteren Einlesen nicht berücksichtigt. Um solche Attribute korrekt zu initialisieren, ist daher beim Einlesen ein spezielles Vorgehen zur Initialisierung erforderlich. Vergisst man dies, so bleiben die Attribute mit ihrem Defaultwert belegt, d. h. `false` für boolesche Variablen, 0 für Zahlen und `null` für Referenzen. Auf Abhilfen gehe ich gleich genauer in Abschnitt 8.3.2 ein.

Serialisierung der Klassenhierarchie

Der Vorgang der Serialisierung wird für alle Bestandteile der Klassenhierarchie durchgeführt. Wie bereits beschrieben, werden referenzierte Objekte wiederum komplett serialisiert. Es wird also der gesamte Objektgraph in den Stream geschrieben. Die Serialisierungsautomatik sorgt allerdings dafür, dass mehrfach referenzierte Objekte nur einmal serialisiert werden. Das Einlesen geschieht ähnlich: Solange alle Attribute und Bestandteile der Klassenhierarchie ***serialisierbar*** sind, erledigt der Automatismus sämtliche Aufgaben. Es erfolgt insbesondere ***kein*** Konstruktoraufruf. Ein Spezialfall ist jedoch zu beachten: Ist eine Basisklasse nicht serialisierbar, so kommt es während der Rekonstruktion unter Umständen zu Exceptions. Betrachten wir dies genauer.

Spezialfall: Nicht serialisierbare Basisklassenbestandteile

Es ist möglich, dass eine Klasse das Interface `Serializable` implementiert, eine Basisklasse dies jedoch nicht tut. In Abbildung 8-2 ist eine solche Situation für die Klasse `SerializableClass` gezeigt.

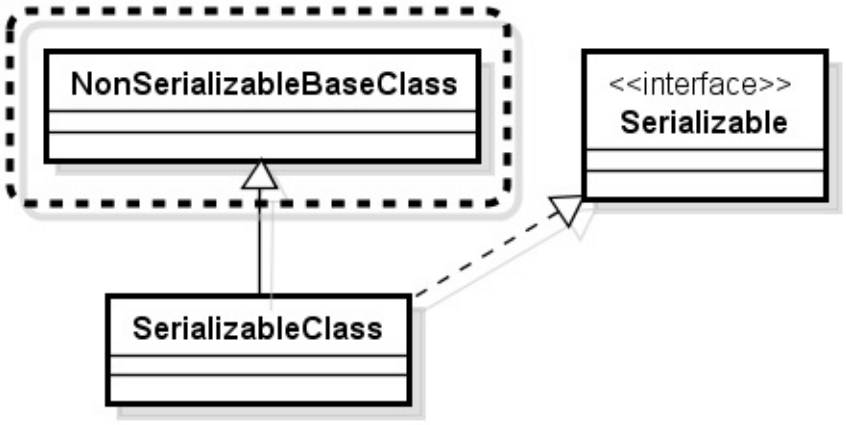


Abbildung 8-2 *Spezialfall: Nicht serialisierbare Basisklassenbestandteile*

Zur Rekonstruktion eines nicht serialisierbaren Basisklassenbestandteils erfolgt beim Deserialisierungsvorgang – im Gegensatz zum Standardfall – ein Konstruktoraufruf: Der Defaultkonstruktor wird dazu genutzt, um den Initialisierungsschritt für diesen Klassenbaustein vorzunehmen. Gibt es keinen Defaultkonstruktor, so kommt es beim Deserialisieren zu einer `java.io.InvalidClassException` mit dem Hinweis „`no valid constructor`". Man fragt sich nun, wieso sich ein derartiges Objekt überhaupt serialisieren lässt und erst beim Einlesen Probleme auftreten. Die Antwort ist einfach: Der Automatismus kann über Reflection zwar alle zur Speicherung notwendigen Bestandteile ermitteln, der Rekonstruktionsprozess benötigt dagegen

weitere Informationen, nämlich die Anzahl und Typen der Attribute sowie deren Speicherbelegung. Für serialisierbare Klassen kümmert sich die Automatik um diese Details. Für nicht serialisierbare Basisklassen macht dies der Defaultkonstruktor.

Es ist eher unwahrscheinlich, dass die Attribute durch einen Aufruf des Defaultkonstruktors in den gewünschten Zustand versetzt werden. Um einen korrekten Objektzustand herzustellen, müssen dann die Daten der Basisklasse vom Entwickler selbst gespeichert und wieder eingelesen werden. Dazu benötigt man eine Möglichkeit, sich in den Ablauf des Automatismus einzuklinken, um erforderliche Daten zu speichern und später wieder einzulesen. Der folgende Abschnitt geht darauf genauer ein.

Tipp: Serialisierung vs. Kapselung und Versionierung

Im Datenformat der Serialisierung wird zur Identifizierung von Typen neben der konkreten Typangabe als voll qualifizierter Name zusätzlich eine spezielle Versionsnummer gespeichert. Diese kann explizit in Form einer Kennung (`serialVersionUID`) im Sourcecode angegeben werden. Teilweise geschieht dies nicht. Dann wird beim Kompilieren automatisch eine solche Versionsnummer berechnet und beim Serialisieren verwendet.

In die Berechnung gehen unter anderem die Methoden einer Klasse und der Zeitpunkt der Kompilierung ein. Jede neu eingeführte Methode verändert damit diese Versionsnummer und führt zu Inkompatibilitäten zu älteren Versionen des Objekts. Die Versionsnummer ändert sich demnach häufiger, als es tatsächlich zu Änderungen an der Schnittstelle oder in der Semantik kommt.

Mit Serialisierung zerstört man zudem Kapselung: Jede Änderung an der Struktur einer Klasse führt zu Inkompatibilitäten mit früher serialisierten Versionen der Objekte. Der Grund ist folgender: Alle nicht explizit von der Serialisierung ausgeschlossenen Attribute einer Klasse werden verarbeitet und in die Ausgabe geschrieben. Das gilt aber auch für private Attribute, die gegebenenfalls Implementierungsdetails darstellen. Bei einem späteren Einlesen werden exakt diese Informationen wieder erwartet.

Finden strukturelle Änderungen statt, so kann ein zuvor serialisiertes Objekt als Folge nicht mehr verarbeitet werden: Diesem fehlen die durch die Änderungen hinzugefügten Informationen. Die Argumentation gilt gleichermaßen für das Entfernen und Ändern von Attributen und Methoden.

Zur Lösung der genannten Probleme ist die Vergabe einer speziellen vom Entwickler definierten `serialVersionUID` möglich. Nur bei Änderungen, die wirklich zu einer Inkompatibilität zwischen verschiedenen Versionen eines Objekts führen, wird die Versionsnummer vom Entwickler angepasst. Sollen verschiedene Versionen verarbeitet werden können, so muss der Serialisierungsvorgang für jede Version einzeln selbst programmiert werden. Abschnitt 8.3.3 beschreibt die Verwaltung von Versionen detaillierter.

8.3.2 Die Serialisierung anpassen

Für viele Anwendungsfälle reicht die von Java bereitgestellte Serialisierung aus. Allerdings gibt es einige Situationen, in denen man den Serialisierungsprozess anpassen möchte oder muss. In der Regel ist dies der Fall, wenn eine Klasse eine andere Klasse referenziert, die nicht serialisierbar ist. Weiterhin gibt es Attribute, die aus anderen Gründen nicht sinnvoll persistiert werden können. Dazu gehören zum einen Attribute, deren Werte nur berechnet werden, und zum anderen solche, die eine Ressource repräsentieren, etwa eine Verbindung zu einer Datenbank. Solche Attribute kann man explizit von der Verarbeitung durch die Serialisierungsautomatik ausschließen. Weitere Anwendungsfälle bestehen darin, eine platzsparendere Speicherung zu erzielen oder verschiedene Versionen einer Klassendefinition zu unterstützen (vgl. Abschnitt 8.3.3).

Problemkontext

Nehmen wir an, die Klasse `NonSerializableClass` modelliert eine Verbindung zu einer Datenbank und erhält zur Konstruktion verschiedene Informationen als String, die zum Aufbau einer Verbindung zur Datenbank genutzt werden. Instanzen dieser Klasse können nicht serialisiert werden, da verschiedene Bestandteile nicht serialisierbar sind. Die folgende Klassendefinition implementiert daher das Interface `Serializable` nicht:

```
public class NonSerializableClass
{
    private final String databaseConnectionParams;

    NonSerializableClass(final String databaseConnectionParams)
    {
        this.databaseConnectionParams = databaseConnectionParams;
    }

    public String getDatabaseConnectionParams()
    {
        return databaseConnectionParams;
    }

    // …
```

Nehmen wir weiter an, dass die Klasse `Person` um ein nicht serialisierbares Attribut `nonSerializable` vom Typ `NonSerializableClass` folgendermaßen erweitert worden ist:

```
public final class Person implements Serializable
{
    private String name;
    private String city;
    private Date birthday;

    // Ausschluss von der Serialisierung
    private transient NonSerializableClass nonSerializable;
```

```
    // …
```

Modifikation des Serialisierungsvorgangs

Zur korrekten Verarbeitung `transient` definierter Attribute muss der Serialisierungsvorgang angepasst werden. Durch ***Implementierung der folgenden zwei privaten Methoden in der zu serialisierenden Klasse*** lässt sich dies erreichen:[4]

```
private void writeObject( ObjectOutputStream outStream) throws IOException
private void readObject( ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
```

Die Methode `writeObject( ObjectOutputStream)` realisiert das Speichern von Objektinformationen in einen Stream, die Methode `readObject( ObjectInputStream)` das Einlesen. Auf den ersten flüchtigen Blick ist kein Unterschied zu den zuvor genutzten Methoden `writeObject( Object)` und `Object readObject()` zu erkennen. Letztere entstammen aber den Klassen `ObjectOutputStream` und `ObjectInputStream` und nutzen den Typ `Object` als Ein- bzw. Rückgabe. Dahingegen müssen die beiden oben gelisteten Methoden innerhalb von eigenen Klassen implementiert werden und Instanzen der Klassen `ObjectInputStream` bzw. `ObjectOutputStream` als Parameter übergeben bekommen.

```
public class Person implements Serializable
{
    // …
    private void writeObject( final ObjectOutputStream outStream) throws
          IOException
    {
            // …
    }

    private void readObject( ObjectInputStream inStream) throws IOException,
          ClassNotFoundException
    {
            // …
    }
}
```

Der Serialisierungsautomatismus ruft diese Methoden, sofern vorhanden, während der Verarbeitung auf. Die Serialisierung der Attribute möglicher Basis- und Subklassen erfolgt nach wie vor durch den Automatismus der Serialisierung. Die beiden Methoden sind dafür zuständig, *alle* Attribute des zugehörigen Klassenbausteins zu verarbeiten, d. h. in den Stream zu schreiben bzw. daraus zu lesen oder anderweitig zu initialisieren.

Realisierung der Serialisierung

Bei der Implementierung der Serialisierung sind alle benötigten Informationen in den Ausgabestream zu schreiben und später wieder korrekt einzulesen. Das bedeutet, dass sowohl alle serialisierbaren Attribute verarbeitet werden müssen als auch eine

Spezialbehandlung für die nicht serialisierbaren, transienten Attribute durchgeführt werden muss.

Bei der Klasse `Person` sind vier Attribute zu serialisieren. Ein erster Ansatz ist, dazu die Methoden des Interface `DataOutput` zu benutzen. Dort sind diverse `write()`-Methoden für primitive Datentypen, Byte-Arrays und UTF8-codierte Strings definiert. Das Interface `DataOutput` wird sowohl von der Klasse `DataOutputStream` als auch von der Klasse `ObjectOutputStream` implementiert. Nutzt man die im Interface `DataOutput` bereitgestellte Funktionalität, so sind für jedes Attribut allerdings spezifische, typabhängige `write()`- und `read()`-Methoden aufzurufen. Für serialisierbare Objekte kann auch die Methode `writeObject(Object)` eingesetzt werden, wie hier für das `Date`-Objekt:

```
private void writeObject(final ObjectOutputStream outStream) throws
IOException
{
    // Verarbeitung aller »normalen« Attribute
    outStream.writeUTF(name);
    outStream.writeUTF(city);
    outStream.writeObject(birthday);

    // Spezialbehandlung des nicht serialisierbaren Attributs
    outStream.writeUTF(nonSerializable.getDatabaseConnectionParams());
}
```

Hier nutzen wir die Tatsache aus, dass sich eine Instanz des nicht serialisierbaren Attributs vom Typ `NonSerializableClass` durch die textuelle Repräsentation der Verbindungsparameter zur Datenbank beschreiben lässt.[5] Beim Speichern schreiben wir diese Information mithilfe der Methode `writeUTF(String)` in den Stream. Auf das Einlesen gehe ich gleich ein, nachdem wir eine Vereinfachung betrachtet haben.

Standardmechanismus aufrufen

All dies von Hand zu programmieren ist jedoch mühselig und fehleranfällig. Das haben wir bereits eingangs bei der Beschreibung der Serialisierung erkannt. Was kann man also tun? Wünschenswert ist es, das Verhalten des Standardmechanismus für serialisierbare Attribute nutzen zu können und nur für die `transient` definierten Attribute eine Spezialbehandlung durchführen zu müssen. Genau für diesen Anwendungsfall sind in den Klassen `ObjectOutputStream` und `ObjectInputStream` zwei Methoden definiert: Die Methode `defaultWriteObject()` speichert alle nicht `transient` definierten Attribute einer Klasse, die Methode `defaultReadObject()` liest sie ein. Diese Methoden können genutzt werden, um die eigene Realisierung der Serialisierung einfacher zu gestalten. Beide Methoden dürfen nur während einer laufenden Serialisierung aufgerufen werden, d. h. nur aus den beiden zuvor genannten privaten Methoden `readObject(ObjectInputStream)` bzw. `writeObject(ObjectOutputStream)`. Ansonsten wird eine

java.io.NotActiveException ausgelöst.

Basierend auf diesen Überlegungen ergeben sich in der Klasse Person folgende Implementierungen der Methoden writeObject(ObjectOutputStream) und readObject(ObjectInputStream):

```
private void writeObject(final ObjectOutputStream outStream) throws
IOException
{
    // Behandlung aller »normalen« Attribute
    outStream.defaultWriteObject();

    // NonSerializableClass wird nicht geschrieben, stattdessen lediglich
    // Informationen, die zur Rekonstruktion benötigt werden
    outStream.writeUTF(nonSerializable.getDatabaseConnectionParams());
}

private void readObject(final ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
{
    // Behandlung aller »normalen« Attribute
    inStream.defaultReadObject();

    // Rekonstruktion der NonSerializableClass aus dem gelesenen String
    final String databaseConnectionParams = inStream.readUTF();
    nonSerializable = new NonSerializableClass(databaseConnectionParams);
}
```

Beim Einlesen nutzen wir die Methode readUTF(). Anschließend erzeugen wir per Konstruktor ein neues Objekt vom Typ NonSerializableClass.

Hinweis

Zur Verarbeitung von Attributen werden Zuweisungen in der Methode readObject() benötigt. Dadurch können diese Attribute nicht final sein. Dies stellt eine Einschränkung beim Design dar. Insbesondere kann eine Klasse dann möglicherweise nicht mehr als unveränderliche Klasse realisiert werden.

8.3.3 Versionsverwaltung der Serialisierung

Der Automatismus der Serialisierung sorgt dafür, dass nur miteinander kompatible Versionsstände einer Klasse verarbeitet werden können. Wie bereits beschrieben, ändert sich die dafür genutzte Versionskennung serialVersionUID vom Typ long bei jeder kleinen Änderung an der Struktur der Klasse sowie bei jedem Kompilieren. Das ist häufig unpraktisch, da es dann zu einer Inkompatibilität zwischen semantisch gleichen Versionen eines Objekts kommt. Als Abhilfe kann man einen fixen Wert im privaten statischen Attribut serialVersionUID festlegen. Wenn Änderungen an Methoden und Attributen der Klasse vorgenommen werden, wird diese Versionskennung dann *nicht* neu berechnet, sondern bleibt konstant. Betreffen die strukturellen Änderungen auch die serialisierten Daten, so wird die

`serialVersionUID` vom Entwickler modifiziert. Damit hat man zwar das Problem der Inkompatibilität semantisch gleicher Klassenstände gelöst, allerdings kann auch dann über die Serialisierung kein Datenaustausch zwischen einer alten und einer neuen Version erfolgen.

Wenn eine Rückwärtskompatibilität erhalten und verschiedene Versionen unterstützt werden sollen, dann muss der Serialisierungsvorgang für jede Version einzeln selbst programmiert werden. Zudem müssen wir mehrere Varianten serialisierter Formen eindeutig im Datenstrom identifizieren können. Dazu verwenden wir eine eigene Versionsverwaltung und fixieren zudem den Wert der `serialVersionUID` beim Erstellen der Klasse einmalig. Das erlaubt selbst bei strukturellen Änderungen an der Klasse deren sichere Identifikation im Datenstrom. Zur Versionsverwaltung verwenden wir ein Attribut `CLASS_VERSION`, das in den Methoden `readObject(ObjectInputStream)` und `writeObject(ObjectOutputStream)` ausgewertet wird.

Da wir sowohl die Versionsverwaltung als auch die Serialisierung der Objekte vollständig selbst realisieren, müssen *alle* Attribute `transient` definiert werden. Wir können damit nicht mehr auf die praktischen Methoden `defaultWriteObject()` bzw. `defaultReadObject()` zurückgreifen, da diese nur nicht transiente Attribute lesen bzw. schreiben.

Im folgenden Listing ist die Klasse `PersonVersion1` gezeigt, die die erste Version einer Klasse `Person` darstellen soll. Vor den eigentlichen Nutzdaten wird eine Versionsnummer in Form eines `int` in den Stream geschrieben. Anschließend werden alle Attribute manuell gespeichert (durch Aufruf entsprechender `write()`-Methoden). Beim späteren Deserialisieren erfolgen die korrespondierenden Schritte zum Einlesen. Hierbei wird zudem die Versionsnummer geprüft. Nur für den Fall eines gültigen Werts findet das Einlesen statt. Ansonsten wird eine `ClassNotFoundException` ausgelöst:

```
public final class PersonVersion1 implements Serializable
{
   // durch beliebige, immer gleichbleibende Kennung
   // Berechnungsautomatismus deaktivieren
   private static final long serialVersionUID = 1L;

   // eigene Kennung der Klassenversion
   private static final long CLASS_VERSION = 1L;

   private transient String name;
   private transient String city;
   private transient Date   birthday;

   public PersonVersion1(final String name, final String city, final Date
         birthday)
   {
      this.name = name;
      this.city = city;
      this.birthday = birthday;
   }
```

```
   private void writeObject( final ObjectOutputStream outStream) throws
         IOException
   {
      // eigene Versionsinformation schreiben
      outStream.writeLong( CLASS_VERSION);

      // Attributdaten schreiben
      outStream.writeUTF( name);
      outStream.writeUTF( city);
      outStream.writeObject( birthday);
   }

   private void readObject( final ObjectInputStream inStream) throws
IOException,
                                                          ClassNotFoundExceptior
   {
      // eigene Versionsinformation lesen
      final long version = inStream.readLong();

      if ( version == CLASS_VERSION)
      {
          // Attributdaten lesen
          name = inStream.readUTF();
          city = inStream.readUTF();
          birthday = ( Date) inStream.readObject();
      }
      else
      {
          // Diese Version unterstützt keine anderen Formate
         throw new ClassNotFoundException( "Unsupported version " + version);
      }
  }
  // …
```

Nehmen wir an, wir hätten die genannten Schritte für die Klasse `Person` durchgeführt und bei einer nachfolgenden Erweiterung der Klasse (Speicherung der Augenfarbe) wäre die interne Klassenversion hochgezählt worden:

```
public final class PersonVersion2 implements Serializable
{
    // beliebige, immer gleichbleibende Kennung
    private static final long serialVersionUID = 1L;

    // eigene Kennung der Klassenversion
    private static final long CLASS_VERSION = 2L;

    private transient String name;
    private transient String city;
    private transient Date   birthday;
    private transient Color  eyeColor;

    // …
```

Um die zwei Versionen deutlich voneinander zu unterscheiden, wurde eine Optimierung der Speicherung in das Datenformat in die Version 2 übernommen. Diese besteht darin, das `Date`-Objekt als `long` zu speichern und anstatt des `Color`-Objekts lediglich die drei Farbwerte für Rot, Grün und Blau als `int` zu verwenden. Darauf gehe ich im nachfolgenden Abschnitt nochmal genauer ein.

```
private void writeObject( final ObjectOutputStream outStream) throws
```

```
IOException
{
    // Versionsinformation schreiben
    outStream.writeLong(CLASS_VERSION);

    // Daten
    outStream.writeUTF(name);
    outStream.writeUTF(city);

    // Optimierte Darstellung für Date
    outStream.writeLong(birthday.getTime());

    // Optimierte Darstellung für Color
    outStream.writeInt(eyeColor.getRed());
    outStream.writeInt(eyeColor.getGreen());
    outStream.writeInt(eyeColor.getBlue());
}
```

Das Einlesen ist etwas komplizierter als das Schreiben, da beim Lesen verschiedene Versionen unterschieden und behandelt werden müssen. In folgender Methode `readObject(ObjectInputStream)` wird dazu zunächst die Versionsnummer aus dem Stream gelesen und anschließend der jeweilige Sourcecode zum Einlesen ausgeführt:

```
private void readObject(final ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
{
    // Versionsinformation ermitteln
    final long version = inStream.readLong();

    // versionsabhängiges Einlesen
    if (version == CLASS_VERSION)
    {
        readDataInCurrentVersion(inStream);
    }
    else if (version == 1)
    {
        readDataOfVersion1(inStream);
    }
    else
    {
        throw new ClassNotFoundException("Unsupported version " + version);
    }
}

private void readDataOfVersion1(final ObjectInputStream inStream) throws
      IOException, ClassNotFoundException
{
    name = inStream.readUTF();
    city = inStream.readUTF();
    birthday = (Date) inStream.readObject();
    eyeColor = Color.BLUE; // Vereinfachung für Beispiel: Defaultwert Blau
}

private void readDataInCurrentVersion(ObjectInputStream inStream) throws
      IOException
{
    name = inStream.readUTF();
    city = inStream.readUTF();
```

```
    final long time = inStream.readLong();
    birthday = new Date(time);

    final int red = inStream.readInt();
    final int green = inStream.readInt();
    final int blue = inStream.readInt();
    eyeColor = new Color(red, green, blue);
}
// …
```

Durch Implementierung dieser beiden Methoden können wir die Serialisierung beliebig anpassen. Damit können verschiedene Versionen eines Objekts verwaltet werden. Es ist relativ aufwendig, die Kompatibilität von Versionen sicherzustellen, dies wird hier nicht weiter behandelt. In diesem Beispiel habe ich stark vereinfachend die Augenfarbe auf Blau gesetzt, da diese Information in einer älteren Version nicht verfügbar ist.

8.3.4 Optimierung der Serialisierung

Bislang haben wir den Standardmechanismus der Serialisierung eingesetzt bzw. nur ein wenig angepasst. Zum Teil kann es zur Performance-Optimierung sinnvoll sein, die Datenmenge gegenüber dem Standardformat zu reduzieren. Dazu müssen Änderungen am Format der herausgeschriebenen Informationen vorgenommen werden.

Um einen Blick für Optimierungsmöglichkeiten zu bekommen, führen wir eine Erweiterung der eingangs vorgestellten Klasse `Person` durch. Hier wird wieder die Augenfarbe als Attribut in Form eines `Color`-Objekts hinzugefügt. Folgende `main()`-Methode zeigt die Erzeugung eines solchen `PersonWithEyeColorV1`-Objekts und dessen Speicherung in der Datei `TestWithEyeColor1.ser`:

```
public static void main(final String[] args) throws IOException
{
    final PersonWithEyeColorV1 original = new PersonWithEyeColorV1("Test",
                                          "TestCity", new Date(),
Color.GREEN);

    try (final ObjectOutputStream objectOutStream = new ObjectOutputStream(
      new BufferedOutputStream(new
FileOutputStream("TestWithEyeColor1.ser"))))
    {
        // Schreibe Objekt in die Datei
        objectOutStream.writeObject(original);
        System.out.println("Wrote to stream:  " + original);
    }
}
```

Listing 8.9 *Ausführbar als* **'SERIALIZATIONOPTIMIZATIONEXAMPLE'**

Serialisiert man diese Klasse durch Aufruf des Programms SERIALIZATIONOPTIMIZATIONEXAMPLE, wird die entstehende Datei 357 Bytes groß.

Suchen wir nun nach Optimierungspotenzialen: Dazu öffnen wir die Datei `TestWithEyeColor1.ser` mit einem HEX- oder Texteditor. Wir erkennen, dass

die Speicherung für das Datum bzw. die Farbinformation noch relativ viele, für diesen Anwendungsfall irrelevante Informationen (voll qualifizierten Klassenname, Transparenz usw.) in die Datei schreibt.

Optimierung durch eigene Repräsentationsform

Statt eines `Color`-Objekts werden lediglich die Farbwerte Rot, Grün und Blau zur Modellierung der Augenfarbe benötigt. Auch die Repräsentation des gespeicherten Geburtsdatums ist umfangreich. Für beide Attribute ist eine Darstellung durch die primitiven Datentypen `int` und `long` relativ einfach möglich. Minimal aufwendiger als das Schreiben wird die Rekonstruktion korrespondierender `Date`- bzw. `Color`-Objekte. Folgendes Listing zeigt die dazu notwendigen Realisierungen der Methoden `readObject(ObjectInputStream)` und `writeObject(ObjectOutputStream)`:

```
// Achtung: falsche Lösung
private void writeObject(final ObjectOutputStream outStream) throws
IOException
{
    outStream.defaultWriteObject();

    outStream.writeLong(birthday.getTime());

    outStream.writeInt(eyeColor.getRed());
    outStream.writeInt(eyeColor.getGreen());
    outStream.writeInt(eyeColor.getBlue());
}

private void readObject(final ObjectInputStream inStream) throws IOException,
      ClassNotFoundException
{
    inStream.defaultReadObject();

    final long time = inStream.readLong();
    birthday = new Date(time);

    final int red = inStream.readInt();
    final int green = inStream.readInt();
    final int blue = inStream.readInt();
    eyeColor = new Color(red, green, blue);
}
```

Serialisiert man diese Klasse durch Aufruf des Programms SERIALIZATIONOPTIMIZATIONEXAMPLE2[6], so entsteht die Datei `TestWithEyeColor2.ser`. Diese benötigt 380 Bytes Speicherplatz. Offensichtlich ist diese Lösung sogar schlechter als der Standard! Wie kann das denn sein?

Die Antwort ist einfach: Es wird zunächst die Standardserialisierung vorgenommen und dann werden zusätzlich die Informationen der primitiven Daten in den Stream geschrieben. Dieser Flüchtigkeitsfehler ist schnell gemacht, aber auch schnell behoben. Soll zwar der Standardmechanismus für eine korrekte Objektserialisierung der `String`-Attribute sorgen und wollen wir lediglich einige Attribute unseres Objekts besonders behandeln, so müssen wir diese natürlich auch

transient definieren:

```
public class PersonWithEyeColor implements Serializable
{
    private String name;
    private String city;
    // beide komplexeren Attribute nicht vom Standard behandeln lassen
    private transient Date  birthday;
    private transient Color eyeColor;

   // …
```

Führen wir die nicht als Listing gezeigte Korrektur des Programms SERIALIZATIONOPTIMIZATIONEXAMPLE3 aus, so entsteht die Datei TestWithEyeColor3.ser, die nur noch 153 Bytes groß ist.

Mit dieser modifizierten Form der Speicherung haben wir einiges an Speicherplatz eingespart. ***Für ein einzelnes Objekt lohnen sich derartige Mühen nicht***. Werden aber umfangreichere Datenstrukturen serialisiert, so kann eine eigene Repräsentation sinnvoll sein. Auch bei einer Übertragung über ein Netzwerk können so Verbesserungen in der Übertragungsgeschwindigkeit erzielt werden.

Hinweis: Serialisierbare Attribute vorgeben

Im Normalfall bestimmt der Serialisierungsautomatismus die zu verarbeitenden Attribute und deren Werte mithilfe von Reflection. Sind nun die Attribute überwiegend transient, so bietet es sich an, die tatsächlich zu serialisierenden Attribute anzugeben. Dazu muss man eine statische Variable mit dem Namen serialPersistentFields in die Klassendefinition aufnehmen, etwa wie folgt für zwei Attribute attr1 und attr2 mit den Typen attr1Type und attr2Type:

```
private static final ObjectStreamField[] serialPersistentFields =
{
    new ObjectStreamField("attr1", attr1Type.class);
    new ObjectStreamField("attr2", attr2Type.class);
};
```

Wir nutzen hier die syntaktische Besonderheit der Kurzschreibweise der Array-Initialisierung statt new ObjectStreamField[], wodurch das Ganze kurz und knackig wird.

Das Interface Externalizable

Manchmal wird eine noch weiter gehende Einflussnahme des Serialisierungsvorgangs benötigt, als dies über die Methoden readObject(ObjectInputStream) und writeObject(ObjectOutputStream) möglich ist. Für diese Anwendungsfälle ist das Interface java.io.Externalizable zu implementieren. Über dieses Interface, das das Interface Serializable erweitert, kann eine Variante der Serialisierung beschrieben werden, die es dem Entwickler erlaubt, das Datenformat vollkommen selbst zu bestimmen, d. h. festzulegen, welche Attribute in welcher Form

verarbeitet (geschrieben und gelesen) werden. Dadurch kann beispielsweise das Datenformat noch kompakter gestaltet werden. Allerdings müssen auch die Attribute der Basisklassen selbst verarbeitet werden. Das Interface `Externalizable` definiert dazu folgende zwei Methoden:

```
public interface Externalizable extends java.io.Serializable
{
   void readExternal(ObjectInput in) throws IOException,
ClassNotFoundException;
   void writeExternal(ObjectOutput out) throws IOException;
}
```

Wir entfernen das zuvor eingeführte Schlüsselwort `transient` für die Attribute und lassen die Klasse `Person` nun das Interface `Externalizable` implementieren:

```
public final class PersonExternalizable implements Externalizable
{
    private String name;
    private String city;
    private Date   birthday;
    private Color  eyeColor;

    // …
```

Durch Implementierung der zuvor genannten Methoden wird man vom Datenformat der Serialisierung unabhängig. Dies ist insbesondere dann von Vorteil, wenn keine Objektgraphen, sondern lediglich flache Objekte verarbeitet werden sollen. Dann können Speicherplatz sparende Dateiformate gewählt werden. Schauen wir uns nun die Realisierung der beiden Methoden an, die die schon vorgestellte Optimierung der Repräsentation von Objekten über primitive Werte ihrer relevanten Daten realisiert:

```
@Override
public void writeExternal(final ObjectOutput objectOut) throws IOException
{
    objectOut.writeUTF(name);
    objectOut.writeUTF(city);

    objectOut.writeLong(birthday.getTime());

    objectOut.writeInt(eyeColor.getRed());
    objectOut.writeInt(eyeColor.getGreen());
    objectOut.writeInt(eyeColor.getBlue());
}

@Override
public void readExternal(final ObjectInput objectIn) throws IOException,
      ClassNotFoundException
{
    name = objectIn.readUTF();
    city = objectIn.readUTF();

    final long time = objectIn.readLong();
    birthday = new Date(time);

    final int red = objectIn.readInt();
    final int green = objectIn.readInt();
    final int blue = objectIn.readInt();
    eyeColor = new Color(red, green, blue);
}
```

Da das Interface `Externalizable` eine Erweiterung des Interface `Serializable` ist, könnte man diese Klasse exakt wie in den vorangegangenen Beispielen verarbeiten. Dies ist für all die Situationen vorteilhaft, in denen der größte Teil der Klassen mit dem Standardmechanismus verarbeitet und nur ein kleiner Teil speziell behandelt werden soll. In den Standardprozess der Serialisierung (durch Aufruf der Methoden `readObject(ObjectInputStream)` und `writeObject(ObjectOutputStream)`) eingefügt, kann man auf diese Weise den Speicherbedarf auf 111 Bytes reduzieren, wie es ein Start des Programm SERIALIZATIONFOREXTERNALIZABLECLASSEXAMPLE zeigt. Das stellt bereits eine deutliche Reduktion des Speicherverbrauchs dar. Aber es geht noch deutlich besser!

Schauen wir darauf, was passiert, wenn man das Ganze in die eigene Hand nimmt und auf die Serialisierungsautomatik verzichtet. Dann nutzt man Aufrufe der Methoden `readExternal(ObjectInput)` und `writeExternal(ObjectOutput)` aus dem Interface `Externalizable`. Betrachten wir die korrespondierende `main()`-Methode, die ein Objekt vom Typ `PersonExternalizable` ohne Einbindung in die Serialisierungsautomatik speichert und wieder einliest:

```
public static void main(final String[] args) throws IOException,
      ClassNotFoundException
{
    final PersonExternalizable original = new PersonExternalizable("Test",
                                          "TestCity", new Date(),
Color.GREEN);

   try (final ObjectOutputStream objectOutStream = new ObjectOutputStream(
        new BufferedOutputStream(new
FileOutputStream("TestExternizable.ser"))))
   {
       // Schreibe Objekt mit writeExternal() in die Datei
       original.writeExternal(objectOutStream);
       System.out.println("Wrote to stream: " + original);
   }

   try (final ObjectInputStream objectInStream = new ObjectInputStream(
        new BufferedInputStream(new FileInputStream("TestExternizable.ser"))))
   {
       // Rücklesen des Objekts
       fileInStream = new FileInputStream("TestExternizable.ser");
       objectInStream = new ObjectInputStream(fileInStream);

       // Konstruktoraufruf und Einsatz von readExternal()
       final PersonExternalizable readInObject = new PersonExternalizable();
       readInObject.readExternal(objectInStream);
       System.out.println("Back from stream: " + readInObject);
   }
}
```

Listing 8.10 *Ausführbar als* **'EXTERNALIZABLEEXAMPLE'**

Der entstehende Sourcecode ist recht ähnlich zu dem der Serialisierung. Allerdings muss im Gegensatz zur Serialisierung ein Konstruktoraufruf der einzulesenden Klasse erfolgen. Für die neue Instanz wird die Methode `readExternal(ObjectInput)`

aufgerufen.[7] Der entscheidende Unterschied liegt darin, dass es sich bei den beiden Methoden des Interface `Externalizable` um Objektmethoden handelt, denen ein Stream übergeben wird. Bei der Serialisierung sind die Zuständigkeiten genau andersherum geregelt: Ein spezieller Stream bekommt eine Objektreferenz, die er verarbeitet.

Führt man das Programm EXTERNALIZABLEEXAMPLE aus und betrachtet dann die Größe der entstehenden Datei `TestExternalizable.ser`, so ermittelt man einen Speicherbedarf von lediglich 42 Bytes. Im Vergleich zum Ausgangspunkt der Optimierung der Serialisierung mit 357 Bytes ist dies nur noch 1/8 des Speicherplatzes. Bezogen auf die letzte optimierte Version der Serialisierung mit 153 Bytes wird der Speicherbedarf auf rund 1/3 reduziert.

Fazit

Die Reduktion des Datenvolumens geht mit einem erhöhten Aufwand bei der Implementierung und insbesondere einer späteren Wartung des Speicherns und Einlesens einher. Derartige Lösungen sind deshalb nur bei Anwendungen sinnvoll zu verwenden, die kritisch bezüglich Performance oder Speicherplatz sind.

8.4 Objektkopien und das Interface `Cloneable`

Neben der Konstruktion über `new` existieren in Java weitere Möglichkeiten, Objekte zu erzeugen: Mithilfe von Reflection und dem Aufruf der Methode `newInstance()` kann man dynamisch zur Laufzeit neue Instanzen beliebiger Klassen anlegen. Das haben wir in Abschnitt 8.1 bereits kennengelernt. Auch über die im vorherigen Abschnitt 8.3 beschriebene Serialisierung ist eine Objekterzeugung möglich. Dieser Abschnitt stellt Techniken zum Kopieren von Objekten vor.

Erwartungshaltung

In den Abschnitten 3.4.1 und 3.4.2 wurden mögliche Probleme durch die Übergabe und die Rückgabe von Referenzen besprochen. Betrachten wir dies nun im Kontext des Kopierens von Objekten. Kopiert man in Java Werte primitiver Typen, so erhält man wieder unabhängige Werte. Kopiert man Referenzattribute, so sind die Referenzattribute selbst auch wieder unabhängig voneinander. Eine Zuweisung in ein kopiertes Referenzattribut besitzt keine Rückwirkungen auf das originale Referenzattribut; im umgekehrten Fall gilt das selbstverständlich auch. Die referenzierten Objekte selbst werden jedoch nicht kopiert. Änderungen an diesen betreffen dann sowohl das Original als auch die Kopie: Wird ein solches referenziertes Objekt (z. B. für die Originalreferenz) geändert, ist die Veränderung auch für die kopierte Referenz sichtbar. Derartige Änderungen im Objektzustand (vgl. Abschnitt 3.1.5) sind meistens unerwartet und unerwünscht.

Aus diesen Betrachtungen leiten wir folgende Forderung ab: Eine Kopie sollte immer ein unabhängiges Objekt liefern. Datenänderungen auf dessen Attributen besitzen dann keine Auswirkungen auf das Original. Das erfordert allerdings häufig eine tiefe Kopie aller referenzierten Objekte.[8]

8.4.1 Das Interface `Cloneable`

Um Objekte zu kopieren und damit eine zweite, gleichartige Instanz zu erhalten, bietet Java einen Automatismus. Ähnlich zur Serialisierung muss eine Klasse ebenfalls ein Marker-Interface implementieren, um diesen Automatismus zu aktivieren. Im JDK ist dafür das Interface `java.lang.Cloneable` definiert.

Grundlagen zur Methode `clone()`

Die Methode `clone()` dient dazu, eine Kopie eines Objekts zu erstellen. Dazu ist die Methode `clone()` in der Basisklasse `Object` mit folgender Signatur definiert:

```
protected Object clone() throws CloneNotSupportedException
```

Weil die Methode nur die Sichtbarkeit `protected` besitzt, können nur Objekte der eigenen Klasse (oder Klassen aus dem gleichen Package) die Methode `clone()` aufrufen. Um auch beliebigen Klienten das Kopieren zu ermöglichen, muss die Methode überschrieben und deren Sichtbarkeit auf `public` geändert werden. Eine Realisierung einer Methode `clone()` könnte folgendermaßen aussehen:

```
public class MyCloneableClass implements Cloneable
{
    public Object clone() throws CloneNotSupportedException
    {
        return super.clone();
    }
    // …
```

Diese Realisierung nutzt die in der Klasse `Object` vorhandene Standardimplementierung, die flache Kopien erstellt, d. h. alle Attribute primitiver Datentypen und Objektreferenzen bitweise kopiert. Erwähnenswert ist, dass zum Erstellen der Kopie *kein* Konstruktoraufruf erfolgt. Während des Kopierens findet eine Typprüfung statt, die sicherstellt, dass ein zu kopierendes Objekt vom Typ `Cloneable` ist. Ansonsten wird eine `java.lang.CloneNotSupportedException` ausgelöst. Da dies eine Checked Exception ist, muss sie mit einem `catch`-Block behandelt oder weiter propagiert werden. Weiterhin können Subklassen diese Exception auslösen, um Kopien zu verhindern.

Die entstehenden flachen Kopien widersprechen jedoch der eingangs aufgestellten Erwartung unabhängiger Objektkopien, weil dabei nur Referenzen, nicht aber die Objekte selbst kopiert werden. Wenn tiefe Kopien gewünscht sind, muss die Methode `clone()` in eigenen Klassen überschrieben und erweitert werden. Das erfordert vom

Entwickler einige Nacharbeiten. Dies gilt insbesondere, wenn Objekte andere Objekte und Containerklassen als Attribute besitzen.

Achtung: Unzulänglichkeiten des `Cloneable`-Interface

Die Methode `clone()` wäre besser als Methode innerhalb des Interface `Cloneable` definiert worden. Auch die Einschränkung der Standardimplementierung auf die Sichtbarkeit `protected` erschwert den Einsatz aus externen Klassen. Darüber hinaus entsteht lediglich eine flache Kopie. Weitere Unzulänglichkeiten beschreibt Joshua Bloch ausführlich in seinem Buch »Effective Java« [9].

Der `clone()`-Kontrakt

Die Java-Dokumentation empfiehlt, folgende Konventionen einzuhalten, wenn ein Objekt `x` durch Aufruf von `clone()` kopiert wird:

- **Objekterzeugung** – Es wird ein neues Objekt erzeugt, sodass `x.clone() != x` gilt.
- **Typgleichheit** – Das neu erzeugte Objekt ist von exakt dem Typ des kopierten Objekts: Es gilt `x.clone().getClass().equals(x.getClass())`.
- **Wertgleichheit** – Für Original und Kopie *sollte* Wertgleichheit gelten, d. h., es gilt `x.clone().equals(x)`. Dies ist beispielsweise nicht gegeben, wenn die Methode `equals(Object)` nicht überschrieben wurde, das es dann lediglich zu einem Referenzvergleich kommt.

Weil wir die Kontraktprüfung nachfolgend benötigen, implementieren wir eine Utility-Klasse `CloneableUtils` mit der Methode `checkContract(Object, Object)`:

```
public static void checkContract(final Object original, final Object clone)
{
    System.out.println("Kontraktprüfung:");
    System.out.println("Original:  " + original);
    System.out.println("Kopie:     " + clone);
    System.out.println("Objekterzeugung? " + (clone != original));
    System.out.println("Typgleichheit?   " + clone.getClass().
                                             equals(original.getClass()));
    System.out.println("Wertgleichheit?  " + clone.equals(original));
}
```

Erste Schritte mit der `clone()`-Methode

Nachdem wir nun einige Grundlagen zur `clone()`-Methode kennengelernt haben, wollen wir dieses Wissen nutzen: Die Klasse `BaseCloneable` implementiert das Interface `Cloneable` und definiert eine öffentliche `clone()`-Methode, die den Automatismus durch Aufruf der `clone()`-Methode aus der Basisklasse `Object` zum Erstellen einer flachen Kopie nutzt:

```
public class BaseCloneable implements Cloneable
{
    private final int id;
    /* private */ long value;   // Zum späteren Zugriff im Package

    public BaseCloneable(final int id, final long value)
    {
        this.id = id;
        this.value = value;
    }

    // Überschreiben zur Erweiterung der Sichtbarkeit
    @Override
    public Object clone() throws CloneNotSupportedException
    {
        // Aufruf des Standardmechanismus, für primitive Attribute ausreichend
        return super.clone();
    }
    // …
```

Mit einem einfachen Programm wollen wir einige Erkenntnisse durch den Einsatz der eben definierten Methode `clone()` gewinnen. Dazu konstruieren wir zunächst ein Objekt vom Typ `BaseCloneable` mit beliebigen Eingabewerten, in diesem Beispiel mit den Werten `47` und `11` für die Attribute `id` bzw. `value`. Von diesem Objekt wird zunächst durch Aufruf der Methode `clone()` eine Kopie erstellt. Anschließend rufen wir die Methoden `checkContract(BaseCloneable, BaseCloneable)` und `modify(BaseCloneable)` auf, um die Einhaltung der einzelnen Bedingungen des Kontrakts zu überprüfen und eine Änderung an der Kopie auszuführen. Dabei wird das Attribut `value` auf den Wert `13` gesetzt:

```
public static void main(final String[] args)
{
    final BaseCloneable original = new BaseCloneable(47, 11);

    try
    {
        // Kopieren durch Klonen
        // Cast erforderlich, damit man sinnvoll mit der Kopie arbeiten kann
        final BaseCloneable typeSafeClone = (BaseCloneable) original.clone();

        // Verschiedene Prüfungen durchführen
        CloneableUtils.checkContract(original, typeSafeClone);
        modify(original, typeSafeClone);
    }
    catch (final CloneNotSupportedException e)
    {
        // Kann nicht auftreten, muss aber abgefangen werden
    }
}

private static void modify(final BaseCloneable original,
                           final BaseCloneable clone)
{
    // Änderung der Kopie
    System.out.println("\nÄnderung der Kopie:");
    clone.value = 13;
    System.out.println("Original:  " + original);
    System.out.println("Kopie:     " + clone);
```

```
}
```

***Listing 8.11** Ausführbar als* **'BASECLONEABLEEXAMPLE'**

Führen wir das Programm BASECLONEABLEEXAMPLE aus, so erhalten wir folgende Ausgabe auf der Konsole:

```
Kontraktprüfung:
Original: BaseCloneable [id=47, value=11]
Kopie:    BaseCloneable [id=47, value=11]
Objekterzeugung? true
Typgleichheit?   true
Wertgleichheit?  false

Änderung der Kopie:
Original: BaseCloneable [id=47, value=11]
Kopie:    BaseCloneable [id=47, value=13]
```

Tatsächlich verhält sich das Programm gemäß dem Kontrakt. Auffällig ist jedoch die Ausgabe für Wertgleichheit. Diese ist für die Attribute offensichtlich gegeben, ein Vergleich mit `equals(Object)` liefert aber `false`! Das ist zunächst unerwartet, aber logisch: In der Klasse `BaseCloneable` existiert keine überschriebene Methode `equals(Object)`. Daher kommt es zu einem Referenzvergleich, und dieser liefert selbstverständlich `false`. Das zuvor erwähnte Detail bezüglich Wertgleichheit und dem Ergebnis `equals(Object)` wird hier deutlich: Die Realisierung der `equals(Object)`-Methode passt nicht und liefert falsche Ergebnisse. Man kann die Schlussfolgerung ziehen, dass es beim Überschreiben der Methode `clone()` sinnvoll ist, ebenfalls die Methode `equals(Object)` derart zu überschreiben, dass ein semantischer Vergleich der Attribute erfolgt (vgl. Abschnitt 4.1.2).

Hilfreiche, wünschenswerte API-Erleichterungen

Für Klienten ist ein Aufruf der Methode `clone()` mit einigen Unannehmlichkeiten verbunden:

1. Die Methode `clone()` liefert standardmäßig einen Rückgabewert vom Typ `Object`. Das erfordert in der Regel eine `instanceof`-Prüfung und einen Cast, wenn Klienten mit dem kopierten Objekt sinnvoll arbeiten wollen.
2. In der Signatur der Methode `clone()` wird eine `CloneNotSupportedException` definiert. Dadurch müssen Klienten diese Exception weiter propagieren oder mit einem `catch`-Block behandeln.

Aufgrund der vorherigen Erläuterungen ergeben sich folgende Verbesserungsmöglichkeiten in der Implementierung:

1. **Die Methode `clone()` wird kovariant überschrieben:** Klienten können dadurch direkt (ohne Cast) mit der gelieferten Kopie arbeiten.
2. **Die Methode `clone()` wird so überschrieben, dass sie keine Exception wirft:** Wenn eine eigene Klasse das Interface `Cloneable` implementiert und die enthaltenen Attributtypen ebenfalls kopierbar sind, sind Objektkopien folglich immer möglich. Eine `CloneNotSupportedException` kann dann technisch

nicht mehr auftreten, muss aber behandelt werden. Für Klienten wird der Einsatz der Methode `clone()` erleichtert, indem die Exception direkt in der überschriebenen Methode abgefangen wird, statt sie zu propagieren. Im `catch`-Block wird als Reaktion auf diese »unmögliche« Situation ein `java.lang.InternalError` ausgelöst. Dieses Vorgehen findet man in vielen Implementierungen des JDKs.

Setzt man diese Hinweise um, ergibt sich folgende Realisierung:

```
public BaseCloneable clone()
{
    try
    {
        return (BaseCloneable) super.clone();
    }
    catch (final CloneNotSupportedException ex)
    {
        throw new InternalError(ex.getMessage());
    }
}
```

Diese Änderungen befreien Klienten sowohl von der Typprüfung und einem Cast als auch von einer Fehlerbehandlung. Der nutzende Sourcecode wird nun deutlich kürzer, einfacher und lesbarer:

```
public static void main(final String[] args)
{
    // Kopieren durch Klonen
    final BaseCloneable original = new BaseCloneable(47, 11);
    final BaseCloneable clone = original.clone();

    // Verschiedene Prüfungen durchführen
    CloneableUtils.checkContract(original, clone);
    modify(original, clone);
}
```

Listing 8.12 *Ausführbar als* **'BASECLONEABLE2EXAMPLE'**

Die gezeigte Realisierung birgt jedoch im Zusammenhang mit Vererbung einige Schwächen. Betrachten wir dies genauer.

Mögliche Probleme bei Vererbung

Nehmen wir an, eine Klasse `DerivedCloneable` wäre von der Klasse `BaseCloneable` abgeleitet und hätte keine speziellen Anforderungen an die `clone()`-Methode. Da diese in der Basisklasse bereits die gewünschten Eigenschaften besitzt (flache Kopie primitiver Attribute und Sichtbarkeit `public`), ist eine Redefinition an dieser Stelle nicht erforderlich.

Ein Objekt der Klasse `DerivedCloneable` wird durch einen Aufruf von `clone()` kopiert:

```
final DerivedCloneable original = new DerivedCloneable(8, 15, "Test");
// Cast wieder notwendig, weil Rückgabetyp BaseCloneable ist
final DerivedCloneable clone = (DerivedCloneable) original.clone();
```

Anhand dieses Beispiels möchte ich auf zwei Probleme eingehen:

1. Das kovariante Überschreiben sollte Casts bei Klienten vermeiden helfen. Überschreibt man die `clone()` -Methode in einer Basisklasse, so müssen Klienten beim Kopieren von Subklassen allerdings doch wieder einen Cast nach dem Klonen durchführen, um auf die Eigenschaften der Subklasse, hier `DerivedCloneable`, Zugriff zu erhalten.
2. Klienten müssen sich zwar nicht mehr um das Exception Handling kümmern. Subklassen können jedoch auch nicht mehr selbst bestimmen, ob sie das Klonen erlauben wollen oder nicht. Dazu war ursprünglich die `CloneNotSupportedException` vorgesehen, die (in guter Absicht zur Erleichterung der Handhabung) aus der Signatur entfernt wurde.

Wir erkennen nun, dass diese Vereinfachungen *nur* für finale Klassen sinnvoll sind. Für die Basisklasse `BaseCloneable` waren diese Änderungen zu voreilig.

Achtung: Konstruktoraufrufe statt Aufruf von `super.clone()`

Konstruktoraufrufe sollten beim Implementieren der Methode `clone()` vermieden werden. Nehmen wir an, in der Klasse `BaseCloneable` würde in der `clone()` -Methode ein Konstruktoraufruf wie folgt ausgeführt:

```
public BaseClonable clone()
{
    return new BaseCloneable(id, value);
}
```

Problematisch wird die obige Realisierung, wenn ein Objekt des abgeleiteten Typs `DerivedCloneable` kopiert werden soll. Durch den Konstruktoraufruf wird immer lediglich ein Objekt vom Basistyp `BaseCloneable` erzeugt. Der Klassenbestandteil der Subklasse `DerivedCloneable` wird weder erzeugt noch kopiert. Beim Einsatz kommt es dadurch zu Typfehlern und `ClassCastException`s.

Damit alle Klassenbestandteile korrekt konstruiert und initialisiert werden, muss beim Implementieren der Methode `clone()` immer ein expliziter Aufruf von `super.clone()` erfolgen. Auf diese Weise wird der in der Klasse `Object` realisierte Automatismus gestartet. Dieser erzeugt dann korrekterweise ein Objekt vom kopierten Typ, hier `DerivedCloneable`.

Spezialbehandlungen für Container

Enthält ein zu kopierendes Objekt Referenzattribute auf Containerklassen, so müssen nach einem Aufruf des Standardmechanismus noch spezielle Nacharbeiten und Anpassungen durch die überschriebene `clone()` -Methode ausgeführt werden. Dies ist notwendig, um das zurückgelieferte Objekt unabhängig von dem ursprünglichen Objekt zu machen. Betrachten wir folgende Realisierung der Klasse

`DerivedCloneable`, die ein Attribut `persons` vom Typ `List<Person>` definiert:

```
public class DerivedCloneable extends BaseCloneable
{
    private final List<Person> persons = new LinkedList<>();

    public DerivedCloneable(final int id, final long value)
    {
        super(id, value);
    }

    @Override
    public DerivedCloneable clone() throws CloneNotSupportedException
    {
        return (DerivedCloneable) super.clone();
    }
    // …
```

Erstellt man durch einen Aufruf von `clone()` eine flache Kopie, so verweisen die `List<Person>`-Referenzen von Original und Kopie auf dieselbe Liste. Änderungen darauf, etwa ein Aufruf der Methode `add(Person)` oder `clear()`, wirken sich somit im Objektzustand zweier Objekte aus. Dies ist im Allgemeinen unerwünscht. Folgende Lösungsmöglichkeiten sind denkbar:

1. **Kopie der Container** – Man kopiert nur den Container, belässt die darin enthaltenen referenzierten Objekte aber gleich:

```
public class DerivedCloneableV1 extends BaseCloneable
{
    // Nachteil: Durch die Wertzuweisungen außerhalb der Konstruktors
    // können die davon betroffenen Attribute nicht final sein
    private /*final*/ List<Person> persons = new LinkedList<Person>();

    public DerivedCloneableV1(final int id, final long value)
    {
        super(id, value);
    }

    @Override
    public DerivedCloneableV1 clone() throws CloneNotSupportedException
    {
        // Standardautomatismus aufrufen
        final DerivedCloneableV1 clone = (DerivedCloneableV1) super.clone();

        // Nachteil: Wertzuweisungen außerhalb
        clone.persons = new LinkedList<Person>(this.persons);

        return clone;
    }
```

In dieser Implementierung der `clone()`-Methode wird zunächst der Standardmechanismus aufgerufen. Danach erfolgt die Konstruktion und Wertzuweisung des Containerobjekts. Dabei nutzen wir ein praktisches Detail: Alle Containerklassen des Collection-Frameworks bieten einen Copy-Konstruktor (vgl. Abschnitt 8.4.2), der das Kopieren der in der Liste gespeicherten Elemente vereinfacht. Dadurch kann die Zusammensetzung des Containers in der Kopie durch

Einfügen, Löschen, Sortieren etc. beliebig ohne Rückwirkungen auf das Original modifiziert werden. Allerdings sind Änderungen an den enthaltenen referenzierten Objekten durch die Referenzsemantik von Java dann auch als Änderungen im Originalcontainer sichtbar, da dieser auf die gleichen Objekte verweist. Manchmal reicht diese Art der Umsetzung bereits aus. Andernfalls muss eine tiefe Kopie erfolgen.

2. **Tiefe Kopie der Container** – Soll eine (weitgehende) Unabhängigkeit von Original und Kopie erreicht werden, so muss man als Programmierer dafür sorgen, dass alle enthaltenen Elemente ebenfalls kopiert werden – streng genommen muss dies beliebig tief für alle Objekte und dort wiederum enthaltene Objekte erfolgen, um eine vollständig unabhängige Kopie zu garantieren. Drei Strategien sind denkbar:

   (a) **Aufruf der Methode `clone()`** – Alle im Container gespeicherten Elemente werden durch Aufruf der Methode `clone()` kopiert. Das setzt voraus, dass die Objekte das Interface `Cloneable` implementieren. Da Objekte vom Typ `Person` im Beispiel das nicht tun, ist dieser Ansatz für die Liste der `Person`-Objekte nicht möglich. Diese Situation trifft man in der Praxis häufig an.

   (b) **Einsatz eines Copy-Konstruktors** – Alle im Container gespeicherten Elemente werden durch Aufruf eines Copy-Konstruktors kopiert. Diesen bietet die Klasse `Person` ebenfalls nicht an.

   (c) **Einsatz eines »normalen« Konstruktors** – Alle im Container gespeicherten Elemente werden durch Aufruf eines Konstruktors kopiert. Dazu lesen wir die Werte aller benötigten Attribute aus und übergeben diese an den Konstruktor.

Alle Versionen sind relativ aufwendig. Eine mögliche Umsetzung ist in folgendem Listing gezeigt:

```
public class DerivedCloneableV2 extends BaseCloneable
{
    // Nachteil: Durch die Wertzuweisungen außerhalb der Konstruktors
    // können die davon betroffenen Attribute nicht final sein
    private /*final*/ List<Person> persons = new LinkedList<Person>();

    public DerivedCloneableV2(final int id, final long value)
    {
        super(id, value);
    }

    @Override
    public DerivedCloneableV2 clone() throws CloneNotSupportedException
    {
        // Standardautomatismus aufrufen
        final DerivedCloneableV2 clone = (DerivedCloneableV2) super.clone();

        // Nachteil: Wertzuweisungen außerhalb
        clone.persons = new LinkedList<Person>();

        // Tiefe(re) Kopie erzeugen
        final Iterator<Person> it = this.persons.iterator();
```

```
        while (it.hasNext())
        {
            final Person obj = it.next();

            // Variante 1: Rekursiver Aufruf von clone()
            // cloneDeep.persons.add((Person) obj.clone()));

            // Variante 2: Copy-Konstruktor
            // cloneDeep.persons.add(new Person(obj));

            // Variante 3: Aufruf eines normalen Konstruktors
            clone.persons.add(new Person(obj.getName(), obj.getCity(), obj.
                  getBirthday()));
        }

        return clone;
    }
```

Alle Lösungen besitzen die gleichen Nachteile: Zum einen können diejenigen Attribute nicht `final` deklariert werden, die tief kopiert werden, für die also Zuweisungen innerhalb der `clone()`-Methode und somit außerhalb des Konstruktors stattfinden. Zum anderen entsteht einiges an Schreibaufwand. Betrachten wir nun zwei Alternativen, die diese Probleme behandeln.

8.4.2 Alternativen zur Methode `clone()`

Mit der Methode `clone()` kann jedes Objekt kopiert werden, wenn dessen Klasse das Interface `Cloneable` implementiert. Allerdings implementieren viele Klassen das Interface nicht. Als Alternative zum Kopieren durch Klonen kann man einen sogenannten Copy-Konstruktor einführen oder aber Serialisierung nutzen.

Beide Varianten sind in der Regel besser verständlich als eine Umsetzung mit `Cloneable`. Zudem erlauben die beiden Alternativen die Definition finaler und sogar unveränderlicher Attribute selbst bei tiefen Kopien. Wie bereits bekannt, ist dies bei einer Implementierung der Methode `clone()` nicht möglich. Dieser Punkt stellt eine Einschränkung beim Design durch das Interface `Cloneable` dar.

Objektkopien mit einem Copy-Konstruktor erstellen

Eine Variante zum Klonen von Elementen ist es, einen speziellen Konstruktor, einen sogenannten ***Copy-Konstruktor***, bereitzustellen. Diesem wird das zu kopierende Objekt als Parameter übergeben. Ein neues Objekt wird basierend auf den Attributwerten des Originals initialisiert:

```
public class PersonWithCopyConstructor
{
    private final String name;
    private final String city;
    private final Date   birthday;

    // Copy-Konstruktor nimmt anderes Objekt entgegen
```

```
    public PersonWithCopyConstructor(final PersonWithCopyConstructor original)
    {
        Objects.requireNonNull(original, "original must not be null");

        // DRY-Prinzip + Konstruktordesign
        // Keine Wertzuweisungen, Delegation an den normalen Konstruktor
        this(original.name, original.city, original.birthday);
    }

    // normaler Konstruktor
    public PersonWithCopyConstructor(final String name, final String city,
                                     final Date birthday)
    {
        // final + unveränderlich, da String unveränderlich
        this.name = Objects.requireNonNull(name, "name must not be null");
        this.city = Objects.requireNonNull(city, "city must not be null");
        // final + unveränderlich durch Kopie
        Objects.requireNonNull(birthday, "birthday must not be null");
        this.birthday = new Date(birthday.getTime());
    }
    // …
}
```

Bei dieser Realisierung muss zur Sicherstellung einer Unveränderlichkeit eine Kopie des übergebenen veränderlichen `Date`-Objekts erfolgen. Für die `String`-Instanzen sind keine Kopieraktionen erforderlich, da die Klasse `String` unveränderlich ist.

Objektkopien mithilfe von Serialisierung erstellen

Eine weitere Alternative zum Kopieren durch Klonen ist der Einsatz von Serialisierung. Die Idee ist folgende: Man serialisiert das zu kopierende Objekt in einen Ausgabestream. Die Kopie erhält man durch das Deserialisieren eines neuen Objekts aus dem Stream. Diese Funktionalität setzen wir in einer Utility-Klasse `CopyObjectUtils` um. Dort definieren wir eine generische Methode `copyObject(T)` wie folgt:

```
@SuppressWarnings("unchecked")
public static <T extends Object & Serializable> T copyObject(final T original)
{
    Objects.requireNonNull(original, "original must not be null");

    try (final ByteArrayOutputStream bout = new ByteArrayOutputStream();
         final ObjectOutputStream objectout = new ObjectOutputStream(bout))
    {
        // Objekt in Byte-Array schreiben
        objectout.writeObject(original);

        // Objekt aus dem Byte-Array einlesen und konstruieren
        final byte[] objBytes = bout.toByteArray();
        try (final ByteArrayInputStream bin = new
ByteArrayInputStream(objBytes)
             ;
             final ObjectInputStream objectin = new ObjectInputStream(bin);)
        {
            return (T) objectin.readObject();
        }
    }
```

```
    catch (final IOException e)
    {
        // unmöglich, da wir ein Byte-Array zur Ein- und Ausgabe nutzen
    }
    catch (final ClassNotFoundException e)
    {
        // unmöglich, da wir ein Objekt der Klasse gerade geschrieben haben
    }

    // unmöglicher Fall, Anweisung nur zur Vermeidung von Kompilierfehlern
    throw new IllegalStateException("copyObject() failed to copy object of " +
                                    "type: " + original.getClass());
}
```

Die Methode verwendet die bereits bekannten Klassen `ByteArrayOutputStream` und `ObjectOutputStream` sowie `ByteArrayInputStream` und `ObjectInputStream` (vgl. Abschnitt 8.3). Erstere nutzt man, um ein Objekt in ein Byte-Array zu serialisieren. Mithilfe der Klassen `ByteArrayInputStream` und `ObjectInputStream` realisieren wir den umgekehrten Weg aus einem Byte-Array in ein neues Objekt.

Durch die generische Definition mit der Typeinschränkung `& Serializable` kann der Aufruf dieser Methode nur für solche Klassen erfolgen, die tatsächlich das Interface `Serializable` implementieren und damit serialisierbar sind. Es müssen trotzdem zwei Arten von Checked Exceptions abgefangen werden. Beide `catch`-Blöcke können in diesem Spezialfall leer implementiert werden. Eine `IOException` kann nicht auftreten, da wir auf dem Speicher arbeiten und nicht mit einem externen Medium kommunizieren. Die in der Signatur der Methode `readObject()` definierte `ClassNotFoundException` kann im Allgemeinen beim Serialisieren immer auftreten. In diesem Fall ist dies aber unmöglich, da die Klasse bereits beim Aufruf der Methode `copyObject(T)` geladen ist und eine Instanz davon an die Methode übergeben wird. Dem Compiler ist es jedoch nicht möglich, diese Situationen zu erkennen. Dieser geht von einer unvollständigen Implementierung aus (ein möglicher Pfad liefert keine Rückgabe vom Typ `T`). Um einen Kompilierfehler zu vermeiden, müssten wir normalerweise eine `return`-Anweisung mit Rückgabewert angeben. Hier gibt es jedoch keinen sinnvollen Rückgabewert. Diese Situation lässt sich am besten durch eine `IllegalStateException` beschreiben.

Info: Clone-Automatismus vs. Serialisierung

Die Gründe für die bitweise Kopie durch den Clone-Automatismus sind wohl, dass man eine solche in jedem Fall anbieten kann, dies mit geringer Laufzeit möglich ist und dafür nichts von Entwicklern programmiert werden muss. Allerdings ist nur in seltenen Fällen eine solche Form der Kopie hilfreich und sinnvoll! Eine Realisierung als tiefe Kopie hätte jedoch erfordert, dass alle enthaltenen Elemente auch wieder kopierbar sind. Bei dem Automatismus der Serialisierung wurde genau dies umgesetzt. Aggregierte Elemente müssen wiederum serialisierbar sein. Wird diese Forderung nicht eingehalten, so kommt es bei der Serialisierung zu Exceptions.

8.5 Garbage Collection

Während der Ausführung eines Java-Programms werden in der Regel neue Objekte angelegt. Dadurch steigt der Speicherverbrauch eines Programms kontinuierlich. Um diesem Trend entgegenzuwirken, besitzt die JVM ein automatisches Speicherbereinigungssystem, das nicht mehr durch Objekte benötigten Speicher wieder freigibt. Alle noch referenzierten Objekte müssen dabei natürlich erhalten bleiben. Erlischt allerdings die letzte Referenz auf ein Objekt, so kann dieses eingesammelt und der dadurch belegte Speicherplatz freigegeben werden. Dieser Aufräumvorgang wird als ***Garbage Collection*** (GC) bezeichnet und findet im Hintergrund parallel zur eigentlichen Applikation statt. Die ausführende Einheit wird ***Garbage Collector*** genannt.

Durch diesen Automatismus werden Probleme verhindert, die in Programmiersprachen mit expliziter Speicherfreigabe durch zu früh oder niemals freigegebene Objekte entstehen.[9] Allerdings benötigt die Garbage Collection einen gewissen Anteil der Rechenzeit: Es entsteht etwas Overhead, der beim expliziten Anfordern und Freigeben von Speicher nicht anfällt. In sehr vielen Applikationen ist dieser Mehraufwand unbedeutend. In Echtzeitsystemen kann dieser Overhead jedoch problematisch werden, da kaum Aussagen, geschweige denn Garantien, zu den einzelnen Zeitpunkten und den jeweiligen Laufzeiten von Garbage-Collection-Vorgängen gemacht werden können.

8.5.1 Grundlagen zur Garbage Collection

Zur Laufzeit eines Programms werden normalerweise dynamisch neue Objekte erzeugt. Erreicht die Speicherbelegung einen gewissen Grenzwert, so erfolgt eine Garbage Collection, um wieder für mehr freien Speicher zu sorgen.

Freizugebene Objekte erkennen und beseitigen

Vor einer Freigabe müssen überhaupt erst einmal die möglichen Kandidaten dafür ermittelt werden. Für die Garbage Collection sind immer folgende Aufgaben zu erfüllen:

1. **Garbage-Erkennung** – Zunächst müssen alle Objekte ermittelt werden, die nicht länger benötigt werden. Dies ist vereinfacht immer dann der Fall, wenn Objekte von keinem anderen Objekt mehr referenziert werden. Die folgende Hintergrundinfo »Erreichbarkeit von Objekten« geht darauf genauer ein.
2. **Garbage-Beseitigung** – Es muss der durch diese unreferenzierten Objekte belegte Speicher dem Programm wieder zur Verfügung gestellt werden. Zuvor wird Objekten noch die Möglichkeit gegeben, letzte Aufräumarbeiten durchzuführen. Dazu existiert die Methode `finalize()`, die automatisch durch den Garbage

Collector aufgerufen wird, bevor das Objekt entfernt wird (vgl. Abschnitt 8.5.6).

Hintergrundinfo: Erreichbarkeit von Objekten

Der Automatismus der Speicherbereinigung setzt voraus, dass man nicht mehr referenzierte Objekte sicher erkennen kann. Ein (naives) Zählen von Referenzen auf Objekte, das sogenannte ***Reference Counting***, ist dazu nicht besonders geeignet, weil damit zyklische Referenzen nicht zu erkennen sind.

Die JVM berechnet daher stattdessen die ***Erreichbarkeit*** durch Nachverfolgen von Objektreferenzen. Objekte sind erreichbar, wenn es im Programm eine Referenz von einer sogenannten Wurzelreferenz über beliebig viele Indirektionen gibt. Alle derart erreichbaren Objekte sind lebendig. Alle anderen können freigegeben werden, da sie unreferenziert sind und keinen Einfluss auf nachfolgende Berechnungen im Programm mehr besitzen.

Die Ausgangsmenge der Wurzelreferenzen setzt sich wie folgt zusammen:

- Alle Referenzen aus lokalen Variablen im momentanen Stack
- Alle Referenzen, die von lebenden Threads ausgehen
- Alle Referenzen in statischen Attributen[a]

[a] Nicht nur aus OO-Sicht sind statische Attribute möglichst zu vermeiden. Diese wirken sich negativ auf die Laufzeit von Garbage Collections aus.

Lebensdauer von Objekten

Zur Optimierung der Speicherverwaltung nutzt man die Erkenntnis, dass die Lebensdauern von Objekten stark variieren, sich aber grob in zwei Kategorien einteilen lassen: *kurzlebige* Objekte und *langlebige* Objekte. Die große Masse an Objekten besitzt nur eine geringe Lebenszeit, da diese Objekte nur temporär zur Durchführung von Aufgaben erzeugt und benötigt werden. Einige wenige davon leben länger. Daneben findet man auch eine von Applikation zu Applikation variierende Anzahl an langlebigen Objekten – meistens sind dies die zentralen Komponenten des Programms und zum Teil auch andere Daten. Man kann daher Objekte gemäß ihrem Alter grob in eine ***junge*** und eine ***alte Generation*** (***Young Generation*** und ***Old Generation***) aufteilen.

Die Lebenszeit eines Objekts wird durch einen speziellen Zähler ausgedrückt, der die Anzahl der überlebten Garbage-Collection-Vorgänge beschreibt und bei jeder Garbage Collection inkrementiert wird. Entsprechend ihrem Generationszähler werden Objekte verschiedenen Generationen zugeordnet. Für jede Generation ist ein spezifischer Speicherbereich reserviert. Mit zunehmendem Alter wandern die Objekte durch diese Generationen und Speicherbereiche. Betrachten wir daher im Folgenden die Aufteilung und Organisation des Speichers genauer.

Aufteilung des Speichers

Nachdem kurz motiviert wurde, warum eine Aufteilung des Speichers in mehrere Bereiche erfolgt, möchte ich auf die einzelnen Bereiche und ihre Funktion eingehen. Abbildung 8-3 visualisiert die Unterteilung des Speichers der JVM (sofern nicht der neue Garbage Collector G1 zum Einsatz kommt).

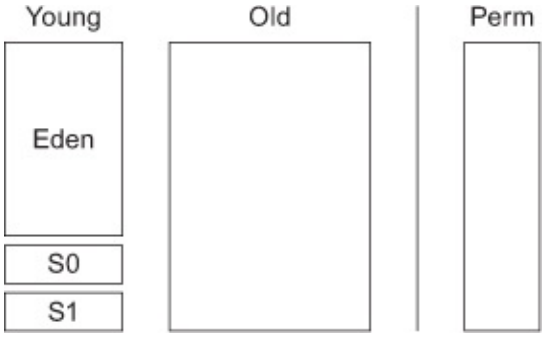


Abbildung 8-3 *Speicherbereiche in der JVM*

1. **Young Generation** – Die Young Generation wird in einen der Größe nach dominierenden ***Eden-Bereich*** und zwei wesentlich kleinere, aber gleich große ***Survivor-Bereiche*** unterteilt. Eden ist der Bereich, in dem alle neuen Objekte angelegt werden und einige Zeit bis zur nächsten Garbage Collection verweilen. Oftmals sind neu erzeugte Objekte recht kurzlebig. Wenige davon leben etwas länger, haben bereits eine oder einige Garbage Collections überstanden und werden dementsprechend als Survivor (»Überlebende«) bezeichnet. Diese Überlebenden werden in den Survivor-Bereichen (Survivor 0 und 1, häufig auch nach ihren Funktionen als From- und To-Survivor benannt) gespeichert.
2. **Old Generation oder Ternured Generation** – Die Old Generation ist nicht unterteilt. Hier befinden sich die Objekte, die länger referenziert werden und bereits mehrere Garbage-Collection-Vorgänge überlebt haben. Einige wenige davon leben sogar so lange wie die Applikation selbst. Dies gilt im Speziellen für statische Collections und deren Elemente (falls diese nicht anderweitig gelöscht werden).
3. **Perm oder Permanent Generation** – Im Perm-Bereich werden alle Klasseninformationen (Klassenobjekte, Methodenobjekte usw.), statische Attribute (nur die Referenzen, nicht aber die referenzierten Objekte) und Stringliterale gespeichert. Dieser Speicher wird durch Verwendung vieler Klassen und durch den Einsatz von Reflection gefüllt. Im Perm finden keine Garbage Collections statt, allerdings werden zum Teil auch hier Aufräumarbeiten durchgeführt, etwa unbenutzte Klassen usw. wieder aus dem Speicher entladen.

Varianten der Garbage Collection

Durch die Untergliederung des Speichers in Generationen können Aufräumvorgänge jeweils auf Teilbereichen stattfinden, wodurch weniger Aufwand entsteht und weniger Zeit für den Vorgang der Garbage Collection verbraucht wird. Des

Weiteren können für jede Generation unterschiedliche Strategien bzw. Algorithmen verwendet werden. Die Art und Anzahl der Aufräumvorgänge kann zudem an die spezifischen Gegebenheiten und die Dynamik der Objekterzeugung angepasst werden: Für die junge Generation werden kleinere Aufräumarbeiten, sogenannte ***Minor Garbage Collections***, durchgeführt. Dies geschieht häufig, da hier eine hohe Fluktuation und »Sterblichkeit« herrscht. Für die ältere Generation werden Aufräumvorgänge seltener benötigt, da die Objekte wahrscheinlich noch referenziert werden. Dort finden aufwendigere Aufräumarbeiten, sogenannte ***Major Garbage Collections***, statt.

Minor GC

Im laufenden Betrieb findet sehr häufig eine sogenannte ***Minor GC*** statt, etwa wenn der Eden-Bereich zu voll geworden ist oder eine gewisse Zeit seit der letzten GC vergangen ist. Bei einer Garbage Collection auf der Young Generation werden zunächst alle noch referenzierten Objekte im Eden-Bereich und From-Survivor ermittelt und dann in den To-Survivor kopiert. Anschließend werden der Eden-Bereich und der From-Survivor als frei betrachtet und die beiden Survivor-Bereiche wechseln ihre Funktion: Aus dem From-Survivor wird der To-Survivor und umgekehrt. Die Existenz der beiden Survivor-Bereiche sorgt dafür, dass auch etwas länger referenzierte Objekte nicht direkt in die Old Generation wandern. Nur für den Fall, dass Objekte für den Survivor-Bereich zu groß sind, werden sie direkt in die Old Generation übertragen. Zudem werden alle Objekte, die eine gewisse Anzahl an Garbage-Collection-Vorgängen überlebt haben, in die Old Generation verschoben. Auf diese Weise verbleibt in der Regel lediglich eine kleine Menge an aktiven Objekten, die kopiert werden müssen. Dadurch können Minor GCs schnell und häufig ausgeführt werden, und das sogar vielfach ohne spürbare Beeinträchtigung der Laufzeit des Programms.

Major GC oder Full GC

Viel seltener als Minor GCs wird die aufwendigere und langsamere Garbage Collection in der Old Generation durchgeführt. Tritt allerdings die Situation ein, dass auch die Old Generation einen gewissen maximalen Füllstand erreicht und Speicherbedarf besteht, so werden alle Speicherbereiche einer Garbage Collection unterzogen, um möglichst viel Speicher freigeben zu können.[10] Ein solcher Komplettaufräumvorgang wird als ***Major GC*** oder auch ***Full GC*** bezeichnet und kann die Programmausführung spürbar verlangsamen. Das liegt daran, dass wesentlich mehr Aufwand als bei einer Minor GC betrieben werden muss. Erstens wird ein größerer Speicherbereich aufgeräumt, was eine Garbage Collection auf der alten Generation bereits zeitaufwendiger macht. Zweitens ist die Wahrscheinlichkeit hoch, dass viele Objekte auch weiterhin referenziert werden und somit eine umfangreiche Analyse nötig wird, um freizugebende Objekte zu ermitteln. Drittens muss im Gegensatz zu dem trickreichen Kopieren der Minor GC, das automatisch für

frischen, zusammenhängenden, freien Speicher sorgt, in der Old Generation dafür mehr Aufwand betrieben werden. Durch den Garbage Collector freigegebene Objekte würden für Löcher im Speicher wie in einem Schweizer Käse sorgen, wenn nicht ein spezieller Kompaktierungsschritt eine solche Fragmentierung des Speichers vermeiden würde. Diese Defragmentierung ist wichtig, um möglichst für ein großes zusammenhängendes Stück freien Speichers zu sorgen, und so jederzeit auch größere Speicheranfragen bedienen zu können.

8.5.2 Herkömmliche Algorithmen zur Garbage Collection

Zur Bereinigung des Speichers existieren verschiedene Algorithmen, die bis zur Einführung des Garbage Collectors G1 mit JDK 7 immer auf der Unterteilung des Speichers in verschiedene Bereiche basierten, die durch Kopier- und Kompaktieroperationen bereinigt wurden. Nachfolgend möchte ich ein wenig ausführlicher auf die Vorgänge eingehen, um das Verständnis zu erleichtern.

Mark-and-Sweep-Algorithmus

Die Arbeitsweise des sogenannten ***Mark-and-Sweep-Algorithmus*** kann man sich in etwa wie folgt vorstellen: Die Aufräumarbeiten finden in zwei Durchgängen statt. In einem ersten Durchgang, der sogenannte Mark-Phase, werden die lebendigen Objekte ermittelt, indem deren Erreichbarkeit durch Nachverfolgen der Wurzelreferenzen geprüft wird. Jedes lebendige Objekt wird »markiert«. Alle nicht markierten Objekte sind unreferenziert und werden in einem zweiten Durchgang, der sogenannten Sweep-Phase, freigegeben. Der durch diese Objekte belegte Speicher steht der Applikation daraufhin wieder zur Verfügung.

Diese Art der Verwaltung und der Freigabe von Speicher kann allerdings zu einer Fragmentierung des Speichers führen, da immer nur Speicherbereiche freigegeben werden – jedoch niemals freie Bereiche zu größeren Einheiten zusammengefasst werden. Mit zunehmender Programmlaufzeit steigt die Wahrscheinlichkeit, dass sich die Objekte zufällig auf den gesamten Speicher verteilen und es so zu einer starken Zerstückelung kommt: Zwischen den Objekten entstehen kleinere und größere freie Bereiche. Diesen Effekt kennt man beispielsweise von Festplatten. Dort hilft dann der Aufruf einer Defragmentierung. Die im Folgenden beschriebenen Algorithmen »Stop and Copy« und »Mark and Compact« vermeiden eine Fragmentierung des Speichers.

Stop-and-Copy-Algorithmus

Für diesen Algorithmus wird der Speicher in zwei Bereiche geteilt: einen aktiven Bereich, in dem Objekte angelegt werden, und einen gleichgroßen inaktiven Bereich, der unbelegt bleibt und nur während eines Garbage-Collection-Vorgangs benötigt

wird.

Wird der verfügbare Speicher im aktiven Bereich knapp, so wird das Programm angehalten (daher der Namensteil »Stop«) und eine Garbage Collection durchgeführt. Alle noch referenzierten Objekte werden von der aktiven in die inaktive Region kopiert (daher der Namensteil »Copy«) und die Referenzen im Programm entsprechend angepasst. Nach diesem Vorgang wird die zuvor inaktive Region aktiv. Sie enthält dann nur noch die momentan tatsächlich benötigten Objekte. Die zu entfernenden Objekte verbleiben in dem nun inaktiven Bereich. Dieser wird anschließend einfach als frei markiert. Dadurch sind dort keine Aufräumarbeiten mehr nötig – außer den Aufrufen an die später beschriebene Methode `finalize()`.

Während des Kopiervorgangs lassen sich (bei Bedarf) alle Objekte anordnen, wodurch ein Fragmentierung vermieden werden kann. Das stellt einen Vorteil gegenüber dem Mark-and-Sweep-Algorithmus dar. Allerdings stehen diesem Vorteil zwei Nachteile gegenüber: Zum einen müssen alle noch lebendigen Objekte kopiert und in ihrer Referenz angepasst werden. Die Verarbeitungskosten steigen linear zu dem durch die Objekte belegten Speicher. Zum anderen wird immer doppelt so viel Speicher benötigt, wie die Applikation eigentlich benutzt, und nach einer Garbage Collection liegt eine Hälfte des Speichers brach.

Für die Young Generation wird daher ein modifizierter Stop-and-Copy-Algorithmus verwendet. Die Aufteilung des Speichers variiert: Die aktive Region entspricht dem Eden-Bereich. Die inaktive Region wird durch einen Survivor-Bereich realisiert. Diese Reduktion der Größe des inaktiven Bereichs im Vergleich zur zuvor beschriebenen Originalversion des Stop-and-Copy-Algorithmus ist aufgrund der Kurzlebigkeit möglich. Es kann hier also ein viel kleinerer Kopierbereich genutzt werden.

Mark-and-Compact-Algorithmus

Der Mark-and-Sweep-Algorithmus führt häufig zu einer Fragmentierung des Speichers. Der Stop-and-Copy-Algorithmus verhindert zwar Fragmentierung, verdoppelt aber den Speicherverbrauch. Der ***Mark-and-Compact-Algorithmus*** ist eine Kombination der zuvor genannten Algorithmen und läuft wiederum in zwei Phasen ab. Die erste Phase ist identisch zu der des Mark-and-Sweep-Algorithmus und markiert alle erreichbaren Objekte. Die zweite Phase ist ähnlich zu der Copy-Phase im Stop-and-Copy-Algorithmus. Im Gegensatz dazu werden allerdings die erreichbaren Objekte kompaktiert, indem diese an eine andere Stelle innerhalb des Speichers kopiert werden. Der restliche Bereich wird anschließend als frei betrachtet. Durch diese Kopieraktion liegen alle Objekte kompakt im Speicher hintereinander. Man vermeidet so Fragmentierung, ohne die Speicherkosten des Stop-and-Copy-Algorithmus zu verursachen.

Für die Old Generation wird standardmäßig ein Mark-and-Compact-Algorithmus verwendet, der eine Fragmentierung des Speichers vermeidet.

8.5.3 Einflussfaktoren auf die Garbage Collection

In der Regel befreit die Automatik der Speicherbereinigung den Programmierer davon, sich allzu viele Gedanken über die Speicherverwaltung machen zu müssen. Als Programmierer hat man kaum Einfluss auf die Garbage Collection, weder auf den Zeitpunkt noch auf den Umfang. Allerdings sollte man doch ein paar Dinge wissen, um die Garbage Collection möglichst effizient werden zu lassen. Warum ist das wichtig?

Beim Vorgang der Garbage Collection werden für einige Aufräumschritte alle laufenden Threads suspendiert, um einen konsistenten Speicherstatus zu erreichen. Das Programm wird erst nach Abschluss der Garbage Collection fortgesetzt. Dadurch kommt es zu einer Pause in der Programmabarbeitung. Um ein flüssiges Arbeiten zu gewährleisten, möchte man die Zeitdauer dieser Unterbrechung möglichst minimieren. Zwar dauert die Garbage Collection vielfach nur einige (Milli-)Sekundenbruchteile – im Extremfall kann dies aber auch einige Sekunden und mehr andauern. Dafür sind dann Komplettaufräumarbeiten, also Full GCs, verantwortlich. Um spürbare Performance-Einbußen zu verhindern, sollten aufwendige Full-GC-Phasen möglichst vermieden werden.

Automatik vs. manueller Aufruf

Normalerweise wird die Garbage Collection automatisch aufgerufen, falls Speicher freigegeben und wieder bereitgestellt werden muss. Darüber hinaus kann man zur Laufzeit eines Programms den Wunsch zur Ausführung einer Garbage Collection explizit durch einen Aufruf der statischen Methode `System.gc()` »äußern«. Tatsächlich löst ein Aufruf von `System.gc()` die Garbage Collection nicht in jedem Fall aus. Der JVM steht es frei, diese Aufforderung zu ignorieren. Ein expliziter Aufruf von `System.gc()` ist in der Regel auch überflüssig und sollte unterbleiben. Eine Ausnahme bilden Situationen, in denen zu einem bestimmten Zeitpunkt so viel Speicher wie möglich freigegeben werden soll, um anschließend einen größeren freien Speicherbereich bereitstellen zu können. Da die Aufrufe von `System.gc()` aber potenziell ignoriert werden können, muss man dies normalerweise mehrfach machen.

Oftmals ist es sinnvoller, dem Garbage Collector indirekt dadurch zu helfen, dass man Referenzvariablen wieder auf `null` setzt – mit dem Ziel, dass diese damit nicht bis in die Old Generation »überleben« und noch durch eine Minor GC entfernt werden können. Gleiches gilt für Collections: Werden größere Datenbestände nicht mehr benötigt, so sollte man diese möglichst schnell aufräumen, etwa indem diese durch einen Aufruf von `remove()` oder `clear()` aus dem Container entfernt werden.

Parametrierung zur Optimierung der Garbage Collection

Durch geschickte Parametrierung der Größenverhältnisse der einzelnen Bereiche kann man aufwendige Full GCs unwahrscheinlich machen. Für die Garbage Collection kann eine Konfiguration durch die in der Aufzählung genannten JVM-Aufrufparameter erfolgen und zu einer Performance-Steigerung führen.

- **`-Xms`** und **`-Xmx`** – Mit `-Xms` legt man die initiale Heap-Größe fest. Diese kann bei Bedarf maximal auf den mit `-Xmx` angegebenen Wert wachsen. Der Heap kann zur Laufzeit nicht nur wachsen, sondern auch wieder schrumpfen. Um Performance-Probleme durch zu wenig Speicher und eigentlich unnötige Garbage Collections zu vermeiden, setzt man in der Regel den Wert für `-Xmx`.
- **`-XX:NewSize`** und **`-XX:MaxNewSize`** – Mit `-XX:NewSize` legt man die initiale Größe der Young Generation fest. Diese kann bei Bedarf maximal auf den mit `-XX:MaxNewSize` angegebenen Wert wachsen. Über diese Parameter steuert man, wie viel Speicherplatz für junge Objekte im Heap-Speicher verbraucht werden darf. Durch eine problemangepasste Einstellung kann man den Aufwand für Garbage Collections reduzieren. Somit lässt sich häufig die Antwortzeit und der Durchsatz einer Applikation leicht verbessern. Wählt man hier einen zu niedrigen Wert, so kommt es vermehrt zu Minor GCs. Eine größere Young Generation verringert die Häufigkeit von Full GCs. Allerdings nimmt dadurch die Dauer der Minor GCs zu. Als Faustregel sollte die Young Generation niemals größer als die Old Generation sein. Für viele Anwendungen haben sich Werte etwa im Bereich von 1/4 bis 1/3 der Größe des Heap-Speichers als ideal erwiesen.
- **`-XX:NewRatio`** – Statt über absolute Größenangaben kann man das Verhältnis von Young Generation und Old Generation relativ festlegen. Dieser Wert wird nicht in Prozent, sondern nach der Formel *Young Generation : Old Generation* festgelegt, wobei die Größe der Young Generation immer auf 1 normiert ist. Ein Wert von 2 bedeutet demnach ein Verhältnis von 1:2 oder 1/3 des Gesamt-Heaps.
- **`-XX:SurvivorRatio`** – Man kann das Größenverhältnis innerhalb der Young Generation zwischen dem Eden-Bereich und den Survivor-Bereichen festlegen. Bei höheren Werten dominiert der Eden-Bereich. Dies ist sinnvoll, wenn sehr viel Objekte mit sehr kurzer Lebensdauer angelegt werden. Werden die Survivor-Bereiche allerdings zu klein, so werden größere, neu angelegte Objekte mit höherer Wahrscheinlichkeit direkt in die Old Generation übertragen, da es für sie keinen Platz mehr im Survivor-Bereich gibt. Auch für diesen Parameter erfolgt eine (nicht intuitive) Angabe in der Form *Survivor : Eden*. Allerdings ist hierbei zu beachten, dass es zwei Survivor-Bereiche gibt. Wählt man einen Wert von 16, gilt statt 1 : 16 vielmehr 2 : 16. Daraus folgt, dass der Eden-Bereich 18-mal so groß ist wie ein Survivor-Bereich.
- **`-XX:MaxTenuringThreshold`** – Bei jeder Garbage Collection ermittelt die JVM, wie häufig Objekte bereits zwischen den Survivor-Bereichen hin und her kopiert wurden. der Parameter `-XX:MaxTenuringThreshold` legt fest, nach

welcher maximalen Anzahl an Kopiervorgängen ein Objekt in die Old Generation wandert.

Anpassungen des Perm-Bereichs

Der Perm-Bereich ist unabhängig vom Heap-Speicher der JVM und wird separat von der JVM verwaltet. Das gilt jedoch nur bis JDK 8, denn dort wurde dieser Speicher abgeschafft (vgl. Abschnitt 15.8). Sofern Sie noch mit JDK 7 oder früher arbeiten, kann man für den Perm-Bereich eine initiale Größe vorgeben. Für normale Applikationen werden standardmäßig 32 MB für den Client-Betrieb bzw. 64 MB für den Server-Modus der JVM bereitgestellt. Werden beispielsweise viele 3rd-Party-Bibliotheken benutzt, kann dieser Standardwert aber zu gering sein. In diesem Fall sollte man über die JVM-Aufrufparameter `-XX:PermSize` und `-XX:MaxPermSize` einen größeren initialen und maximalen Wert festlegen. Im Gegensatz zur Einstellung von `-Xmx`, bei der der Wert direkt dem Aufrufparameter folgt (z. B. `-Xmx768m`), wird für die Angabe der Größe des Perm-Bereichs folgende Notation verwendet: `-XX:MaxPermSize=128m`.

8.5.4 Der Garbage Collector »G1«

Die bisher in der JVM (von Sun) verwendeten Garbage-Collection-Algorithmen, etwa der Concurrent Mark and Sweep Collector (CMS), teilen den Heap in unterschiedliche Bereiche für kurz lebende und lang lebende Objekte ein (vgl. Abschnitt 8.5.1). Mit JDK 7 wird ein neuer Garbage Collector namens G1 (als Abkürzung für »Garbage First«) eingeführt, der eine andere Speicheraufteilung verwendet.

Ein Problem bei den bisher beschriebenen Garbage-Collection-Varianten besteht darin, dass man als Entwickler wenig Einflussmöglichkeiten hat: Zwar lassen sich verschiedene Einstellungen vornehmen, jedoch kann man nicht festlegen, wie viel Speicher in einem Durchlauf freigegeben werden soll oder wie lang der Vorgang der GC die Programmausführung maximal unterbrechen darf. Daher kann es teilweise zu störenden Pausen kommen. Insbesondere lassen sich keine Echtzeitanforderungen mit garantierten Antwortzeiten erfüllen. Auch für Action-Spiele und eine flüssige Darstellung komplexerer Landschaften mit butterweichem Scrolling muss man sich in Java einiger Tricks bedienen. Der neue G1-Collector adressiert die Schwachpunkte der bisherigen Verfahren und erlaubt es, mehr Einfluss auf die Garbage Collection zu nehmen und im Speziellen die Unterbrechungen kurz zu halten.

Arbeitsweise

Der neue G1-Collector funktioniert wie folgt: Der Heap wird in eine Reihe von fixen Teilbereichen zerlegt. Zu jedem Teilbereich wird eine Liste mit Referenzen von

Objekten geführt. Diese wird »Remember Set« genannt und enthält Einträge von Referenzen auf die Objekte in diesem Bereich. Veränderungen von Referenzen werden durch die JVM registriert und führen zu einer Anpassung der Remember Sets.

Wenn eine Garbage Collection notwendig wird, werden zunächst die Regionen aufgeräumt, in denen sich nur wenig »lebende«, d. h. weiterhin referenzierte Objekte befinden, in denen sich also viel »Müll« angesammelt hat. Daher stammt auch der Name »Garbage First«. Im besten Fall kann ein ganzer Teilbereich ohne großen Aufwand freigegeben werden, wenn das Remember Set leer ist. Zudem besteht der Vorteil, dass »Müll« leicht erkannt werden kann: Existieren keine eingehenden Referenzen, so ist kein aufwendiges Markieren und Kompaktieren (Mark-and-Sweep) notwendig, sondern ein Bereich kann dann einfach als frei betrachtet werden. Außerdem kompaktiert der G1, indem er alle aktiven Objekte an das Ende des Heaps bewegt. Dadurch wird einer Fragmentierung entgegengewirkt und es steht immer ein möglichst großes Stück freier Speicher zur Verfügung.

8.5.5 Memory Leaks: Gibt es die auch in Java?!

Die automatische Freigabe des Speichers von nicht länger referenzierten Objekte durch die Garbage Collection verleitet viele Entwickler dazu, sich keine Gedanken um die Speicherverwaltung und mögliche Memory Leaks zu machen. Häufig ist das auch in Ordnung. Bei Implementierungen von eigenen Containerklassen ist allerdings besondere Vorsicht geboten. In diesem Abschnitt stelle ich dies anhand einer simplen Implementierung eines Stacks dar. Die Ideen basieren auf Item 6 aus »Effective Java« [9].

Nehmen wir an, folgende Klasse `FixedSizeStack` sei unsere Implementierung eines größenbeschränkten Stacks:

```
// Achtung: Problematische Implementierung: Potenzielles Memory Leak
// in pop() durch fehlende Freigabe der Referenz im Stack selbst
public final class FixedSizeStack
{
    private final Object[] stack;
    private int            index = 0;

    private FixedSizeStack(final int size)
    {
        stack = new Object[size];
    }

    public void push(final Object objectToInsert)
    {
        if(index >= stack.length)
            throw new IllegalStateException("capacity of stack exceeded!");

        stack[index] = objectToInsert;
        index++;
    }
```

```
    public Object pop()
    {
        if( index == 0)
            throw new IllegalStateException( "stack is empty! ");

        index--;
        return stack[ index];
    }
}
```

Würden Sie denken, dass diese Klasse ein Memory Leak enthält? Wahrscheinlich nicht! Überlegen wir einmal, was passiert, wenn die Methode `pop()` aufgerufen wird. Offensichtlich wird eine Referenz auf das gespeicherte Objekt zurückgeliefert. Das Gefährliche daran ist, dass die Klasse `FixedSizeStack` selbst noch eine Referenz auf dieses Objekt behält, wie dies Abbildung 8-4 zeigt. Selbst wenn ein Aufrufer von `pop()` seine Arbeit mit dem Objekt beendet hat und die erhaltene Referenz freigibt, kann der Garbage Collector das Objekt nicht löschen, bis der Stack erneut wächst und die Referenz dadurch befreit wird, dass der Speicherplatz im Array überschrieben wird.

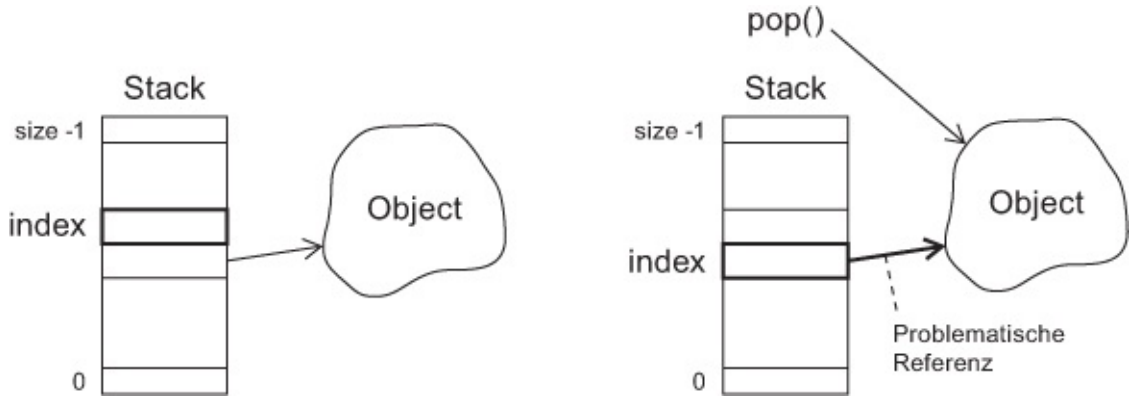


Abbildung 8-4 *Referenzen auf Objekte nach Aufruf von* `pop()`

Wie kann die Methode korrigiert werden? Wir müssen dafür sorgen, dass die Referenz auf das zurückgelieferte Objekt in der Klasse `FixedSizeStack` ausgetragen wird:

```
public E pop()
{
    if( index == 0)
        throw new IllegalStateException( "stack is empty! ");

    index--;

    final E obj = stack[ index];
    // Referenz auf null setzen => Freigabe durch Garbage Collector möglich!
    stack[ index] = null;

    return obj;
}
```

Diese Korrektur verdeutlicht, dass der von Objekten belegte Speicher erst dann freigegeben werden kann, wenn keine Referenzierung des Objekts mehr erfolgt.

Es gibt jedoch Spezialfälle von zirkulären Abhängigkeiten, die unter dem Namen »***isolierte Inseln***« bekannt sind. Im einfachsten Fall referenzieren sich zwei Objekte

gegenseitig, werden aber von keinem anderen Objekt mehr referenziert. Das zeigt Abbildung 8-5. Derartige Konstellationen von gegenseitigen Referenzierungen können vom Garbage Collector erkannt und der belegte Speicher freigegeben werden.

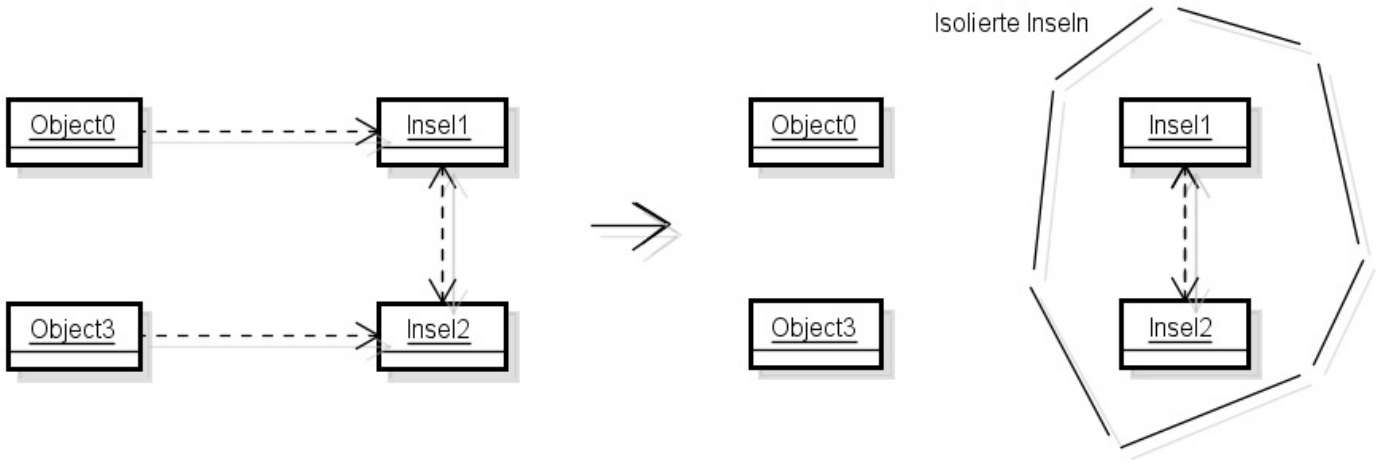


Abbildung 8-5 *Zirkuläre Referenzen*

8.5.6 Objektzerstörung und `finalize()`

Nachdem der Garbage Collector freizugebende Objekte ermittelt hat, wird für jedes dieser Objekte dessen `finalize()`-Methode aufgerufen. Eine Implementierung wird ***Finalizer*** genannt. Man kann diese als »letzten Willen« des Objekts vor der Zerstörung betrachten. Um diesen zu formulieren, kann die Methode `finalize()` überschrieben werden. Dabei sollte auf jeden Fall die Implementierung der Oberklasse mit `super.finalize()` aufgerufen werden, um in allen Oberklassenbestandteilen eventuell notwendige Aufräumarbeiten durchführen zu können.

Die Analogie zum »letzten Willen« ist sehr zutreffend, denn ob und wann dieser befolgt wird oder nicht, ist fraglich. Ein Aufruf von Methode `finalize()` kann unter Umständen erst beim Programmende ausgeführt werden. Bei einem Absturz erfolgt aber möglicherweise auch kein Aufruf.

Es ist guter Programmierstil, sich nicht auf die Aufräumarbeiten der `finalize()`-Methode zu verlassen. Der Grund dafür ist, dass diese eventuell erst sehr spät im Programmablauf oder sogar erst bei der Terminierung der JVM ausgeführt wird. Es empfiehlt sich daher stattdessen, dass eine Klasse spezielle Methoden zum Aufräumen anbietet, die von Nutzern aufgerufen werden sollten. Diese Aufräummethoden können auch als letztes Sicherheitsnetz in der Methode `finalize()` aufgerufen werden.

Im folgenden Listing der Klasse `FinalizeExample` ist dieses Vorgehen durch eine `cleanUp()`-Methode angedeutet, die belegte Ressourcen auf eine definierte Art und Weise freigibt. Wird die Methode `cleanUp()` nicht explizit vor dem Lebensende eines Objekts aufgerufen, so erfolgt dies später implizit im Prozess der

Garbage Collection beim Aufruf von `finalize()`.

```
public final class FinalizeExample
{
    // null ist Indikator, ob bereits initialisiert oder nicht
    private InputStream inStream = null;

    public FinalizeExample(final String inputFileName) throws IOException
    {
        inStream = new FileInputStream(inputFileName); // IOException
    }

    protected void finalize() throws Throwable
    {
        try
        {
            cleanUp(); // Erst eigene Aufräumarbeiten ausführen
        }
        finally
        {
            super.finalize(); // Dann Methode der Basisklasse ausführen
        }
    }

    public synchronized void cleanUp()
    {
        IOUtils.closeQuietly(inStream);
        // Markiere InputStream als nicht mehr benutzbar
        inStream = null;
    }
}
```

Auch im JDK findet man dieses Verfahren, um Ressourcen freizugeben. So besitzen etwa die Klassen `FileInputStream` und `FileOutputStream` eine `finalize()`-Methode, die die Dateiressourcen wieder freigibt, sofern dies nicht innerhalb der Applikation korrekt geschieht. ***Allerdings sollte man sich auf diesen Mechanismus nicht abstützen, da ansonsten Ressourcen möglicherweise deutlich länger – nämlich bis zum Lebensende des Objekts – belegt bleiben, als dies tatsächlich notwendig ist***. Als letzte Sicherheit kann ein Finalizer sinnvoll sein, um zu vermeiden, dass durch Programmierfehler Ressourcen endlos (genauer: bis zum Ende der JVM und dem Zeitpunkt, zu dem sie vom Betriebssystem wieder freigegeben werden) belegt bleiben.

Tipp: Spezielle Referenzen

Normale Referenzen (zum Teil zur besseren Unterscheidung auch »starke« Referenzen genannt) verhindern, dass der Garbage Collector ein derart referenziertes Objekt entfernt. Manchmal, etwa zur Realisierung speichersensitiver Caches, ist es aber durchaus wünschenswert und sinnvoll, diese Forderung etwas schwächer zu fassen, d. h., für Objekte selbst dann eine Garbage Collection zu erlauben, wenn diese noch referenziert werden. In dem genannten Beispiel würde man sich wünschen, dass ein Objekt freigegeben werden kann, wenn keine Referenzen aus der Applikation selbst mehr existieren, d. h., eine Freigabe bei

alleiniger Referenzierung aus dem Cache erfolgen kann.

Im Package `java.lang.ref` sind dazu spezielle Referenzklassen definiert, mit denen diese Funktionalität umgesetzt werden kann. Deren Basis bildet die abstrakte Klasse `Reference`. Diese besitzt unter anderem folgende Subklassen:

- **`SoftReference`** – Soft References sind eine spezielle speichersensitive Form normaler Referenzen. Solange genügend Hauptspeicher vorhanden ist, ist der einzige Unterschied eine Indirektion beim Zugriff. Wird der einer JVM zugeteilte Hauptspeicher knapp, können durch Soft References referenzierte Objekte durch den Garbage Collector freigegeben werden.
- **`WeakReference`** – Weak References sind »schwächer« als Soft References und werden von der Klasse `WeakReference` realisiert. Wenn ein Objekt nur noch durch eine oder mehrere solcher Weak References referenziert wird, kann dieses vom Garbage Collector eingesammelt und (später) gelöscht werden.

8.6 Weiterführende Literatur

Die in diesem Kapitel behandelten Themen Reflection und Garbage Collection sind komplex und umfangreich. Daher konnte in diesem Kapitel nur ein Einblick gegeben werden. Weiterführende Informationen finden Sie in folgenden Büchern:

- »**Java Reflection in Action**« von Ira R. und Nate Forman [26]

  Dies ist eines der wenigen Bücher, die das Thema Reflection mit größerem Tiefgang behandeln.

- »**Core Java – Band 2 Expertenwissen**« von Cay S. Horstmann und Gary Cornell [46]

  In diesem Buch liefern Cay S. Horstmann und Gary Cornell einen umfassenden Überblick über fortgeschrittene Java-Themen. Insbesondere Annotations werden dort ausführlicher als in diesem Kapitel dargestellt.

- »**Java 6 Core Techniken**« von Friedrich Esser [23]

  In diesem Buch geht Friedrich Esser tief gehend auf fortgeschrittene Java-Themen ein. Insbesondere werden Generics und Reflection sowie die Verarbeitung von Annotations ausführlich behandelt.

Weiterführende Informationen zur Garbage Collection finden Sie in folgenden Quellen:

- `http://www.artima.com/insidejvm/ed2/gcP.html`
- `http://docs.oracle.com/javase/8/docs/technotes/guides/vm/g`

9 Programmierung grafischer Benutzeroberflächen mit Swing

Dieses Kapitel gibt einen Einstieg in die Programmierung grafischer Benutzeroberflächen mithilfe der Swing-Bibliothek des JDKs. Die hier vermittelten Kenntnisse sollen Ihnen dabei helfen, Ihre Programme zuverlässig, ergonomisch und intuitiv, kurzum gut bedienbar zu machen. Weil heutzutage die Anforderungen und Erwartungen immer mehr steigen, wurde mit JavaFX eine neue GUI-Technologie eingeführt, die Swing in Zukunft ablösen soll. Doch bis einschließlich JDK 8 fehlt JavaFX noch die letzte Reife und sogar elementare Dinge wie die in Business-Applikationen gebräuchlichen Dialoge. Insbesondere für Fancy-Design-Oberflächen mit Animationen und Effekten ist JavaFX geradezu ideal. JavaFX werde ich später separat in Kapitel 14 behandeln. In diesem Kapitel stelle ich zunächst die Grundlagen zur GUI-Programmierung vor, wodurch das Verständnis der Ausführungen zu JavaFX erleichtert wird.

Abschnitt 9.1 beginnt mit Basiswissen zu Bedienelementen und Containern, zum Layoutmanagement und zur Ereignisbehandlung. Neben diesen Grundlagen ist beim Entwurf grafischer Benutzeroberflächen zu berücksichtigen, dass zum Teil länger dauernde Aktionen ausgeführt werden müssen, die die Bedienung des Programms nicht behindern sollen. Das erfordert häufig den Einsatz von Bearbeitungs-Threads, deren (Zwischen-)Ergebnisse korrekt mit der Benutzeroberfläche abgeglichen werden müssen. Abschnitt 9.2 beschreibt gängige Probleme und mögliche Lösungen beim Einsatz von Multithreading in Swing-Benutzeroberflächen.

Nach Lektüre der ersten beiden Abschnitte haben Sie dann bereits ein gutes Verständnis für die Gestaltung klassischer Benutzeroberflächen erworben. Manchmal möchte man das GUI aber optisch ein wenig interessanter gestalten. In Abschnitt 9.3 lernen wir dazu die Grafikbibliothek Java 2D und einige ihrer Möglichkeiten kennen, mit deren Hilfe man verschiedenste Grafikeffekte, wie Transparenz, Transformationen, Farbverläufe usw., in eigene GUIs integrieren kann. Als Schmankerl werden wir ein Tachometer als eigenes GUI-Element entwickeln.

Nach diesem Ausflug in die Welt selbst gestalteter GUIs komme ich am Ende dieses Kapitels zu wichtigen, komplexeren Bedienelementen klassischer Benutzeroberflächen zurück. In Abschnitt 9.4 stelle ich dazu Bäume, Listen und Tabellen vor. Diese sind aus professionellen Anwendungen kaum wegzudenken, einfach weil diese Bedienelemente die strukturierte Darstellung umfangreicher Datenmengen, die Sortierung bzw. Gruppierung der Daten sowie die Selektion einzelner und mehrerer Einträge unterstützen.

9.1 Grundlagen zu grafischen Oberflächen

Dieser Abschnitt beschreibt zum besseren Verständnis dieses Kapitels und als kurze Auffrischung einiges an Basiswissen zu grafischen Benutzeroberflächen. Zunächst gebe ich einen kurzen Überblick über das AWT (Abstract Window Toolkit) sowie über Swing und Java 2D. Im Anschluss daran erfolgt als Schnelleinstieg eine erste Beschreibung der vier Themengebiete Bedienelemente, Container, Layoutmanagement und Ereignisverarbeitung. Eine inhaltliche Verfeinerung liefern dann die nachfolgenden Abschnitte.

Einführung

Die meisten Anwender erwarten heutzutage eine ansprechende und ergonomische Gestaltung der Benutzeroberfläche, die es erlaubt, das Programm einfach, schnell und intuitiv zu bedienen. Moderne grafische Benutzeroberflächen bieten dazu eine große Auswahl an Bedienelementen, etwa spezielle Eingabefelder für Datumsangaben oder sortierbare Tabellen. Neben dem Entwurf mit eher konventionellen Bedienelementen kann man Benutzeroberflächen aber auch realen Geräten nachempfinden, indem z. B. ein Drehregler oder eine Tachometeranzeige als Bedienelemente nachgebildet werden. Derartige Bedienelemente gewinnen seit dem Boom von stark grafisch interaktiv orientierten Smartphones und Tablet-PCs zunehmend an Bedeutung. Wir werden sehen, wie man Derartiges mithilfe von Java 2D realisieren kann. Gerade in diesem Bereich liegen die Stärken von JavaFX.

Begriffe: Benutzeroberfläche vs. Benutzungsoberfläche

Die Begriffsbildung zur Beschreibung grafischer Oberflächen ist nicht ganz eindeutig. Im Englischen verwendet man den Begriff ***GUI*** für Graphical User Interface. Direkt übersetzt würde man also von einer grafischen Benutzeroberfläche sprechen. Einige Fachleute argumentieren nicht zu Unrecht, dass der Begriff nicht ganz korrekt sei. Es wird ja nicht die Oberfläche des Benutzers, sondern die Oberfläche zur Benutzung eines Programms beschrieben. Insofern sprechen diese Fachleute von grafischen Benutzungsoberflächen. Manchmal hört und liest man auch die Begriffe Bedien-, Bedienungs- und Bedieneroberfläche. Im folgenden Text werde ich den weitverbreiteten Begriff Benutzeroberfläche verwenden.

Überblick

Das JDK enthält zur Gestaltung grafischer Benutzeroberflächen das Toolkit AWT und die Swing-Bibliothek. Beide entstammen den JFC (Java Foundation Classes), die sich grob wie folgt unterteilen lassen:

- **AWT** – Das AWT nutzt ausschließlich diejenigen Bedienelemente (im Folgenden manchmal auch GUI-Komponenten genannt), die vom Betriebssystem nativ zur Verfügung gestellt werden. Aufgrund der Forderung nach Plattformunabhängigkeit von Java umfasst das AWT jedoch nur all diejenigen GUI-Komponenten, die in jedem von einer JVM unterstützten

Betriebssystem vorhanden sind. Das sind beispielsweise die Bedienelemente und zugehörigen Klassen `Button`, `Label`, `List` und `Textfield` aus dem Package `java.awt`. Komplexere Bedienelemente wie Tabellen- und Baumdarstellungen fehlen jedoch im AWT.

- **Swing** – Die Swing genannte GUI-Bibliothek stellt eine Erweiterung des AWTs dar und bietet GUI-Komponenten, die nahezu alle vollständig in Java geschrieben sind – nur einige Fensterkomponenten nutzen vom Betriebssystem bereitgestellte Funktionalität. Somit erzielt man eine (möglichst große) Unabhängigkeit von den nativen GUI-Komponenten: Die Swing-Bedienelemente werden durch die JVM gezeichnet, wodurch auch solche GUI-Komponenten bereitgestellt werden können, für die es kein Pendant aufseiten des Betriebssystems gibt. Viele Swing-Bedienelemente nutzen die ***Model-View-Controller-Architektur*** (kurz MVC). Bei MVC wird ein Bedienelement in die Bausteine Datenmodell, Darstellung und Kontrolleinheit untergliedert.
- **Java 2D** – Mitunter ist es zur Gestaltung einer ansprechenden grafischen Benutzeroberfläche erforderlich, die Bedienelemente optisch interessanter zu gestalten, z. B. indem man zusätzlich einige Zeichenoperationen nutzt. Mit der Java-2D-Bibliothek kann man vielfältige, komplexere Zeichenoperationen ausführen: Man kann verschiedene Liniendicken, Füllmuster, Farbverläufe, Rotationen usw. nutzen.

Bedienelemente

In GUIs stellen verschiedene Bedienelemente wie Buttons, Texteingabefelder usw. die Basisbausteine dar. Die Bedienelemente werden durch Klassen modelliert, die im AWT von der Basisklasse `java.awt.Component` abgeleitet sind. In Swing bildet die Klasse `javax.swing.JComponent` die Basis. Abbildung 9-1 zeigt einen Ausschnitt der Vererbungshierarchie gebräuchlicher Bedienelemente.

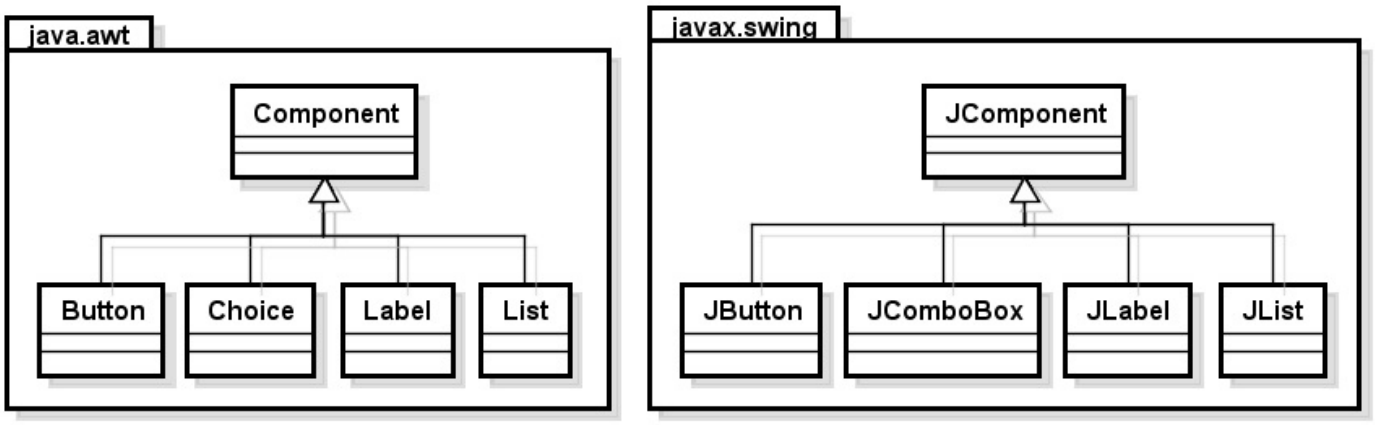


Abbildung 9-1 *Vererbungshierarchie der Bedienelemente in AWT und Swing*

Es fällt auf, dass die Bedienelemente von Swing (bis auf wenige Ausnahmen) analog zu denen des AWTs benannt sind (mit Präfix ’`J`’ im Namen). ***Obwohl es naheliegend scheint, sind die Swing-Bedienelemente*** **nicht** ***von den korrespondierenden AWT-***

Pendants abgeleitet. Wie sich AWT- und Swing-Bedienelemente dennoch in eine gemeinsame Vererbungshierarchie eingliedern, erläutere ich nachfolgend.

Container

Bedienelemente werden in Dialogen oder Fenstern angeordnet, die im AWT durch die Klassen `Dialog` und `Frame` aus dem Package `java.awt` modelliert sind. Swing definiert dafür im Package `javax.swing` die Klassen `JDialog` und `JFrame`. Erwähnenswert ist, dass die Container im AWT einer streng hierarchischen Anordnung folgen. ***Die Verbindung zu den entsprechenden Containerklassen in Swing wird durch Vererbung gelöst***: Die beiden Swing-Containerklassen `JDialog` und `JFrame` erben direkt von der jeweiligen korrespondierenden AWT-Containerklasse `Dialog` bzw. `Frame`. Das hat verschiedene Auswirkungen:

1. Die Swing-Containerklassen bilden untereinander keine Vererbungshierarchie.
2. Neben den AWT-Containerklassen erben auch die Swing-Containerklassen von der Basisklasse `Container` und der übergeordneten Basisklasse `Component`. Beide Basisklassen stammen aus dem AWT.
3. Somit ist weder `JComponent` die Basisklasse für die Swing-Containerklassen noch existiert eine Entsprechung der AWT-Containerbasisklasse `Container`, die etwa `JContainer` heißen müsste.

Um das Ganze zu verdeutlichen, zeigt Abbildung 9-2 die Vererbungshierarchie der Containerklassen in AWT und Swing. Zur leichteren Unterscheidbarkeit sind dort alle Swing-Klassen grau eingefärbt.

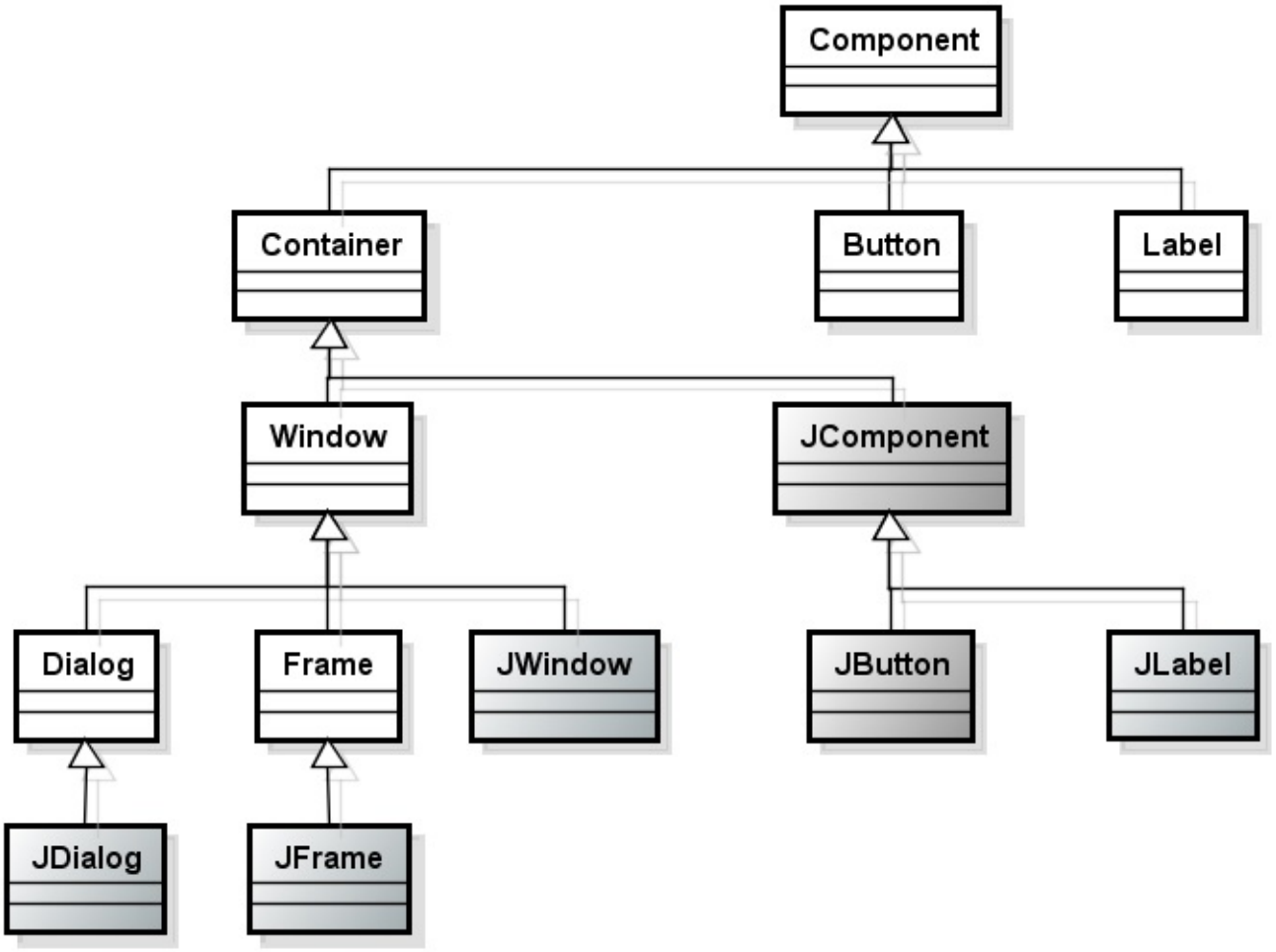


Abbildung 9-2 *Vererbungshierarchie der Containerklassen in AWT und Swing*

Beim Betrachten der Abbildung wird deutlich, dass neben den Swing-Containerklassen `JDialog` und `JFrame` auch *jedes* Swing-Bedienelement einen Container im Sinne von AWT darstellt, da die Basisklasse `JComponent` der Swing-Bedienelemente die AWT-Containerklasse `java.awt.Container` erweitert. Demnach könnte man also ein `javax.swing.JLabel` als einen Container ansehen und dort beispielsweise zwei `javax.swing.JButtons` platzieren:

```
// Achtung: Unsinnigerweise JLabel als Container für Bedienelemente
erzeugen
final JLabel label = new JLabel("Label:");
label.add(new JButton("First Button"));
label.add(new JButton("Second Button"));

// Liefert den Wert 2, die Buttons werden jedoch nicht dargestellt!
System.out.println(label.getComponentCount());
```

Der Aufruf dieser Funktionalität macht keinen Sinn und ist der Vererbungshierarchie geschuldet: Anhand dieses kurzen Sourcecode-Ausschnitts wird klar, dass einige Klassen laut API zwar gewisse Funktionalität anbieten, die man jedoch eigentlich nicht nutzen sollte. Die eher ungewöhnliche Vererbungshierarchie zeigt zudem, dass Swing mit etwas Mühe in die Basisklassenhierarchie des AWTs eingepasst wurde. Andererseits werden auch einige nützliche Eigenschaften vererbt, etwa Zugriffe auf die Größe und Ausrichtung eine GUI-Komponente.

Layoutmanagement

Die Anordnung von Bedienelementen innerhalb einer Containerkomponente erfolgt durch sogenannte Layoutmanager. Deren verschiedene Ausprägungen positionieren Bedienelemente unterschiedlich, z. B. in einem Raster ausgerichtet. Der große Vorteil daran ist, dass das Ganze automatisch geschieht und dabei gewisse Anordnungsvorgaben beachtet werden, wie der zur Verfügung stehende Platz genutzt werden soll, etwa wenn es zu einer Größenänderung eines Fensters kommt. Details dazu lernen wir bald kennen.

Alternativ zu einer Positionierung mithilfe eines Layoutmanagers ist es zwar möglich, wenn auch in der Regel nicht empfehlenswert, die Bedienelemente durch absolute Koordinaten »per Hand« anzuordnen.

Ereignisbehandlung

Alle bisher genannten Bestandteile eines GUIs sorgen lediglich für die Optik der Benutzeroberfläche. Dafür, dass tatsächlich etwas passiert, wenn ein Benutzer beispielsweise auf einen Button drückt, ist die Ereignisbehandlung zuständig. Mithilfe eigener Realisierungen vordefinierter Interfaces und Klassen kann man auf Ereignisse gezielt reagieren, also etwa auf einen Mausklick.

9.1.1 Überblick: Bedienelemente und Container

Nach diesem kurzen Einstieg sollen nun die eben genannten vier Hauptbestandteile von GUIs, also Bedienelemente, Container, Layoutmanagement und Ereignisbehandlung, detaillierter behandelt werden. Beginnen wir in diesem Abschnitt mit einer kurzen Darstellung zu Bedienelementen und Containern, bevor wir in den folgenden Abschnitten das Layoutmanagement und die Ereignisbehandlung genauer kennenlernen werden.

Das zentrale Element in grafischen Benutzeroberflächen sind Fenster. Wir wissen bereits, dass in einem Fenster verschiedene Bedienelemente wie Labels, Buttons, Texteingabefelder usw. angeordnet werden können. Ein Fenster dient somit als Sammelplatz oder ***Container*** für Bedienelemente.

Betrachten wir das anhand folgenden Beispiels: Einem Fenster vom Typ `JFrame` werden zwei Bedienelemente hinzugefügt. Das sind ein Beschriftungstextfeld mit dem Typ `JLabel` und ein Bedienknopf vom Typ `JButton`:

```
public static void main(final String[] args)
{
    // JFrame als Container für Bedienelemente erzeugen
    final JFrame frame = new JFrame("First Swing");

    // JLabel bzw. JButton erzeugen und zum JFrame hinzufügen
    frame.add(new JLabel("Text-Label:"));                    // #1
```

```
        frame.add( new JButton( "Button – Press Me! " ));              // #2

        // Größe festlegen und sichtbar machen
        frame.setSize( 400, 200);
        frame.setVisible( true);
    }
```

Listing 9.1 *Ausführbar als* **'FIRSTSWINGEXAMPLE'**

Abhängig vom persönlichen Vorwissen wundern wir uns beim Ausführen des obigen Programms darüber, wo das `JLabel` geblieben ist (Abbildung 9-3). Vertauschen wir die Reihenfolge der Zeilen (`#1` und `#2`), verschwindet stattdessen der `JButton`.

Abbildung 9-3 *Ein erstes Fenster mit Swing*

Wir erkennen, dass in dem Fenster immer nur eines der beiden Bedienelemente sichtbar ist, je nachdem, in welcher Reihenfolge die beiden Bedienelement hinzugefügt werden. Die Erklärung dieses zunächst merkwürdigen Verhaltens führt uns im nächsten Abschnitt 9.1.2 zur Thematik der ***Layoutmanager***. Zuvor gilt es noch auf zwei Besonderheiten des Beispiels sowie auf einige nützliche Container hinzuweisen.

Wissenswertes zum Beispiel

Wenn wir die paar Zeilen des Beispiels betrachten, scheint alles relativ klar und natürlich zu sein, doch das Ganze »hat es in sich«. Zumindest auf zwei Dinge möchte ich eingehen, und zwar auf die folgenden beiden – vermeintlich harmlosen – Aufrufe:

1. `setSize()` – Der Aufruf von `setSize( int, int)` bestimmt die Größe des `JFrames`. Diese ist damit aber nicht fix, sondern kann nachträglich beliebig verändert werden. In diesem Beispiel passt sich die Größe des enthaltenen `JButtons` dynamisch an die des Fensters an. Dafür finden wir die Erklärung in Abschnitt 9.1.2 über Layoutmanager.
2. `setVisible()` – Wie erwartet führt der Aufruf von `setVisible( boolean)` dazu, dass der `JFrame` auf dem Bildschirm sichtbar wird – im Swing-Sprachjargon spricht man auch davon, dass das Fenster »realisiert« wird. Damit ist gemeint, dass hinter den Kulissen deutlich mehr passiert, als die reine Darstellung vermuten lässt: Es wird zudem die Verarbeitung von Ereignissen aktiviert, sodass etwa die Reaktion auf das Betätigen des Schließkreuzes oder eine Änderung der Fenstergröße erfolgen kann. Das alles besitzt Konsequenzen: Nach dieser

»Realisierung« dürfen Methodenaufrufe an Swing-Komponenten nur noch gemäß einem später in Abschnitt 9.2 vorgestellten Ablauf erfolgen, um Inkonsistenzen bzw. Probleme zu vermeiden.

Für uns ist hier und für die nachfolgenden Beispiele erstmal nur wichtig, dass diese beiden Aufrufe immer *nach* allen Aufrufen erfolgen, die die Benutzeroberfläche erstellen.

Nützliche Container im Überblick

In den nachfolgenden Beispielen werden wir einige Containerklassen nutzen, die es erlauben, ein GUI optisch zu strukturieren. Das sind im Einzelnen:

- `javax.swing.JPanel` – Zur Verschachtelung von Containern kann man die Klasse `javax.swing.JPanel` einsetzen, deren Instanzen als Sammelstelle für Bedienelemente dienen und bei der hierarchischen Komposition von Containerkomponenten helfen. Dadurch können komplexere Layouts realisiert werden, was gleich bei der Diskussion von Layoutmanagern deutlich wird.
- `javax.swing.JSplitPane` – Die `JSplitPane` ist wie das `JPanel` eine GUIKomponente, die als Container für weitere GUI-Komponenten dient und dazu in zwei größenveränderliche, horizontal bzw. vertikal ausgerichtete Bereiche mit Trenner aufgeteilt ist. Jedem Bereich kann eine GUI-Komponente zugeordnet werden. Deren Ausdehnung kann durch Verschieben des Trenners angepasst werden.
- `java.swing.JScrollPane` – Die `JScrollPane` stellt einen Container für ein anderes Bedienelement oder einen anderen Container dar, dessen Ausmaße möglicherweise recht groß werden können. Eine `JScrollPane` ist zur Darstellung umfangreicher Listen, Tabellen oder Baumstrukturen sehr hilfreich. Übersteigen die Ausmaße der aufgenommenen GUI-Komponente die durch die `JScrollPane` vorgegebenen Maße, so werden je nach Bedarf eine horizontale und eine vertikale Scrollbar hinzugefügt, was eine Navigation über die Gesamtfläche erlaubt – eine `JScrollPane` verwaltet also eine virtuelle Arbeitsfläche.

Ganz allgemein findet man bei der Gestaltung grafischer Benutzeroberflächen immer wiederkehrende Muster: In einem `JPanel` werden Containerkomponenten und Bedienelemente zu komplexeren Strukturen kombiniert. Abbildung 9-4 zeigt das. Dort sehen wir eine Baumdarstellung, die in eine `JScrollPane` eingefügt ist. Beides wird durch eine `JSplitPane` mit einer Tabelle kombiniert. Das Ganze wiederum steckt in einem `JPanel`, das oben eine Toolbar und unten eine Statuszeile ergänzt.

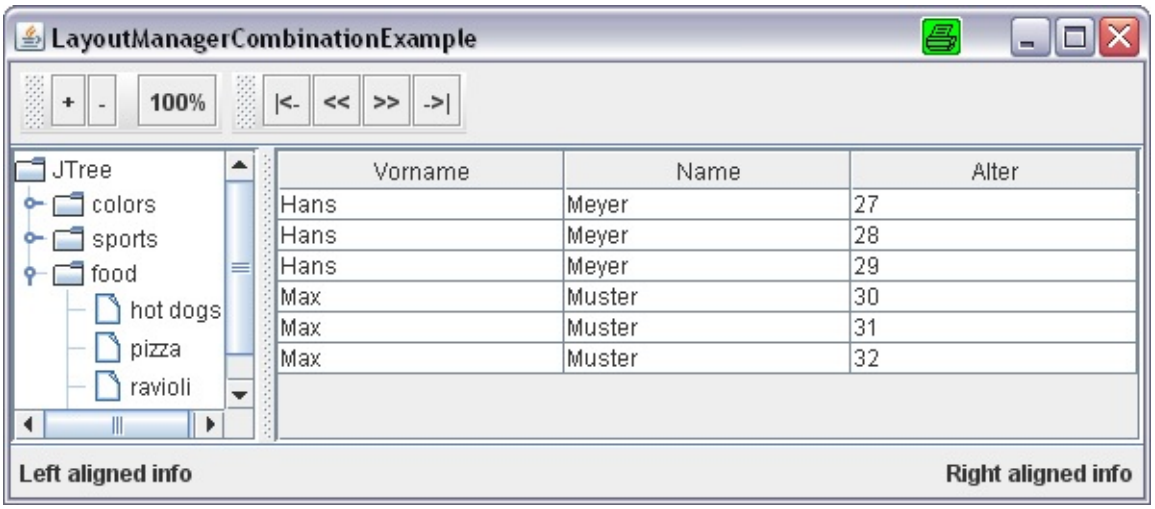


Abbildung 9-4 *Kombination von Containerkomponenten*

9.1.2 Einführung in das Layoutmanagement

Im initialen Beispiel haben wir nirgends den Einsatz eines Layoutmanagers direkt gesehen. Dennoch war ein Layoutmanager aktiv, und sein Einfluss wurde durch die (unerwartete) Anordnung der beiden Bedienelemente (`JLabel` und `JButton`) spürbar. Wie man Layoutmanager aktiv und sinnvoll einsetzen kann, erläutere ich nun.

Motivation für Layoutmanager

Bekanntlich bestimmen Layoutmanager die Anordnung der Bedienelemente innerhalb von Containern. Man mag sich fragen, wozu der Einsatz einer speziellen Komponente für die Positionierung nützlich sein soll, da es doch möglich ist, die Bedienelemente eines Dialogs oder eines Fensters bereits im Vorhinein so zu platzieren, dass sie optimal für einen Anwendungsfall angeordnet sind. Voraussetzung dafür ist aber, dass für jede GUI-Komponente eine absolute Position und Größe bekannt ist. Auch das klingt erstmal durchaus vernünftig. Warum existieren also Layoutmanager? Versuchen wir darauf eine Antwort zu finden, indem wir mögliche Schwachpunkte einer pixelweisen oder zumindest statischen Positionierung analysieren.

Mögliche Probleme einer statischen Positionierung

Ist ein Layout statisch definiert, so treten keine Probleme auf, solange die gesamten Rahmenbedingungen und die Anordnungen der Bedienelemente unverändert bleiben. Aber selbst dann ergeben sich folgende Schwierigkeiten:

1. Wenn das Layout angepasst werden soll, beispielsweise um neue Auswahlmöglichkeiten in Form weiterer GUI-Komponenten präsentieren zu können, entsteht unweigerlich einiges an Neupositionierungsaufwand.

2. Wird ein Fenster oder Dialog vergrößert, so ist die Erwartungshaltung vielfach die, dass sich einige GUI-Komponenten (z. B. Textfelder oder Auswahllisten) in der Größe anpassen und der neu zur Verfügung stehende Platz sinnvoll ausgefüllt wird. Dieses Verhalten selbst auszuprogrammieren ist aufwendig.

Höchstwahrscheinlich wird man sich daher irgendwann eine (Utility-)Klasse schreiben, die eine ähnliche Funktionalität wie ein Layoutmanager bereitstellt.

Rahmenbedingungen

Wenn sich dann noch die Rahmenbedingungen ändern, entstehen weitere Probleme: Es könnte etwa sein, dass ein Benutzer die Standardschriftgröße ändert. Danach sind Größenangaben in Pixeln hinfällig. Texte sind entweder falsch ausgerichtet, ragen in andere GUI-Komponenten hinein oder werden abgeschnitten. Wünschenswert ist es daher, dass sich das Layout der GUI-Komponenten automatisch und dynamisch an sich verändernde Gegebenheiten anpasst. Genau das ist das Einsatzgebiet der Layoutmanager.

Zur Positionierung und Größenberechnung müssen die GUI-Komponenten einige Informationen, z. B. deren minimale, bevorzugte und maximale Ausdehnung, dem Layoutmanager mitteilen. Diese Angaben dienen dann als Richtwerte, um die tatsächlich zu verwendenden Größen und Ausrichtungen der GUI-Komponenten zu berechnen.

Überblick über einige vordefinierte Layoutmanager des JDKs

Nach der Motivation zum Einsatz von Layoutmanagern stelle ich Ihnen nun einige im JDK vordefinierte Layoutmanager vor, die jeweils charakteristische Anordnungen von GUI-Komponenten innerhalb eines Containers vornehmen. Im JDK stehen unter anderem folgende Layoutmanager zur Auswahl, die Sie mithilfe des Programms LayoutManagerExample ausprobieren können:

1. `java.awt.BorderLayout` – Dieser Layoutmanager teilt die Fläche des Containers in fünf Bereiche auf, die jeweils entweder von genau einer GUI-Komponente belegt werden oder aber auch ungenutzt bleiben können. Als Positionierungsinformationen für die GUI-Komponenten sind im `BorderLayout` die Konstanten `NORTH`, `SOUTH`, `WEST`, `EAST` und `CENTER` definiert, um die gewünschte Anordnung festzulegen. Die Besonderheit dieses Layoutmanagers besteht darin, dass der zentrale Bereich lediglich denjenigen Platz einnimmt, den die anderen Teile initial nicht für sich benötigen: Die GUI-Komponenten im oberen und unteren Bereich erhalten ihre bevorzugte Höhe und werden in ihrer Breite auf die Containerbreite angepasst. Für die linke und rechte GUI-Komponente gilt das Gesagte andersherum: Sie erhalten ihre bevorzugte Breite und werden in ihrer Höhe angepasst. Insbesondere heißt dies aber auch, dass bei zunehmender Größe des Containers der zentrale Bereich immer mehr Raum erhält. ***Mithilfe dieses Layoutmanagers lassen sich sehr gut größensensitive***

Dialoge gestalten. Abbildung 9-5 zeigt ein Beispiel, an dem man die Ähnlichkeit zum Layout typischer Webseiten erkennt, die oben eine Menüleiste, links eine Navigation und den Inhalt in der Mitte präsentieren.

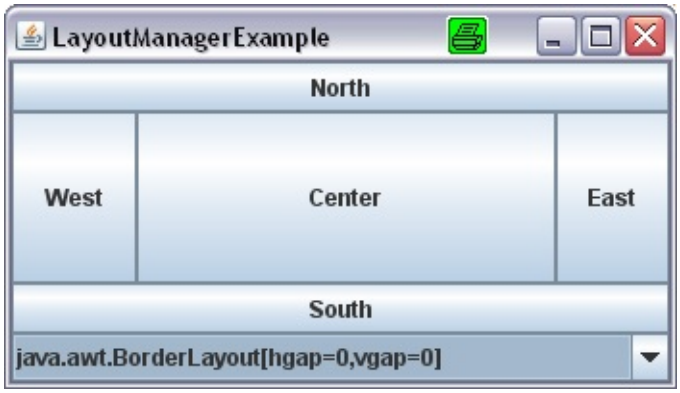


Abbildung 9-5 *Beispiel* `BorderLayout`

2. `java.awt.FlowLayout` – Das `FlowLayout` ordnet GUI-Komponenten horizontal in einer Reihe an und verwendet dazu die bevorzugten Größen der GUI-Komponenten. Passen auf diese Weise nicht alle anzuordnenden GUIKomponenten in den Container, so werden diese umbrochen – ähnlich wie Worte eines Absatzes in einer Textverarbeitung (vgl. Abbildung 9-6). Die Ausrichtung der GUI-Komponenten kann über die im `FlowLayout` definierten Konstanten `LEFT`, `CENTER`, `RIGHT` spezifiziert werden, wobei standardmäßig eine zentrierte Anordnung erfolgt. ***Das `FlowLayout` eignet sich daher besonders zur Gestaltung von Toolbars oder zur Ausrichtung von Bestätigungsbuttons in Dialogen.***

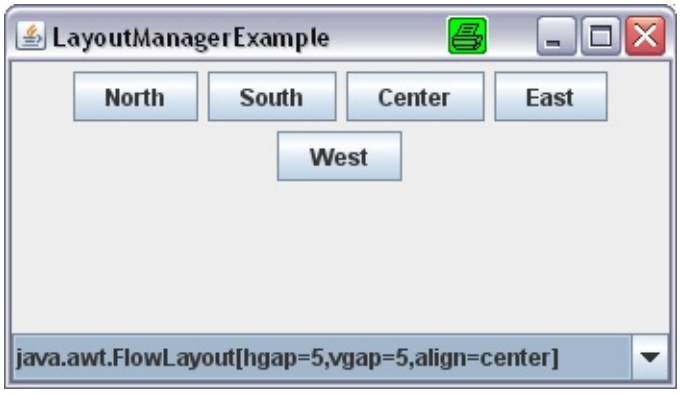


Abbildung 9-6 *Beispiel* `FlowLayout`

3. `java.awt.CardLayout` – Die Anordnung der GUI-Komponenten ist bei diesem Layoutmanager einem Stapel Karten nachempfunden. Es ist immer nur die obere GUI-Komponente sichtbar (vgl. Abbildung 9-7), deren Größe an die Abmessungen des Containers angepasst wird. Mithilfe der Methoden `first()`, `last()`, `prev()` und `next()` kann man zwischen den GUI-Komponenten wechseln. Es ist aber auch eine direkte Darstellung einer gewünschten GUI-Komponente über die Methode `show()` möglich. ***Aufgrund der Eigenschaften des `CardLayouts` lassen sich damit insbesondere sogenannte Wizards[1] recht einfach realisieren.***

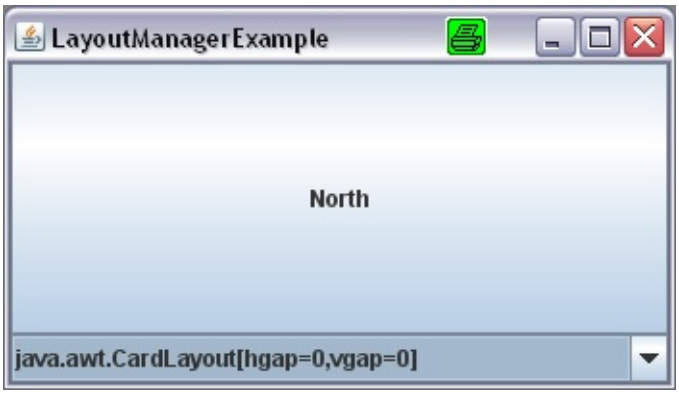


Abbildung 9-7 *Beispiel* `CardLayout`

4. `java.awt.GridLayout` – Seinem Namen entsprechend, ordnet das `GridLayout` die GUI-Komponenten in Form eines Gitters an. Dieses besteht aus gleich großen Gitterzellen, die von genau einer GUI-Komponente besetzt werden oder frei bleiben. Die Größen der GUI-Komponenten werden so angepasst, dass sie jeweils eine Gitterzelle ausfüllen. Abbildung 9-8 zeigt das.

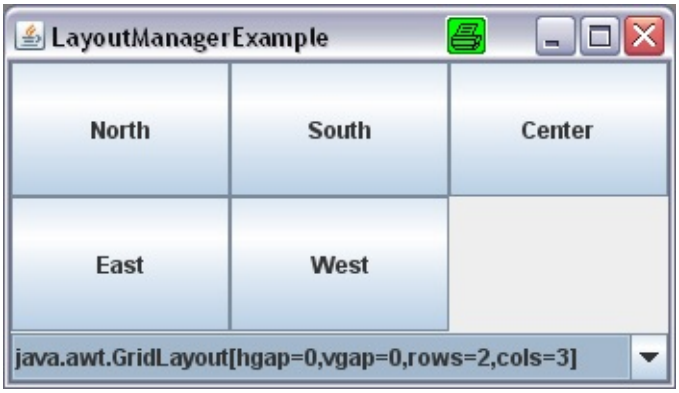


Abbildung 9-8 *Beispiel* `GridLayout`

Alternative: Die Klasse `GridBagLayout`

Haben Sie weiter gehende Anforderungen an das Layout, so können Sie das viel Flexibilität bietende, aber recht kompliziert zu handhabende `java.awt.GridBagLayout` nutzen. Dessen Arbeitsweise ähnelt der des `GridLayouts`, jedoch kann eine GUI-Komponente durchaus auch mehrere Zellen (sowohl Spalten als auch Zeilen) im Gitter einnehmen. Insbesondere können unterschiedlich breite und hohe Zellen realisiert werden, in denen die Bedienelemente standardmäßig ihre gewünschte Größe beibehalten. Bei Bedarf ist es zudem möglich, die Ausrichtung einer GUI-Komponente innerhalb des Gitters sowie die Anpassung ihrer Größe an das Gitter zu spezifizieren. Diese und weitere Einstellungen lassen sich über die Klasse `java.awt.GridBagConstraints` festlegen.

Layoutmanager und Container

Einer Containerkomponente kann man im Konstruktor, aber auch später über einen Aufruf von `setLayout(LayoutManager)`, einen gewünschten Layoutmanager

explizit zuordnen. Ansonsten besitzen die Container des JDKs (`JDialog`, `JFrame` usw.) einen vordefinierten Layoutmanager:

- Für die Klassen `JDialog`, `JFrame` und `JWindow` ist das ein Layoutmanager vom Typ `BorderLayout`.
- Für die Klasse `JPanel` wird standardmäßig das `FlowLayout` verwendet.

Tipp: GUI-Entwurf mit Skizzen

Beim Entwurf eines GUIs ist es ratsam, zunächst eine Skizze auf einem Blatt Papier oder mithilfe von Mockup-Tools zu erstellen – dazu empfehle ich das Tool Balsamiq Mockups, das unter `http://www.balsamiq.com/` als Demoversion verfügbar ist.

Layoutmanager im Einsatz

Nachdem wir nun einen Überblick über einige Layoutmanager des JDKs gewonnen haben, kommen wir zu unserem ersten Beispiel zurück. Dort wurde entweder nur das `JLabel` oder nur der `JButton` im Fenster dargestellt. Das war so nicht gewünscht, daher soll dieser Fehler nun behoben werden.

Aus dem vorherigen Abschnitt wissen wir, dass für ein `JFrame` standardmäßig das `BorderLayout` zum Einsatz kommt, wenn wir nichts anderes spezifizieren. Zur Korrektur des Platzierungsverhaltens kann man also entweder explizit einen anderen Layoutmanager wählen oder aber den Einsatz des `BorderLayouts` derart korrigieren, dass beide Bedienelemente dargestellt werden. Durch Nachschlagen in der Dokumentation der Klasse `Container` finden wir eine überladene Variante der `add()`-Methode, die es uns erlaubt, eine Positionierungsinformation anzugeben:

```
void add(Component comp, Object constraints)
```

Auf die Möglichkeit zur Angabe von Constraints als Platzierungsinformation hatte ich bereits bei der Beschreibung des `BorderLayouts` hingewiesen: Die Werte der Constraints bestimmen die Position der jeweiligen GUI-Komponente im Container. Für das `BorderLayout` sind die Constraints zwar textuell, jedoch nicht beliebig wählbar, sondern durch spezielle Konstanten (`BorderLayout.NORTH` usw.) fest vorgegeben. Fehlt die Angabe der Constraints, so wird standardmäßig `CENTER` genutzt. Bei mehrfachem Belegen eines Bereichs wird nur die zuletzt hinzugefügte GUI-Komponente verwendet und dargestellt.

Zur Korrektur der Platzierung der beiden Bedienelemente nutzen wir für das `JLabel` die Konstante `WEST` und für den `JButton` den Wert `CENTER` aus der Klasse `BorderLayout`:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("First Swing Improved Layout");

    // JLabel bzw. JButton mit Positionierungsangaben zum JFrame hinzufügen
```

```
        frame.add( new JLabel( "Text-Label: "), BorderLayout.WEST);
        frame.add( new JButton( "Button – Press Me! "), BorderLayout.CENTER);

        frame.setSize( 400, 200);
        frame.setVisible( true);
    }
```

Listing 9.2 *Ausführbar als* **'FIRSTSWINGEXAMPLEIMPROVED'**

Als Folge der Modifikation sind nun – wie gewünscht – beide Bedienelemente sichtbar, wenn Sie das Programm FIRSTSWINGEXAMPLEIMPROVED starten (vgl. Abbildung 9-9). Die Größe des `JButtons` wurde vom Layoutmanager automatisch so angepasst, dass das `JLabel` vollständig (in seiner bevorzugten Breite) im linken Teil des Fensters direkt neben dem `JButton` dargestellt wird. Allerdings ist eine Ausdehnung der Bedienelemente in der Vertikalen auf die Fensterhöhe normalerweise nicht gewollt. Nun werden wir verschiedene Layoutmanager kombinieren, um Abhilfe zu schaffen.

Abbildung 9-9 *Verbesserung des initialen Beispiels*

9.1.3 Komplexere Layouts (Kombination von Layoutmanagern)

Das im vorherigen Beispiel gezeigte Layout ist doch recht rudimentär. Um ansprechendere Layouts zu erzielen, kann man (verschiedene) Layoutmanager miteinander kombinieren. Dazu werden mehrere Containerkomponenten ineinander verschachtelt und dann wird jeder einzelnen der gewünschte, passende Layoutmanager zugeordnet. Zur Verschachtelung von Containerkomponenten nutzen wir `JPanel`s, die eine hierarchische Komposition ermöglichen.

Typisches Layout

An einem kleinen, aber durchaus praxisnahen Beispiel wollen wir ein etwas komplexeres Layout nachvollziehen: Es soll folgende Aufteilung des Fensters erfolgen: Oben wird eine Toolbar und unten eine Statuszeile angezeigt. Die besonders wichtigen Bedienelemente, etwa eine Tabelle, sind größensensitiv und mittig angeordnet.

Mit dem `BorderLayout` lässt sich diese Anordnung gut umsetzen. Das wird in Abbildung 9-10 verdeutlicht, die ich bereits zur Vorstellung einiger Containerkomponenten genutzt habe. Hier dient sie zur Visualisierung für die nachfolgenden Beschreibungen der benötigten Verschachtelungen von Layoutmanagern und Containern.

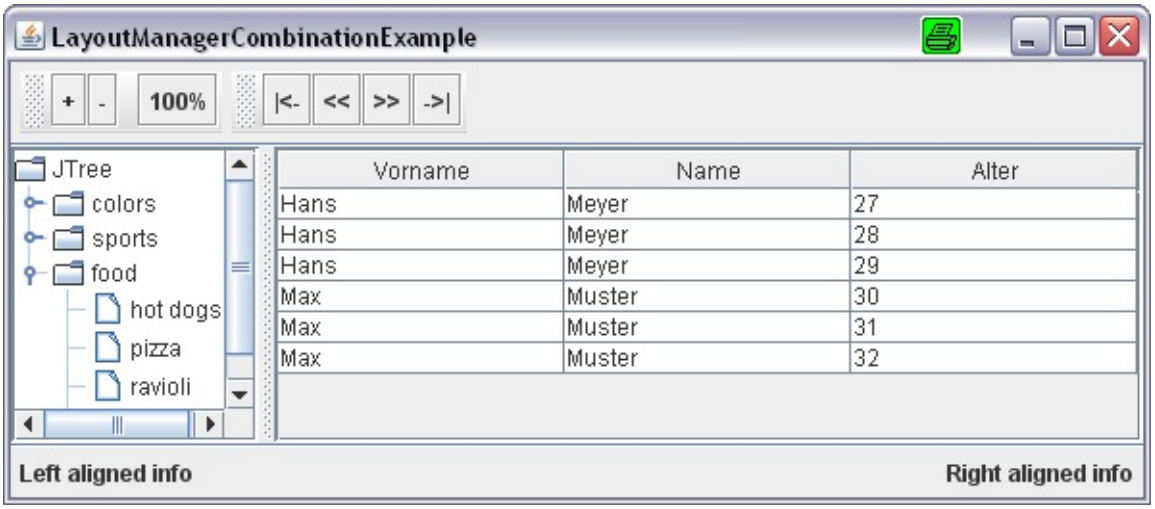


Abbildung 9-10 *Kombination von Containerkomponenten*

Links neben einer Tabelle ist eine Baumdarstellung gezeigt, wie dies in der Praxis zur Navigation in den Daten häufig gewünscht ist. Diese Baumdarstellung kann man entweder im `WEST`-Bereich anordnen oder zusätzlich im `CENTER`-Bereich. Letzteres ist hier der Fall. Da man bekanntlich aber nicht mehrere GUI-Komponenten in einem Bereich des `BorderLayouts` nutzen kann, muss hier eine `javax.swing.JSplitPane` zum Einsatz kommen, die die Teilung des `CENTER`-Bereichs erlaubt.

In diesem Beispiel setzen wir noch eine weitere nützliche Containerkomponente ein: die `java.swing.JScrollPane`, die bekanntermaßen eine Navigation über die Gesamtfläche der integrierten GUI-Komponente ermöglicht. Hier erlaubt uns das die Navigation in der Baumdarstellung. Aber auch die Tabelle wurde in eine `JScrollPane` eingebettet, ohne dass dies sichtbar ist, da derzeit weder Scrollbars benötigt noch dargestellt werden. Warum erfolgt dann die Integration in die `JScrollPane`? Das hat zwei Gründe. Zum einen werden nur so die Spaltenüberschriften für Tabellen angezeigt (vgl. Abschnitt 9.4.3). Zum anderen geschieht dies vorausschauend, weil die Menge an Daten für die Tabelle in der Praxis häufig umfangreicher wird, wodurch dann Scrollbars zur Navigation benötigt werden. Außerdem ist das für die Fälle hilfreich, in denen der Benutzer das Fenster verkleinert.

Hierarchische Struktur

Die hierarchische Struktur von Containern und Layouts wird deutlich, wenn man das durch diese Verschachtelungen erstellte GUI als Baum darstellt, wie in Abbildung 9-11. Dort entsprechen die Begriffe `CENTER`, `EAST`, `NORTH`, `SOUTH` und `WEST` den korrespondierenden Ausrichtungen innerhalb eines `BorderLayouts`. Mit `left` und

right sind die zwei Bereiche einer JSplitPane referenziert. Für das Toolbar-Panel forcieren wir eine linksbündige Ausrichtung durch die Angabe von FlowLayout.LEFT. Ansonsten nutzen wir die jeweiligen Standards der Container, die hier aus Gründen der Übersichtlichkeit nicht gesondert aufgeführt sind.

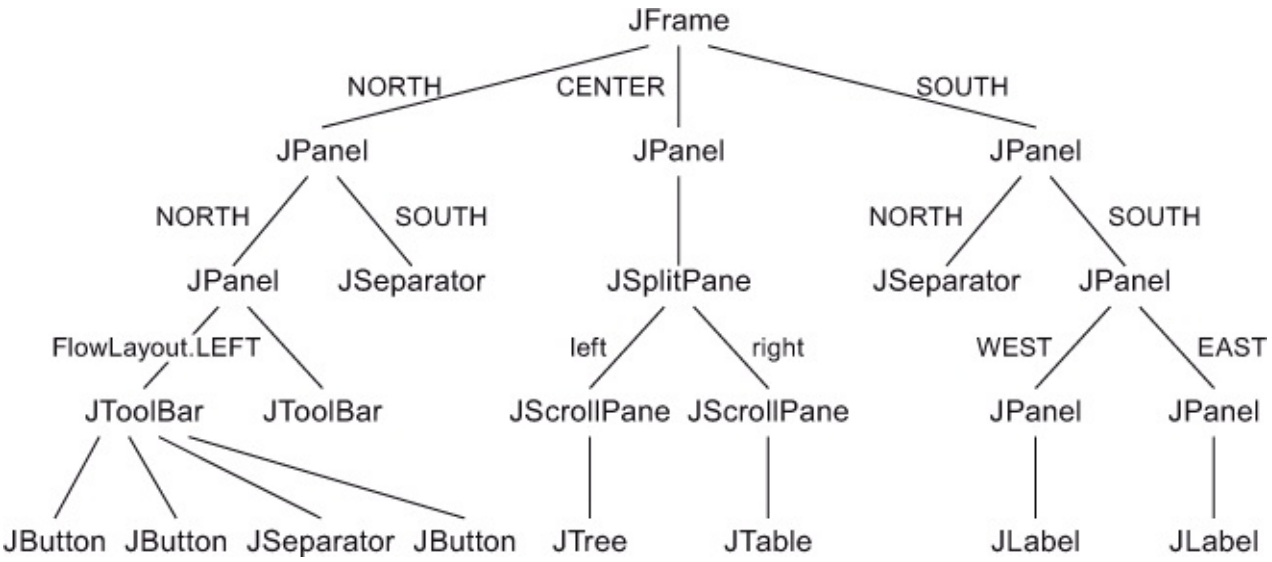


Abbildung 9-11 *Hierarchische Struktur der Container des Beispiels*

Implementierung

Nach diesem gedanklichen Ausflug zur Kombination von Layoutmanagern werden wir nun das in Abbildung 9-10 gezeigte Layout realisieren. Dazu wird eine Untergliederung, im Speziellen in die drei JPanels des JFrames, benötigt. Das erreichen wir mit drei Erzeugungsmethoden, die wiederum den gesamten darunter befindlichen Baum an Bedienelementen und GUI-Komponenten aufbauen.

In diesem Beispiel wird ein JFrame mit dem BorderLayout als Standard konstruiert, das sich ideal für die gewünschte Dreiteilung des Applikationsfensters eignet:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("LayoutManagerCombinationExample");

    // Das Hauptfenster wird in drei Bereiche unterteilt
    frame.add(createToolBarPanel(), BorderLayout.NORTH);
    frame.add(createCenterPanel(), BorderLayout.CENTER);
    frame.add(createStatusBarPanel(), BorderLayout.SOUTH);

    frame.setSize(600, 300);
    frame.setVisible(true);
}
```

Listing 9.3 *Ausführbar als* **'LAYOUTMANAGERCOMBINATIONEXAMPLE'**

Die drei Bereiche erzeugen wir mit den bereits erwähnten drei spezifischen Erzeugungsmethoden (vgl. Abschnitt 18.1.1 zum Entwurfsmuster ERZEUGUNGSMETHODE). Die Realisierung der Methode createToolBarPanel() geschieht folgendermaßen:

```
private static JPanel createToolBarPanel()
{
    // Zwei Toolbars erzeugen
    final JToolBar zoomToolBar = new JToolBar();
    zoomToolBar.add(new JButton("+"));
    zoomToolBar.add(new JButton("-"));
    zoomToolBar.addSeparator();
    zoomToolBar.add(new JButton("100%"));

    final JToolBar skipToolBar = new JToolBar();
    skipToolBar.add(new JButton("|<-"));
    skipToolBar.add(new JButton("≪"));
    skipToolBar.add(new JButton("≫"));
    skipToolBar.add(new JButton("->|"));

    // Ausrichtung LEFT ist wichtig, da die Toolbars sonst mittig sind.
    final JPanel toolbarPanel = new JPanel(new FlowLayout(FlowLayout.LEFT));
    toolbarPanel.add(zoomToolBar);
    toolbarPanel.add(skipToolBar);

    // Gemeinsames JPanel für Toolbar-Panel und JSeparator
    final JPanel compoundPanel = new JPanel(new BorderLayout());
    compoundPanel.add(toolbarPanel, BorderLayout.NORTH);
    compoundPanel.add(new JSeparator(), BorderLayout.SOUTH);

    return compoundPanel;
}
```

Für die einzelnen Knöpfe in einer Toolbar scheint der Einsatz eines `FlowLayouts` angebracht. Tatsächlich ist dies aber nicht erforderlich, da die hier eingesetzte GUIKomponente `javax.swing.JToolBar` praktischerweise die Details (z. B. die Anordnung der Buttons) mithilfe eines eigenständigen privaten Layoutmanagers für uns erledigt. Erst wenn, wie hier, mehrere Toolbars hintereinander angezeigt werden sollen, ist es sinnvoll, die `JToolBars` durch ein `FlowLayout` in einem `JPanel` anzuordnen.

Zwischen Toolbars und dem zentralen Hauptbereich soll eine Trennlinie gezeichnet werden. Daher nutzen wir ein weiteres `JPanel` mit einem `BorderLayout`. Dort platzieren wir das Toolbar-Panel oben (`BorderLayout.NORTH`) und die durch die Klasse `javax.swing.JSeparator` erstellte Trennlinie unten (`BorderLayout.SOUTH`).

Die Bedienelemente des zentralen Bereichs werden in einer `JSplitPane` angeordnet. Das geschieht mit der Erzeugungsmethode `createCenterPanel()` wie folgt:

```
private static JComponent createCenterPanel()
{
    // Tabellenspalten und -daten
    final String[] headers = {"Vorname", "Name", "Alter"};
    final String[][] values = { { "Hans", "Meyer", "27" },
                                { "Hans", "Meyer", "28" },
                                { "Hans", "Meyer", "29" },
                                { "Max", "Muster", "30" },
                                { "Max", "Muster", "31" },
                                { "Max", "Muster", "32"} };
```

```
    final JScrollPane scrollPane = new JScrollPane( new JTable( values,
headers));

    final JSplitPane splitPane = new JSplitPane();
    splitPane. setLeftComponent( new JScrollPane( new JTree()));
    splitPane. setRightComponent( scrollPane);

    return splitPane;
}
```

Die verbleibende Erzeugungsmethode `createStatusbarPanel()` arbeitet nach dem gleichen Prinzip und wird daher hier nicht gezeigt.

9.1.4 Grundlagen zur Ereignisbehandlung

Während wir uns bisher auf die Anordnung der Bedienelemente konzentriert haben, wollen wir uns nun dem dynamischen Verhalten in GUIs widmen. Wenn man in unseren bisherigen Beispielen einen Button anklickt, dann passiert wenig: Der Button wird im gedrückten Zustand durch Swing neu gezeichnet, ansonsten wird jedoch keine Aktion ausgeführt. Wie man Bedienelemente mit Aktionen verknüpft, erklärt dieser Abschnitt.

Wichtige Interfaces und Klassen bei der Ereignisbehandlung

Bei der Bedienung eines GUIs führt der Benutzer zu beliebigen Zeitpunkten Mausklicks, Mausbewegungen, Tastatureingaben usw. durch. Diese Aktionen werden als ***Ereignis*** (engl. ***Event***) modelliert. Events sind in AWT und Swing Objekte vom Basistyp `java. util. EventObject`. Für verschiedene Arten von Events sind entsprechende Subtypen definiert, etwa ein `java. awt. event. MouseEvent` für Mauseingaben. Bei einem Klick auf einen `JButton` wird ein `java. awt. event. ActionEvent` ausgelöst.

Ein solches Event besitzt immer eine Quelle (***Event Source***) und wird an beliebig viele (oder auch keinen) Interessenten (sogenannte ***Event Listener***) mit dem Basistyp `java. util. EventListener` weitergeleitet. Diverse Spezialisierungen davon helfen bei der Verarbeitung von Ereignissen – ein `java. awt. event. MouseListener` dient etwa zur Behandlung von Mauseingaben. Die Art der Reaktion auf ein Ereignis wird durch die Implementierung des jeweiligen `EventListener`-Interface bestimmt. Dort sind jeweils spezifische Callback-Methoden definiert, etwa `mouseMoved( MouseEvent)`. Durch die Implementierung der Methoden kann man vorgeben, welche Aktionen als Reaktion auf ein Event auszuführen sind.

Um bei einer Benutzeraktion aber tatsächlich benachrichtigt zu werden, müssen sich Event Listener zuvor bei einer Ereignisquelle registrieren. Weil die Ereignisse an die registrierten Interessenten weitergeleitet (delegiert) werden, spricht man für diese Art der Ereignisbehandlung auch vom ***Delegation (Based) Event Model***.

Einen ersten Überblick gibt Abbildung 9-12, in der eine vereinfachte Vererbungshierarchie von `EventObject` und `EventListener` gezeigt ist.

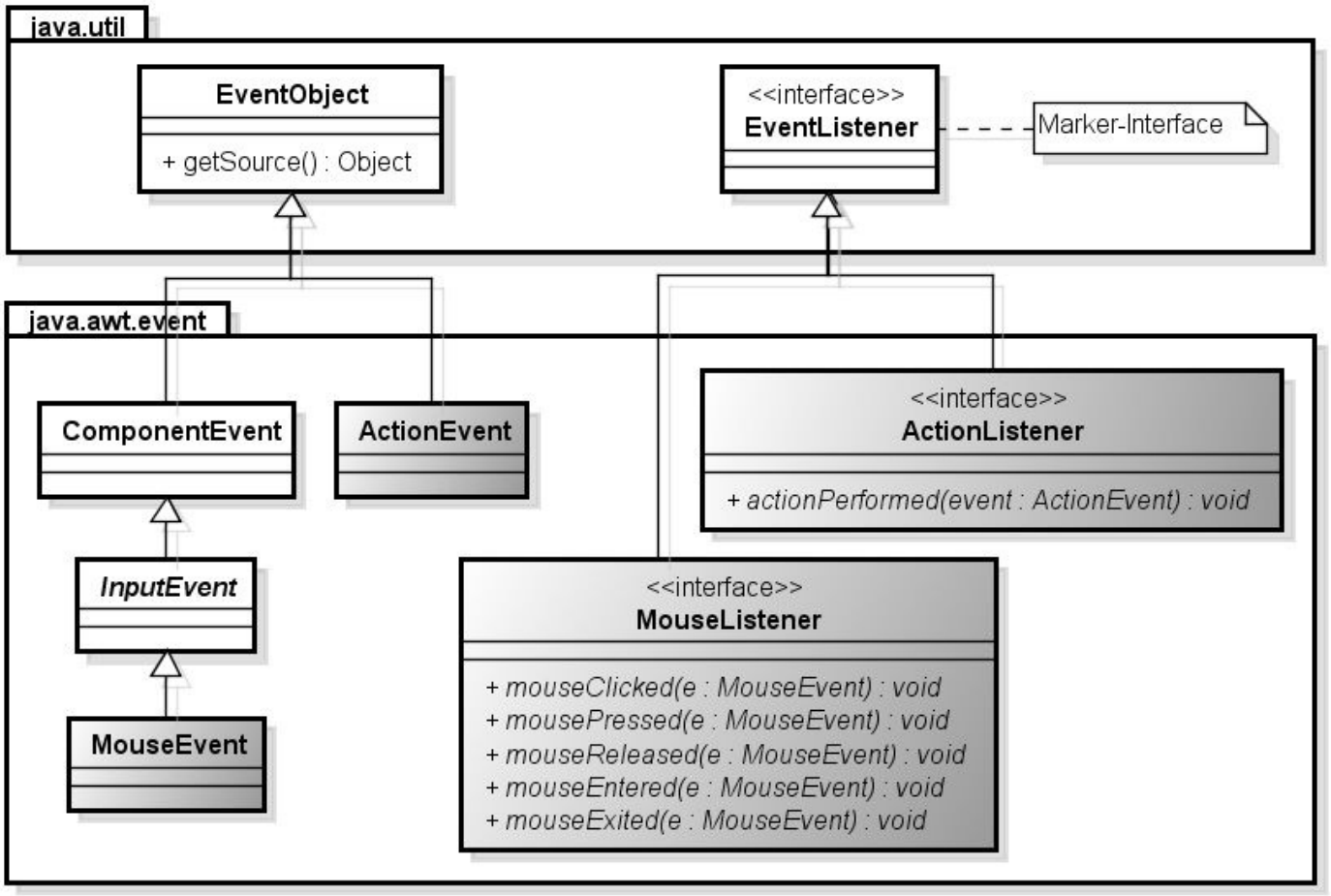


Abbildung 9-12 *Vererbungshierarchie von* `ActionEvent` *und* `ActionListener`

Verarbeitung von `Events`

Schauen wir uns am Beispiel eines `JButton` exemplarisch die Verarbeitung von Events an. Ein Event bleibt so lange unbehandelt, wie kein Interessent vom Typ `java.awt.event.ActionListener` angemeldet wurde. Ist hingegen ein solcher Listener registriert, so wird die im Interface `ActionListener` definierte CallbackMethode `actionPerformed(ActionEvent)` aufgerufen, die in eigenen Listener-Implementierungen mit Leben gefüllt werden muss. Abbildung 9-13 stellt die Zusammenhänge als Klassendiagramm dar. Dort sieht man auch, dass die Basisklasse `AbstractButton` die angemeldeten Listener verwaltet und Zugriff darauf bietet.

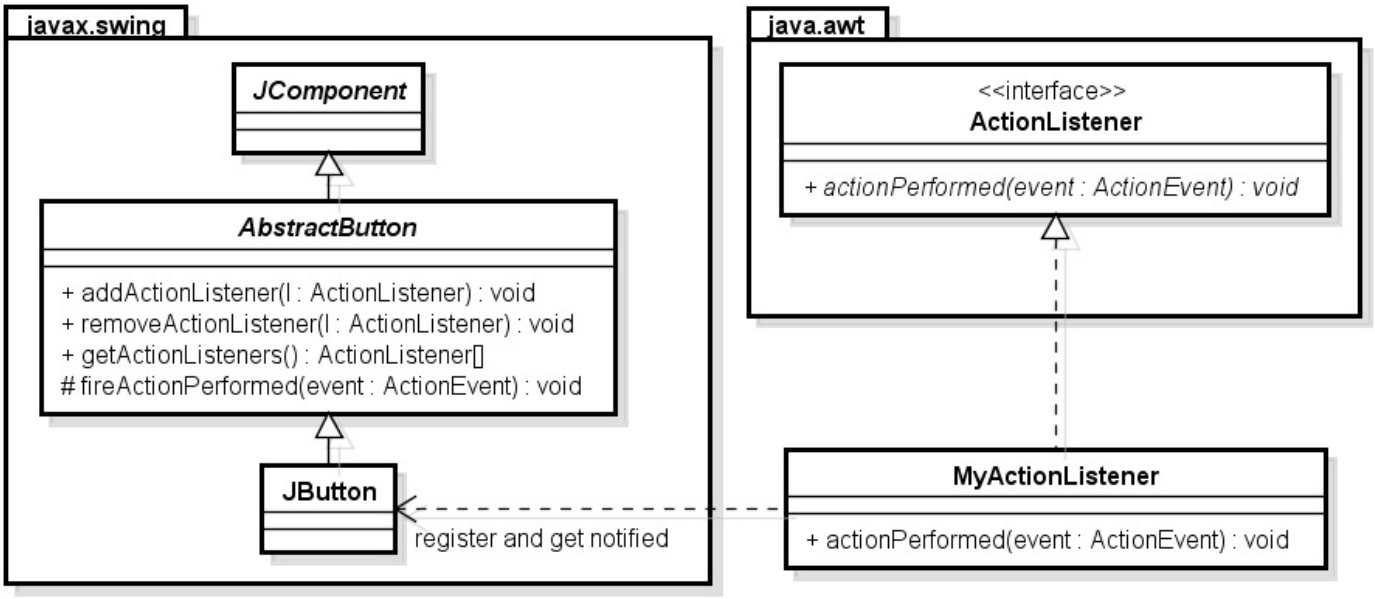


Abbildung 9-13 *Klassendiagramm eines eigenen* `ActionListener`*s*

Mit diesem Wissen wollen wir nun unser Beispiel vom Anfang leicht modifizieren und mit einer Ereignisverarbeitung versehen. Bei jedem Klick auf den `JButton` soll ein Ausrufezeichen an dessen Text angefügt werden. Dazu registrieren wir für den `JButton` eine Instanz einer anonymen Klasse vom Typ `ActionListener` und implementieren dort die Methode `actionPerformed( ActionEvent)` wie folgt:

```
public static void main( final String[] args)
{
    final JFrame frame = new JFrame( "ActionEventExample");
    final JButton button = new JButton( "Button – Press Me! ");
    frame.add( button);

    // Event Listener registrieren
    button.addActionListener( new ActionListener()
    {
        @Override
        public void actionPerformed( final ActionEvent e)
        {
            button.setText( button.getText() +"! ");
        }
    });

    frame.setSize( 300, 200);
    frame.setVisible( true);
}
```

Listing 9.4 *Ausführbar als* **'ACTIONEVENTEXAMPLE'**

Nach diesen Anpassungen führt jeder Klick auf den `JButton` zu einem veränderten Text, was Sie nachprüfen können, wenn Sie das Programm ACTIONEVENTEXAMPLE starten. Dieses Programmverhalten scheint ganz normal, aber eigentlich müssten wir uns folgende Fragen stellen:

1. Wieso läuft das Programm denn überhaupt noch? Wir haben doch in Kapitel 7 gelernt, dass ein Programm terminiert, sobald die letzte Anweisung der `main()`-Methode abgearbeitet ist, es sei denn, es wurde ein User-Thread gestartet.
2. Wie werden die `ActionEvents` erzeugt und an die Bedienelemente

weitergeleitet?

Frage 1: Wieso läuft das Programm denn überhaupt noch?

Die Situation ist tatsächlich etwas mysteriös, da wir selbst keinen Thread in der `main()`-Methode gestartet haben. Weil das Programm aber augenscheinlich noch läuft, muss mindestens noch ein vom `main()`-Thread abgespaltener User-Thread aktiv sein. Eben darin besteht die Erklärung: Immer dann, wenn wir GUI-Komponenten nutzen, werden von der JVM automatisch weitere Threads zur Verarbeitung der Interaktionen in der Benutzeroberfläche gestartet. Um diesen Sachverhalt zu verifizieren, erweitern wir unser Beispielprogramm. In dessen `main()`-Methode geben wir alle aktuell laufenden Threads aus, wozu wir die in Abschnitt 7.1.2 erstellte Utility-Klasse `ThreadUtils` einsetzen:

```
ThreadUtils.dumpThreads();
```

Es kommt in etwa zu folgender Ausgabe, wobei die Angaben in eckigen Klammern die Informationen Thread-Name, Priorität und Parent-Thread darstellen:

```
Thread-Group java.lang.ThreadGroup[name=main,maxpri=10] contains 4 threads
Thread Thread[main,5,main]
Thread Thread[AWT-Shutdown,5,main]
Thread Thread[AWT-Windows,6,main]
Thread Thread[AWT-EventQueue-0,6,main]
```

In der Ausgabe ist der Thread `AWT-EventQueue` aufgelistet. Dieser sorgt für die Verarbeitung der Ereignisse in der Benutzeroberfläche. Der Name ist etwas irreführend, denn es handelt sich nicht um die Event-Queue des AWTs bzw. von Swing (häufig auch AWT-Event-Queue genannt), sondern der Thread ist tatsächlich vom Typ `java.awt.EventDispatchThread`. Da er aber aus der Klasse `java.awt.Event-Queue` erzeugt wird, kommt dieser Name zustande.

Frage 2: Wer erzeugt die `ActionEvents` und wie werden diese an die Bedienelemente weitergeleitet?

Die JVM registriert Tastatur- und Mauseingaben und wandelt diese in Objekte um, die als Ereignisse in die AWT-Event-Queue eingestellt werden. Der zuvor genannte Thread `EventDispatchThread` verarbeitet die Ereignisse, indem die Event-Objekte sukzessive durch den Aufruf von `dispatchEvent(AWTEvent)` an Bedienelemente vom Basistyp `Component` weitergeleitet werden. Abhängig vom konkreten Typ des Bedienelements erfolgt dann die weitere Verarbeitung. Abbildung 9-14 zeigt diesen Ablauf leicht vereinfacht (es wurden einige zur Darstellung des Prinzips irrelevante Methodenaufrufe weggelassen).

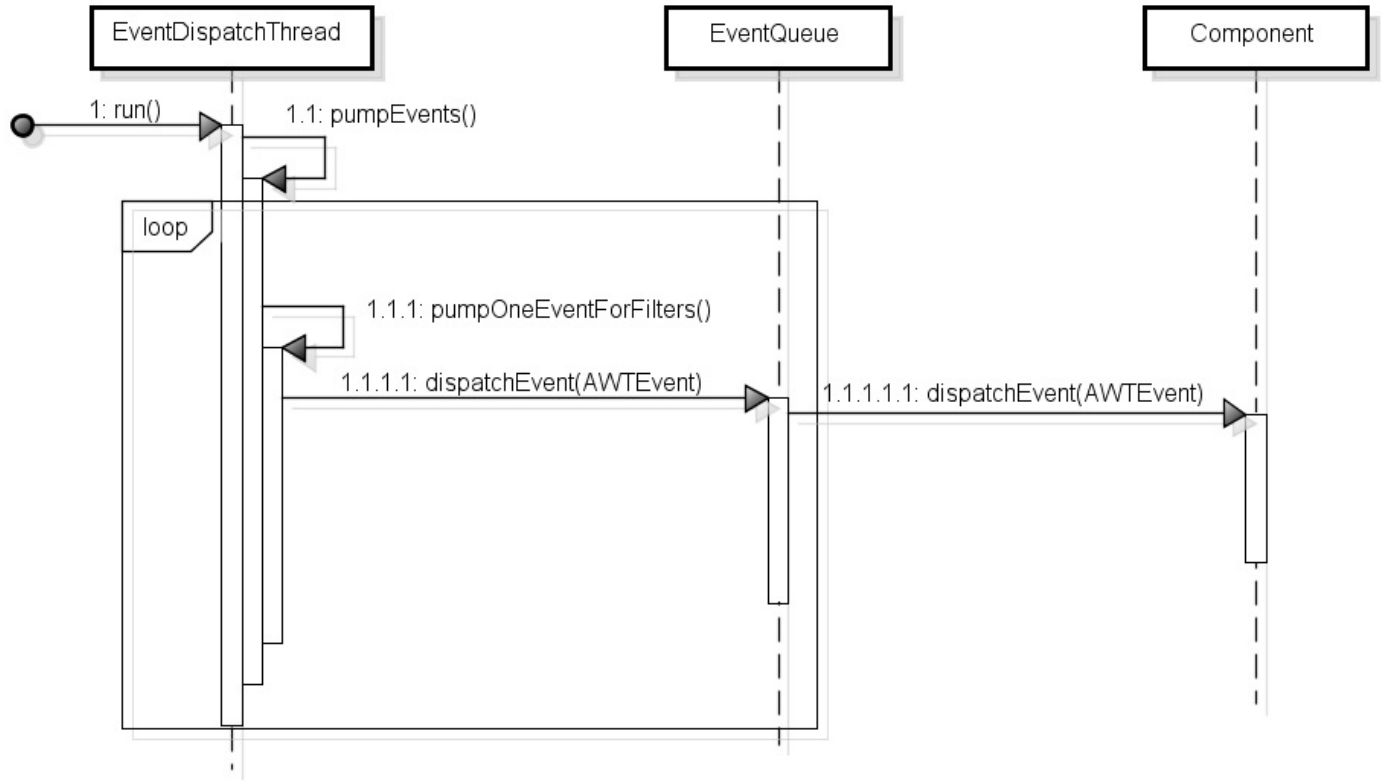


Abbildung 9-14 *Allgemeiner Ablauf beim Verarbeiten von Events*

Der weitere Ablauf zur Verarbeitung von Events ist komplexer, sodass sich die Aufrufhierarchie über ein Sequenzdiagramm nur recht unübersichtlich darstellen lässt. Daher habe ich hier einen Stacktrace zur Verdeutlichung gewählt (vgl Abbildung 9-15).

Abbildung 9-15 *Verarbeitung eines* `ActionEvent`*s in Swing*

Man erkennt, dass die Verarbeitung einige Indirektionen enthält. Für das Beispiel des Buttonklicks wird

1. zunächst die Verarbeitung in Form eines `MouseEvents` vom `JFrame` mithilfe

eines `LightweightDispatchers` an den `JButton` geleitet und

2. in späteren Verarbeitungsschritten das `MouseEvent` in ein `ActionEvent` umgewandelt.
3. Nach einigen weiteren Indirektionen landet das Event dann wieder im `JButton`. Dort wird zur Benachrichtigung der registrierten `ActionListener` die Methode `fireActionPerformed( ActionEvent)` aufgerufen. Das führt dazu, dass alle angemeldeten `ActionListener` mithilfe der Callback-Methode `actionPerformed( ActionEvent)` über das Ereignis benachrichtigt werden.

Hinweis: High-Level- und Low-Level-Events

Bewegt der Benutzer die Maus oder drückt er einen Mausknopf, so löst dies Events aus. Wir können in diesem Zusammenhang zwischen sogenannten High-Level- und Low-Level-Events unterscheiden. Die JVM erstellt für Tastatur- und Mauseingaben zunächst Low-Level-Events. Stehen diese Aktionen im Zusammenhang mit einem Bedienelement, so werden daraus automatisch High-Level-Events erzeugt. Das haben wir zuvor für den `JButton` kennengelernt: Ein Mausklick über einem `JButton` führt zu einem `MouseEvent`, woraus später ein `ActionEvent` entsteht.

9.1.5 Weitere gebräuchliche Event Listener

Nachdem nun ein grundsätzliches Verständnis für den Ablauf bei der Verarbeitung von Ereignissen aufgebaut wurde, stellt dieser Abschnitt einige in der Praxis gebräuchliche Event Listener vor, die das JDK bereitstellt. Ein `WindowListener` erlaubt es, auf verschiedene Ereignisse im Zusammenhang mit Fenstern reagieren zu können, z. B. auf eine Aktivierung oder ein Schließen eines Fensters. Mausklicks und -bewegungen kann man mithilfe eines `MouseListeners` bzw. eines `MouseMotionListeners` auswerten. Tastatureingaben lassen sich durch einen `KeyListener` verarbeiten.

Exemplarisch schauen wir uns `KeyListener` und `WindowListener` genauer an. Wobei ich für Letztere zuvor noch auf einige Besonderheiten beim Schließen von Fenstern hinweise.

Besonderheiten beim Schließen von Fenstern

Die bisher erstellten schlichten Beispielapplikationen verlieren recht schnell ihren Reiz. Dann klicken wir auf das Schließkreuz, wie wir es zum Beenden von Programmen gewohnt sind. Das Applikationsfenster wird geschlossen. Doch Beenden wir so auch unser Programm? Nein, denn standardmäßig wird das Fenster lediglich unsichtbar und verschwindet daher vom Bildschirm. Dass dies tatsächlich so ist, sehen Sie daran, dass das Programm bei einem Start aus Eclipse heraus noch nicht als beendet angezeigt wird.[2]

Diese Reaktion ist meiner Meinung nach unerwartet! Ein Programm soll in der Regel bei einem Klick auf das Schließkreuz beendet werden.[3] Besonders ungünstig finde ich, dass dieses Verhalten dem voreingestellten Standard entspricht. Was können wir tun, um das anzupassen?

Für jede Instanz der Klasse `JFrame` lässt sich durch den Aufruf der Methode `setDefaultCloseOperation(int)` die Programmreaktion beim Betätigen des Schließ-kreuzes spezifizieren. Als Parameter sind die folgenden, in der Klasse `JFrame` definierten Werte vorgesehen:

- `DISPOSE_ON_CLOSE` – Das Fenster wird unsichtbar. Falls keine weiteren Fenster der Applikation geöffnet sind, endet diese.
- `EXIT_ON_CLOSE` – Das Programm wird durch Aufruf von `System.exit()` umgehend beendet.
- `HIDE_ON_CLOSE` – Das Fenster wird lediglich unsichtbar, aber die Applikation wird nicht beendet. Das ist die ***Standardeinstellung***.
- `NOTHING_ON_CLOSE` – Es erfolgt keine Reaktion.

Diese vier Verhaltensweisen verstecken die Details der im Hintergrund stattfindenden Verarbeitung. Für kleinere Applikationen, die keine spezifische Aktion beim Schließen eines Fensters benötigen, sind die angebotenen Verhaltensweisen ausreichend. Nahezu immer ist das durch die Konstante `DISPOSE_ON_CLOSE` festgelegte Verhalten die bevorzugte Wahl. Möchte man die Programmreaktion allerdings im Detail steuern, so muss man dazu stattdessen einen `WindowListener` nutzen. Wie man dies macht und in welchen Situationen das sinnvoll ist, beschreibt der folgende Abschnitt.

`WindowListener` – Verarbeitung von `WindowEvents`

Wenn sich Änderungen am Status eines Fensters ergeben, wird ein `WindowEvent` ausgelöst und an alle registrierten `WindowListener` propagiert. Dies ist z. B. der Fall, wenn ein Fenster geöffnet oder geschlossen, aktiviert oder deaktiviert wird. Zur Reaktion auf ein solches Ereignis sind im Interface `WindowListener` folgende Callback-Methoden mit sprechenden Namen definiert:

```
public interface WindowListener extends EventListener
{
    public void windowOpened(WindowEvent event);
    public void windowClosing(WindowEvent event);
    public void windowClosed(WindowEvent event);
    public void windowIconified(WindowEvent event);
    public void windowDeiconified(WindowEvent event);
    public void windowActivated(WindowEvent event);
    public void windowDeactivated(WindowEvent event);
}
```

Aus diesen Methoden ist nun die für unseren Anwendungsfall passende zu wählen: Wollen wir auf einen Klick auf das Schließkreuz reagieren, so müssen wir dazu die

Methode `windowClosing(WindowEvent)` implementieren. Dadurch haben wir die Möglichkeit, vor dem Schließen des Fensters steuernd einzugreifen. Die Callback-Methode `windowClosed(WindowEvent)` wird nach dem Schließen des Fensters aufgerufen. In deren Realisierung können z. B. Aufräumarbeiten stattfinden.

Realisierung eines `WindowListeners`

Vor dem Schließen des Hauptfensters einer Applikation, also unmittelbar vor dem möglichen Programmende, bietet es sich oftmals an, eine Dialogbox zu präsentieren und Rückfrage zu halten, ob das Programm tatsächlich beendet werden soll. Außerdem können in der Implementierung der Methode `windowClosing(WindowEvent)` einige Aktionen erfolgen, etwa die Speicherung bisher nicht gesicherter Daten oder die Freigabe von Ressourcen. ***Derartige Aktionen sind bei einer Behandlung über die Standardverhaltensweisen leider nicht möglich, sodass dabei eventuell Systemressourcen belegt bleiben***. Daran sieht man, wie wichtig eine Realisierung eines `WindowListeners` sein kann. Eine mögliche Umsetzung einer Rückfrage vor Programmende sieht wie folgt aus:

```
public final class WindowClosingListener implements WindowListener
{
   // Alle leer implementiert, da nicht von Interesse
   public void windowOpened(final WindowEvent event) {}
   public void windowClosed(final WindowEvent event) {}
   public void windowIconified(final WindowEvent event) {}
   public void windowDeiconified(final WindowEvent event) {}
   public void windowActivated(final WindowEvent event) {}
   public void windowDeactivated(final WindowEvent event) {}

   public void windowClosing(final WindowEvent event)
   {
      final int answer = JOptionPane.showConfirmDialog(parentFrame, "Wollen "
+
                                                    "Sie das Programm
beenden?");
      if (answer == JOptionPane.YES_OPTION)
      {
         // Fenster unsichtbar machen und (bei Bedarf) Ressourcen freigeben
         parentFrame.setVisible(false);
         parentFrame.dispose();

         // Programmende forcieren
         System.exit(0);
      }
   }
}
```

Neben der Methode `windowClosing(WindowEvent)` sind im Interface `WindowListener` noch weitere Methoden definiert. All diese müssen für eine eigene Realisierung auch noch implementiert werden, selbst wenn man – wie auch hier – nur auf *ein* spezielles Ereignis reagieren möchte. Diese Methoden erhalten dann eine leere Implementierung, wie dies im obigen Listing gezeigt ist. Wie man sieht, verursacht das lediglich Schreibaufwand für Ereignisse, die nicht von Interesse

sind. In diesem Fall habe ich eine recht kompakte Schreibweise gewählt, sodass dieser Negativpunkt nicht wirklich auffällt. Dabei habe ich bewusst gegen meine ansonsten verwendeten Coding Conventions verstoßen – denn im Normalfall sind öffnende und schließende geschweifte Klammern in eigenen Zeilen zu notieren. Weitere Details finden Sie in Kapitel 19.

Adapterklassen und die Klasse `WindowAdapter`

Das Problem mit dem Schreibaufwand haben die JDK-Entwickler erkannt und bieten für einige Interfaces sogenannte Adapterklassen als Implementierungsvereinfachung an. Im JDK versteht man unter einer Adapterklasse eine Klasse, die alle Methoden des jeweiligen Interface leer implementiert. Leitet man eine Listener-Implementierung von einer solchen Adapterklasse ab, so müssen folglich nur noch diejenigen Methoden überschrieben werden, für die tatsächlich Funktionalität bereitgestellt werden soll.

Folgendes Listing zeigt dies am Beispiel der Klasse `ExitListener`, die die Funktionalität der eben erstellten Klasse `WindowClosingListener` bereitstellt, aber wesentlich übersichtlicher implementiert werden kann, da hier die Basisklasse `WindowAdapter` zum Einsatz kommt. Zur Reaktion auf ein Schließereignis muss dort nur die Methode `windowClosing( WindowEvent)` überschrieben werden:

```
public final class ExitListener extends WindowAdapter
{
    private final JFrame parentFrame;

    public ExitListener( final JFrame parentFrame)
    {
        this. parentFrame = parentFrame;
    }

    public void windowClosing( final WindowEvent event)
    {
        final int answer = JOptionPane. showConfirmDialog( parentFrame, "Wollen
              Sie das Programm beenden?");
        if ( answer == JOptionPane. YES_OPTION)
        {
            // Fenster unsichtbar machen und ( bei Bedarf) Ressourcen freigeben
            parentFrame. setVisible( false);
            parentFrame. dispose();

            // Programmende forcieren
            System. exit( 0);
        }
    }
}
```

Hinweis

Achten Sie darauf, dass die Registrierung eines `WindowListeners` den Standardmechanismus nicht automatisch deaktiviert! Ohne weiteres Zutun entspricht

dieser dem Wert `JFrame.HIDE_ON_CLOSE` oder aber einer über `setDefaultCloseOperation(int)` gewählten Einstellung! Das so spezifizierte Verhalten wird *zusätzlich* zu den registrierten `WindowListenern` aufgerufen, was häufig unerwartet und ungewünscht ist. Daher wird man normalerweise den Standardmechanismus deaktivieren wollen, indem man explizit die Methode `setDefaultCloseOperation(int)` mit dem Übergabewert `JFrame.DO_NOTHING_ON_CLOSE` aufruft. Somit wird im Beispiel dann nur noch die Funktionalität des `ExitListeners` ausgeführt.

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("WindowClosingExample");

    // ExitListener registrieren
    frame.addWindowListener(new ExitListener(frame));
    // Wichtig, um Standardfunktionalität auszuschalten
    frame.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);

    frame.setSize(300, 200);
    frame.setVisible(true);
}
```

Listing 9.5 *Ausführbar als* **'WINDOWCLOSINGEXAMPLE'**

`KeyListener` – Verarbeitung von `KeyEvents`

Tastatureingaben lassen sich recht einfach verarbeiten. Benachrichtigungen werden in Form von `KeyEvents` an interessierte `KeyListener` propagiert. Zur Vereinfachung der Handhabung ist im JDK auch hier wieder eine Adapterklasse `KeyAdapter` definiert.

Betrachten wir nun einen in der Praxis häufig anzutreffenden Anwendungsfall: Abhängig von den Eingaben innerhalb eines Textfelds soll ein Button (z. B. der OKButton) aktiviert oder deaktiviert werden – je nachdem, ob der eingegebene Text eine Mindestlänge von drei Zeichen besitzt. Diese Funktionalität wird durch die anonyme `KeyListener`-Klasse sowie die Utility-Klasse `TextLengthChecker` realisiert:

```
public final class KeyListenerExample
{
    public static void main(final String[] args)
    {
        final JFrame frame = new JFrame("KeyListenerExample");
        initGuiForKeyListenerExample(frame);

        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);
    }

    private static void initGuiForKeyListenerExample(final JFrame frame)
    {
        final JLabel     label     = new JLabel("Enter some text " +
                                                "(3 or more letters):");
```

```
        final JTextField textField = new JTextField(20);
        final JButton    button    = new JButton("Press me!");
        button.setEnabled(false);

        // KeyListener registrieren
        textField.addKeyListener(new KeyAdapter()
        {
            @Override
            public void keyReleased(final KeyEvent e)
            {
                button.setEnabled(TextLengthChecker.
                                  checkRequiredLength(textField, 3));
            }
        });
        // Layout gestalten
        final JPanel inputPanel = new JPanel();
        inputPanel.add(label);
        inputPanel.add(textField);

        final JPanel buttonPanel = new JPanel(new
FlowLayout(FlowLayout.RIGHT));
        buttonPanel.add(button);

        frame.add(inputPanel, BorderLayout.NORTH);
        frame.add(buttonPanel, BorderLayout.SOUTH);
    }
}
```

Listing 9.6 *Ausführbar als* **'KEYLISTENEREXAMPLE'**

Die Textlängenprüfung haben wir in folgende Klasse ausgelagert, da wir diese Funktionalität später noch ein paarmal benötigen.

```
public class TextLengthChecker
{
    public static boolean checkRequiredLength(final JTextField textField,
                                              final int requiredLength)
    {
        return textField.getText() != null &&
               textField.getText().length() >= requiredLength;
    }
}
```

9.1.6 Varianten der Ereignisverarbeitung

Zur Realisierung der Ereignisverarbeitung kann man verschiedene Varianten nutzen, die wir größtenteils schon kennengelernt haben:

1. **Listener-Klasse als anonyme Klasse** – Diese Variante haben wir bereits für die Typen `ActionListener`, `MouseListener` und `KeyListener` eingesetzt.
2. **Listener-Klasse als (statische) innere Klasse** – Diese Form der Umsetzung wurde zur Behandlung von `ComponentEvents` genutzt.
3. **Listener-Klasse als separate Klasse** – Diese Variante haben wir am Beispiel der Klasse `ExitListener` zur Verarbeitung von `WindowEvents` kennengelernt.
4. **Listener-Implementierung durch Komponentenklasse** – Diese Form wurde

bisher noch nicht beschrieben.

Für die folgenden Beispiele nehmen wir zur Betrachtung der jeweiligen Vor- und Nachteile an, dass die Bedienelemente `button` und `textField` wie zuvor definiert sind.

Listener-Klasse als anonyme Klasse

Als Erstes schauen wir auf die Variante, die die Listener-Klasse als anonyme Klasse direkt an Ort und Stelle beim Registrieren des Listeners – unter Zuhilfenahme der zuvor erstellten Utility-Klasse `TextLengthChecker` und deren Methode `checkRequiredLength(JTextField, int)` – beispielsweise folgendermaßen implementiert:

```
textField.addKeyListener(new KeyAdapter()
{
    public void keyReleased(final KeyEvent event)
    {
        button.setEnabled(TextLengthChecker.checkRequiredLength(textField,
3));
    }
});
```

Bewertung

Eine anonyme Klasse eignet sich immer dann zur Ereignisverarbeitung, wenn diese sehr spezifische und relativ triviale Dinge erledigt. Dann ist es zudem sehr wahrscheinlich, dass nur ein Listener-Objekt benötigt wird.

Listener-Klasse als (statische) innere Klasse

Bei dieser Form der Realisierung wird eine innere Klasse implementiert, die die Listener-Schnittstelle erfüllt. Das ist in der folgenden Klasse `MinLengthKeyHandler` der Fall. Dort wird basierend auf dem vorherigen Beispiel der Längenprüfung folgende Erweiterung realisiert: Wenn zu wenig Zeichen eingegeben wurden, dann wird sowohl die Farbe auf Rot als auch der Stil der Schriftart auf fett geändert. Damit ergibt sich folgende Implementierung (zur besseren Lesbarkeit wird die Methode `checkRequiredLength()` aus der `TextLengthChecker` statisch importiert):

```
public final class InnerListenerExample
{
    // …
    textField.addKeyListener(new MinLengthKeyHandler());
    // …

    final class MinLengthKeyHandler extends KeyAdapter
    {
        private final Color originalColor = textField.getForeground();
        private final Font originalFont = textField.getFont();
        private final Font boldFont = originalFont.deriveFont(Font.BOLD);
```

```
        @Override
        public void keyReleased(final KeyEvent event)
        {
            final boolean hasEnoughText = checkRequiredLength(textField, 3);
            button.setEnabled(hasEnoughText);
            adjustTextField(textField, hasEnoughText);
        }

        public void adjustTextField(final JTextField textField,
                                    final boolean hasEnoughText)
        {
            if (!hasEnoughText)
            {
                textField.setForeground(Color.RED);
                textField.setFont(boldFont);
            }
            else
            {
                textField.setForeground(originalColor);
                textField.setFont(originalFont);
            }
        }
    }
}
```

Bewertung

Diese Variante eignet sich immer dann, wenn die Ereignisverarbeitung etwas komplexer ist. Im Beispiel erkennt man aber auch, dass sich leicht eine enge Bindung zwischen der äußeren Klasse und der Listener-Klasse ergibt. Für einige Anwendungsfälle ist das akzeptabel. In der Regel erschwert dies aber die Wartbarkeit, da Abhängigkeiten nicht unmittelbar ersichtlich sind.

Variante: Statische innere Klasse

Wenn man die Listener-Klasse statisch definiert, so lassen sich Abhängigkeiten klar erkennen und äußern sich in Kompilierfehlern:[4] Für das Beispiel würde aber der Zugriff auf das `JTextField` und der Zugriff auf den `JButton` benötigt. Die Abhängigkeiten können als Parameter bei der Konstruktion übergeben werden. Da diese Variante nahezu identisch zu der im Folgenden vorgestellten Umsetzung der Listener-Klasse als separate Klasse ist, verzichte ich hier auf eine Darstellung der Implementierung.

Listener-Klasse als separate Klasse

Zur Realisierung umfangreicherer Listener-Funktionalität bietet es sich an, eine eigenständige Klasse zu nutzen. Wie bereits für die Variante der statischen inneren Klassen erwähnt, erfordert diese Variante die Übergabe benötigter Daten – meistens bietet sich eine Parameterübergabe im Konstruktor an, etwa folgendermaßen:

```
public final class MinLengthKeyHandler extends KeyAdapter
```

```
{
    private final JTextField textField;
    private final JButton button;
    private final Color originalColor;
    private final Font originalFont;
    private final Font boldFont;

    public MinLengthKeyHandler(final JTextField textField, final JButton
button)
    {
        this.textField = textField;
        this.button = button;
        this.originalColor = textField.getForeground();
        this.originalFont = textField.getFont();
        this.boldFont = originalFont.deriveFont(Font.BOLD);
    }

    @Override
    public void keyReleased(final KeyEvent event)
    {
        final boolean hasEnoughText = checkRequiredLength(textField, 3);
        button.setEnabled(hasEnoughText);
        adjustTextField(textField, hasEnoughText);
    }

    public void adjustTextField(final JTextField textField,
                                final boolean hasEnoughText)
    {
        // Wie zuvor gezeigt
    }
}
```

Bewertung

Diese Art der Implementierung ermöglicht eine gute Trennung von Benutzeroberfläche und Ereignisverarbeitung. Das liegt daran, dass im Gegensatz zu den vorherigen Ansätzen keine Möglichkeit zum direkten Zugriff aus dem Listener auf Attribute und Methoden des Bedienelements besteht. Für kleinere Applikationen mag das teilweise unpraktisch sein. Für größere Applikationen erreicht man so jedoch eine bessere Strukturierung. Auch werden Abhängigkeiten klarer ersichtlich, da die Kommunikation nur mehr durch Aufrufe öffentlicher Methoden erfolgen kann und somit Direktzugriffe unterbunden werden. Das ist insofern erstrebenswert, als durch anonyme Klassen sogar Zugriffe auf privat definierte Elemente des ereignisauslösenden Bedienelements möglich sind.

Listener-Implementierung durch Komponentenklasse

Statt der zuvor genannten Varianten kann man auch das Bedienelement selbst als Event Listener realisieren. Die Listener-Funktionalität ist dann Bestandteil der Klasse der GUI-Komponente und wird nicht in einer separaten Klasse gekapselt.

Betrachten wir als Beispiel die Klasse `MultiEventListenerFrame`, die von `JFrame` erbt und die Interfaces `ActionListener` und `KeyListener`

implementiert:

```
public class MultiEventListenerFrame extends JFrame implements ActionListener,
                                                                KeyListener
{
    private final JTextField textField = new JTextField( "Enter some text! ");
    private final JButton button = new JButton( "Press Me! ");
    private final Color originalColor;
    private final Font originalFont;
    private final Font boldFont;

    public MultiEventListenerFrame()
    {
        super( "MultiEventListenerFrame");

        add( new JLabel( "Textlabel: "), BorderLayout.WEST);
        add( this.textField, BorderLayout.CENTER);
        add( this.button, BorderLayout.SOUTH);

        this.originalColor = textField.getForeground();
        this.originalFont = textField.getFont();
        this.boldFont = originalFont.deriveFont( Font.BOLD);

        this.button.addActionListener( this);
        this.textField.addKeyListener( this);
    }

    @Override
    public void actionPerformed( ActionEvent event)
    {
        JOptionPane.showMessageDialog( this, "Button clicked! ");
    }

    @Override
    public void keyReleased( final KeyEvent event)
    {
        final boolean hasEnoughText = checkRequiredLength( textField, 3)
        button.setEnabled( hasEnoughText);
        adjustTextField( textField, hasEnoughText);
    }

    // …
}
```

Bewertung

Zwar bietet diese Art der Implementierung des Listeners vollen Zugriff auf alle Attribute des Bedienelements, jedoch geht dies leider mit einer schlechten Trennung von Zuständigkeiten einher: Die GUI-Komponentenklasse ist für diverse Funktionalität von der Darstellung bis zur Verarbeitung von Benutzereingaben verantwortlich. Dadurch wird es mühsamer, zu erkennen, welche Methoden für welche Funktionalität zuständig sind. Schnell entstehen auch Querabhängigkeiten. Diese wiederum erschweren es, die Listener-Funktionalität aus der GUI-Komponente herauszulösen oder aber an anderer Stelle zu verwenden.

Berechtigterweise fragen Sie sich jetzt vielleicht, warum ich diese Variante dann

so ausführlich darstelle. Der Grund ist, dass man deren Gebrauch in der Praxis leider immer wieder sieht. Meine Intention ist es, Ihnen zu vermitteln, warum das ungünstig ist und Sie besser eine der anderen zuvor vorgestellten Varianten der Listener-Realisierung nutzen sollten. Schauen wir uns also einige Schwachstellen an.

Probleme bei Listener-Implementierung durch Komponentenklasse

Solange Sie nur sehr kleine Applikationen erstellen, werden Sie vielleicht nicht sofort auf Schwierigkeiten stoßen. Allerdings kommt es recht schnell dazu: Es reicht bereits aus, im obigen Beispiel einen weiteren `JButton` mitsamt Ereignisverarbeitung ergänzen zu wollen. Wir erkennen dann, dass es für den neuen `JButton` nicht möglich ist, eine weitere Realisierung der Methode `actionPerformed( ActionEvent)` hinzuzufügen, da diese Methode ja bereits für den ersten `JButton` implementiert ist. Soll nun die Ereignisverarbeitung auch für den zweiten `JButton` erfolgen, so bleibt bei diesem Ansatz als einzige Lösung, eine Fallunterscheidung in die Methode `actionPerformed( ActionEvent)` einzubauen. Das könnte in etwa wie folgt aussehen:

```
public void actionPerformed( ActionEvent event)
{
    if ( event. getSource( ) == button)
    {
        JOptionPane. showMessageDialog( this, "Button clicked! ");
    }
    else if ( event. getSource( ) == otherButton)
    {
        performActionForOtherButton( );
    }
}
```

Bereits für zwei Bedienelemente erkennt man, wie wenig elegant diese Umsetzung ist. Schnell leiden sowohl die Übersichtlichkeit als auch die Modularität darunter, wodurch die Wiederverwendbarkeit erschwert wird: Die einzelnen Funktionalitäten zur Ereignisverarbeitung lassen sich mit zunehmender Zahl an Bedienelementen immer mühsamer voneinander trennen – insbesondere dann, wenn die Fallunterscheidung weniger klar als im Beispiel realisiert ist. Die Nachteile gelten verstärkt dann, wenn eine Vielzahl von Ereignissen beobachtet werden soll und dadurch der Umfang der Klasse wächst. Spätestens dann bietet es sich an, Event Listener in eigenständigen Klassen zu realisieren, um eine klare Struktur zu erreichen.

9.2 Multithreading und Swing

Zu einem gelungenen GUI gehören neben einer ansprechenden grafischen Gestaltung auch ergonomische Aspekte, etwa die schnelle Reaktion auf Benutzereingaben. Dadurch entsteht das Gefühl der Kontrolle und der Benutzer

empfindet das Programm als gut bedienbar. Leider sieht man in der Praxis immer mal wieder weniger reaktive GUIs, die eher einen gegenteiligen Eindruck hinterlassen. Die folgenden Abschnitte greifen das zuvor kurz behandelte Thema Event Handling wieder auf und bieten einen Überblick zur Gestaltung reaktiver und Thread-sicherer Swing-GUIs.

Bei der Verarbeitung von Ereignissen sollte einer guten Antwortgeschwindigkeit unbedingt Beachtung geschenkt werden. Folglich muss die Bearbeitung eines Ereignisses sehr schnell abgeschlossen sein. Allerdings erfordern Benutzeraktionen teilweise umfangreiche oder länger dauernde Berechnungen. Würden diese innerhalb des Threads der Ereignisverarbeitung ablaufen, so käme dadurch die Abarbeitung anderer Ereignisse ins Stocken. Daher bietet es sich an, länger dauernde Berechnungen in eigene Threads auszulagern, um so das GUI reaktiv halten zu können und dort Rückmeldungen über den Berechnungsfortschritt zu geben. Abschließend sind die Berechnungsergebnisse im GUI darzustellen. Die Aktualisierung auf Basis von (Zwischen-) Ergebnissen kann jedoch Probleme bereiten, wenn dies aus beliebigen Threads heraus erfolgt, da eine Mehrheit der Methoden in Swing nicht Thread-sicher ausgelegt ist. Daher muss das Zusammenführen von Berechnungsergebnissen und GUI einem speziellen Ablauf folgen. Ansonsten drohen verschiedene Probleme bis hin zu Programmabstürzen: Manchmal sieht man lediglich veraltete Werte; zum Teil entstehen aber schwerwiegendere Darstellungsprobleme, etwa leere Zeilen am Ende einer Tabelle, die beim Auswählen zu Exceptions führen.

9.2.1 Crashkurs Event Handling in Swing

Bei der Beschreibung zu den Grundlagen der Ereignisverarbeitung in Abschnitt 9.1.4 habe ich den ***Event Dispatch Thread*** (EDT) erwähnt, der für Swing bzw. AWT ein zentrales Element darstellt: Dieser spezielle Thread dient dazu, verschiedenste Arten von Aktionen im Zusammenhang mit dem GUI auszuführen. Ebenso wie viele andere GUI-Frameworks arbeitet auch Swing mit Singlethreading, d. h., alle GUI-Ereignisse werden sequenziell abgearbeitet. Dazu gehört etwa, Fensterinhalte neu zu zeichnen oder Ereignisse (Buttonklick, Menüauswahl usw.) zu verarbeiten. Man kann sich die Abarbeitung wie an einem Event-Förderband vorstellen. Der EDT entnimmt aus einer speziellen `EventQueue` ständig Aufgaben und führt diese aus. Neue Aufgaben entstehen in der Regel als Reaktion auf Benutzeraktionen, etwa wenn Menüs ausgewählt werden oder Buttonklicks erfolgen. Abbildung 9-16 stellt die Vorgänge schematisch dar.

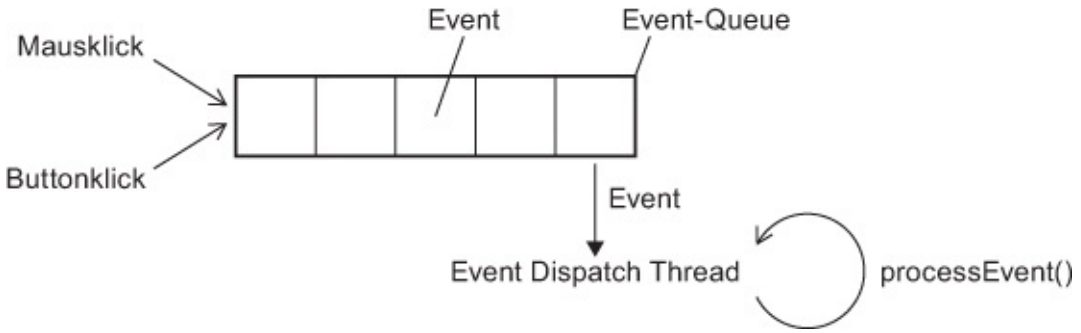


Abbildung 9-16 *Ereignisverarbeitung im EDT*

Die Grafik verdeutlicht die Arbeitsweise und auch die Schwachstelle: Stockt es in der Bearbeitung der aktuellen Aufgabe, so werden dadurch alle nachfolgenden Schritte blockiert. Um das zu vermeiden, gilt folgende Regel: ***Alle Aktionen, die direkt im EDT ablaufen, sollten möglichst schnell ausgeführt werden, um die Verarbeitung anderer Ereignisse nicht zu behindern.*** Bei deren Missachtung kommt es zu einem wenig reaktiven GUI und zu Repaint-Problemen, etwa einem nicht neu gezeichneten Hintergrund. Das werden wir in Abschnitt 9.2.2 anhand eines Beispiels nachvollziehen.

Einstellen von Aktionen mit `invokeLater()` bzw. `invokeAndWait()`

Neben der Reaktion auf Benutzereingaben ist es darüber hinaus auch möglich, aus dem Programm heraus Aufgaben in die AWT-Event-Queue einzutragen. Diese Aufgaben werden durch Klassen beschrieben, die das Interface `Runnable` implementieren. Die Utility-Klasse `javax.swing.SwingUtilities` bietet zu deren Ausführung die zwei statischen Methoden `invokeLater(Runnable)` und `invokeAndWait(Runnable)`. Die beiden Methoden erlauben unterschiedliche Arten der Ausführung:

- **`invokeLater(Runnable)`** arbeitet ***asynchron***: Das übergebene `Runnable`-Objekt wird in die `EventQueue` eingefügt. Die Abarbeitung des aufrufenden Programmteils wird fortgesetzt. Ein Aufrufer weiß nicht, wann die Aufgabe abgearbeitet wird.
- **`invokeAndWait(Runnable)`** arbeitet ***synchron***: Das übergebene `Runnable`-Objekt wird in die AWT-Event-Queue eingefügt. Anschließend wird an der aufrufenden Programmstelle so lange gewartet, bis alle vorherigen Aufgaben sowie die eingestellte Aufgabe durch den EDT abgearbeitet sind.

Mithilfe beider Methoden ist es möglich, die Aktionen innerhalb des EDTs abzuarbeiten. Mit `invokeLater(Runnable)` kann man eine neue Aufgabe in Form eines `Runnables` in die `EventQueue` einstellen. Das ist sowohl während der Abarbeitung innerhalb des EDTs als auch aus jedem beliebigen anderen Thread möglich. Man nutzt das beispielsweise, wenn Werte im GUI oder den zugrunde liegenden Datenmodellen verändert werden müssen. Bekanntermaßen müssen diese Änderungen im EDT erfolgen, da ansonsten Inkonsistenzen drohen. Um

festzustellen, ob Anweisungen im EDT ausgeführt werden, bietet die Klasse `SwingUtilities` die Methode `isEventDispatchThread()`. Liefert diese den Rückgabewert `false`, werden die Anweisungen nicht durch den EDT ausgeführt. Es gilt die folgende ***Single-Thread-Regel***: ***Swing-Operationen und Änderungen an den Datenmodellen dürfen nur während der Abarbeitung durch den EDT ausgeführt werden***.

Es gibt noch eine Besonderheit bei der Ausführung im EDT: Bei einem Aufruf von `invokeAndWait(Runnable)` ist zu beachten, dass dieser niemals aus dem EDT heraus erfolgen darf, denn das würde zu einem Deadlock führen! Die momentane Abarbeitung würde so lange warten, bis die durch sie selbst neu eingestellte Aufgabe abgearbeitet ist. Diese kann jedoch nicht abgearbeitet werden, weil die aktuelle Aufgabe noch nicht beendet ist. Damit das nicht passieren kann, wird eine solche Ausführung automatisch verhindert: Erfolgt (versehentlich) ein Aufruf von `invokeAndWait(Runnable)` aus dem EDT, so kommt es zu folgendem Fehler:`java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread`.

9.2.2 Ausführen von Aktionen

Nachdem wir die Grundlagen zum Einstellen von Aufgaben in die AWT-Event-Queue kennengelernt haben, stelle ich nun drei Varianten der Ausführung von Aufgaben vor:

1. **Synchron im EDT** – Dieser einfache Ansatz führt die Aktion direkt in der Callback-Methode des aufgerufenen Event Listeners aus und nutzt keinen Aufruf von `invokeLater(Runnable)`. Das kann zu spürbaren Verzögerungen und Repaint-Problemen im GUI führen. Daher ist diese Variante nur für Prototypen oder zur direkten Abarbeitung sehr schnell ausführbarer Aktionen geeignet.
2. **Verzögert im EDT** – Dieser Ansatz ist minimal besser als der vorherige, aber selten ausreichend. Hierbei wird eine länger dauernde Aktion durch `invokeLater(Runnable)` in die AWT-Event-Queue eingestellt, sodass zunächst alle dort bereits vorhandenen Events verarbeitet werden können. Im Speziellen wird auch ein Neuzeichnen des Fensters zur Darstellung von Zustandsänderungen möglich.
3. **Parallel zum EDT** – Dies ist die komplizierteste, aber zu bevorzugende Umsetzung. Sie ermöglicht sowohl eine gute Abarbeitungsgeschwindigkeit im GUI als auch parallele Berechnungen. Allerdings muss man sich dann an die Spielregel halten, Ergebnisse durch `invokeLater(Runnable)` bzw. `invokeAndWait(Runnable)` zu kommunizieren.

Synchron im EDT

Beginnen wir die Betrachtung mit der synchronen Abarbeitung von Aktionen. Im folgenden Listing ist dieser Ansatz exemplarisch anhand der Methode `actionPerformed(ActionEvent)` gezeigt. Verzögerungen der Abarbeitung im EDT durch aufwendige Berechnungen werden dabei mithilfe von Aufrufen der Methode `safeSleep(long)` der Klasse `SleepUtils` (vgl. Abschnitt 7.1.3) simuliert:

```
public static final class LongRunningMenuAction extends AbstractAction
{
    public LongRunningMenuAction(final String resourceKey)
    {
        super(resourceKey);
    }

    public void actionPerformed(ActionEvent e)
    {
        resultTextfield.setText("Computation started");

        for (int i=1; i < 4; i++)
        {
            // Rückmeldung ausgeben => wirkt nicht
            resultTextfield.setText("Computation " + i);
            // Verzögerung (stellvertretend für aufwendige Berechnung)
            SleepUtils.safeSleep(1000);
        }

        resultTextfield.setText("Computation finished");
    }
}
```

Listing 9.7 *Ausführbar als* **'AWTEDTEXAMPLE'**

Führt man das Programm AWTEDTEXAMPLE aus, so erscheint ein Applikationsfenster. Startet man dort die länger dauernde Aufgabe über das Menü und ändert anschließend die Größe des Applikationsfensters, so erhält man eine Ausgabe ähnlich zu der in Abbildung 9-17. Offensichtlich erfolgt kein Neuzeichnen: Zum einen bleibt das Menü, das die Aktion ausgelöst hat, sichtbar. Zum anderen wird der neu erscheinende Hintergrund schwarz dargestellt.

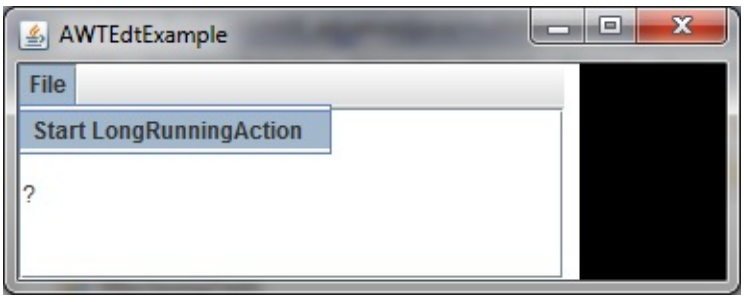


Abbildung 9-17 *Repaint-Probleme bei der direkten Ereignisverarbeitung im EDT*

Dieses Beispiel verdeutlicht die Nachteile dieser Art der Verarbeitung. Der offensichtlichste ist, dass die Verarbeitung anderer Events verzögert wird. Häufig wird von Benutzern die Möglichkeit zum Abbruch einer Berechnung gewünscht, was ebenfalls mit der synchronen Abarbeitung im EDT nicht realisierbar ist. Gleiches gilt für die Darstellung von Zwischenergebnissen: Zwar erfolgen innerhalb der Schleife

einige Aufrufe von `setText(String)` an das Textfeld zur Darstellung von Zwischenergebnissen. Diese werden jedoch nicht gezeigt. Erst im Anschluss an die synchrone Abarbeitung der Methode `actionPerformed(ActionEvent)` werden die durch `setText(String)` automatisch ausgelösten Events zum Neuzeichnen bearbeitet und dadurch Änderungen im GUI sichtbar, wie dies in Abbildung 9-18 angedeutet ist. Als Besonderheit werden mehrere Events zum Neuzeichnen zu einem einzigen Event zusammengefasst.

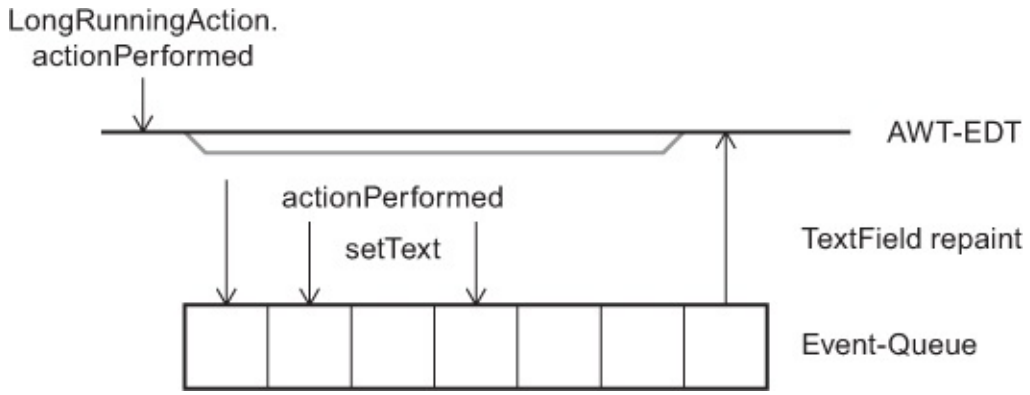


Abbildung 9-18 *Verzögerung des Neuzeichnens*

Verzögert im EDT

Zur verzögerten Ausführung im EDT wird die Aktion in Form eines `Runnables` beschrieben, das dann durch Aufruf der Methode `invokeLater(Runnable)` in die AWT-Event-Queue eingestellt wird. Allerdings beobachtet man hierbei prinzipiell die gleichen Probleme wie bei der rein synchronen Variante: Während der Abarbeitung der länger dauernden Aktion ist das GUI für eine Weile nicht reaktiv. Diese Variante führt jedoch zuvor eingetroffene Ereignisse, etwa Repaints, noch vor der Aktion selbst aus, was eine kleine Verbesserung in der Darstellung bewirkt: So wird beispielsweise verhindert, dass ein Menü so lange sichtbar bleibt, bis eine damit gestartete Aktion vollständig abgearbeitet ist. Das ist aber nur eine marginale Verbesserung.

Parallel zum EDT

Intuitiv kommt man auf die Idee, länger dauernde Aktionen durch separate Threads abarbeiten zu lassen. Dabei ist allerdings die Single-Thread-Regel zu beachten. Im Speziellen muss ***die Propagation von Berechnungsergebnissen unbedingt im EDT geschehen***, um Probleme zu vermeiden.

Im folgenden Listing ist die Klasse `AsyncLongRunningMenuAction` gezeigt, in deren Methode `safeSetText(String)` es durch Aufruf der Methode `invokeLater(Runnable)` sicher möglich wird, Berechnungsergebnisse im EDT zu propagieren und im GUI darzustellen:

```
public static final class AsyncLongRunningMenuAction extends AbstractAction
{
    public AsyncLongRunningMenuAction(final String resourceKey)
```

```
    {
        super(resourceKey);
    }

    public void actionPerformed(final ActionEvent e)
    {
        resultTextfield.setText("Computation started");

        // Worker-Thread abspalten
        new Thread()
        {
            @Override
            public void run()
            {
                for (int i=1; i < 4; i++)
                {
                    // Propagation in den EDT
                    safeSetText("Computation " + i);
                    // Verzögerung (stellvertretend für aufwendige Berechnung)
                    SleepUtils.safeSleep(1000);
                }

                // Propagation in den EDT
                safeSetText("Computation finished");
            }
        }.start();
    }
}

private static void safeSetText(final String text)
{
    SwingUtilities.invokeLater(new Runnable()
    {
        @Override
        public void run()
        {
            resultTextfield.setText(text);
        }
    });
}
```

Listing 9.8 *Ausführbar als* **'AWTEDTEXAMPLE2'**

9.2.3 Die Klasse `SwingWorker`

Beim Implementieren der vorangegangenen Beispiele fällt bereits auf, dass es im Kontext von Swing-Applikationen aufwendig und fehleranfällig sein kann, länger dauernde Berechnungen unabhängig in eigenen Threads auszuführen. Insbesondere gilt dies, wenn man diese nicht nur separat starten, sondern auch abbrechen können möchte und deren (Zwischen-)Ergebnisse wieder korrekt ins GUI propagiert werden sollen. Seit Java 6 wird dies durch die Klasse `SwingWorker<T, V>` erleichtert, wobei die Typangabe `T` der des Endergebnisses entspricht und der Typ `V` dem Typ von Zwischenergebnissen, wodurch Zwischenergebnisse bei Bedarf einen anderen Typ als das Endergebnis besitzen können.

Der Einsatz von `SwingWorker<T, V>` reduziert die Komplexität innerhalb der

Anwendung selbst. Die zuvor aufwendige Abstimmung mit dem EDT wird enorm vereinfacht. Innerhalb der Applikation muss weniger Aufwand bezüglich Thread-Sicherheit betrieben werden, sodass diese eher am zu lösenden Problem ausgerichtet programmiert werden kann, wie das der folgende Abschnitt zeigt.

Wichtige Methoden der Klasse `SwingWorker`

Die Aufgabe wird durch Ableitung von der Klasse `SwingWorker<T, V>` beschrieben. Dort ist zumindest die abstrakte Methode `doInBackground()`, die die eigentliche Berechnung durchführt und ein Ergebnis vom Typ `T` liefert, zu implementieren. Die folgende Aufzählung nennt weitere wichtige Methoden:

- `publish(V)` – Für ein visuelles Feedback kann man Zwischenergebnisse durch Aufruf der Methode `publish(V)` aus der Verarbeitung in `doInBackground()` bekannt geben. Dies ist beispielsweise zur Darstellung eines Fortschrittsbalkens oder eines Fortschrittsdialogs mit Zwischenresultaten denkbar.
- `process()` – Ein Aufruf der Methode `publish()` sorgt dafür, dass ein Aufruf der Callback-Methode `process()` in die `EventQueue` eingetragen und somit Thread-sicher im EDT ausgeführt wird. Man kann daher ohne Probleme auf Swing-Komponenten und deren Datenmodelle lesend und schreibend zugreifen.
- `void done()` – Diese Methode wird automatisch aus dem EDT aufgerufen, wenn die Abarbeitung in `doInBackground()` abgeschlossen ist.

Um die Bearbeitung einer Aufgabe parallel zum EDT zu starten, bietet die Klasse `SwingWorker<T, V>` die Methode `execute()`. Besonders erwähnenswert ist, dass die Verarbeitung durch Aufruf der Methode `cancel(boolean)` auch auf einfache Weise abgebrochen werden kann. Dabei spezifiziert der boolesche Parameter, ob ein Abbruch durch `Thread.interrupt()` forciert werden soll oder nur durch eine Änderung des internen Status erfolgt. Beides basiert auf der Funktionalität der in Abschnitt 7.6 vorgestellten Concurrency Utilities: Die Klasse `SwingWorker<T, V>` implementiert das `Future<V>`-Interface. Daher ist es möglich, Ergebnisse zwischen aufrufendem und ausführendem Thread auszutauschen. Man sollte allerdings darauf achten, dass ein Aufrufer der Methode `get()` so lange blockiert, bis die Berechnung in `doInBackground()` abgeschlossen ist.

Im folgenden Listing nutzen wir die obigen Methoden, um die zuvor per Hand mit `invokeLater(Runnable)` realisierte Ausführung einer länger laufenden Aktion zu verbessern. Die Applikation wird außerdem um einen Stop-Button erweitert, um die Verarbeitung zu beliebigen Zeitpunkten abbrechen zu können. Das ist insbesondere für zeitaufwendige Berechnungen sehr nützlich.

```
public static final class LongRunningMenuAction extends AbstractAction
{
    public LongRunningMenuAction(final String resourceKey)
    {
        super(resourceKey);
```

```
    }

    public void actionPerformed(final ActionEvent e)
    {
        // Start eines SwingWorkers
        currentWorker = new DemoSwingWorker();
        currentWorker.execute();

        stopBtn.setEnabled(true);
        startAction.setEnabled(false);
    }
}

public static class DemoSwingWorker extends SwingWorker<Integer, String>
{
    // doInBackground() wird in einem eigenen Thread ausgeführt.
    // Hier dürfen keine Manipulationen an Swing-Komponenten stattfinden!
    @Override
    protected Integer doInBackground() throws Exception
    {
        publish("Computation started");

        for (int i = 1; i < 4; i++)
        {
            // Abbruch mehrfach testen, um schnell zu reagieren
            if (isCancelled())
                break;

            // Verzögerung (stellvertretend für aufwendige Berechnung)
            SleepUtils.safeSleep(1000);

            // Abbruch mehrfach testen, um schnell zu reagieren
            if (isCancelled())
                break;

            // Zwischenergebnis ausgeben
            publish("Computation " + i);
       }

        // Ergebnis durch Aufruf von get() in der done()-Methode abfragbar
        return 4711;
    }

    // process() erlaubt es, Zwischenergebnisse darzustellen.
    // Aufruf im EDT: => Manipulationen an GUI-Elementen sicher möglich
    @Override
    protected void process(final List<String> info)
    {
        resultTextfield.setText(info.get(0));
    }

    // Worker wurde beendet. done() wird innerhalb des EDTs aufgerufen.
    // Manipulationen an GUI-Elementen sicher möglich
    @Override
    protected void done()
    {
        try
        {
            // Ergebnis aus doInBackground() ermitteln
            final Integer value = get();
            publish("Computation finished: result = " + value);
```

```
        }
        catch (final ExecutionException e)
        {
            throw new RuntimeException(e);
        }
        // JDK 7: Multi Catch
        catch (final InterruptedException | CancellationException e)
        {
            publish("Canceled");
        }

        startAction.setEnabled(true);
        stopBtn.setEnabled(false);
    }
}
```

Listing 9.9 *Ausführbar als* **'SWINGWORKERDEMO'**

Führt man das Programm SWINGWORKERDEMO aus, so wird klar, dass alle in den vorherigen Abschnitten negativ angemerkten Punkte durch den Einsatz der Klasse `SwingWorker<T, V>` adressiert werden. Abbildung 9-19 verdeutlicht das. Dort sieht man zum einen die Ausgabe eines Zwischenergebnisses sowie einen `JButton` zum Stoppen der Abarbeitung.

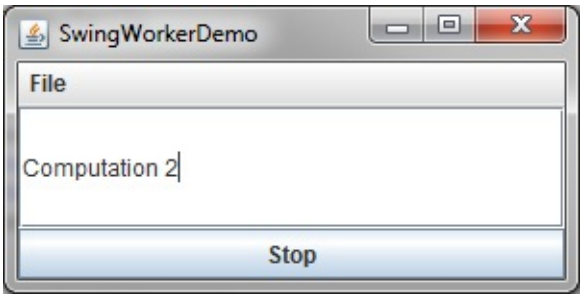


Abbildung 9-19 *Screenshot des Programms* SWINGWORKERDEMO

Die Verarbeitung kann durch Aufruf der Methode `cancel(boolean)` abgebrochen werden. Dazu wird folgender `ActionListener` für den `JButton` registriert:

```
stopBtn.addActionListener(new ActionListener()
{
    public void actionPerformed(final ActionEvent actionEvent)
    {
        startAction.setEnabled(true);
        stopBtn.setEnabled(false);

        currentWorker.cancel(false);
    }
});
```

Fazit

In diesem Abschnitt habe ich zunächst verschiedene Varianten der Ausführung von Aktionen in und parallel zum EDT vorgestellt. Dabei blieb das wichtige Thema des Abbruchs einer laufenden Aktion außen vor, da dies aufwendig von Hand selbst zu realisieren ist. Das und die asynchrone Abarbeitung von Aufgaben wird durch den Einsatz der Klasse `SwingWorker<T, V>` erleichtert, deren Arbeitsweise einführend im letzten Beispiel vorgestellt wurde. Es bietet sich nahezu immer an, einen

`SwingWorker<T,V>` einzusetzen. Ein ausführlicheres Tutorial finden Sie online unter
`http://www.oracle.com/technetwork/articles/javase/swingworker137249.html`.

9.3 Zeichnen in GUI-Komponenten

Bisher haben wir Bedienelemente eingesetzt, um eine klassische grafische Benutzeroberfläche zu realisieren. Für den Fall, dass Bedienelemente anders gezeichnet oder um Funktionalität erweitert werden sollen, stelle ich Ihnen in Abschnitt 9.3.1 einige Grundlagen zum Zeichnen in Bedienelementen vor. Die Möglichkeiten zur Veredelung der grafischen Darstellung sind damit aber noch lange nicht erschöpft: Die Java-2D-Bibliothek erlaubt es, vielfältige Zeichenfunktionen und diverse grafische Effekte wie z. B. Transparenz zu nutzen. Zum Einstieg bietet Abschnitt 9.3.2 eine Kurzeinführung in Java 2D. Mit dem erworbenen Wissen erstellen wir in Abschnitt 9.3.3 eine Tachometeranzeige.

9.3.1 Generelles zum Zeichnen in GUI-Komponenten

In diesem Abschnitt widmen wir uns dem Zeichnen in Swing-GUI-Komponenten. Dazu lernen wir zunächst etwas über den prinzipiellen Ablauf beim Zeichnen und über das Koordinatensystem der GUI-Komponenten.

Grundsätzliches zum Zeichnen und Neuzeichnen

Wie schon erwähnt, zeichnet die Swing-Bibliothek ihre GUI-Komponenten selbst und nutzt nicht die vom Betriebssystem bereitgestellten Bedienelemente, wie dies in AWT der Fall ist. Wenn in Swing ein Fenster oder ein Bedienelement gezeichnet werden soll, so wird durch die Implementierung der Methode `paint(java.awt.Graphics)` festgelegt, was und wie dies genau geschehen soll. Diese Methode wird durch Swing bei Bedarf automatisch aufgerufen. Initial ist das der Fall, wenn die GUI-Komponente zum ersten Mal sichtbar wird. Außerdem wird das Neuzeichnen angestoßen, wenn es eine Zustandsänderung im Modell des Bedienelements gab oder das Bedienelement durch andere Fenster überdeckt und anschließend wieder sichtbar wurde. Ein sogenannter Repaint-Manager stellt dann fest, welche Bereiche welcher Bedienelemente als Folge neu dargestellt werden müssen. Das geschieht allerdings nicht durch einen direkten Aufruf der Methode `paint(Graphics)`, sondern vielmehr erhalten alle betroffenen Bedienelemente eine indirekte Aufforderung, sich neu zu zeichnen: Es wird ein Event zum Neuzeichnen erzeugt und in die AWT-Event-Queue eingestellt. Erst bei der späteren Abarbeitung dieses Events wird dann die Methode `paint(Graphics)` aufgerufen, wo dann die Zeichenoperationen tatsächlich stattfinden.

Tipp: Typ des Übergabeparameters der Methode `paint(Graphics)`

Der Methodenparameter der Methode `paint(Graphics)` ist laut Signatur vom Typ `Graphics`. Tatsächlich wird aber seit JDK 1.2 immer ein spezielleres `Graphics2D`-Objekt übergeben. Dieses Wissen werden wir später in den Beispielprogrammen nutzen, um auch ohne eine vorherige explizite Typprüfung einen Down Cast (vgl. Abschnitt 3.1.1) auf den spezielleren Typ `Graphics2D` vorzunehmen.

Ablauf beim Zeichnen

Das Zeichnen einer GUI-Komponente ist ein dreistufiger Prozess. In der Methode `paint(Graphics)` werden nacheinander folgende Methoden aufgerufen: `paintComponent(Graphics)`, `paintBorder(Graphics)` und `paintChildren(Graphics)`. Demnach wird zunächst die GUI-Komponente selbst gezeichnet, anschließend ihre Umrahmung und abschließend alle möglicherweise enthaltenen Subkomponenten.

Tipp: Besonderheiten der Methode `paint(Graphics)`

Wenn man die Methode `paint(Graphics)` statt der Methode `paintComponent(Graphics)` überschreibt, so muss man selbst dafür sorgen, dass die Funktionalitäten der Basisklasse ausgeführt werden – sonst erfolgt weder das Zeichnen des Rahmens per `paintBorder(Graphics)` noch das von Subkomponenten mit `paintChildren(Graphics)`. Neben den Aufrufen der zuvor aufgezählten Methoden sorgt die Implementierung der Methode `paint(Graphics)` der Klasse `JComponent` auch für Double Buffering und dafür, dass das `Graphics`-Objekt bereits für den jeweiligen Anwendungsfall korrekt konfiguriert wurde. Diese Aufgabe sollte man dem Swing-System überlassen. Anders formuliert: Um die korrekte Funktion des Neuzeichnens zu gewährleisten, sollte man die `paint(Graphics)`-Methode möglichst nicht durch Überschreiben verändern und auch nicht selbst aufrufen.

Eingriffe beim Zeichnen

Möchte man Einfluss darauf nehmen, wie und was im Bereich eines Bedienelements gezeichnet wird, kann man dazu die Methode `paintComponent(Graphics)` überschreiben. Häufig sollen zunächst die Zeichenoperationen der Basisklasse durch Aufruf von `super.paintComponent(Graphics)` ausgeführt werden, wodurch die GUI-Komponente an sich dargestellt wird, z. B. die Schaltfläche und der Text eines `JButtons`. Anschließend können eigene Zeichenoperationen ausgeführt werden.

Folgendes Listing verdeutlicht das: Zusätzlich zu dem eigentlichen Bedienelement

JButton wird ein blaues Rechteck der Größe 50 x 50 Pixel gezeichnet:

```
final JButton button = new JButton( "Button mit blauem Rechteck")
{
    @Override
    public void paintComponent( final Graphics graphics)
    {
        super. paintComponent( graphics);

        graphics. setColor( Color. BLUE);
        graphics. fillRect( 0, 0, 50, 50);
    }
};
```

Listing 9.10 *Ausführbar als* **'JBUTTONRECTDRAWINGEXAMPLE'**

In der Regel möchte man komplexere Zeichenoperationen ausführen und dabei auch die Ausmaße des Bedienelements beachten. Das wurde eben vollständig vernachlässigt, wodurch das Quadrat abgeschnitten und als Rechteck dargestellt wird, wie dies Abbildung 9-20 zeigt.

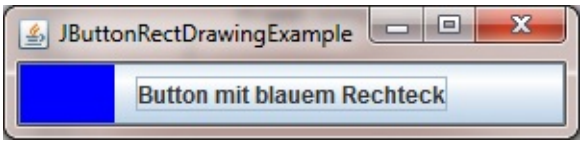


Abbildung 9-20 *Screenshot des Programms* JBUTTONRECTDRAWINGEXAMPLE

Abmessungen, Koordinatensystem und Grafikkontext

Betrachten wir nun das Koordinatensystem und die Abmessungen von Bedienelementen. Jede GUI-Komponente besitzt dazu ihr eigenes pixelbasiertes Koordinatensystem, das bei Position x=0, y=0 links oben startet. Die Größe der GUI-Komponente kann man mit den Methoden `getWidth()` und `getHeight()` erfragen. Im Gegensatz zur mathematischen Ausrichtung verläuft das Koordinatensystem auf der y-Achse nach unten: Demnach reicht das Bedienelement in seiner Ausdehnung bis zur Position mit der Koordinate x=`getWidth()-1`, y=`getHeight()-1` rechts unten.

Außerdem ist für das Verständnis wichtig zu wissen, dass die Klasse `Graphics` einen sogenannten ***Grafikkontext*** repräsentiert. Darunter versteht man eine Abstraktion eines realen Ausgabegeräts bzw. einer Zeichenfläche. Dort sind Zeichenfarben, Schriftarten und weitere – später für Java 2D beschriebene – Eigenschaften wie Liniendicken, Füllmuster und Transparenz einstellbar.

Hier schauen wir zunächst auf die Funktionalitäten der Klasse `Graphics`. Diese bietet verschiedene Methoden zum Zeichnen bzw. Füllen von grafischen Figuren wie Linien, Rechtecken und Kreisen sowie zur Ausgabe von Text und Bildern. Die Figuren werden immer mit einer Liniendicke von 1 Pixel in der zuvor durch `setColor(java.awt.Color)` festgelegten Farbe (bzw. ohne einen solchen Aufruf in der Standardfarbe Schwarz) dargestellt. Unter anderem werden folgende Methoden durch die Klasse `Graphics` bereitgestellt:

- `drawLine(int x1, int y1, int x2, int y2)` – Zeichnet eine Linie zwischen den beiden angegebenen Punkten.
- `drawRect(int x, int y, int width, int height)` – Zeichnet ein Rechteck ab Position (x, y) mit der angegebenen Breite und Höhe.
- `drawString(String text, int x, int y)` – Gibt die angegebene textuelle Information startend an der Position (x, y) aus, wobei die y-Koordinate die Grundlinie der Schriftart bestimmt. Das ist eine gedachte Hilfslinie, entlang der die Buchstaben gezeichnet werden. Im Zusammenhang mit den Abmessungen einer Schriftart gehe ich darauf später detaillierter ein. Wichtig ist hier nur, dass die y-Koordinate nicht die obere Kante der Textausgabe bestimmt.

Neben diesen Zeichenmethoden gibt es korrespondierende Methoden zum Füllen von Figuren, z. B. `fillRect(int x, int y, int width, int height)`. Natürlich gilt dies für Linien und Texte nicht.

An den Methodensignaturen erkennt man, dass die Zeichenoperationen, die in der Klasse `Graphics` angeboten werden, eher funktional ausgerichtet sind. Die zu zeichnenden Figuren, etwa ein Rechteck, werden durch `int`-Werte und nicht z. B. durch ein Rechteck vom Typ `java.awt.Rectangle` beschrieben. Das gilt ebenso für das Zeichnen von Linien, Ellipsen usw. Die später vorgestellte Klasse `Graphics2D`, die die Funktionalität von Java 2D bereitstellt, arbeitet eher objektorientiert. Dort existieren zwei Methoden `draw(Shape)` und `fill(Shape)`, die beliebige Figuren malen bzw. füllen, die dem Figuren-Interface vom Typ `java.awt.Shape` genügen.

Im nachfolgenden Listing nutzen wir das gewonnene Wissen zu Koordinatensystem und Abmessungen, um einen blauen Rahmen mithilfe von fünf einzelnen Rechtecken zu zeichnen. Wir definieren eine anonyme Klasse, basierend auf der Klasse `JLabel`, und überschreiben die Methode `paintComponent(Graphics)` wie folgt:

```
final JLabel borderedLabel = new JLabel("Dies ist ein Text mit blauem Rand")
{
    @Override
    public void paintComponent(final Graphics graphics)
    {
        super.paintComponent(graphics);

        // 5 Rechtecke mit 1 Pixel Breite zeichnen
        graphics.setColor(Color.BLUE);
        for (int i = 0; i < 5; i++)
        {
            final int width = getWidth() - 1 - i * 2;
            final int height = getHeight() - 1 - i * 2;

            graphics.drawRect(i, i, width, height);
        }
    }
};
```

Listing 9.11 *Ausführbar als* **'JLABELDRAWINGEXAMPLE'**

Führt man das Programm JLABELDRAWINGEXAMPLE aus, so erhält man ein Fenster ähnlich zu dem aus Abbildung 9-21.

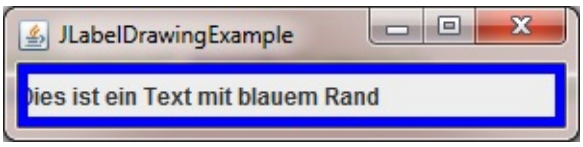


Abbildung 9-21 *Screenshot des Programms* JLABELDRAWINGEXAMPLE

Diese Realisierung einer farbigen Umrahmung für das `JLabel` ist offensichtlich nicht besonders geschickt, denn mit zunehmender Dicke wird immer mehr Text übermalt, wie man dies ansatzweise bereits in Abbildung 9-21 erkennt. Wie geht es besser?

Rahmen und das Interface `Border`

Bei der Beschreibung der Abläufe beim Zeichnen eines Bedienelements wurde die Methode `paintBorder(Graphics)` erwähnt, die – sofern gewünscht – einen Rahmen zeichnet. Dazu kann jeder GUI-Komponente ein Rahmen vom Typ `javax.swing.border.Border` zugeordnet werden. Für dieses Interface existieren verschiedene Implementierungen. Die nachfolgend genannten stammen aus dem Package `javax.swing.border`. Ein `TitledBorder` dient beispielsweise zur Darstellung eines rechteckigen Rahmens mit Titel. Ein `LineBorder` zeichnet eine Umrahmung beliebiger Liniendicke und Farbe. Das Besondere an den Rahmen ist, dass sich diese durch die Klasse `CompoundBorder` miteinander kombinieren lassen (vgl. Abschnitt 18.2.4 zum KOMPOSITUM-Muster). Damit ist es möglich, auch komplexere Rahmen einheitlich und auf einfache Weise anzusprechen.

Folgendes Listing zeigt die Kombination zweier Rahmen der eben beschriebenen Typen durch einen `CompoundBorder` und dessen Zuweisung an ein `JLabel`:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("JLabelWithCompoundBorderExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final JLabel label = new JLabel("Text mit zwei Rändern vom Typ Border");

    // Verschiedene Umrahmungen erzeugen
    final Border blueBorder = BorderFactory.createLineBorder(Color.BLUE, 5);
    final Border titledBorder = BorderFactory.createTitledBorder("Title");
    final Border compoundBorder = BorderFactory.createCompoundBorder(
                                                titledBorder, blueBorder);

    // Umrahmungen mit dem JLabel verknüpfen
    label.setBorder(compoundBorder);

    frame.add(label);
    frame.setSize(350, 90);
    frame.setVisible(true);
}
```

Listing 9.12 *Ausführbar als* **'JLABELWITHCOMPOUNDBORDEREXAMPLE'**

Führt man das Programm JLABELWITHCOMPOUNDBORDEREXAMPLE aus, so erhält man ein Fenster ähnlich zu dem aus Abbildung 9-22.

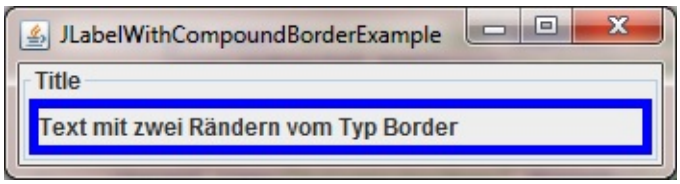


Abbildung 9-22 *Ausgabe des Programms* JLABELWITHCOMPOUNDBORDEREXAMPLE

Der Vorteil beim Einsatz der Rahmen vom Typ `Border` ist, dass die Größenberechnungen und das Zeichnen der GUI-Komponente automatisch so durchgeführt werden, dass für das `JLabel` als Folge der Text ohne Überdeckung dargestellt wird.

Anhand dieses Beispiels wird klar, dass man beim Zeichnen in GUI-Komponenten nicht einfach überall innerhalb der gesamten Fläche zeichnen sollte, sondern gegebenenfalls die Abmessungen von Rahmen zu beachten sind, die einer GUI-Komponente als `Border`-Instanzen zugeordnet sind. Die Abmessungen der Umrahmung selbst bestimmen zu müssen, wäre unpraktisch. Es geht glücklicherweise einfacher.

Details zum Zeichnen in GUI-Komponenten

Die inneren Abmessungen einer GUI-Komponente werden durch `java.awt.Insets`-Objekte beschrieben. Diese erhält man durch Aufruf der Methode `getInsets()`. Die dort hinterlegten Werte muss man als Offset nutzen, um den Bereich zu ermitteln, in dem Zeichenoperationen (ohne Einfluss auf den Rand) ausgeführt werden können. Weil das etwas abstrakt klingt, erweitern wir das obige Beispiel. Die Anweisung zur Erzeugung des `JLabels` ersetzen wir mit folgenden Programmzeilen:

```
final JLabel label = new JLabel("Dieser Text wird überschrieben!")
{
    @Override
    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);

        final Insets insets = getInsets();
        final Dimension dimension = getSize();

        setText("<HTML><ul><li>Size:  " + dimension +
                          "<li>Inset:  " + insets + "</ul></HTML>");

        g.setColor(Color.RED);
        g.drawRect(insets.left, insets.top,
                   dimension.width-insets.left-insets.right-1,
                   dimension.height-insets.top-insets.bottom-1);
    }
};
```

Listing 9.13 *Ausführbar als* **'DIMENSIONANDINSETSEXAMPLE'**

Mithilfe der Methoden `getInsets()` und `getSize()` ermitteln wir einige Größenangaben, die wir für die Beschriftung des `JLabel`s nutzen. Den Inhalt setzen wir mit `setText(String)`. Ganz nebenbei sieht man, dass es in Swing-Komponenten problemlos möglich ist, HTML auszugeben. Hier wird aus den ermittelten Informationen eine ungeordnete Liste (`ul`) mit zwei Einträgen (`li`) aufgebaut. Außerdem wird ein rotes Rechteck im inneren Bereich gemalt, wie es Abbildung 9-23 zeigt.

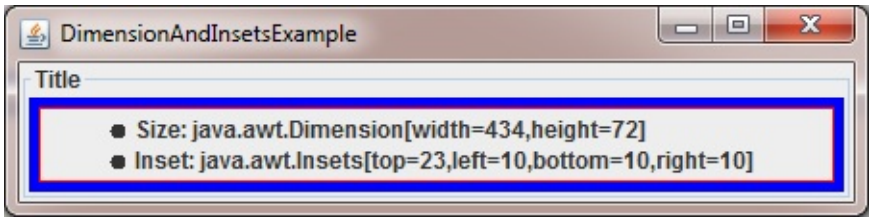


Abbildung 9-23 *Ausgabe des Programms* DIMENSIONANDINSETSEXAMPLE

Die Klasse `FontMetrics`

Die Klasse `java.awt.FontMetrics` spielt eine wichtige Rolle bei der Ausgabe von Text, weil man damit die Breite eines Zeichens bzw. einer Zeichenkette und auch die Höhe der Schriftart bestimmen kann, sowie unter anderem zu Ober- und Unterlängen (Ascent und Descent). Abbildung 9-24 stellt das schematisch dar. Die dort gezeigten Abmessungen aus einem `FontMetrics`-Objekt beziehen sich immer auf eine Schriftart, die in Swing durch die Klasse `java.awt.Font` repräsentiert wird. Ein korrespondierendes `FontMetrics`-Objekt erhält man durch Aufruf der Methode `getFontMetrics(Font)` einer GUI-Komponente.

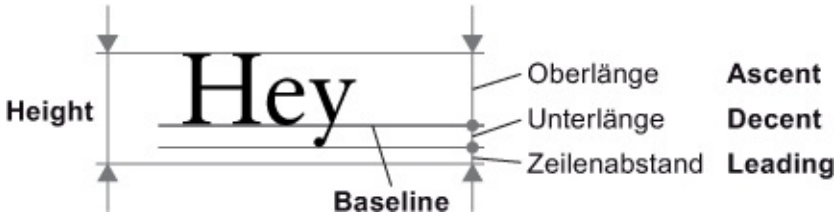


Abbildung 9-24 *Abmessungen aus einem* `FontMetrics`*-Objekt*

Um den Einsatz eines `FontMetrics`-Objekts kennenzulernen, erstellen wir eine anonyme Bedienelementklasse, die wir von der Klasse `JComponent` ableiten. Für diese überschreiben wir die Methode `paintComponent(Graphics)`. Dort erzeugen wir ein `Font`-Objekt für die Schriftart Arial in fett mit der Größe 96 pt. Diese Größe wähle ich hier, weil sich daran die einzelnen Abmessungen der Schriftart besser visualisieren lassen. Besonders erwähnenswert sind zwei Dinge: Zum einen ist bei einer Textausgabe immer die sogenannte Basislinie (oder Grundlinie) zu beachten. Eine Textausgabe per `drawString(String, int, int)` zeichnet auf dieser Grundlinie. Dies sollte man unbedingt für die Angabe der y-Koordinate beachten, ansonsten wird der Text fälschlicherweise nach oben versetzt gezeichnet.

Eine weitere Kennlinie eines Zeichensatzes ist dessen Mittellinie, also die Linie, die man etwa zum Durchstreichen eines Worts nutzen würde. Die Bestimmung von deren y-Position ist nicht ganz so leicht, wie es scheinen mag. Es gibt nämlich

verschiedene (Definitions-)Möglichkeiten, basierend auf unterschiedlichen Informationen aus einem `FontMetrics`-Objekt. Einige davon stelle ich im folgenden Listing vor. Grafisch sind die Varianten der Mittellinie in Abbildung 9-25 dargestellt.

```
final JComponent fontMetricsComponent = new JComponent()
{
   @Override
   public void paintComponent( final Graphics graphics)
   {
      super.paintComponent( graphics);

      // Schriftart Arial, Bold, 96 pt wählen
      final Font font = new Font( "Arial", Font.BOLD, 96);
      graphics.setFont( font);

      // Textausgabe an Position x=20, y=96
      final int textStartXPos = 20;

      final int baseLinePos = 96;
      final String demotext = "Test QÄqä";
      graphics.drawString( demotext, textStartXPos, baseLinePos);

      // Ermitteln der Werte zum Zeichnen der Begrenzungslinien
      final FontMetrics fontMetrics = graphics.getFontMetrics();
      final int ascent = fontMetrics.getAscent();
      final int descent = fontMetrics.getDescent();
      final int leading = fontMetrics.getLeading();

      // Basislinie/Grundlinie
      graphics.drawLine( 10, baseLinePos, 550, baseLinePos);

      // Obere + untere Begrenzungslinie: Ascent und Descent
      graphics.setColor( Color.GRAY);
      graphics.drawLine( 10, baseLinePos - ascent, 550, baseLinePos - ascent);
      graphics.drawLine( 10, baseLinePos + descent, 550, baseLinePos +
descent);

      // Zeilenabstand
      graphics.drawLine( 10, baseLinePos + descent + leading,
                         550, baseLinePos + descent + leading);

      // Mittellinie basierend auf Ascent
      graphics.setColor( Color.BLUE);
      graphics.drawLine( 10, baseLinePos - ascent/2,
                         550, baseLinePos - ascent/2);

      // Vermutete Mittellinie
      graphics.setColor( Color.CYAN);
      graphics.drawLine( 10, baseLinePos - ascent/3,
                         550, baseLinePos - ascent/3);

      // Mittellinie basierend auf Ascent und Descent
      graphics.setColor( Color.GREEN);
      graphics.drawLine( 10, baseLinePos + descent - ( ascent + descent)/2,
                         550, baseLinePos + descent - ( ascent + descent)/2);

      // Bounding Box des gesamten Textes
      graphics.setColor( Color.RED);
      final Rectangle2D rect2D = fontMetrics.getStringBounds( demotext,
```

```
                                                              graphics);
        final Rectangle rect = rect2D.getBounds();
        graphics.drawRect(rect.x + textStartXPos, rect.y + baseLinePos,
                          rect.width, rect.height);

        // X-Offset für 3. Zeichen berechnen
        int startPosX = 0;
        for (int i = 0; i< 3; i++)
           startPosX += fontMetrics.charWidth(demotext.charAt(i));

        // Bounding Box für Zeichen 3 bis 7
        final Rectangle2D innerRect2D = fontMetrics.getStringBounds(demotext,
                                                                    3, 7,
                                                                    graphics);
        final Rectangle innerRect = innerRect2D.getBounds();
        graphics.setColor(Color.ORANGE);
        graphics.drawRect(innerRect.x + textStartXPos + startPosX,
                          innerRect.y + baseLinePos,
                          innerRect.width, innerRect.height);
    }
};
```

Listing 9.14 *Ausführbar als* **'FONTMETRICSEXAMPLE'**

Starten Sie das Programm FONTMETRICSEXAMPLE, so erhalten Sie eine Ausgabe ähnlich zu der in Abbildung 9-25. Dort sind verschiedene Begrenzungslinien zu sehen, die sich aus den Werten des `FontMetrics`-Objekts ergeben, etwa die grauen Begrenzungslinien für Ober- und Unterlänge. Außerdem sind die Begrenzungen des gesamten Textes (rotes Rechteck) sowie des Textausschnitts, gebildet aus den Buchstaben 3 bis 7 (oranges Rechteck) eingezeichnet.[5] Zur Berechnung der Startposition nutzen wir die Methode `charWidth(char)` und addieren die Werte aller Zeichen bis zur gewünschten Position auf. Um die farbigen Markierungen zu sehen, starten Sie bitte das obige Progamm.

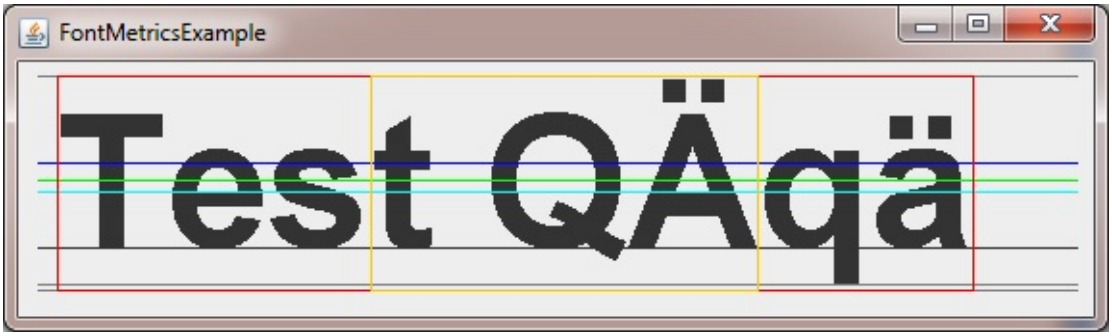


Abbildung 9-25 *Visualisierung verschiedener Werte aus einem* `FontMetrics`*-Objekt*

9.3.2 Kurzeinführung in Java 2D

Java 2D ist eine Sammlung von Klassen zur Darstellung zweidimensionaler Text- und Bildinformationen. Bekanntermaßen bietet das AWT mit der Klasse `Graphics` nur recht primitive Zeichenfunktionen für eine Menge an vordefinierten Figuren (Rechteck, Oval usw.). Es stehen Methoden wie etwa `drawRect(int, int, int, int)` zur Verfügung. Anhand dieser Methodensignatur erkennt man schon die recht funktionale Ausrichtung. Im Gegensatz dazu arbeitet Java 2D und der

zugehörige Grafikkontext in Form der Klasse `Graphics2D` eher objektorientiert. Hier wird anstelle einer Menge von Punkten mit dem Konzept der Figur gearbeitet. Daher existieren in der Klasse `Graphics2D` statt mehrerer spezifischer Methoden für jede Figur jeweils nur eine Methode zum Zeichnen und zum Füllen von Figuren: `draw(Shape)` und `fill(Shape)`. Figurenklassen müssen dazu das Interface `java.awt.Shape` implementieren. Das gilt z. B. für die Klassen `Ellipse2D` oder `Rectangle2D` aus dem Package `java.awt.geom`.[6] In den Figurenklassen sind jeweils zwei innere Klassen `Float` und `Double` definiert, die ihre Koordinaten im Typ gleichen Namens speichern. Dadurch wird ersichtlich, dass bei Java 2D kein pixelbasiertes, sondern ein logisches Koordinatensystem zum Einsatz kommt. Aufgrund dessen lassen sich Figuren leichter verändern, etwa drehen, strecken und dehnen. Des Weiteren können bei den Zeichenoperationen unterschiedliche Liniendicken und -stile, Füllmuster, Transparenzeffekte usw. genutzt werden.

Zur Demonstration einiger der genannten Funktionalitäten wollen wir einen Text mit der Schriftart Arial der Größe 28 pt in Fettschrift ausgeben und einen Kreis darum malen. Außerdem wird der Kreis mit einer 7 Pixel dicken, unterschiedlich gestrichelten Linie gezeichnet und nutzt dabei einen zyklischen Farbverlauf von Blauviolet nach goldenem Rot. Das Ganze lässt sich als Programm FIRSTJAVA2DEXAMPLE starten. In Abbildung 9-26 ist ein Screenshot dargestellt.

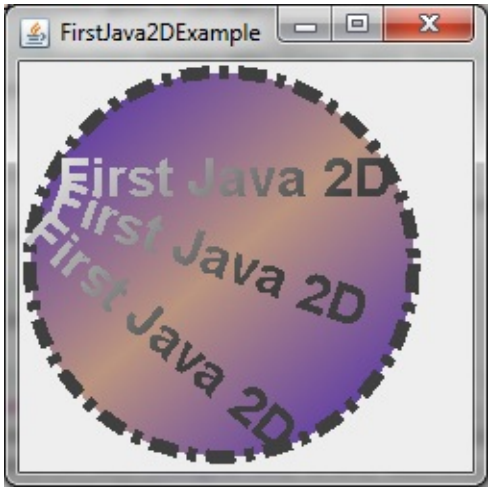


Abbildung 9-26 *Erstes Java-2D-Beispiel*

Figuren mit der Klasse `Path2D` erstellen

Bevor wir Java 2D für das gezeigte Beispiel im Einsatz erleben, möchte ich hier noch auf eine Besonderheit eingehen. Man kann ähnlich zum Zeichnen auf einem Blatt Papier mit Karos dies auch mit Java 2D tun. Schauen wir dazu Abbildung 9-27 an.

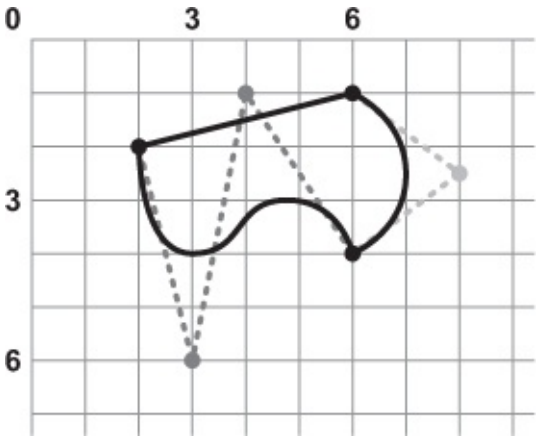


Abbildung 9-27 *Skizze einer Figur auf Karo-Papier*

Derartige Zeichnungen sind mithilfe der Klasse `java.awt.geom.Path2D` leicht nachzubilden. Dazu bietet diese verschiedene Methoden, unter anderem um den Zeichenstift auf eine spezielle Position mit `moveTo(float, float)` zu bewegen und eine Linie von der aktuellen zur angegebenen Position mit `lineTo(float, float)` zu malen. Außerdem können noch spezielle Kurven, sogenannte ***Bézierkurven***, in zweierlei Ausprägungen (mit einem und mit zwei Stützpunkten) mit `quadTo(float, float, float, float)` bzw. `curveTo(float, float, float, float, float, float)` gezeichnet werden.[7] In Abbildung 9-27 habe ich die Stützpunkte hellgrau bzw. mittelgrau markiert und mit den Ausgangspunkten des Kurventeils gestrichelt verbunden. Diese Darstellung erleichtert das Verständnis für die Arbeitsweise des Kurvenzeichnens.

Folgendes Listing nutzt die Klasse `java.awt.geom.GeneralPath`, die eine Spezialisierung der Klasse `Path2D.Float` ist, die wiederum das Interface `Shape` erfüllt und sich somit leicht durch Aufruf der Methode `draw(Shape)` zeichnen lässt:

```
public Path2D.Float createDemoPath()
{
    final Path2D.Float demoPath = new GeneralPath();

    demoPath.moveTo(2f, 2f);
    demoPath.lineTo(6f, 1f);
    demoPath.quadTo(8f, 2.5f, 6f, 4f);
    demoPath.curveTo(4f, 1f, 3f, 6f, 2f, 2f);

    return demoPath;
}
```

Würden wir diese Programmzeilen ausführen, dann würde nur ein kleiner Pixelhaufen entstehen, da zunächst von Pixelangaben ausgegangen wird. Jetzt kommen wir aber zu einer der großen Stärken von Java 2D. Durch die Beschreibung von Figuren in logischen Einheiten können wir auf einfache Weise eine Skalierung erreichen. Betrachten wir zunächst das Resultat in Abbildung 9-28.

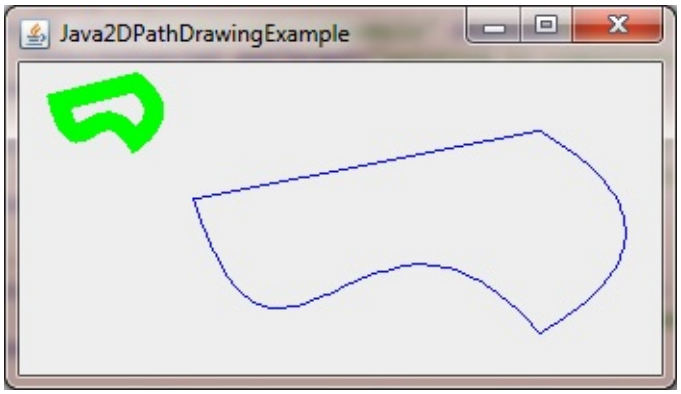


Abbildung 9-28 *Nachbau der Skizze mit der Klasse* `Path2D.Float`

In der Abbildung sehen wir zwei Varianten der Skalierung: Zum einen wird die gesamte Ausgabe per `scale(int, int)` in der Größe angepasst, was auch Auswirkungen auf die Liniendicke hat. Alternativ kann man auch die Figur an sich skalieren, indem man eine Transformation durch Aufruf der Methode `createTransformedShape(AffineTransform)` nutzt:

```
@Override
public void paintComponent(final Graphics g)
{
    super.paintComponent(g);

    final Graphics2D g2d = (Graphics2D) g;
    final Path2D.Float path = createDemoPath();

    // Nur Figur skalieren, Linie bleibt schmal
    final AffineTransform transform = AffineTransform.getScaleInstance(45,
35);
    final Shape shape = path.createTransformedShape(transform);
    g2d.setColor(Color.BLUE);
    g2d.draw(shape);

    // Koordinatensystem skalieren => Linien werden dicker
    g2d.scale(10, 10);
    g2d.setColor(Color.GREEN);
    g2d.draw(path);
}
```

Listing 9.15 *Ausführbar als* **'JAVA2DPATHDRAWINGEXAMPLE'**

Java 2D am Beispiel

Kommen wir nun zum eingangs gezeigten Beispiel zurück und versuchen uns an dessen Implementierung. Dazu stelle ich zur ersten Orientierung bisher nicht verwendete Funktionalität aus Java 2D in zwei Aufzählungen stichpunktartig vor. Für Einzelheiten verweise ich auf die Onlinedokumentation des JDKs oder auf Spezialliteratur zum Thema Java 2D. Dabei sind insbesondere die Bücher »Java 2D Graphics« von Jonathan Knudsen [55] und »Java 2D API Graphics« von Vincent J. Hardy [37] empfehlenswert. Zweifellos ist die Lektüre der Bücher hilfreich, allerdings ergibt sich vieles auch während der Realisierung des Beispiels. Dabei nutzen wir folgende Interfaces und Klassen:

- `Shape`, `Ellipse2D` – Eine Figur, hier ein Kreis als Spezialfall einer Ellipse
- `Paint`, `GradientPaint` – Eine Füllung oder ein Farbverlauf
- `Stroke`, `BasicStroke` – Ein Linienstil

Außerdem verwenden wir folgende Methoden der Klasse `Graphics2D`:

- `setPaint(Paint)` – Setzen einer Füllung bzw. eines Farbverlaufs
- `setStroke(Stroke)` – Setzen eines Linienstils
- `draw(Shape)` – Malen einer Figur
- `fill(Shape)` – Füllen einer Figur
- `rotate(double)` – Rotation des Koordinatensystems

Wir leiten die Klasse `DrawingComponent` von der Klasse `JComponent` ab und überschreiben die Methode `paintComponent(Graphics)` für die Grafikausgabe. Dort wird zunächst ein Farbverlauf vom Typ `GradientPaint` definiert, ein Kreis mit dem Typ `Ellipse2D` als Figur vom Basistyp `Shape` erstellt und anschließend durch einen Aufruf von `fill(Shape)` gefüllt. Danach definieren wir einen speziellen Linienstil vom Typ `Stroke`, den wir für die Kreisfigur zum Zeichnen der Umrandung per `draw(Shape)` wie folgt verwenden:

```
public static class DrawingComponent extends JComponent
{
    private static final Color violetBlue = new Color(0x5533AA);
    private static final Color goldenRed = new Color(0xBB9580);

    @Override
    public void paintComponent(final Graphics g)
    {
        super.paintComponent(g);

        final Graphics2D g2d = (Graphics2D) g;

        // Kreis als Figur mit Interface Shape definieren
        final Shape shape = new Ellipse2D.Float(5, 5, 200, 200);

        // Diagonaler Farbverlauf von Rot nach Blau
        final Paint paint = new GradientPaint(20, 20, violetBlue,
                                              100, 110, goldenRed, true);
        g2d.setPaint(paint);

        // Figur ausfüllen
        g2d.fill(shape);

        // Zeichenstift mit Dicke 7 (=> new BasicStroke(7))
        final float[] dashes = new float[] { 15f, 5f, 5f, 5f };
        final Stroke stroke = new BasicStroke(7, BasicStroke.CAP_BUTT,
                                                 BasicStroke.JOIN_MITER,
                                                 1.0f, dashes, 0);
        g2d.setStroke(stroke);

        // Figur malen
        g2d.setColor(Color.DARK_GRAY);
        g2d.draw(shape);
```

```
        // Farbverlauf für die Textausgabe
        final Paint paint2 = new GradientPaint(20, 20, Color.WHITE,
                                               120, 120, Color.DARK_GRAY);

        // Gedrehte Textausgabe
        g2d.setFont(new Font("Arial", Font.BOLD, 28));
        for (int i = 0; i < 3; i++)
        {
            g2d.rotate(Math.toRadians(i * 20));
            g2d.setPaint(paint2);
            g2d.drawString("First Java 2D", 20 + i * 20, 70);
            g2d.rotate(-Math.toRadians(i * 20));
        }
    }
}
```

Listing 9.16 *Ausführbar als* **'FIRSTJAVA2DEXAMPLE'**

Zum Schluss erfolgt die Textausgabe, die sich für die Rotation eines Tricks bedient: Statt den Text zu rotieren, tun wir dies mit dem Koordinatensystem und zeichnen dann den Text.

9.3.3 Bedienelemente mit Java 2D selbst erstellen

Wir wollen mit dem erworbenen Wissen zu Java 2D nun ein grafisch etwas komplexeres Bedienelement selbst gestalten: Es soll eine Analoganzeige in Form eines Tachometers entwickelt werden, wie sie in Abbildung 9-29 zu sehen ist.

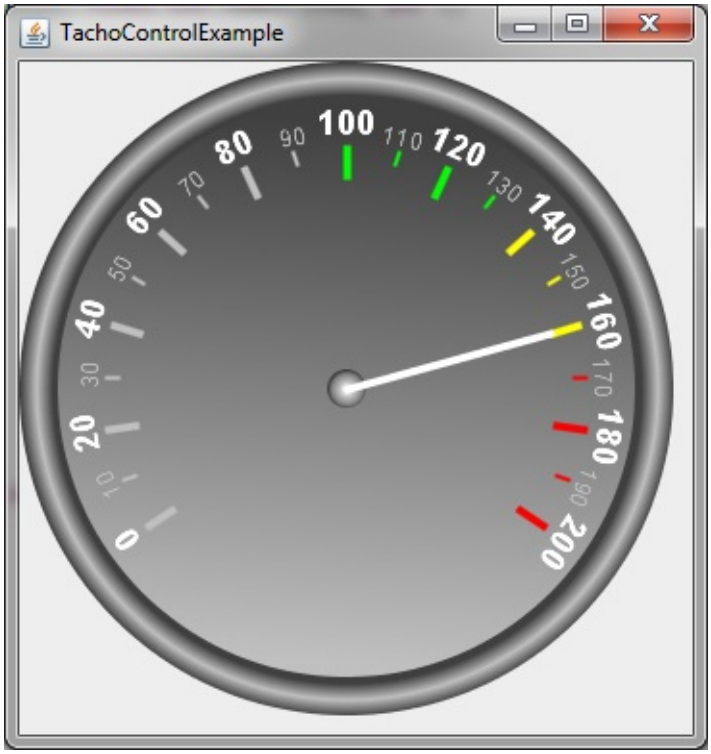


Abbildung 9-29 *Das Bedienelement* `TachoControl`

Die Implementierung dieses Bedienelements ist etwas aufwendiger. Schauen wir uns das Ganze schrittweise an. Beginnen wir mit dem Malen des Hintergrunds.

Hinweis: Analoge Anzeigen

Analoge Anzeigen sind zwar weniger präzise als digitale Anzeigen. Sie werden jedoch immer dann gerne eingesetzt, um auf einen Blick erfassen zu können, in welchem Bereich sich ein Messwert, hier die Geschwindigkeit, befindet. Neben dem aktuellen Stand sind auch Grenzwerte und Relationen leicht ersichtlich.

Schritt 1: Außenring und Hintergrund zeichnen

Der Hintergrund besteht aus zwei Kreisen mit unterschiedlichen Füllungen. Deutlich zu sehen ist der innere Kreis, der einen Farbverlauf von oben Dunkelgrau nach unten Hellgrau enthält. Weniger erkennbar ist, dass die radiale Wölbung des Außenrings auch mithilfe eines Kreises sowie eines `RadialGradientPaints`, dem man mehrere Farben sowie Gewichtungen vorgeben kann, gemalt wird. Vom Mittelpunkt startet das Zeichnen des Außenrings, der zunächst eine schwarze Fläche ausfüllt. Erst in den letzten 10 % des Radius, also ziemlich am Rand, werden die Farbübergänge durch die Gewichtung `{0.90f, 0.95f, 1.0f}` in verschiedenen Grautönen gemalt. Bei dieser Implementierung ist es also wichtig, zuerst den Farbverlauf für den Außenring zu malen und im Anschluss daran den inneren Kreis mit seiner Abstufung.

```
public void drawBackground(final Graphics2D graphics2D, final Point2D center)
{
    // Außenring: Farben sowie Abstände
    final float[] dist = {0.90f, 0.95f, 1.0f};
    final Color[] colors = {Color.DARK_GRAY, Color.LIGHT_GRAY,
Color.DARK_GRAY};

    // Kreisförmiger Gradient
    final RadialGradientPaint paint = new RadialGradientPaint(center,
                                                    outerRadius, dist, colors);

    graphics2D.setPaint(paint);

    // Gezeichnet wird zwar vom Mittelpunkt aus, aber es wird nur im
    // äußeren Bereich der Farbverlauf gezeichnet
    final Ellipse2D outerEllipse = new Ellipse2D.Double(0, 0, outerRadius * 2,
                                                        outerRadius *
2);

    graphics2D.fill(outerEllipse);

    // Hintergrund mit Farbverlauf nach dem Außenring malen
    final GradientPaint gradient = new GradientPaint(
                                        innerStart, innerStart,
Color.DARK_GRAY,
                                        0, innerRadius * 2, Color.LIGHT_GRAY);
    graphics2D.setPaint(gradient);

    final Ellipse2D ellipse = new Ellipse2D.Double(innerStart, innerStart,
                                            innerRadius * 2, innerRadius *
2);
    graphics2D.fill(ellipse);
}
```

Schritt 2: Der Zeiger und seine Verankerung

Zum Zeichnen des Geschwindigkeitszeigers sowie der Befestigungskugel werden die bereits bekannten Klassen `Ellipse2D` und `RadialGradientPaint` genutzt. Der für die Befestigungskugel zu zeichnende Bereich ergibt sich aus dem Mittelpunkt des äußeren Kreises und dem hier gewählten Radius von 10 Pixeln. Außerdem nutzen wir einen vierstufigen Farbverlauf von Weiß nach Dunkelgrau:

```
private void drawAnchor(final Graphics2D graphics2D, final Point2D center)
{
    final float radius = 10;

    // Gewichtung und Farben für den kreisförmigen Gradienten
    final float[] dist = { 0.05f, 0.2f, 0.4f, 0.5f };
    final Color[] colors = { Color.WHITE, Color.LIGHT_GRAY,
                             Color.GRAY, Color.DARK_GRAY };

    // Kreisförmiger Gradient mit vierstufigem Verlauf
    final RadialGradientPaint paint = new RadialGradientPaint(center,
                                                              radius * 2,
                                                              dist, colors);
    graphics2D.setPaint(paint);

    // Befestigungskugel malen
    final Ellipse2D ellipse = new Ellipse2D.Double(center.getX() - radius,
                                                   center.getY() - radius,
                                                   radius * 2, radius * 2);
    graphics2D.fill(ellipse);
}
```

Der Zeiger wird hier durch eine einfache vier Pixel breite weiße Linie dargestellt. Je nach Geschwindigkeit, repräsentiert durch `model.getValue()`, wird die Linie um einen gewissen Winkel gedreht gezeichnet. Dazu verschieben wir zunächst den Ursprung und drehen dann das Koordinatensystem. Dadurch können wir beim Aufruf von `drawLine(int, int, int, int)` einfache Koordinaten angeben, ohne dafür aufwendige Berechnungen abhängig vom Winkel ausführen zu müssen.

```
private void drawIndicator(final Graphics2D graphics2D, final Point2D center)
{
    // Ursprung verschieben
    graphics2D.translate(center.getX(), center.getY());

    // Koordinatensystem drehen
    final double rotation = model.getValue();
    graphics2D.rotate(Math.toRadians(adjustment + -180 + rotation * stretch));

    // Den Zeiger malen
    graphics2D.setStroke(new BasicStroke(4, BasicStroke.CAP_ROUND,
                                         BasicStroke.JOIN_ROUND));
    graphics2D.setColor(Color.WHITE);
    graphics2D.drawLine(0, 0, indicatorLength, 0);

    // Transformation zurücknehmen
    graphics2D.rotate(-Math.toRadians(adjustment + -180 + rotation *
stretch));
    graphics2D.translate(-center.getX(), -center.getY());
}
```

Schritt 3: Die Skala

Am anspruchsvollsten ist das Zeichnen der Skala, da hier eine fortlaufende Rotation erfolgen muss. Zunächst sind die Markierungslinien zu zeichnen. Jede zweite wird dicker und länger gezeichnet. Ähnlich verfahren wir mit der Geschwindigkeitsangabe über der Markierungslinie. Als Besonderheit muss der Text nochmal um 90° zur Markierungslinie gedreht werden – streng genommen um -90°, da hier eine Drehung gegen den Uhrzeigersinn erfolgt. Daraus ergibt sich die verwendete Gradangabe von 270.

Die beschriebene Darstellung wird durch die Methode `drawTicksAndValueTexts(Graphics2D, Point2D)` realisiert, indem in einer Schleife wiederholt die beiden Methoden `drawTick(Graphics2D, int, int, int)` und `drawValueText(Graphics2D, int, int, int)` zum Zeichnen einer Markierungslinie bzw. eines Geschwindigkeitswerts aufgerufen werden:

```
private void drawTicksAndValueTexts(final Graphics2D graphics2D,
                                    final Point2D center)
{
    graphics2D.translate(center.getX(), center.getY());

    // Radius basierend auf dem des Zeigers
    final int radius = indicatorLength + 15;

    final double valueRange = model.getMaximum() - model.getMinimum();
    final int extend = model.getExtent();
    final int numOfTicks = (int) valueRange / extend;

    for (int i = 0; i < numOfTicks; i++)
    {
        graphics2D.rotate(Math.toRadians(adjustment + stretch * i * extend));

        // Markierungslinie zeichnen
        final int tickLength = drawTick(graphics2D, radius, i, numOfTicks);

        // Geschwindigkeitsangabe zeichnen
        drawValueText(graphics2D, radius, i, tickLength);

        graphics2D.rotate(-Math.toRadians(adjustment + stretch * i * extend));
    }

    graphics2D.translate(-center.getX(), -center.getY());
}

private int drawTick(final Graphics2D graphics2D, final int tickRadius,
                     final int value, final int numOfTicks)
{
    final int tickLength = calcTickLength(value);

    prepareGraphicsForTick(graphics2D, value, numOfTicks);
    graphics2D.drawLine(-tickRadius, 0, -tickRadius + tickLength, 0);

    return tickLength;
}
```

Die Methode `drawValueText(Graphics2D, int, int, int)` gibt einen Wert über der Markierungslinie aus. Dazu ermittelt sie verschiedene Abstände und auch die Abmessungen des Texts mithilfe von `FontMetrics` wie folgt:

```
private void drawValueText(final Graphics2D graphics2D, final int tickRadius,
                           final int value, final int tickLength)
{
    final int tickTextGap = calcTickTextGap(value);
    prepareGraphicsForValueText(graphics2D, value);

    // Wert mittig über Trennlinie schreiben
    final String strValue = Integer.toString(value * model.getExtent());
    final FontMetrics fm = graphics2D.getFontMetrics(getFont());
    final Rectangle2D stringBounds = fm.getStringBounds(strValue, graphics2D);
    final int textYPos = tickLength + tickTextGap;

    // Text 90 Grad zu Linien drehen
    graphics2D.translate(-tickRadius + tickLength, 0);
    graphics2D.rotate(Math.toRadians(270));
    graphics2D.drawString(strValue, -(int) stringBounds.getWidth() / 2,
                                    -textYPos);
    graphics2D.rotate(-Math.toRadians(270));
    graphics2D.translate(+tickRadius - tickLength, 0);
}
```

Um die Realisierung beider Hilfsmethoden möglichst einfach zu halten, bediene ich mich eines Implementierungstricks: Die Berechnung bzw. das Setzen verschiedener Attribute für die grafische Zeichenumgebung (`Graphics2D`) werden durch zwei Hilfsmethoden `prepareGraphicsForTick/ValueText(...)` durchgeführt.

```
private void prepareGraphicsForTick(final Graphics2D graphics2D,
                                    final int value, final int numOfTicks)
{
    BasicStroke tickStroke = new BasicStroke(2);

    if (value % 2 == 0)
    {
        tickStroke = new BasicStroke(4);
    }

    graphics2D.setColor(getTickColor(value, numOfTicks));
    graphics2D.setStroke(tickStroke);
}

private int prepareGraphicsForValueText(final Graphics2D graphics2D,
                                        final int value)
{
    Font font = TICK_FONT;
    Color fontColor = Color.LIGHT_GRAY;

    if (value % 2 == 0)
    {
        font = LONG_TICK_FONT;
        fontColor = Color.WHITE;
    }

    graphics2D.setFont(font);
    graphics2D.setColor(fontColor);
}
```

Es verleiben ein paar Hilfsmethoden zur Bestimmung von Abständen sowie von Farben:

```
private int calcTickLength( final int value)
{
    if ( value % 2 == 0)
        return LONG_TICK_LENGTH;

    return TICK_LENGTH;
}

private int calcTickTextGap( final int value)
{
    if ( value % 2 == 0)
        return 7;

    return 4;
}

private static Color getTickColor( final int value, final int maxNumOfTicks)
{
    // Werte könnten auch als Konstanten definiert oder aus
    // einem komplexeren Modell gelesen werden
    if ( value < 100)
        return Color. LIGHT_GRAY;
    if ( value < 140)
        return Color. GREEN;
    if ( value < 170)
        return Color. YELLOW;

    return Color. RED;
}
```

Letztere Methode dient dazu, das Bedienelement optisch noch interessanter zu gestalten und ein besseres Feedback zu geben. Dazu werden die Markierungsstriche zunächst hellgrau, dann grün und gelb und schließlich rot gezeichnet.

Schritt 4: Die GUI-Komponente `TachoControl`

Damit haben wir alle Basisbausteine zusammen, die wir zur Darstellung für die GUIKomponente `TachoControl` benötigen. Was fehlt noch? Das Datenmodell! Dieses realisieren wir nicht selbst, sondern nutzen dafür die Klasse `javax. swing. DefaultBoundedRangeModel` aus dem JDK, das unter anderem für die Klassen `JSlider` und `JScrollBar` eingesetzt wird. Bei Wertänderungen wird durch dieses Modell ein `javax. swing. event. ChangeEvent` ausgelöst, das im `TachoControl` ein Neuzeichnen per `repaint()` veranlasst. Damit das möglich wird, implementiert die Klasse `TachoControl` das Interface `javax. swing. event. ChangeListener`. Das folgende Listing zeigt die Klasse `TachoControl`:

```
public class TachoControl extends JComponent implements ChangeListener
{
    private static final int TICK_LENGTH = 6;
    private static final Font TICK_FONT = new Font( "Arial", Font. PLAIN, 12);
    private static final int LONG_TICK_LENGTH = 14;
```

```
    private static final Font LONG_TICK_FONT = new Font("Arial", Font.BOLD,
18);

    // Stretch-Faktor, damit die Zahlen nicht so nah beieinander stehen
    private static final double adjustment = -35;
    private static final double stretch = 1.25;

    private final int innerRadius = 150;
    private final int innerStart = 20;
    private final int outerRadius = innerRadius + innerStart;
    private final int tickRadius = innerRadius - 25;
    private final int indicatorLength = innerRadius - 40;

    private final BoundedRangeModel model;

    TachoControl(final BoundedRangeModel model)
    {
        this.model = model;
        this.model.addChangeListener(this);
    }

    @Override
    public void stateChanged(final ChangeEvent event)
    {
        repaint();
    }

    @Override
    public void paintComponent(final Graphics graphics)
    {
        super.paintComponent(graphics);

        final Graphics2D graphics2D = (Graphics2D) graphics;

        // Aktivierung von Antialiasing
        graphics2D.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                    RenderingHints.VALUE_ANTIALIAS_ON);

        graphics2D.setRenderingHint(RenderingHints.KEY_TEXT_ANTIALIASING,
                                    RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

        final Point2D center = new Point2D.Float(outerRadius, outerRadius);
        drawBackground(graphics2D, center);
        drawAnchor(graphics2D, center);
        drawTicksAndValueTexts(graphics2D, center);
        drawIndicator(graphics2D, center);
    }

    // …
}
```

Im obigen Listing sehen Sie zwei Aufrufe an die bisher unbekannte Methode `setRenderingHint(RenderingHints.Key, Object)`. Diese schalten das sogenannte ***Antialiasing*** für Zeichen- und Textausgaben ein. Dadurch erfolgt eine Berechnung von »Zwischenpixeln«, wodurch Farbabstufungen, Ränder von Figuren und Buchstaben nicht mehr pixelig erscheinen, sondern sanft in den Hintergrund übergehen.

Schritt 5: Ein kleines Testprogramm

Zur Demonstration der beschriebenen Funktionalität implementieren wir in der `main()`-Methode das fortlaufende Setzen des Werts im Datenmodell wie folgt:

```
public static void main(final String[] args) throws InterruptedException
{
    final JFrame appFrame = new JFrame("TachoControlExample");
    appFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final BoundedRangeModel model = new DefaultBoundedRangeModel(0, 10, 0,
200);
    final TachoControl tachoControl = new TachoControl(model);

    appFrame.add(tachoControl, BorderLayout.CENTER);
    appFrame.pack();
    appFrame.setVisible(true);

    // nun verschiedene Werte simulieren
    for (int i = 0; i <= 240; i += 10)
    {
        model.setValue(i);
        Thread.sleep(500);
    }
}
```

Listing 9.17 *Ausführbar als* **'TACHOCONTROLEXAMPLE'**

Fazit

Zuletzt wurde beschrieben, wie ein GUI mithilfe von Java 2D attraktiv gestaltet werden kann. Am Beispiel der Tachometeranzeige und der dazu erstellten Hilfsmethoden erkennt man sehr schön die Vorteile eines sauberen Sourcecode-Designs, wie dies der folgende Praxistipp argumentiert.

Tipp: Vorteile sinnvoll gewählter, überschaubarer Methoden

In der Praxis sieht man leider immer mal wieder GUI-Code, der recht verworren und wenig strukturiert ist. In diese Situation gerät man schnell, wenn man nicht fortlaufend refaktorisiert (Kapitel 17 beschreibt einige nützliche Refactorings) und für Einfachheit sorgt. Der vorgestellte Sourcecode für das Bedienelement `TachoControl` ist ein gutes Beispiel dafür: Er war zunächst auch nicht ganz so übersichtlich, wurde dann aber mehrmals überarbeitet. Dazu habe ich möglichst viel Funktionalität in separate Methoden ausgelagert. So konnte ich diese jeweils recht kurz halten, wodurch sich der Programmablauf besser nachvollziehen lässt. Zudem erleichtern kurze, übersichtliche Methoden mit wenigen Parametern die Testbarkeit, weil die korrekte Arbeitsweise jeder dieser Methoden gut mithilfe von Unit Tests prüfbar ist (Details zum Unit-Testen liefert Ihnen Kapitel 20). Darüber hinaus erhöht sich die Wartbarkeit, da kurze, prägnante Methoden leicht verständlich und beherrschbar und meistens auch ohne viel Mühe an neue Gegebenheiten anpassbar sind.

9.4 Komplexe Bedienelemente

Bisher haben wir eher grundlegende Bedienelemente wie Labels und Buttons genutzt. Nachfolgend stelle ich Ihnen mit Listen, Tabellen und Bäumen drei komplexere Bedienelemente vor, die zur Darstellung umfangreicherer Datenmengen gewinnbringend eingesetzt werden können. In Abschnitt 9.4.1 arbeite ich einige Gemeinsamkeiten dieser Bedienelemente heraus, wodurch die Grundlage für das Verständnis der folgenden Abschnitte geschaffen wird. Abschnitt 9.4.2 beschäftigt sich mit einer `JList<E>`. Im Anschluss daran zeigt Abschnitt 9.4.3 die Verarbeitung tabellarischer Daten mithilfe einer `JTable`. Als Letztes bespreche ich in Abschnitt 9.4.4 hierarchisch strukturierte Daten und das Bedienelement `JTree`.

9.4.1 Grundlagen

Befassen wir uns nun einführend mit einigen Grundlagen. Wir beginnen mit einer Einführung in die ***Model-View-Controller-Architektur*** (MVC-Architektur oder kurz ***MVC***), die in nahezu allen[8] Swing-GUI-Komponenten Anwendung findet. Danach lernen wir Wissenswertes zu sogenannten Modellen kennen, die als Datenlieferanten für die Darstellung von Listen, Tabellen und Bäumen dienen. Abschließend widmen wir uns dann dem View, im Speziellen dem Erscheinungsbild der Einträge: Die Architektur der komplexeren Bedienelemente sieht vor, mithilfe eigener Implementierungen von ***Renderern*** im Detail steuern zu können, wie ein einzelner Eintrag gezeichnet wird.

Model-View-Controller

Die Idee bei der MVC-Architektur ist, die Verarbeitung und Darstellung von Daten durch drei voneinander getrennte Komponenten ausführen zu lassen:

- Das ***Model*** dient als Datenquelle und zur Aufbereitung und Verwaltung von Daten.
- Die Anzeige der Daten ist Aufgabe der Visualisierung (auch ***View*** genannt).
- Die Steuerungseinheit (***Controller***) verbindet Model und View. Der Controller empfängt und verarbeitet Benutzereingaben und sorgt für die passende Reaktion.

Neben der Funktion als Datenlieferant für die Anzeige, speichert das ***Model*** auch Selektionen (d. h. vom Benutzer ausgewählte Daten). Kommt es zu Änderungen im Modell, so müssen Mitteilungen an den View kommuniziert werden, um die Darstellung aktuell zu halten. Dazu muss sich der View zunächst einmal als Beobachter bzw. Änderungsinteressent beim Modell anmelden.[9] Der ***View*** ist für die Darstellung auf dem Bildschirm (den *Look*) zuständig und zeichnet je nach Datenlage und Selektion. Der ***Controller*** steuert das Verhalten (das *Feel*): Dazu nimmt er Benutzeraktionen und andere Ereignisse entgegen und bestimmt, wie darauf reagiert

wird, z. B. welcher Eintrag angesprungen oder selektiert wird.

Auswirkungen von MVC

Ein wesentliches Merkmal von MVC ist die Auftrennung in drei logische Bestandteile. Deren möglichst lose Kopplung bringt neben der Austauschbarkeit einzelner Komponenten auch weitere Vorteile wie einfache Wart- und Erweiterbarkeit: ***Jeder Bestandteil kann durch einen anderen ausgetauscht werden, sofern dieser die benötigte Schnittstelle erfüllt***. Insbesondere lassen sich beliebige eigene Modelle zur Datenversorgung verwenden. Darüber hinaus kann man ein Modell als Grundlage für mehrere Views nutzen.

Datenmodelle

Die Architektur des Swing-Frameworks sieht vor, Funktionalitäten bevorzugt in Form von Interfaces anzusprechen. Dadurch lässt sich eine saubere Trennung von Datenmodell und Darstellung realisieren. Zudem wird es möglich, beliebige Datenbestände im GUI zu visualisieren, weil auf diese mithilfe speziell dafür vorgegebener Modell-Interfaces zugegriffen wird: Im JDK sind dazu für Listen, Tabellen und Bäume die Interfaces `javax.swing.ListModel<E>`, `javax.swing.table.TableModel` und `javax.swing.tree.TreeModel` definiert. Mit Ausnahme von `TreeModel` gibt es für diese Interfaces abstrakte Basisimplementierungen, in denen bereits einige Funktionalität zur Verfügung gestellt wird, etwa die Verarbeitung von Events und die Verwaltung von Event Listenern. Das Design der abstrakten Basisklassen ermöglicht maximale Flexibilität bei der Datenbereitstellung: Es findet dort bewusst keine Datenspeicherung statt, da diese typischerweise anwendungsspezifisch ist und somit in eigenen Erweiterungen realisiert werden sollte. Dementsprechend bieten die abstrakten Basisklassen *keine* Methoden, um Daten zu modifizieren. Stattdessen wird dort nur ein rein lesender Zugriff angeboten, der für die Darstellung essenziell ist. Zur Vervollständigung der Implementierung des Modells sind dann nur noch wenige Methoden entsprechend der gewählten Datengrundlage zu realisieren, wie wir dies gleich exemplarisch für ein `ListModel<E>` sehen werden.

Beispiel eines `ListModels`

Nehmen wir an, dass in einer `JList<E>` eine Menge von Namen dargestellt werden soll, die als `String[]` vorliegt. Zur Verarbeitung in einer `JList<E>` müssen die Daten durch ein entsprechendes Modell bereitgestellt werden, das das Interface `ListModel<E>` erfüllt. Dieses ist im JDK wie folgt definiert:

```
// Kommentierter Auszug aus dem JDK
public interface ListModel<E>
{
    // Verwaltung der Daten
    int getSize();
```

```
    E getElementAt(int index);

    // Für Benachrichtigungen bei Änderungen im Modell
    void addListDataListener(ListDataListener listener);
    void removeListDataListener(ListDataListener listener);
}
```

Die gezeigten Methoden sind für die Funktion des Bedienelements `JList<E>` erforderlich: Man kann die Gesamtzahl an Einträgen mit `getSize()` ermitteln bzw. auf einzelne Einträge mit `getElementAt(int)` indiziert zugreifen. Darüber hinaus können sich Interessenten am Modell anmelden, um über Veränderungen informiert zu werden. Die `JList<E>` nutzt dies, um Änderungen im Modell im View reflektieren zu können: Als Folge werden betroffene Listenzeilen neu gezeichnet.

Im JDK ist die abstrakte Basisklasse `javax.swing.AbstractListModel<E>` definiert, die die Verwaltung von Interessenten vom Typ `ListDataListener` vornimmt und diverse Benachrichtungsmethoden bereitstellt, sodass in eigenen Realisierungen lediglich die Methoden `getSize()` und `getElementAt(int)` implementiert werden müssen. Darüber hinaus gibt es vorgefertigte Implementierungen, sogenannte Defaultmodelle, die ich nun kurz beschreiben möchte.

Defaultmodelle

Die Defaultmodelle implementieren alle Methoden des jeweiligen Interface und speichern Daten. Insbesondere erlauben die Defaultmodelle eine Veränderung der gespeicherten Daten, also das Hinzufügen, Verändern und Löschen von Einträgen. Für das Bedienelement `JList<E>` nutzt die Modellklasse `javax.swing.DefaultListModel<E>` einen `Vector<E>` zur Speicherung der Daten. Für eine `JTable` bietet die Klasse `javax.swing.table.DefaultTableModel` eine vollständige Implementierung des Interface `TableModel`, die ebenfalls einen `Vector<E>` für die Zeilen nutzt, der wiederum `Vector<E>`-Instanzen zur Speicherung der Spalten einer Zeile der Tabelle beinhaltet.

Kommen wir zu der zuvor genannten Liste von Namen zurück. Das folgende Sourcecode-Fragment nutzt ein `String[]` zur Definition von Namen als Eingabedaten und füllt diese in ein `DefaultListModel<String>`:

```
private static final String[] NAMES = { "Alex", "Andi", "Andy", "Bernd"
                                        "Clemens", "Merten", "Michael",
                                        "Peter", "Rudolf" };

// DefaultListModel aufwendig in der Befüllung
final DefaultListModel<String> listModel = new DefaultListModel<>();
for (final String name : NAMES)
{
    listModel.addElement(name);
}
```

Hier sieht man, dass das Füllen des Modells mit Daten bei Einsatz eines

`DefaultListModel<E>` etwas unhandlich ist: Man kann die Daten nur über mehrere Methodenaufrufe, beispielsweise `addElement(E)`, hinzufügen, nicht jedoch – wie es praktisch wäre – initial an den Konstruktor des Datenmodells übergeben. Bald werden wir für eine eigene Realisierung eines `ListModel<String>` lernen, wie wir dieses Manko beheben. Vorher möchte ich allerdings noch auf einen weiteren wichtigen Negativpunkt beim Einsatz der Defaultmodelle hinweisen.

Schwachpunkte der Defaultmodelle

Während für Listen eine Speicherung als `Vector<E>` häufig noch recht natürlich ist, so gilt dies für Tabellen, in der Regel eine Verschachtelung aus mehreren `Vector<E>`, nicht mehr. Hierbei speichert ein `Vector<E>` die Zeilen, deren Spaltenwerte jeweils durch einen weiteren `Vector<E>` modelliert werden.[10]

Spalten lassen sich meistens statt in einem `Vector<E>` besser und natürlicher über die Attribute eines Objekts beschreiben, weil der Spaltenvektor lediglich für den allgemeinen Typ `Object` definiert werden kann. Das ist unschön, da Attribute beliebige Typen besitzen können. Ein typsicherer Zugriff auf Attribute ist dadurch nur noch abgesichert über `instanceof` möglich. Je mehr Attribute eine Klasse besitzt, desto aufgeblähter und unleserlicher wird der Sourcecode. Die Nutzung eines `DefaultTableModel`s ist somit selten ratsam.

Ganz allgemein gilt, dass Daten häufig eine beliebige, nicht direkt mit einem der vordefinierten Modelle für Listen, Tabellen oder Bäume verarbeitbare Struktur besitzen. Spätestens dann benötigt man eine von den Defaultmodellen abweichende Verarbeitung oder Speicherung von Daten in Form von eigenen Modellen, was wir im Folgenden betrachten.

Eigene Modelle

Bei der Implementierung eigener Datenmodelle wird man häufig die abstrakten Basisklassen nutzen, da diese schon viele Methoden sinnvoll vordefinieren und dadurch nur noch wenige selbst zu implementieren sind. Eine komplett eigenständige Implementierung der Interfaces ist deutlich aufwendiger und wird daher in der Praxis eher selten genutzt.

Schauen wir nun, wie wir für eine `JList<String>` die Eingabedaten mithilfe eines eigenen Modells verwalten können. Das folgende Listing zeigt die Klasse `StringListModel`, die eine Implementierung eines eigenen Datenmodells mithilfe der abstrakten Basisklasse `AbstractListModel<String>` realisiert. Wie zuvor erwähnt, sind zur Fertigstellung des Modells nur noch wenige Methoden selbst zu realisieren, in diesem Fall `getSize()` und `getElementAt(int)`. Die Klasse `StringListModel` ist so implementiert, dass die Eingabedaten vom Typ `String[]` an ein `ListModel<String>` angepasst werden. Diese Realisierung folgt dem ADAPTER-Muster (vgl. Abschnitt 18.2.2).

```
public class StringListModel extends AbstractListModel<String>
{
    private final String[] values;

    public StringListModel(final String[] inputData)
    {
        // Kopie vermeidet Änderungen an der Zusammensetzung, hier
        // auch am Inhalt aufgrund der Unveränderlichkeit von Strings
        this.values = Arrays.copyOf(inputData, inputData.length);
    }

    @Override
    public int getSize()
    {
         return values.length;
    }

    @Override
    public String getElementAt(int index)
    {
        return values[index];
    }
}
```

Hier soll nur der prinzipielle Ablauf bei der Realisierung eigener Modelle verdeutlicht werden, daher ist nur eine einfache Variante gezeigt, die nachträgliche, potenziell problematische Datenmodelländerungen vermeidet, indem man eine Kopie des Arrays mit `Arrays.copyOf(T[], int)` (vgl. Abschnitt 5.1.2) erstellt – was in diesem Fall aufgrund der Unveränderlichkeit des Typs `String` ausreichend ist, um sich vor Modifikationen zu schützen. Generell sollte man vermeiden, mit hereingereichten Referenzen auf Daten zu arbeiten. Zwar könnte man meinen, dass so automatisch immer die aktuellsten Änderungen im GUI dargestellt werden können. Aber aufgrund der Single-Thread-Regel können nicht im EDT ablaufende Datenmodelländerungen zu verschiedenen Arten von Problemen (z. B. Darstellungsfehlern) führen – später erfahren Sie dazu mehr.

Nutzung

Betrachten wir nun Datenmodelle im Einsatz. Ein `ListModel<E>` kann man entweder bei der Konstruktion an eine `JList<E>`-Instanz übergeben oder später durch einen Aufruf der Methode `setModel(ListModel<E>)` setzen. Da ein Bedienelement immer nur ein Modell besitzen kann, wird dieses durch den Aufruf ersetzt, sofern eins vorhanden war.

Nachfolgend sind die Varianten der Zuweisung eines Modells an eine `JList<E>` für ein `StringListModel` und für ein `DefaultListModel<String>` gezeigt:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("JListModelExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    // Neu in JDK 7: typisierte JList und Diamond Operator
    final JList<String> list = new JList<>(new StringListModel(NAMES));
```

```
    // JList mit DefaultListModel aufwendig in der Konstruktion/Befüllung
    final DefaultListModel<String> listModel = new DefaultListModel<>();
    for (final String name : NAMES)
    {
        listModel.addElement(name);
    }

    final JList<String> otherList = new JList<>();
    // Alternative: Modell per setModel() setzen
    otherList.setModel(listModel);

    // Integration der JList in eine JScrollPane
    frame.add(new JScrollPane(list), BorderLayout.CENTER);
    frame.add(new JScrollPane(otherList), BorderLayout.EAST);

    frame.setSize(400, 150);
    frame.setVisible(true);
}
```

Listing 9.18 *Ausführbar als* **'JLISTMODELEXAMPLE'**

Starten wir das Programm JLISTMODELEXAMPLE, so erscheint in etwa die in Abbildung 9-30 gezeigte Darstellung.

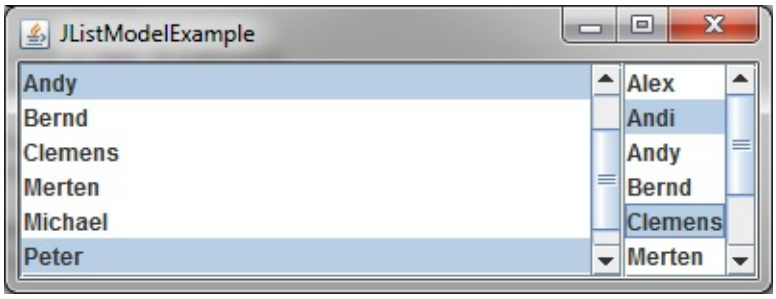


Abbildung 9-30 *Darstellung einer* `JList` *mit textuellen Werten*

Tipp: Bedienelemente und das Interface `Scrollable`

Im obigen Beispiel lernen wir als Vorgriff auf die Beschreibung der Darstellung im View noch ein Detail kennen: die Einbettung der `JList<String>` in eine `JScrollPane`. Nur dadurch wird bei umfangreicheren Datenmengen oder beschränktem Platz automatisch eine Scrollbar angezeigt.

Die komplexeren Bedienelemente Listen, Tabellen und Bäume erfüllen alle das Interface `javax.swing.Scrollable`, das deren Integration in eine `JScrollPane` erlaubt. Auf diese Weise wird die Navigation über die Einträge mithilfe von Scrolling möglich. Diese Art der Bedienung ist insbesondere bei umfangreichen Datenmengen wünschenswert, um die Auswahl beliebiger (möglicherweise gerade nicht dargestellter) Einträge zu ermöglichen.

View – Darstellung mit Renderern

Im eben gezeigten Beispiel erfolgte auch ohne unser Zutun eine korrekte Darstellung der textuellen Informationen für die Eingabedaten vom Typ `String[]`. Das ist aber nicht – wie es zunächst scheinen mag – selbstverständlich. Sofern nicht anders

festgelegt, und demnach auch im vorherigen Beispiel, wird automatisch ein sogenannter ***Default-Renderer*** zur Aufbereitung der Darstellung einzelner Einträge verwendet, der auf einem `JLabel` basiert, wodurch sich Strings oder Grafiken vom Typ `javax.swing.Icon` problemlos zeichnen lassen. Mit diesem Wissen wird klar, weshalb die Liste von Strings wie erwartet dargestellt wird.

Doch wie sieht es aus, wenn wir in einer `JList<E>` statt textueller Informationen beliebige Objekte zur Auswahl anbieten wollen, also etwa statt der Namen beispielsweise `Person`-Objekte, wie im folgenden Listing gezeigt:[11]

```
// …
private static final Person[] PERSONS = { new Person("Alex", "Name",
birthday1),
                                         new Person("Mike", "Inden",
birthday2)
                                         new Person("Joe", "Name",
birthday3)};
final JList<Person> list = new JList<>(PERSONS);
frame.add(new JScrollPane(list), BorderLayout.CENTER);
// …
```

Berechtigterweise kann man sich fragen, wie die Darstellung dieser Objekte erfolgen soll, da die `JList<Person>` ja nicht weiß, welche Attribute eines `Person`-Objekts zur Visualisierung genutzt werden sollen. Zunächst einmal zeige ich Ihnen, wie die obige `JList<Person>` dargestellt wird (vgl. Abbildung 9-31), wenn Sie das Programm JLISTPERSONEXAMPLE ausführen.

Abbildung 9-31 *Darstellung einer* `JList<Person>`

Wie kommt es zu dieser Darstellung? Ohne weiteres Zutun wird durch den Default-Renderer mittels Aufruf der Methode `toString()` für jedes `Person`-Objekt eine Stringrepräsentation erzeugt. Das ist möglich, da in Java jede Klasse diese Methode besitzt (vgl. Abschnitt 4.1). Bekanntermaßen ist diese Stringrepräsentation jedoch nicht sehr aussagekräftig – wobei Zahlentypen und wie bereits gesehen natürlich auch Strings Ausnahmen sind. Als erste Lösungsidee könnte man für andere Typen darauf kommen, `toString()` so zu überschreiben, dass diese eine gewünschte Darstellung liefert. Das klingt zunächst vernünftig, jedoch sind die gewünschten Attribute sowie deren Darstellung meistens von dem jeweiligen Anwendungsfall abhängig. Was kann man also tun?

Für einfachere Bedienelemente wie `JButtons` und `JLabels` haben wir das Überschreiben von `paintComponent(Graphics)` kennengelernt. Für die komplexeren Bedienelemente `JList<E>`, `JTable` und `JTree` wäre es aber wenig sinnvoll, deren Darstellung vollständig oder in großen Teilen selbst zu gestalten, denn das würde eine aufwendige Implementierung erfordern. Daher kommen hier die bereits erwähnten ***Renderer*** ins Spiel, die für die Darstellung einzelner Einträge

verantwortlich sind. Die Beschreibung, wie dies für die Darstellung von `Person`-Objekten eingesetzt wird, verschiebe ich auf Abschnitt 9.4.2, weil wir zunächst einmal einige Grundlagen zu Renderern kennenlernen müssen, die sich an einem einfachen Beispiel einer farbigen Präsentation der Namen leichter erklären lassen.

Grundlagen zu eigenen Renderern

Die Renderer-Funktionalität wird durch verschiedene Interfaces modelliert und bietet viele Gestaltungsmöglichkeiten. Folgende Interfaces und Default-Renderer-Klassen werden durch das JDK bereitgestellt:

- `ListCellRenderer<E>` und `DefaultListCellRenderer`
- `TableCellRenderer` und `DefaultTableCellRenderer`
- `TreeCellRenderer` und `DefaultTreeCellRenderer`

Betrachten wir exemplarisch Listen und die Klasse `DefaultListCellRenderer` sowie das zugrunde liegende Interface `ListCellRenderer<E>` aus dem JDK:

```
// Auszug aus dem JDK
public interface ListCellRenderer<E>
{
    Component getListCellRendererComponent(JList<? extends E> list,
                                           E value,
                                           int index,
                                           boolean isSelected,
                                           boolean cellHasFocus);
}
```

Die Implementierung der Methode `getListCellRendererComponent()` legt für die Einträge einer `JList<E>` fest, welche Farben, Texte, Icons zur Darstellung verwendet werden. Diese Methode wird für jeden darzustellenden Eintrag automatisch beim Zeichnen des Bedienelements aufgerufen. Zur individuellen Konfiguration eines Eintrags kann man steuernd eingreifen und pro Eintrag verschiedene Dinge bzw. Zeichenattribute wie Farben und Rahmen abhängig von Selektion und Fokus usw. festlegen. Dieser Vorgang wiederholt sich für alle sichtbaren Einträge der Liste.

Einen eigenen Renderer erstellen

Nehmen wir an, wir wollten die Liste der Namen optisch etwas bunter gestalten. Beispielsweise kann man diese abhängig vom Startbuchstaben einfärben: Für alle mit A beginnenden Namen nutzen wir Rot, für alle mit M startenden Namen Blau und für die restlichen Startbuchstaben Schwarz.

Das folgende Listing zeigt die Klasse `ColorListCellRenderer`, die die beschriebene Einfärbung umsetzt und als Basis die Klasse `DefaultListCellRenderer` nutzt, die wiederum auf einem `JLabel` basiert. Dieses bietet u. a. die Methoden `setIcon(Icon)`, `setText(String)` und

`setForeground(Color)`. Für die gewünschte Einfärbung des Namens ist in diesem Fall für jeden Eintrag abhängig vom Startbuchstaben die Vordergrundfarbe durch einen Aufruf von `setForeground(Color)` zu setzen:

```
public class ColorListCellRenderer extends DefaultListCellRenderer
{
    @Override
    public Component getListCellRendererComponent(final JList list,
                          final Object value, final int index,
                          final boolean isSelected, final boolean
cellHasFocus)
    {
        super.getListCellRendererComponent(list, value, index,
                                           isSelected, cellHasFocus);

        final String name = (String)value;

        Color color = Color.BLACK;
        if (name.startsWith("A"))
            color = Color.RED;
        else if (name.startsWith("M"))
            color = Color.BLUE;

        setForeground(color);

        return this;
    }
}
```

Je nach Vorwissen stolpern Sie eventuell über die letzte Zeile `return this`. Daher möchte ich hier kurz darauf eingehen. Die Methode `getListCellRendererComponent()` liefert normalerweise die gleiche Instanz zurück: Das ist oftmals die Renderer-Instanz selbst, die zuvor entsprechend konfiguriert wurde.

Die angedeutete Vorgehensweise nutzt man, um Ressourcen zu schonen. Dazu wird bei Renderern häufig folgendermaßen verfahren: Statt für jeden Eintrag eine neue Instanz des `Component`-Objekts zu erzeugen, wird immer wieder die gleiche `Component`-Instanz verwendet, die nur anders konfiguriert ist. Die Default-Renderer besitzen immer den Basistyp `Component` und geben die `this`-Referenz zurück. Diese Wiederverwendung einer Instanz ist sinnvoll, wenn eine große Menge an Einträgen anzuzeigen ist: Auf diese Weise vermeidet man die Konstruktion vieler Objekte und spart viel Speicher, da man stattdessen nur mit einem Objekt und unterschiedlichen Parametrierungen arbeitet.

Einsatz von Renderern

Weil die Klasse `JList<E>` die Funktionalität des Renderers nur über dessen Interface nutzt, bleibt für sie die eventuell vorhandene Komplexität der Renderer-Implementierung vollständig verborgen. Zur Aktivierung der Zeichenfunktionalität des obigen Renderers ist lediglich folgender Aufruf notwendig:

```
public static void main(final String[] args)
{
```

```
        final JFrame frame = new JFrame("ColoredJListExample");
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

        final JList<String> list = new JList<>(new StringListModel(NAMES));
        frame.add(new JScrollPane(list), BorderLayout.CENTER);

        list.setCellRenderer(new ColorListCellRenderer());

        frame.setSize(300, 150);
        frame.setVisible(true);
    }
```

Listing 9.19 *Ausführbar als* **'COLOREDJLISTEXAMPLE'**

Beim Start des Programms COLOREDJLISTEXAMPLE erhält man eine Listendarstellung ähnlich zu der in Abbildung 9-32.

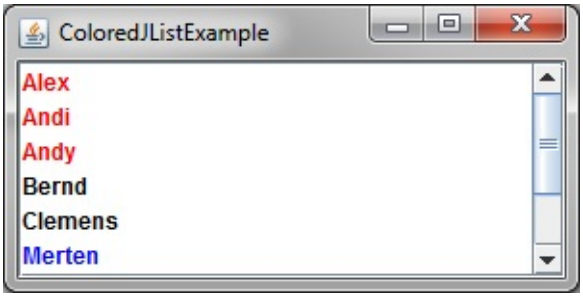


Abbildung 9-32 *Darstellung einer* `JList<String>` *mit farbigen textuellen Werten*

Fazit

Bisher haben wir die Klasse `JList<E>` nur exemplarisch zur Verdeutlichung wesentlicher Grundlagen der komplexeren Bedienelemente genutzt. Dabei haben wir die MVCArchitektur sowie Modelle und Renderer kennengelernt. Nun erfolgt eine detailliertere Betrachtung der `JList<E>` und im Anschluss daran eine Vorstellung von `JTable` und `JTree`.

9.4.2 Die Klasse `JList`

Das Bedienelement `JList<E>` dient dazu, eine Liste gleichartiger Daten darzustellen und Einträge zur Auswahl anzubieten. Seit JDK 7 ist sowohl das Bedienelement `JList<E>` selbst als auch der zugehörige Modelltyp `ListModel<E>` generisch definiert und ermöglicht so neben dem gebräuchlichen Anwendungsfall textueller Einträge auch die Visualisierung und typsichere Auswahl von Objekten beliebiger Typen.

Bereits ohne eigene Anpassungen bietet eine `JList<E>` diverse Funktionalitäten zur Verarbeitung listenartiger Daten. So besteht etwa die Möglichkeit der Navigation über Cursortasten oder Startbuchstaben zum Ansprung von Einträgen.

Interfaces und Klassen im Zusammenspiel mit einer `JList`

Die Klasse `JList<E>` erledigt ihre Aufgaben nur zum Teil selbst. Sie delegiert z. B. das Zeichnen und die Datenbereitstellung für die Anzeige an weitere Klassen. Aus den vorangegangenen Abschnitten wissen wir, dass Ersteres ein Renderer vom Typ `ListCellRenderer<E>` erledigt und Letzteres mithilfe einer Realisierung eines `ListModel<E>` geschieht. Noch unbekannt ist bisher, dass die selektierten Einträge durch ein `ListSelectionModel<E>` repräsentiert werden. Die genannten Interfaces und Klassen sind in Abbildung 9-33 in einen größeren Kontext eingebettet.

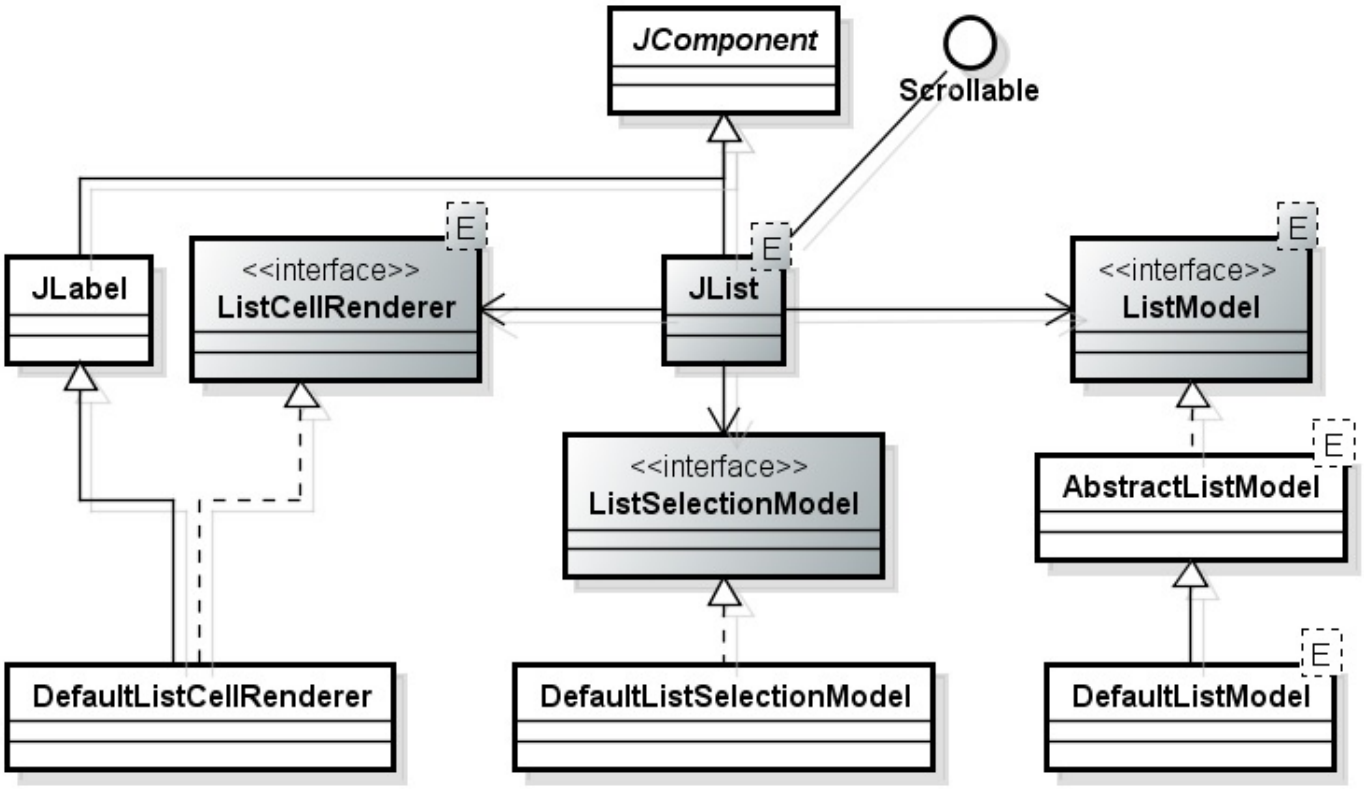


Abbildung 9-33 *Klassendiagramm* `JList<E>`

In der Abbildung sind die zentrale Klasse `JList<E>` sowie die eben genannten Interfaces grau eingefärbt. Deren Zusammenspiel werde ich nachfolgend anhand eines einfachen Beispiels verdeutlichen, das ich dann schrittweise ausbauen werde. Die erste Ausbaustufe entspricht der Liste von Namen und wurde in Abbildung 9-32 gezeigt. Diesen Stand erweitern wir nun um die Verarbeitung von Selektionen.

Selektionen

Während diverse Bedienelemente, z. B. `JComboBox<E>`, nur eine Einfachselektion erlauben, ist für `JList<E>`, `JTable` und `JTree` auch eine Mehrfachselektion zulässig. Aufgrund dieser komplexeren Selektionsmöglichkeiten erfolgt die Verarbeitung zu selektierten Einträgen in eigenständigen Klassen anstatt im Datenmodell.

Die in einer `JList<E>` gewählten Einträge werden durch eine Instanz eines `ListSelectionModel`s verwaltet. Den gewünschten Selektionsmodus kann man mithilfe der Methode `setSelectionMode(int)` festlegen. Dabei sind folgende Konstanten aus dem Interface `ListSelectionModel` als Eingabewerte erlaubt:

- `SINGLE_SELECTION` – Es kann maximal ein Element ausgewählt werden.
- `SINGLE_INTERVAL_SELECTION` – Die Auswahl mehrerer Elemente ist möglich, sie müssen aber einen zusammenhängenden Bereich bilden.
- `MULTIPLE_INTERVAL_SELECTION` – Es können mehrere beliebige Elemente ausgewählt werden. Das ist der ***Standard***.

Wenn man auf eine Veränderung der Selektion mit einer Aktion reagieren möchte, kann man dazu einen `ListSelectionListener` einsetzen. Dieses Interface definiert nur die Callback-Methode `valueChanged(ListSelectionEvent)`.

Das folgende Listing zeigt die Realisierung eines `ListSelectionListeners` in Form einer anonymen Klasse. Dort werden Selektionsereignisse verarbeitet und ein Detaildialog mithilfe der Klasse `javax.swing.JOptionPane` dargestellt:

```
list.addListSelectionListener(new ListSelectionListener()
{
    @Override
    public void valueChanged(final ListSelectionEvent selectionEvent)
    {
        if (!selectionEvent.getValueIsAdjusting())
        {
            @SuppressWarnings("unchecked")
            final JList<String> list = ((JList<String>)
selectionEvent.getSource
                    ());

            // JDK 7: neu im JList-API: getSelectedValuesList()
            final List<String> selectedValues = list.getSelectedValuesList();
            // Schachstelle: GUI-Aktion im EDT (siehe Text)
            JOptionPane.showMessageDialog(null, "Folgende Einträge gewählt: "
+
                                                    selectedValues);
        }
    }
```

Listing 9.20 *Ausführbar als* **'JLISTSELECTIONEXAMPLE'**

Als Besonderheit sieht man hier den Aufruf der Methode `getValueIsAdjusting()`. Das ist notwendig, weil bereits während der Bedienhandlungen zur Selektion verschiedene Events ausgelöst werden. Prüft man den Rückgabewert der Methode `getValueIsAdjusting()`, kann man verhindern, dass verschiedene Zwischenzustände der Selektion propagiert werden (z. B. wenn man mit gedrückter Maustaste über die Einträge fährt) und so kann man nur auf den Endzustand der Selektion reagieren.[12]

Schwachstelle GUI-Aktion im Listener

Führt man das Programm JLISTSELECTIONEXAMPLE aus, so wird eine kleine Unschönheit sichtbar: Ändert man die Selektion durch Drücken der Cursortasten, so erscheint zwar wie gewünscht die Hinweisdialogbox, jedoch sind in der Listendarstellung (manchmal) zwei Einträge so lange selektiert, bis die Dialogbox geschlossen wird. Abbildung 9-34 zeigt zur Verdeutlichung einen Screenshot.

Abbildung 9-34 *Problem beim Selektieren von Einträgen in einer* `JList<String>`

Zu diesem Verhalten kommt es, weil wir in diesem Beispiel bewusst zur Verdeutlichung möglicher Probleme innerhalb der Ereignisverarbeitung, also noch innerhalb der Methode `valueChanged(ListSelectionEvent)`, auf die Benachrichtigung der Selektionsänderung mit der Anzeige eines Dialogs eine weitere GUI-Aktion synchron im EDT durchführen. Das verhindert aber die Abarbeitung von Repaints der `JList<String>`: Zwar hat Swing bereits ein Event zum Neuzeichnen in die AWT-Event-Queue eingestellt, dieses wird jedoch erst nach der Abarbeitung der Methode `valueChanged(ListSelectionEvent)` ausgeführt.

In Abschnitt 9.2 habe ich die Verarbeitung von Events im EDT detailliert behandelt, daher nenne ich hier nur kurz die Lösung für die korrekte Darstellung der Selektion: Es muss eine Aktion in Form eines `Runnables` mithilfe der Methode `invokeLater(Runnable)` in die AWT-Event-Queue eingestellt werden.

Datenmodell

Wenn Daten in einer `JList<E>` angezeigt werden sollen, empfiehlt es sich, diese durch ein Modell vom Typ `ListModel<E>` zu verwalten. Normalerweise wird man für eigene Modelle die abstrakte Basisklasse `AbstractListModel<E>` erweitern, wie wir das schon bei der Besprechung der Grundlagen komplexerer Bedienelemente kennengelernt haben. Zum leichteren Verständnis der nachfolgenden Erläuterungen zeige ich hier erneut die Definition des Interface `ListModel<E>` aus dem JDK:

```
// Kommentierter Auszug aus dem JDK
public interface ListModel<E>
{
    // Verwaltung der Daten
    int getSize();
    E getElementAt(int index);

    // Für Benachrichtigungen bei Änderungen im Modell
    void addListDataListener(ListDataListener listener);
    void removeListDataListener(ListDataListener listener);
}
```

Basierend auf diesem Interface möchte ich das Beispiel einer Liste von Personen wieder aufgreifen, um ein erstes eigenes Datenmodell zu erstellen. Hier sehen wir zunächst die Definition einer Klasse `PersonListModel`, die von der abstrakten Basisklasse `AbstractListModel<Person>` erbt und für Eingabedaten in Form

einer `List<Person>` nur noch die zwei Methoden `getSize()` und `getElementAt(int)` folgendermaßen selbst realisieren muss:

```
public class PersonListModel extends AbstractListModel<Person>
{
    private final List<Person> persons;

    public PersonListModel(final List<Person> persons)
    {
        this.persons = new ArrayList<>(persons); // Vorsicht bei
Datenänderungen
    }

    @Override
    public int getSize()
    {
        return persons.size();
    }

    @Override
    public Person getElementAt(final int index)
    {
        return persons.get(index);
    }
}
```

Eine hereingereichte Referenz ist immer dann problematisch, wenn sich die Zusammensetzung der Daten oder deren Inhalte ändern. Die gesamte Problematik wollen wir hier zunächst nicht weiter betrachten, da dies für das grundsätzliche Verständnis des `ListModel<E>` nicht wichtig ist. Da dieses Thema in der Praxis für Anwendungen aber eine wesentliche Rolle spielt, werde ich später noch explizit darauf eingehen.

Beispieldaten

Für die nachfolgenden Ausführungen und Beispiele definiere ich als Datenbasis eine Liste von bekannten Personen aus dem Java-Umfeld wie folgt:

```
public enum Gender { MALE, FEMALE };

public static final List<Person> famousJavaPersons = new ArrayList<>();

static
{
    famousJavaPersons.add(new Person("Joshua", "Bloch", Gender.MALE));
    famousJavaPersons.add(new Person("Neil", "Gafter", Gender.MALE));
    famousJavaPersons.add(new Person("James", "Gosling", Gender.MALE));
    famousJavaPersons.add(new Person("Bart", "Bates", Gender.MALE));
    famousJavaPersons.add(new Person("Kathy", "Sierra", Gender.FEMALE));
    famousJavaPersons.add(new Person("Angelika", "Langer", Gender.FEMALE));
}
```

Zum Verständnis fehlt uns noch die Definition der Klasse `Person`. Diese ist recht einfach und besitzt lediglich drei Attribute. Außerdem wird die Klasse hier als unveränderliche Klasse (vgl. Abschnitt 3.4.2) realisiert, um mögliche Schwierigkeiten und Inkonsistenzen bei der Darstellung im GUI von vornherein auszuschließen.

```
public class Person
{
    private final String firstname;
    private final String lastname;
    private final Gender gender;

    public Person(final String firstname, final String lastname,
                  final Gender gender)
    {
        this.firstname = firstname;
        this.lastname = lastname;
        this.gender = gender;
    }

    public String getFirstname()          { return firstname; }
    public String getLastname()           { return lastname; }
    public Gender getGender()             { return gender; }
}
```

Renderer

Sehen wir uns nun an, wie wir `Person`-Objekte sinnvoll in einer `JList<Person>` anzeigen können. Hinlänglich bekannt ist, dass eine Darstellung von Objekten in Listen, basierend auf `toString()`, für die meisten Typen unzureichend ist – das gilt natürlich auch für die Klasse `Person`. Um diesen Missstand zu beheben, entwerfen wir nachfolgend einen eigenen Renderer, der das Geschlecht als grafisches Symbol (♂, ♀), gefolgt von den Namensbestandteilen in der Form "Nachname, Vorname", ausgeben soll.

Mit diesen Anforderungen im Hinterkopf machen wir uns nun an die Umsetzung. Dazu wollen wir die Symbole in Form von zwei Icons implementieren. Häufig verbindet man mit Icons vorgefertigte (kleine) Pixelgrafiken. Swing bietet hier viel mehr: Man kann Icons beliebiger Größe selbst zeichnen und dazu Java 2D nutzen. Bevor wird das angehen, schauen wir uns die Definition des Interface `Icon` im JDK an:

```
// Kommentierter Auszug aus dem JDK
public interface Icon
{
    // Zeichnet das Icon an der Position x,y
    void paintIcon(Component c, Graphics g, int x, int y);

    // Abmessungen des Icons
    int getIconWidth();
    int getIconHeight();
}
```

Um das Symbol für das männliche Geschlecht nachzubilden, nutzen wir neben der Kreisfigur `Ellipse2D` auch die Klasse `GeneralPath`, mit der wir einen blauen Pfeil nach rechts oben zeichnen. Das implementieren wir in der Klasse `GenderMaleIcon`:

```
public final class GenderMaleIcon implements Icon
```

```
{
    public void paintIcon(final Component component, final Graphics graphics,
                          final int x, final int y)
    {
        final Graphics2D graphics2d = (Graphics2D) graphics;

        graphics2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                                    RenderingHints.VALUE_ANTIALIAS_ON);
        graphics2d.setStroke(new BasicStroke(2));

        // Ursprung des Koordinatensystems verschieben
        // Figur kann dann bezogen auf 0,0 gemalt werden
        graphics2d.translate(x, y);

        // Schwarzer Kreis
        graphics2d.setColor(Color.BLACK);
        final Shape circle = new Ellipse2D.Float(9, 9, 14, 14);
        graphics2d.draw(circle);

        // Blauer Pfeil nach rechts oben
        graphics2d.setColor(Color.BLUE);
        final GeneralPath path = new GeneralPath();
        path.moveTo(22f, 11f);
        path.lineTo(29f, 4f);
        path.lineTo(22f, 4f);
        path.moveTo(29f, 4f);
        path.lineTo(29f, 11f);
        graphics2d.draw(path);

        // Ursprung des Koordinatensystems wieder zurück verschieben
        graphics2d.translate(-x, -y);
    }

    public int getIconWidth()
    {
        return 32;
    }

    public int getIconHeight()
    {
        return 32;
    }
}
```

Die Realisierung der Klasse `FemaleGenderIcon` wird hier nicht gezeigt, da sie zur obigen recht ähnlich ist. Dort nutzen wir lediglich andere Farben sowie zwei Linien, um ein rotes Kreuz unterhalb des Kreises zu malen.

Nachdem wir nun die Grundbausteine beisammen haben, ist die Realisierung des Renderers auf Basis der Klasse `DefaultListCellRenderer` sehr einfach. Wie bereits erwähnt, erbt diese Klasse von der Klasse `JLabel`, sodass wir dort problemlos durch entsprechende Methoden sowohl das Icon als auch die kommaseparierte Kombination aus Vor- und Nachname darstellen können. Das zeigt folgendes Listing:

```
public class PersonGenderListCellRenderer extends DefaultListCellRenderer
{
    private static final Icon maleIcon = new GenderMaleIcon();
    private static final Icon femaleIcon = new GenderFemaleIcon();
```

```
    @Override
    public Component getListCellRendererComponent(final JList<?> list,
                          final Object value, final int index,
                          final boolean isSelected, final boolean
cellHasFocus)
    {
        super.getListCellRendererComponent(list, value, index,
                                           isSelected, cellHasFocus);
        final Person person = (Person) value;
        setText(person.getLastname() + ", " + person.getFirstname());

        if (person.getGender() == Gender.MALE)
            setIcon(maleIcon)
        else if (person.getGender() == Gender.FEMALE)
            setIcon(femaleIcon);

        return this;
    }
}
```

Abschließend kombinieren wir nun das zu Beginn dieses Abschnitts implementierte Datenmodell `PersonListModel` und den eben erstellten Renderer `PersonGenderListCellRenderer` und setzen diese beiden für eine `JList<Person>` ein:

```
public static void main(String[] args)
{
    final JFrame frame = new JFrame("JListPersonRendererExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final ListModel<Person> listModel = new
PersonListModel(famousJavaPersons);
    final JList<Person> list = new JList<>(listModel);
    list.setCellRenderer(new PersonGenderListCellRenderer());

    frame.add(new JScrollPane(list), BorderLayout.CENTER);
    frame.setSize(300, 250);
    frame.setVisible(true);
}
```

Listing 9.21 *Ausführbar als* **'JLISTPERSONRENDEREREXAMPLE'**

Starten wir das Programm JLISTPERSONRENDERERXAMPLE, so erscheint eine Ausgabe ähnlich zu der in der nachfolgenden Abbildung 9-35.

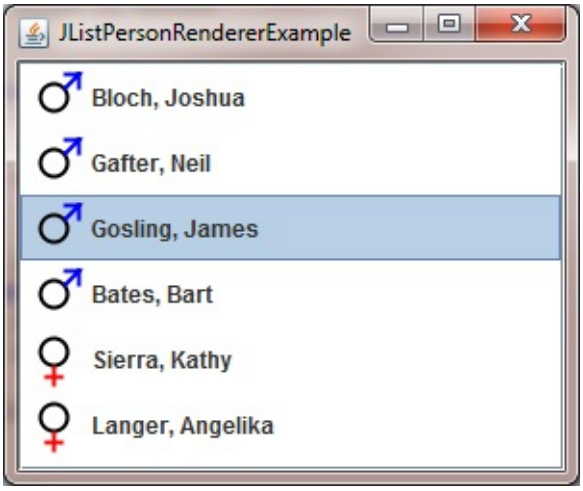


Abbildung 9-35 *Darstellung mit* `Person`*-Renderer*

Verarbeitung dynamischer Listeninhalte

Bisher haben wir die Visualisierung und Auswahl statischer Daten kennengelernt. Zur Demonstration und zum Einstieg sowie für einige Anwendungsfälle ist das vollkommen in Ordnung. Allerdings besitzen die Daten in der Praxis ein gewisse Dynamik. Damit ist gemeint, dass die Datengrundlage nicht konstant ist, also Daten neu hinzukommen, inhaltlich verändert oder auch gelöscht werden. Manchmal bleibt zwar die Zusammensetzung der Daten konstant (oder ändert sich sehr selten), aber einzelne Inhalte (Attribute) der dargestellten Objekte unterliegen fortlaufenden Wertänderungen. Beispielsweise könnte das für die Übersicht eines Aktiendepots von Privatpersonen gelten: Die Zusammensetzung mag sich selten ändern, der Wert einer Aktie variiert jedoch ständig. Was es bei dieser Form der Datenänderungen zu beachten gilt, werden wir im Folgenden betrachten. Bevor wir uns konkret mit Änderungen an der Zusammensetzung und den Inhalten der anzuzeigenden Daten auseinandersetzen, möchte ich hier nochmals auf die Single-Thread-Regel und das Ändern von Modellinhalten im EDT hinweisen.

Single-Thread-Regel und Änderungen in Modellen

Aus Abschnitt 9.2 wissen wir bereits, dass in Swing das Neuzeichnen und die Abarbeitung der CallbackMethoden von Event Listenern standardmäßig im EDT stattfinden.[13] Ich hatte empfohlen, länger dauernde Berechnungen in eigene Threads auszulagern, um die Ereignisverarbeitung im GUI nicht zu blockieren. Als Folge müssen dann die Berechnungsergebnisse der Worker-Threads wieder Thread-sicher durch einen Aufruf von `invokeLater(Runnable)` zurück in das GUI kommuniziert werden. Nur diese Ausführung im EDT vermeidet Darstellungsprobleme oder Exceptions. Wie kann es dazu kommen? Da diese Thematik leider in der Einsteigerliteratur eher stiefmütterlich behandelt wird, möchte ich nun darauf eingehen, um Schwierigkeiten vorzubeugen.

Schauen wir uns den Ablauf beim Zeichnen einer `JList<E>` an. Nehmen wir weiter an, während der Ausführung des Zeichnens würden Elemente aus dem `ListModel<E>` gelöscht und Daten bereits gezeichneter Einträge verändert. Zum Zeichnen wird aber vereinfacht folgender Algorithmus abgearbeitet:

```
// Achtung: Nur Pseudocode!
int entryCount = listModel.getSize();
for (int i=0; i< entryCount; i++)
{
    E entry = listModel.getElementAt(i);

    drawEntryWithRenderer(entry);
}
```

Mit dem Wissen über Multithreading aus Kapitel 7 erkennen wir folgendes Problem: Der Zugriff über `getElementAt(int)` ist nicht Thread-sicher, weil es nach dem

Ermitteln des Wertebereichs des Index (`getSize()`) nebenläufig zu Datenänderungen kommen kann, sodass die durch den Index beschriebene Position ungültig wird und so eine `IndexOutOfBoundsException` ausgelöst wird. Das lässt sich einfach dadurch beheben, dass man Datenänderungen an Modellen ausschließlich im EDT erledigt.

Änderungen in Modellen vom Typ `DefaultListModel`

Solange man sich an die Single-Thread-Regel hält, werden Änderungen im `DefaultListModel<E>` automatisch im View reflektiert. Dafür sind eine Vielzahl an Methoden definiert, die der Schnittstelle der Klasse `Vector<E>` nachempfunden sind. Der Nachteil besteht bei einer solch breiten Schnittstelle darin, dass es bei einer großen Anzahl an Methoden schwierig sicherzustellen ist, dass alle Aufrufe, die Daten verändern, auch wirklich im EDT erfolgen und somit die Single-Thread-Regel immer eingehalten wird.

Es empfiehlt sich daher häufig, eigene Implementierungen von Modellen auf Basis der Klasse `AbstractListModel<E>` zu erstellen, die nur einige wenige Methoden anbieten, die Daten modifizieren. Das wollen wir nun besprechen.

Änderungen in Modellen vom Typ `AbstractListModel`

Das `AbstractListModel<E>` definiert lediglich Lesezugriff auf die Daten. Allerdings ist diese Basisklasse schon darauf vorbereitet, Subklassen bei der Verarbeitung von Datenänderungen zu unterstützen. Dazu gehört die Verwaltung von Event Listenern sowie die Erzeugung von Änderungsmitteilungen. Für Änderungen in der Zusammensetzung der Daten einer `JList<E>` (neue, inhaltlich geänderte oder gelöschte Einträge) sowie für Veränderungen an den Einträgen selbst muss man die Single-Thread-Regel beachten.

Zwei typische Arten von Änderungen findet man in der Praxis: Zum einen kann man Daten z. B. über einen Add-Button aus dem GUI heraus modifizieren. Zum anderen wird man den Datenbestand des GUIs mit einem externen Datenbestand etwa einer Datenbank abgleichen wollen, sofern sich dort Änderungen ergeben haben. Was ist für diese beiden Fälle zu beachten? Wenn im ersten Fall Änderungen des Modells innerhalb der Ereignisverarbeitung (die bekanntlich im EDT abläuft) stattfinden, müssen diese nur noch an den View propagiert werden. Die dazu notwendigen Benachrichtigungsmethoden lernen wir in Kürze kennen. Erfolgen dagegen Änderungen durch andere Threads, also außerhalb des EDTs, so muss man für einen korrekten Abgleich der Daten im EDT selbst sorgen. Beispielsweise könnte der in einer Liste anzuzeigende Inhalt in einem Worker-Thread per Datenbankabfrage bestimmt werden. Um der Single-Thread-Regel zu genügen, muss man den geänderten Inhalt anschließend mithilfe von `invokeLater(Runnable)` im EDT in das GUI propagieren.

Das folgende Listing zeigt ein Listenmodell, das beide Arten von Änderungen

unterstützt: Die generische Klasse `DynamicListModel<E>` erlaubt sowohl das Hinzufügen neuer Elemente über `addEntry(E)` als auch den Austausch des gesamten Datenbestands durch `setNewContent(List<E>)`. Darüber hinaus bieten wir die Methode `rowContentChanged(int, int)`, die immer dann im EDT aufgerufen werden sollte, wenn sich an Einträgen etwas geändert hat, um dies im View zu reflektieren.

```
public class DynamicListModel<E> extends AbstractListModel<E>
{
    private final List<E> entries;

    public DynamicListModel(final List<E> entries)
    {
        this.entries = new ArrayList<>(entries);
    }

    @Override
    public int getSize()
    {
        return entries.size();
    }

    @Override
    public E getElementAt(final int index)
    {
        return entries.get(index);
    }

    public void addEntry(final E newEntry)
    {
        entries.add(newEntry);
        fireIntervalAdded(this, getSize() - 1, getSize() - 1);
    }

    public void setNewContent(final List<E> newEntries)
    {
        entries.clear();
        entries.addAll(newEntries);
        fireContentsChanged(this, 0, getSize()-1);
    }

    public void rowContentChanged(final int startIndex, final int endIndex)
    {
        fireContentsChanged(this, startIndex, endIndex);
    }
}
```

Kommunikation von Änderungen

Um den View auch für umfangreiche Datenbestände performant aktualisieren zu können, sollte man bei Änderungen möglichst präzise kommunizieren, welche Einträge sich tatsächlich geändert haben. Dazu kann man jeweils einen Wertebereich für die im Listing genutzten Methoden `fireIntervalAdded(Object, int, int)` und `fireContentsChanged(Object, int, int)` angeben. Erstere signalisiert neue Zeilen und letztere eine Änderung der Daten in einem Intervall. Vermeiden Sie aus Bequemlichkeit statt die tatsächlichen Intervallgrenzen einfach

den Bereich 0 bis `getSize()-1` anzugeben, denn das löst das Neuzeichnen des gesamten Views aus. Das ist nur dann sinnvoll, wenn – wie im obigen Fall der Methode `setNewContent(List<Person>)` – der gesamte Inhalt ausgetauscht wird. Ein solcher Fauxpas bei der Bereichsangabe wird sich für kleinere Datenmengen und wenige Änderungen kaum als störend bemerkbar machen. Das gilt jedoch nicht mehr, wenn das Datenvolumen oder die Häufigkeit von Änderungen stark zunehmen.

Protokollierung der Änderungen

Wir wollen eine Protokollierung der Änderungen am Listenmodell realisieren. Dazu können wir den Typ `ListDataListener` aus dem JDK nutzen, der folgendermaßen definiert ist:

```
// Auszug aus dem JDK
public interface ListDataListener extends EventListener
{
    void intervalAdded(ListDataEvent event);
    void intervalRemoved(ListDataEvent event);
    void contentsChanged(ListDataEvent event);
}
```

Für dieses Interface erstellen wir die Klasse `SimpleListModelModificationReporter`, die Änderungen durch entsprechende Konsolenausgaben protokolliert:

```
public class SimpleListModelModificationReporter implements ListDataListener
{
    @Override
    public void intervalAdded(final ListDataEvent event)
    {
        System.out.println("intervalAdded(ListDataEvent)\n" + event +"\n");
    }

    @Override
    public void intervalRemoved(final ListDataEvent event)
    {
        System.out.println("intervalRemoved(ListDataEvent)\n" + event +"\n");
    }

    @Override
    public void contentsChanged(final ListDataEvent event)
    {
        System.out.println("contentsChanged(ListDataEvent)\n" + event +"\n");
    }
}
```

Um bei Änderungen entsprechende Rückmeldungen zu erhalten, registrieren wir eine Instanz der obigen Klasse beim Listenmodell wie folgt:

```
listModel.addListDataListener(new SimpleListModelModificationReporter());
```

Beispiel

Im folgenden Listing werden die zuvor erstellten Bausteine kombiniert. Als

Datengrundlage dient eine `List<Person>`, die wir parallel zum EDT im `main()`-Thread aktualisieren. Den aktuellen Stand kann man durch einen Refresh-Button in die `JList<Person>` übernehmen. Außerdem ist es durch Betätigen eines Add-Buttons möglich, neue Einträge in das `DynamicListModel<Person>` einzufügen.

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final List<Person> persons = DataDefinitions.famousJavaPersons;
    final DynamicListModel<Person> listModel = new DynamicListModel<>
(persons);
    listModel.addListDataListener(new SimpleListModelModificationReporter());

    final JList<Person> list = new JList<>(listModel);
    list.setCellRenderer(new PersonGenderListCellRenderer());

    final JButton refreshButton = new JButton("Refresh");
    initRefreshListener(refreshButton, persons, listModel);
    frame.add(refreshButton, BorderLayout.NORTH);

    frame.add(new JScrollPane(list), BorderLayout.CENTER);

    final JButton addEntryButton = new JButton("Add Entry");
    initAddEntryListener(addEntryButton, list, listModel);
    frame.add(addEntryButton, BorderLayout.SOUTH);

    frame.setSize(400, 200);
    frame.setVisible(true);

    // Inhaltsänderung parallel zum GUI: Also NICHT im EDT
    System.out.println("Changing Joshua Blochs name");
    persons.get(0).setLastname("Bloch – The Java Guru");
    SwingUtilities.invokeLater(new Runnable()
    {
        public void run()  { listModel.rowContentsChanged(0, 0); }
    });

    // Änderungen der Ausgangsdaten parallel zum GUI: Also NICHT im EDT
    for (int i = 0; i < 20; i++)
    {
        persons.add(new Person("New ", "Entry " + i, Gender.MALE));
        SleepUtils.safeSleep(2000);
    }
}

private static void initRefreshListener(final JButton refreshButton,
           final List<Person> persons, final DynamicListModel<Person>
listModel)
{
    refreshButton.addActionListener(new ActionListener()
    {
        @Override
        public void actionPerformed(final ActionEvent event)
        {
            System.out.println("Refresh: Resetting business model content");
            listModel.setNewContent(persons);
        }
    });
```

```
}
private static void initAddEntryListener(final JButton addEntryButton,
            final JList<Person> list, final DynamicListModel<Person> listModel)
{
    addEntryButton.addActionListener(new ActionListener()
    {
        @Override
        public void actionPerformed(final ActionEvent event)
        {
            System.out.println("Adding new entry");
            listModel.addEntry(new Person("Misses", "Test", Gender.FEMALE));
            // Neues Element sichtbar machen, evtl. selektieren
            // list.setSelectedIndex(listModel.getSize()-1);
            list.ensureIndexIsVisible(listModel.getSize() - 1);
        }
    });
}
}
```

Listing 9.22 *Ausführbar als* **'JLISTMODIFICATIONEXAMPLE'**

Dieses Progamm demonstriert die bereits angesprochenen zwei typischen Anwendungsfälle aus der Praxis. Zum einen bieten wir einen Add-Button an, der durch einen `ActionListener` bearbeitet wird. Dessen Methode `actionPerformed(ActionEvent)` wird bekanntermaßen immer im EDT ausgeführt und kann somit Thread-sicher neue Einträge erstellen. In diesem Zusammenhang sehen wir den Aufruf der noch unbekannten Methode `ensureIndexIsVisible(int)`. Diese sorgt dafür, dass der neu erzeugte Listeneintrag sichtbar wird, d. h., dass bei größeren Datenmengen der Listeninhalt dementsprechend verschoben wird. Zusätzlich ist es gegebenenfalls gewünscht, einen neu erzeugten Eintrag zu selektieren. Der dafür notwendige Aufruf von `setSelectedIndex(int)` ist im obigen Listing auskommentiert.

Zum anderen wird man den Datenbestand des GUIs mit einem externen Datenbestand abgleichen wollen. Als Simulation eines externen Datenbestands und dessen Variabilität führen wir am Ende der `main()`-Methode einige Änderungen bewusst parallel zum EDT durch. Mithilfe eines Refresh-Buttons lassen sich diese dann Thread-sicher ins GUI übernehmen. Eine Ausführung des Progamms JLISTMODIFICATIONEXAMPLE zeigt Abbildung 9-36.

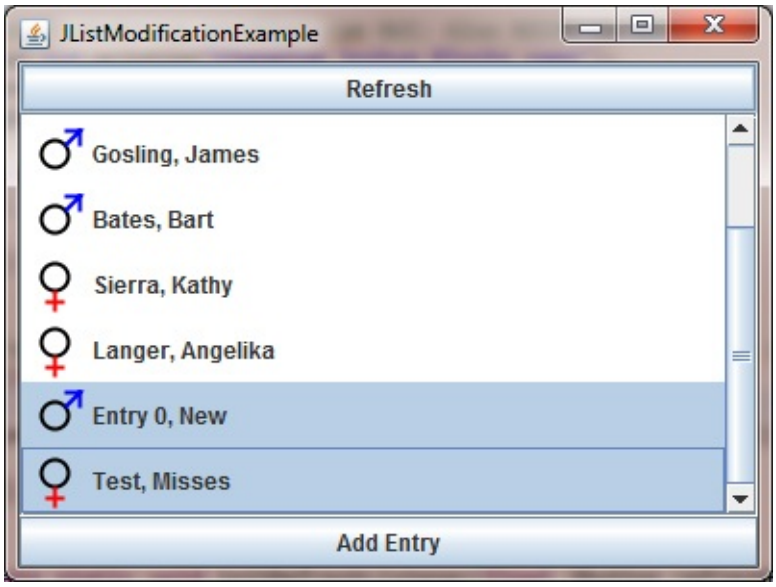


Abbildung 9-36 *Beispiel zu Modifikationen in einer* `JList<Person>`

Alternative: Austausch des Modells

Eine Variante beim Umgang mit Änderungen besteht darin, das Modell immer vollständig neu aufzubauen, dann ein neues `ListModel<E>` zu erzeugen und dieses per `setModel( ListModel<E>)` an die `JList<E>` zu übergeben. Dieses Vorgehen besitzt jedoch den Nachteil, dass diverse Einstellungen und auch die Selektion verloren gehen. Das ist extrem störend, wenn dies im Rahmen einer automatischen Aktualisierung etwa alle 30 Sekunden abläuft.

Hintergrundinfo: Speicherebenen der im GUI sichtbaren Daten

Die ganze Sache der Datenbereitstellung ist eigentlich noch komplexer. Die im GUI angezeigten Daten entstammen in der Regel einer Datenbasis (etwa einer Datenbank), die wiederum in Form von Datenstrukturen (z. B. Listen) ein Business-Modell bilden. Beide stellen jeweils Speicherebenen mit unterschiedlichen Aktualisierungszyklen und Formen der Parallelität dar, wie ich es gleich näher erläutere.

Darüber hinaus können sich beide Datenbestände unabhängig von den Daten des GUI-Modells (z. B. einem `ListModel<E>`) ändern. Auch hier wird ein Abgleich benötigt, spätestens dann, wenn im GUI geänderte Daten in das Business-Modell oder die Datenbasis übertragen werden sollen. Des Weiteren ist häufig auch der umgekehrte Weg gewünscht: Das GUI soll Änderungen im Business-Modell bzw. der Datenbasis reflektieren. Dazu kann etwa ein periodischer oder manueller Abgleich der Daten mit dem GUI-Modell erfolgen.

Kommen wir aber zu den Speicherebenen zurück. In der Computerarchitektur unterscheidet man z. B. Prozessorregister und -Caches, Hauptspeicher und Festplatten. Vergleichbare Ebenen kann man auch für GUI-Daten unterscheiden (in Klammern sind vergleichbare Ebenen aus der Computerarchitektur angegeben):

- Renderer (Prozessorregister)

- GUI-Modell (Prozessor-Cache)
- Business-Modell (Hauptspeicher)
- Datenbasis/-bank (Festplatte)

Die beiden Ebenen GUI-Modell und Business-Modell haben wir in den vorherigen Beispielen immer gesehen, ohne dass ich darauf explizit eingegangen bin. Nachfolgend ist dies angedeutet:

```
// Business-Modell
final List<Person> persons = DataDefinitions.famousJavaPersons;

// GUI-Modell
final DynamicListModel<Person> listModel =
                                new DynamicListModel<>(persons);
```

9.4.3 Die Klasse `JTable`

Zur Visualisierung und Verarbeitung komplexerer Datenbestände bietet sich oftmals eine tabellarische Anzeige an, d. h. eine Darstellung mit mehreren Zeilen und Spalten. Das gilt eigentlich auch schon für das vorherige Beispiel der Liste von `Person`-Objekten: Die Anzeige der verschiedenen Attribute eines `Person`-Objekts war in einer `JList<Person>` nur durch die Implementierung eines eigenen Renderers möglich. In der Regel wird man aber verschiedene Attribute eines Objekts nicht nur darstellen, sondern auch bearbeiten wollen: Das erfordert beim Einsatz einer `JList<Person>` einigen Zusatzaufwand. Schnell kommt der Wunsch auf, Attribute durch eigene Spalten zu repräsentieren, weil dadurch auf einfache Weise eine Bearbeitung der Inhalte sowie eine Sortierung oder Filterung nach einem einzelnen Attribut möglich sind. Beim Einsatz der Klasse `JTable` werden unter anderem folgende Annehmlichkeiten geboten:

- Spalten besitzen Überschriften, die anders dargestellt werden als Zellen. Durch Klick auf eine Spaltenüberschrift lässt sich eine Sortierung ausführen.
- Spalten können in der Breite angepasst und beliebig durch Drag-and-Drop umgeordnet werden.
- Falls aktiviert, können Werte in den Zellen editiert werden.

Interfaces und Klassen im Zusammenspiel mit einer `JTable`

Analog zur `JList<E>` delegiert auch die `JTable` diverse Aufgaben an weitere Klassen: Das Zeichnen einzelner Zellen wird durch einen Renderer vom Typ `TableCellRenderer` erledigt. Die anzuzeigenden Daten werden durch ein `TableModel` bereitgestellt. Die Speicherung von Spalten und deren Überschriften erfolgt in einem `TableColumnModel`. Selektionen von Zeilen werden in Tabellen – ebenso wie für Listen – durch ein `ListSelectionModel` repräsentiert. Abbildung 9-37 bettet die genannten Interfaces und Klassen in einen größeren Kontext ein.

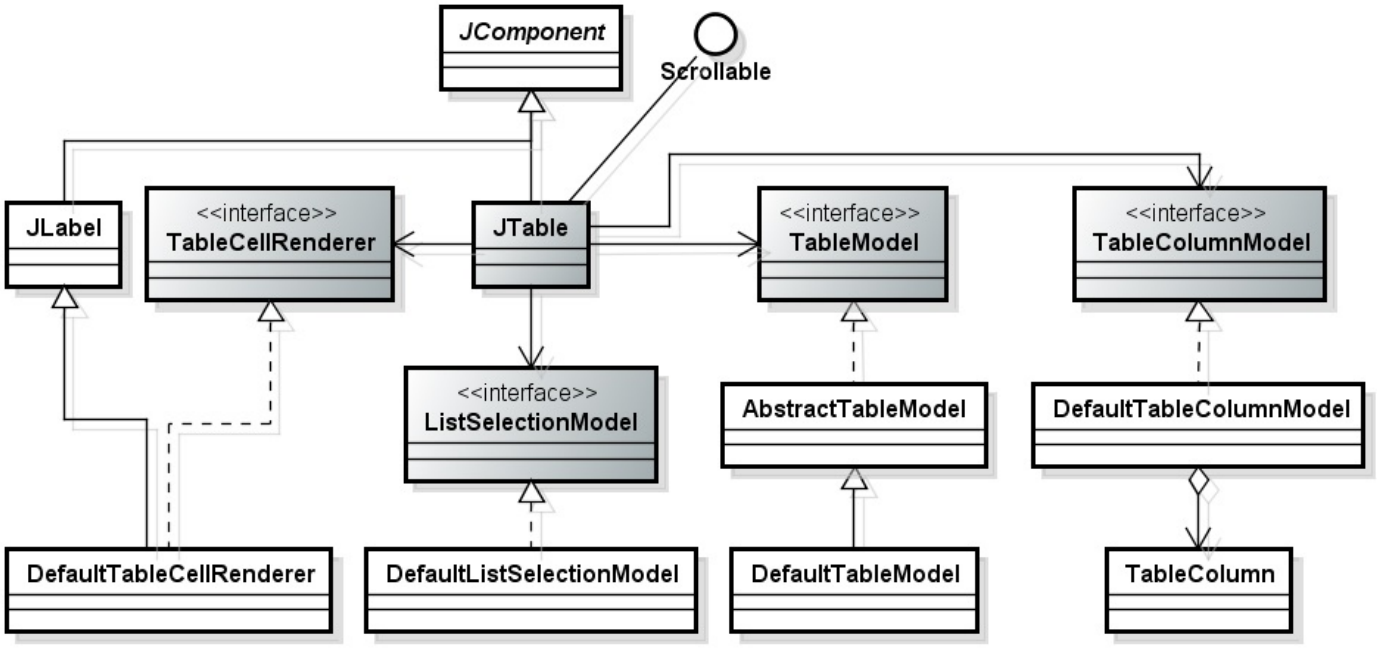


Abbildung 9-37 *Klassendiagramm* `JTable`

Selektionen

Wie schon erwähnt, werden die in einer `JTable` gewählten Einträge durch eine Instanz eines `ListSelectionModel`s verwaltet, das bereits im Zusammenhang mit der `JList<E>` in Abschnitt 9.4.2 besprochen wurde. Daher verweise ich für die Verarbeitung von Selektionen in Tabellen auf diesen Abschnitt und gehe im Folgenden nicht weiter darauf ein. Stattdessen beginne ich die Beschreibung der Klasse `JTable` und der unterstützenden Klassen und Interfaces mit dem `TableModel`.

Datenmodell

Den in einer `JTable` angezeigten Daten liegt ein Tabellenmodell zugrunde, das das Interface `TableModel` aus dem JDK erfüllt. Dieses definiert folgende Methoden:

```
// Kommentierter Auszug aus dem JDK
public interface TableModel
{
    // Liefert die Anzahl der Zeilen.
    public int getRowCount();

    // Liefert die Anzahl der Spalten.
    public int getColumnCount();

    // Gibt den Namen der Spalte zurück.
    public String getColumnName(int columnIndex);

    // Liefert den Typ der Werte einer Spalte.
    public Class<?> getColumnClass(int columnIndex);

    // Bieten Lese-/Schreib-Zugriff auf einzelne Zellen.
    public Object getValueAt(int rowIndex, int columnIndex);
```

```
    public void setValueAt(Object aValue, int rowIndex, int columnIndex);

    // Fragt ab, ob eine bestimmte Zelle der Tabelle editierbar ist.
    public boolean isCellEditable(int rowIndex, int columnIndex);

    // Verarbeiten von Benachrichtigungen bei Änderungen im Modell.
    public void addTableModelListener(TableModelListener listener);
    public void removeTableModelListener(TableModelListener listener);
}
```

Die Methoden des `TableModel`s werden von der `JTable` aufgerufen, etwa um die Anzahl an Spalten und Zeilen zu ermitteln oder Daten zu lesen bzw. zu setzen. Zudem registriert sich die `JTable` als Interessent vom Typ `TableModelListener`, damit sie über Veränderungen informiert wird, wenn beispielsweise Zeilen hinzukommen, entfernt werden oder sich deren Daten ändern. Um diese Modifikationen auch im View zu reflektieren, müssen die gesamte Tabelle oder Teile davon neu gezeichnet werden.

Realisierung eigener Tabellenmodelle

Zur Implementierung eigener Tabellenmodelle kann man auf die vorgefertigten Implementierungen des JDKs zurückgreifen: Man findet dort die abstrakte Basisklasse `AbstractTableModel` und die konkrete Klasse `DefaultTableModel`. Bei letzterer ist man allerdings für die Datentypen auf `Object[][]` bzw. `Vector<Object>`, der als Elemente wiederum `Vector<Object>` enthält, eingeschränkt. Daher basieren eigene Tabellenmodelle normalerweise auf der Klasse `AbstractTableModel`, in der nur noch die folgenden drei abstrakten Methoden implementiert werden müssen:

- `getColumnCount()` – Anzahl der Spalten
- `getRowCount()` – Anzahl der Zeilen
- `getValueAt(int, int)` – Wert einer Zelle

Als Eingabedaten soll – wie schon zuvor für das `DynamicListModel<E>` – eine `List<Person>` dienen. Wiederum müssen wir die Methoden des Tabellenmodells so realisieren, dass diese die Methoden der `List<Person>` bzw. einzelner `Person`-Objekte aufrufen: Dabei werden die Zeilen durch die Einträge der `List<Person>` vorgegeben und die Spalten entsprechen den Attributen eines `Person`-Objekts. Mit diesem Wissen implementieren wir die Klasse `PersonTableModel` folgendermaßen:

```
public class PersonTableModel extends AbstractTableModel
{
    // Namen der Spalten
    private static final String[] COLUMN_NAMES = { "Firstname", "Name",
                                                   "Gender" };
    // Indizes der Spalten
    private static final int COLUMN_IDX_FIRSTNAME = 0;
    private static final int COLUMN_IDX_NAME = 1;
    private static final int COLUMN_IDX_GENDER = 2;
```

```
    // Datenspeicherung
    private final List<Person> persons;

    public PersonTableModel(final List<Person> persons)
    {
        this.persons = new ArrayList<>(persons);
    }

    @Override
    public int getColumnCount()
    {
        return COLUMN_NAMES.length;
    }

    @Override
    public int getRowCount()
    {
        return persons.size();
    }

    @Override
    public Object getValueAt(final int rowIndex, final int columnIndex)
    {
        final Person person = persons.get(rowIndex);

        // Abbildung der Spalten auf Attribute
        if (columnIndex == COLUMN_IDX_FIRSTNAME)
            return person.getFirstname();
        if (columnIndex == COLUMN_IDX_NAME)
            return person.getLastname();

        if (columnIndex == COLUMN_IDX_GENDER)
            return person.getGender();

        throw new IllegalArgumentException("Invalid columIndex " +
columnIndex);
    }
}
```

Zunächst erfolgt die Definition eines `String[] COLUMN_NAMES`. Daraus ergibt sich die Anzahl der Spalten sowie deren Namen. Diese Informationen nutzen wir in der Methode `getColumnCount()` zur Rückgabe der Spaltenanzahl sowie bei der Realisierung der Methode `getColumnName(int)`, um sinnvolle Spaltennamen zurückliefern zu können. Die Anzahl der Zeilen bestimmt die Methode `getRowCount()` basierend auf den Eingabedaten. Die Werte einzelner Zellen werden durch die Methode `getValueAt(int, int)` bereitgestellt. Die Wertermittlung erfolgt zweistufig: Als Erstes erfolgt ein indizierter Zugriff auf die `List<Person>` und als Zweites werden die zuvor als Konstanten definierten Spaltenindizes zum Zugriff auf die korrespondierenden Attribute eines `Person`-Objekts benutzt.

Die Basisklasse `AbstractTableModel` liefert standardmäßig fortlaufende Buchstabenkennungen ('A', 'B', …) als Spaltenüberschriften, daher sollte man die Methode `getColumnName(int)` überschreiben und wie folgt selbst implementieren:

```
@Override
```

```
public String getColumnName(final int columIndex)
{
    return COLUMN_NAMES[columIndex];
}
```

Schauen wir nun, wie wir für das eben erstellte `PersonTableModel` die Berühmtheiten aus dem Java-Umfeld als Eingabedaten in Form einer `List<Person>` nutzen und in einer `JTable` anzeigen können:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("JTableExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final List<Person> persons = DataDefinitions.famousJavaPersons;
    final TableModel tableModel = new PersonTableModel(persons);
    final JTable table = new JTable(tableModel);

    frame.add(new JScrollPane(table), BorderLayout.CENTER);
    frame.setSize(500, 150);
    frame.setVisible(true);
}
```

Listing 9.23 *Ausführbar als* **'JTABLEEXAMPLE'**

Abbildung 9-38 zeigt eine Ausführung des Programms JTABLEEXAMPLE. In der dort gezeigten Tabelle fallen zwei Dinge (negativ) auf: Zum einen sind die Spalten alle gleich breit. Zum anderen wird das Attribut `gender`, basierend auf `toString()`, textuell angezeigt. Das ist störend, weil wir für Listen schon eine deutlich ansprechendere Darstellung realisiert haben. Das wollen wir gleich verbessern. Zuvor gehe ich noch kurz darauf ein, wie man bei Bedarf die Spaltenbreiten anpassen kann.

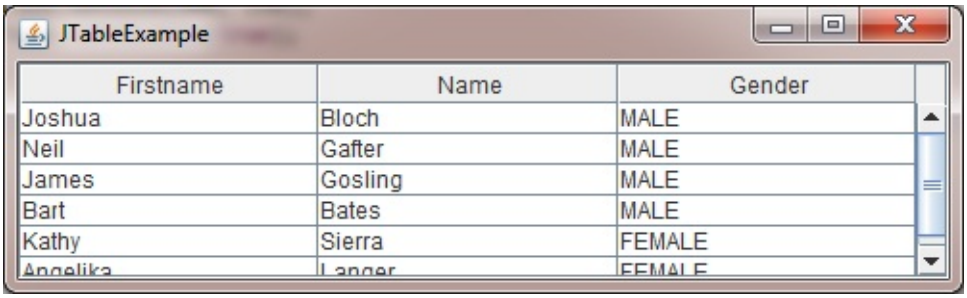


Abbildung 9-38 *Screenshot des Programms* JTABLEEXAMPLE

Anmerkung: Besonderheiten bei Spaltenüberschriften

Es werden nur dann Spaltenüberschriften für eine `JTable` dargestellt, wenn diese in eine `JScrollPane` integriert wird. Das Verhalten ist insofern merkwürdig, als die Überschriften der `JTable` zugeordnet sind und nicht der `JScrollPane`.

Anpassungen der Spalten im `TableColumnModel`

Die Breite und Anordnung von Spalten einer Tabelle lassen sich im Nachhinein durch den Benutzer ändern, jedoch ist es häufig wünschenswert, gewisse Spaltenbreiten im Vorhinein einzustellen. Dazu muss man auf einzelne Spalten zugreifen können. Wie eingangs erwähnt, werden die Spalten in einer `JTable` durch

ein Spaltenmodell vom Typ `TableColumnModel` repräsentiert. Dieses wird automatisch von der `JTable` ohne unser Zutun erzeugt und durch Aufruf der Methode `getColumnModel()` zurückgeliefert. Ein `TableColumnModel` speichert Spalten als `TableColumn`-Objekte, die sich über `getColumn(int)` ermitteln lassen.

Wenn die Namensspalten 150 bzw. 200 Pixel breit werden sollen und die Spalte für das Geschlecht 75 Pixel, dann kann man dafür folgende Methode implementieren:

```
private static void initColumnWidths(final JTable table)
{
    final TableColumnModel tableColumnModel = table.getColumnModel();

    // Aus Gründen der Lesbarkeit im Buch verwende ich hier fixe Zahlenwerte!
    // Ansonsten für den Index die Konstanten aus dem PersonTableModel nutzen.
    tableColumnModel.getColumn(0).setPreferredWidth(150);
    tableColumnModel.getColumn(1).setPreferredWidth(200);
    tableColumnModel.getColumn(2).setPreferredWidth(75);
}
```

Die gezeigte Methode können wir in der `main()`-Methode aufrufen, um die Spaltenbreiten zu setzen. Wenn Sie die Änderungen in Aktion erleben wollen, dann starten Sie bitte das Programm JTABLECOLUMNSIZINGEXAMPLE.

Renderer

Ebenso wie für Listen kann man die Darstellung von Daten in Tabellen für einzelne Spalten mithilfe von Renderern anpassen. Das ist immer dann notwendig, wenn die durch den Default-Renderer bereitgestellte, auf `toString()` basierende textuelle Darstellung nicht ausreichend ist, was für nahezu alle Klassen gilt.

Daher haben wir zur grafischen Darstellung des Geschlechts (♂, ♀) in Listen einen eigenen Renderer in Form der Klasse `PersonGenderListCellRenderer` entwickelt. Diese können wir für die `JTable` nicht direkt verwenden, da sie vom Interface her explizit für Listen, nicht aber für Tabellen einsetzbar ist. Nichtsdestotrotz nutzen wir die zugrunde liegende Idee und implementieren folgenden eigenen Renderer:

```
public class GenderTableCellRenderer extends DefaultTableCellRenderer
{
    private static final Icon maleIcon = new GenderMaleIcon();
    private static final Icon femaleIcon = new GenderFemaleIcon();

    @Override
    public Component getTableCellRendererComponent(final JTable table,
                        final Object value, final boolean isSelected,
                        final boolean hasFocus, final int row, final int
column)
    {
        super.getTableCellRendererComponent(table, value, isSelected,
                                            hasFocus, row, column);

        final Gender gender = (Gender) value;
        if (gender == Gender.MALE)
```

```
            setIcon(maleIcon);
        else if (gender == Gender.FEMALE)
            setIcon(femaleIcon);

        // Anpassung der Zeilenhöhe an die Grafik mit 5 Pixel oben und unten
        table.setRowHeight(getIcon().getIconHeight() + 10);

        return this;
    }
}
```

Dieser Renderer basiert auf der Klasse `DefaultTableCellRenderer`, die wiederum von der Klasse `JLabel` erbt. Somit kann man auch hier wieder die Methode `setIcon(Icon)` aufrufen, um das passende grafische Symbol zu setzen. Erwähnenswert ist an dieser Implementierung außerdem, dass wir für jede Zeile deren Höhe durch einen Aufruf von `setRowHeight(int)` festlegen, sodass genug Platz für die Icons ist.[14] Diese Anpassung wird notwendig, weil Tabellenzeilen standardmäßig eine fixe Höhe von 16 Pixeln besitzen und diese sich weder am Zelleninhalt noch am gewählten Zeichensatz orientiert. Dadurch würden die Icons abgeschnitten dargestellt.

Um unseren `GenderTableCellRenderer` für Zellen vom Typ `Gender` bei der `JTable` zu registrieren, rufen wir `setDefaultRenderer(Class<?>, TableCellRenderer)` in der `main()`-Methode auf:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("JTableGenderRendererExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final List<Person> persons = DataDefinitions.famousJavaPersons;
    final TableModel tableModel = new PersonTableModel(persons);
    final JTable table = new JTable(tableModel);
    // Spaltenbreiten anpassen
    initColumnWidths(table);

    // Renderer registrieren
    table.setDefaultRenderer(Gender.class, new GenderTableCellRenderer());

    frame.add(new JScrollPane(table), BorderLayout.CENTER);
    frame.setSize(500, 150);
    frame.setVisible(true);
}
```

Listing 9.24 *Ausführbar als* **'JTABLEGENDERRENDERER EXAMPLE'**

Probieren wir das Ganze aus und starten dazu das Programm JTABLEGENDERRENDERER EXAMPLE. Wir erleben eine Überraschung! Trotz der Registrierung des Renderers erhalten wir wider Erwarten immer noch eine textuelle Darstellung des Geschlechts (vgl. Abbildung 9-39). Wie kann das sein?

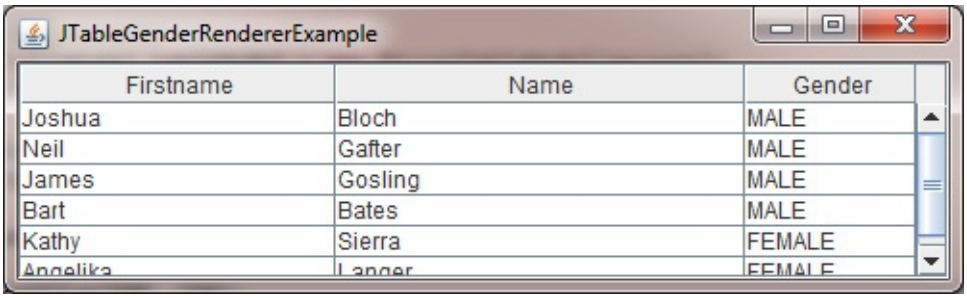


Abbildung 9-39 *Unerwartete Darstellung trotz speziellem Renderer*

Zur Erklärung kehren wir gedanklich zur `JList<E>` zurück. Dort war das Registrieren eines Renderers ausreichend, um die Darstellung der einzelnen Einträge anzupassen, weil alle vom gleichen (Basis-)Typ `E` sind. Im Gegensatz dazu besitzen die Spalten einer Tabelle aber beliebige und zum Teil unterschiedliche Typen. Wenn wir nun genauer auf die Registrierung des Renderers im Listing schauen, bemerken wir, dass dabei zwar der Typ `Gender` angegeben wird, aber unser `GenderTableCellRenderer` trotzdem nicht genutzt wird. Stattdessen kommt der Default-Renderer zum Einsatz, der beliebige Objekte mithilfe deren Stringrepräsentation anzeigt.

Das liegt am Rückgabewert der Methode `getColumnClass(int)`, die für jede Spalte den dort dargestellten Typ spezifiziert und darüber steuert, welcher Renderer für eine Spalte verwendet wird.[15] Die Basisklasse `AbstractTableModel` ist aber so implementiert, dass sie möglichst viele Anwendungsfälle gut abdeckt. Dazu gehört insbesondere auch, beliebige Werte darstellen zu können. Daher liefert die Methode `getColumnClass(int)` den Wert `Object.class` zurück![16] Das führt wiederum dazu, dass kein spezieller Renderer gewählt wird, sondern alle Spalten durch den Default-Renderer textuell dargestellt werden.

Um dieses Verhalten anzupassen, muss die Methode `getColumnClass(int)` so überschrieben werden, dass diese den tatsächlichen Typ der in einer Spalte darzustellenden Werte zurückliefert. Dazu ergänzen wir folgende Definition und Methode in unserem Tabellenmodell:

```
public class PersonTableModel extends AbstractTableModel
{
    // Definition der Spaltentypen
    private static final Class<?>[] COLUMN_CLASSES = { String.class,
                                        String.class, Gender.class };
    PersonTableModel(final List<Person> persons)
    {
        this.persons = new ArrayList<>(persons);
    }

    // …

    // Ermittlung der Spaltentypen
    @Override
    public Class<?> getColumnClass(final int columnIndex)
    {
        // columnIndex == 2 => Gender.class
        return COLUMN_CLASSES[columnIndex];
    }
```

```
}
```

Neben der obigen Erweiterung des `PersonTableModels` muss eine typabhängige Registrierung des Renderers für das `Gender`-Attribut zur korrekten Anzeige erfolgen:

```
// Renderer für den Typ Gender registrieren
table.setDefaultRenderer(Gender.class, new GenderTableCellRenderer());
```

Wenn Sie das korrespondierende Programm JTABLEGENDERRENDEREREXAMPLE2 starten, erfolgt eine Ausgabe wie in Abbildung 9-40.

Sortierbarkeit mit `TableRowSorter`

Wenn man vor JDK 6 eine Sortierung einer Tabelle vornehmen wollte, so musste man dies aufwendig selbst programmieren oder eine Bibliothek nutzen. Das Ganze kann man sich jetzt zum Glück sparen, denn mit JDK 6 wird das Sortieren von in einer `JTable` dargestellten Daten durch die Klasse `javax.swing.table.TableRowSorter<M extends TableModel>` direkt unterstützt. Praktischerweise ist dazu sehr wenig Aufwand seitens der Anwendung notwendig. Teilweise reicht es schon aus, die `JTable` anzuweisen, eine Instanz von `TableRowSorter<M extends TableModel>` durch folgenden Aufruf erzeugen zu lassen:

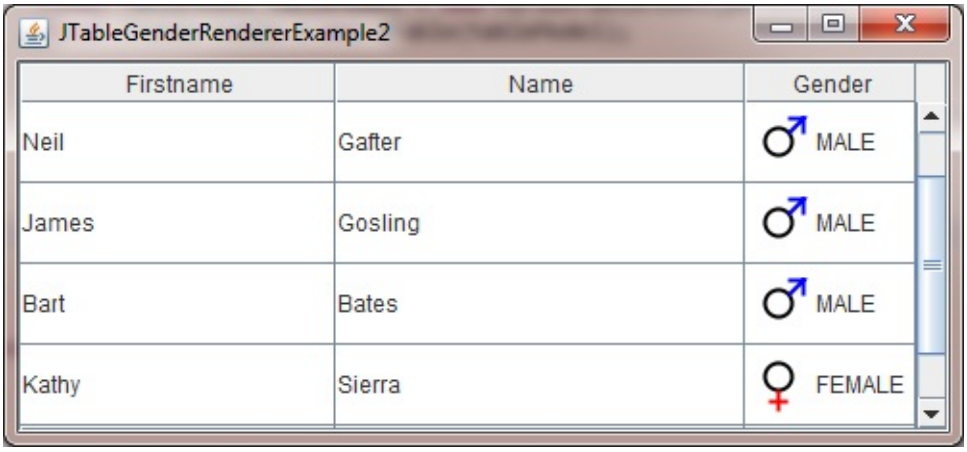


Abbildung 9-40 *Korrekte Darstellung mit* `Person`*-Renderer*

```
table.setAutoCreateRowSorter(true);
```

Der dadurch automatisch generierte `TableRowSorter<M extends TableModel>` arbeitet genauso, wie man es erwartet: Bei einem Klick auf eine Spaltenüberschrift werden die Werte der entsprechenden Spalte sortiert und die Sortierreihenfolge wird durch einen kleinen Pfeil in der Spaltenüberschrift angezeigt.

Führen Sie das Programm JTABLESORTINGEXAMPLE aus, so erhalten Sie eine Darstellung analog zu der in Abbildung 9-41, wenn Sie die Einträge per Klick in die Spaltenüberschrift absteigend nach deren Geschlecht sortieren.

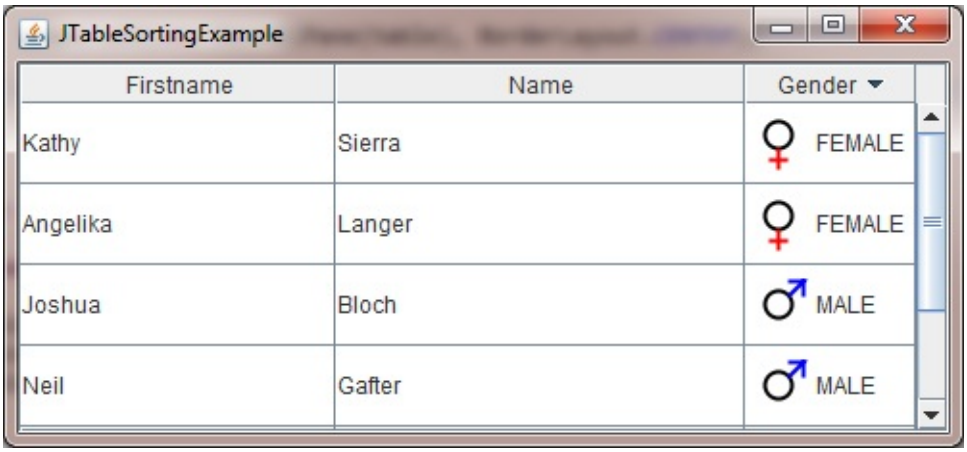


Abbildung 9-41 *Sortierfunktionalität in einer* `JTable`

Nachdem wir nun einen `TableRowSorter<M extends TableModel>` in Aktion erlebt haben, sollten wir noch kurz über dessen Arbeitsweise nachdenken, denn eigentlich müssten wir uns fragen, wieso und wie die gezeigte Sortierung überhaupt durchgeführt werden kann.

Sortierverhalten

Das von einem `TableRowSorter<M extends TableModel>` bereitgestellte (Standard-)Sortierverhalten beruht auf einem Vergleich der Werte mithilfe des Interface `Comparable<T>` bzw. `Comparator<T>` (vgl. Abschnitt 5.1.8). Das ist auch der Grund, weshalb sich die Spalte mit dem Geschlecht sortieren lässt: Alle `enum`-Aufzählungen erfüllen automatisch `Comparable<T>`. Für Spalten, in denen Werte vom Typ `String` enthalten sind, wird mit einem `Collator` eine Spezialisierung von `Comparator<T>` genutzt (vgl. Abschnitt 10.1.8).

Alles Letztes fehlt noch die Betrachtung von Spalten, deren Werte sich weder durch `Comparable<T>` bzw. `Comparator<T>` noch durch einen `Collator` sortieren lassen. Für diese werden die Werte automatisch zunächst per `toString()` in `String`-Instanzen überführt und dann mithilfe eines `Collators` sortiert. Dadurch lassen sich immer alle Spalten sortieren – manchmal jedoch nur sehr bedingt sinnvoll.

Einflussmöglichkeiten auf den `TableRowSorter`

Für einige Anwendungsfälle benötigt man mehr Kontrolle über den Sortiervorgang als dies mit der automatisch generierten `TableRowSorter<M extends TableModel>`-Instanz möglich ist. Dann kann man diese Instanz parametrieren oder eine neue `TableRowSorter<M extends TableModel>`-Instanz erzeugen, je nach Bedarf konfigurieren und schließlich der gewünschten Tabelle zuweisen:

```
// RowSorter von der Tabelle ermitteln - Variante 1
final TableRowSorter<TableModel> autoRowSorter = (TableRowSorter<TableModel>)
                                                  table.getRowSorter();

// RowSorter neu erzeugen und später der Tabelle zuweisen - Variante 2
final TableRowSorter<TableModel> tableRowSorter =
```

```
                                    new TableRowSorter<>(table.getModel());
table.setRowSorter(tableRowSorter);

// Für Spalte 2 einen zuvor definierten customComparator festlegen
tableRowSorter.setComparator(2, customComparator);

// Für Spalte 0 einen Komparator festlegen, der Stringlängen vergleicht
tableRowSorter.setComparator(0, new Comparator<String>()
{
    @Override
    public int compare(final String value1, final String value2)
    {
        // Neu in JDK 7: Integer.compare(int, int)
        return Integer.compare(value1.length(), value2.length());
    }
});

// Für eine Spalte COLUMN_IDX_PORTRAIT die Sortierung deaktivieren
tableRowSorter.setSortable(COLUMN_IDX_PORTRAIT, false);
```

Im Listing sehen wir, dass man einzelnen Spalten explizit eine `Comparator<T>`-Instanz zuordnen kann. Außerdem kann die Sortierfunktionalität deaktiviert werden, was wir hier für eine bisher nicht erwähnte Spalte nutzen, die Bilder bzw. Portraits enthält und den Index `COLUMN_IDX_PORTRAIT` besitzt.

Tabelle programmatisch sortieren

In der Regel ist es wünschenswert, wenn die Tabelle bereits bei ihrer ersten Darstellung, z. B. beim Öffnen eines Dialogs, eine Sortierung ihrer Inhalte vorgibt. Beispielsweise könnte die Liste der Personen nach deren Nachnamen sortiert sein. Es wäre umständlich, wenn der Benutzer zunächst immer erst auf die Spaltenüberschrift `Name` klicken müsste, um die Sortierung einzustellen. Praktischerweise kann man das programmgesteuert mithilfe von `TableRowSorter<M extends TableModel>` erledigen. Dabei bestimmt die Klasse `SortKey`, die innerhalb der Klasse `javax.swing.RowSorter<M>` definiert ist, eine Sortierreihenfolge. Diese wird durch eine `List<RowSorter.SortKey>` beschrieben und kann an einem `TableRowSorter<M extends TableModel>` durch Aufruf von `setSortKeys(List<? extends SortKey>)` übergeben werden. Das klingt noch etwas abstrakt, deshalb verdeutliche ich das an einem Beispiel.

Das folgende Listing zeigt die Definition von zwei Sortierungen: Mit `genderDescending` wird eine absteigende Sortierung der Spalte `gender` erreicht. Wenn sowohl das Geschlecht als auch die Vornamen aufsteigend sortiert werden sollen, so kann man die Sortierung `genderAndFirstname` nutzen.[17] Das Ganze lässt sich mithilfe des Programms JTableInitalSortingExample ausführen (vgl. Abbildung 9-42).

```
// Absteigende Sortierung der Spalte gender
final RowSorter.SortKey genderDescending =
                        new RowSorter.SortKey(2, SortOrder.DESCENDING);
tableRowSorter.setSortKeys(Collections.singletonList(genderDescending));

// Sortierung der Spalte gender aufsteigend, firstname aufsteigend
```

```
final List<RowSorter.SortKey> genderAndFirstname = new ArrayList<>();
genderAndFirstname.add(new RowSorter.SortKey(2, SortOrder.ASCENDING));
genderAndFirstname.add(new RowSorter.SortKey(0, SortOrder.ASCENDING));
tableRowSorter.setSortKeys(genderAndFirstname);
```

Listing 9.25 *Ausführbar als* **'JTABLEINITIALSORTINGEXAMPLE'**

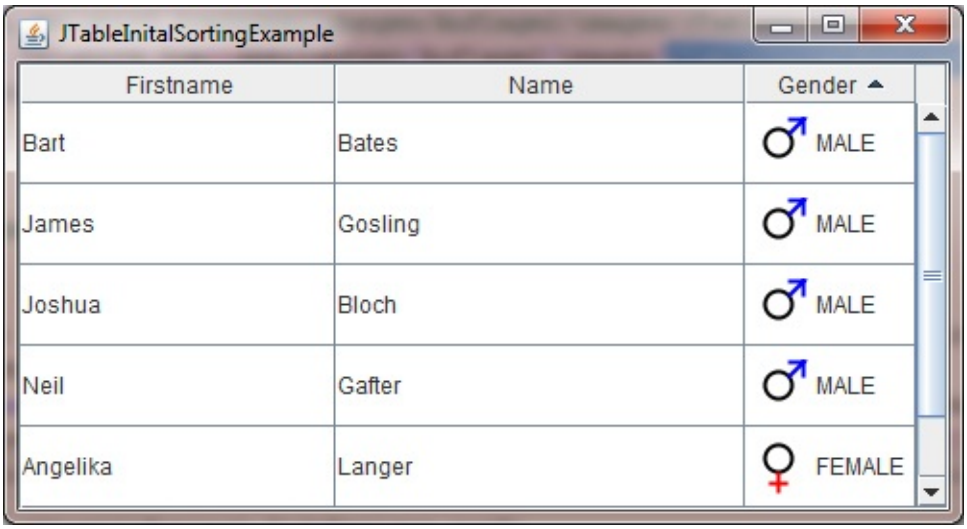


Abbildung 9-42 *Initiales Sortieren einer* `JTable`

Filterung

Neben dem Sortieren stellt das Filtern der Inhalte einer Tabelle eine weitere nützliche Funktionalität dar. Auch das ist seit JDK 6 einfach möglich. Hierbei bildet die abstrakte Klasse `javax.swing.RowFilter<M,I>` die Basis für die Implementierung eigener Filter. Dabei steht der generische Parameter `M` für den Typ des Modells und das `I` für den Typ einer Identifikation eines Eintrags – für Tabellen ist das immer der Typ `Integer`. Ein solcher Filter wird einer `JTable` indirekt mithilfe eines `TableRowSorters` durch Aufruf von dessen Methode `setRowFilter(RowFilter<M,I>)` zugeordnet.

Die Filterung mit der Klasse `RowFilter<M,I>` wird durch deren Methode `include(Entry<? extends M, ? extends I>)` realisiert. Deren Rückgabewert bestimmt, ob Daten aus dem Modell in die Darstellung übernommen werden sollen oder nicht. Die in der Methodensignatur genutzte Klasse `Entry<M,I>` ist eine innere statische Hilfsklasse in `RowFilter<M,I>` und nicht mit dem gleichnamigen inneren Interface `Map.Entry<K,V>` zu verwechseln. Die Klasse `Entry<M,I>` kapselt ein zu filterndes Objekt – für Tabellen ist das eine Zeile und für die später vorgestellten Bäume ein Knoten.

Um die gewünschte Filterung zu realisieren, muss die Methode `include(Entry<? extends M, ? extends I>)` entsprechend implementiert werden. Dabei sind in der Regel folgende Methoden sehr nützlich:

- `getValueCount()` – Liefert die Anzahl der durch ein `Entry<M,I>`-Objekt repräsentierten Werte. Für Tabellen ist das die Anzahl der Spalten.
- `getValue(int)` – Bietet Zugriff auf die einzelnen Werte einer Zeile.

- `getStringValue(int)` – Dies ist eine Convenience-Methode, die eine Stringrepräsentation des Werts liefert, wobei für `null`-Werte ein leerer String zurückgegeben wird.

Die reine Beschreibung der Methoden mag noch Fragen offen lassen. Das Ganze wird nach folgendem Beispiel dann aber sicher verständlich.

Ein eigener Filter

Mit den genannten Klassen `RowFilter<M, I>` und `Entry<M, I>` kann man recht einfach Filter erstellen, wie ich dies in folgendem Listing mit der Implementierung einer Klasse `ContainsFilter` zeige. Als Besonderheit wird dort – wie man es normalerweise als Benutzer erwartet – bei leeren Eingaben keine Filterung vorgenommen und alle Elemente ausgewählt. Aus demselben Grund erfolgt hier eine Umwandlung der zu vergleichenden Werte in Kleinbuchstaben, damit wir case-insensitive filtern, weil dies häufig recht komfortabel ist.

Die eigentliche Filterung ist folgendermaßen implementiert: Man läuft über alle Spalten und ermittelt durch Aufruf von `getStringValue(int)` den dort gespeicherten Wert als Stringrepräsentation. Basierend auf der Methode `contains(CharSequence)` der Klasse `String` werden dann Inhalte textuell ausgewählt, die den gewünschten Filterwert enthalten:

```
public class ContainsFilter extends RowFilter<Object, Object>
{
   private String filterValue = "";

   public ContainsFilter(final String filterValue)
   {
      this.filterValue = filterValue.toLowerCase();
   }

   public boolean include(final Entry<? extends Object, ? extends Object>
entry)
   {
      // Leere Filterbedingung (kein Filterkriterium)
      // => alle Elemente ausgewählt
      if (filterValue.isEmpty())
          return true;

      for (int i = 0; i < entry.getValueCount(); i++)
      {
         if (entry.getStringValue(i).toLowerCase().contains(filterValue))
            return true;
      }
      return false;
   }

   public void setFilterValue(final String filterValue)
   {
      this.filterValue = filterValue.toLowerCase();
   }
}
```

Damit ist ein Basisbaustein zum Filtern erstellt, der nutzbringend eingesetzt werden

kann. Auf ähnliche Weise lassen sich beliebige andere Filter implementieren.

Filter im Einsatz

Wir wollen den eben erstellten Filter nun für unsere Tabelle einsetzen und erweitern dazu das GUI der vorangegangenen Beispiele um ein Textfeld zur Eingabe eines Filterwerts. Dieser bestimmt, ob Einträge dargestellt oder ausgefiltert werden. Allerdings soll nur dann gefiltert werden, wenn tatsächlich ein Filterkriterium definiert wurde. Exakt darauf haben wir unseren `ContainsFilter` ausgelegt.

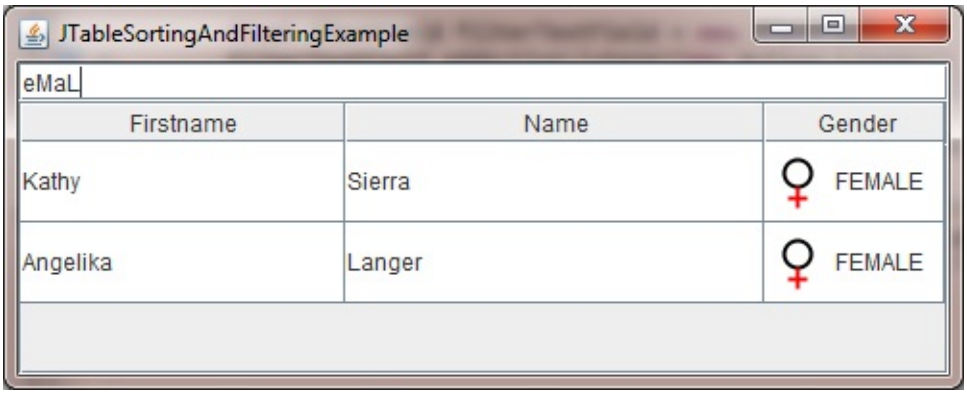


Abbildung 9-43 *Filterfunktionalität in einer* `JTable`

Zur Verarbeitung von Eingaben registrieren wir einen `ActionListener` bei dem Textfeld. Immer dann, wenn Sie in diesem Textfeld Return drücken, wird der Inhalt des Textfelds ausgelesen und an den `ContainsFilter` per `setFilterValue(String)` übergeben. Das reicht aber noch nicht aus, um die Filterung durchzuführen, weil es keine direkte Verbindung zwischen Filter und Tabelle gibt, sondern nur über die Indirektion `RowFilter` → `TableRowSorter` → `JTable`. Deshalb wird noch ein Aufruf von `sort()` an den `TableRowSorter` benötigt, wodurch dann sowohl sortiert als auch gefiltert wird. Daher wäre `sortAndFilter()` als Methodenname wohl geeigneter gewesen. Eine Trennung von Sortieren und Filtern wäre ein besseres Design gewesen.

Kommen wir mit diesem Wissen zur `main()`-Methode zurück, in der wir das GUI, basierend auf den vorherigen Beispielen, um ein `JTextField` zur Eingabe von Filterwerten erweitern. Das Ganze implementieren wir folgendermaßen:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("JTableSortingAndFilteringExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    final List<Person> persons = DataDefinitions.famousJavaPersons;
    final TableModel tableModel = new PersonTableModel(persons);
    final JTable table = new JTable(tableModel);

    initColumnWidths(table);
    table.setDefaultRenderer(Gender.class, new GenderTableCellRenderer());

    // Sortierung und Filterung initialisieren
    final TableRowSorter<TableModel> tableRowSorter =
                                        new TableRowSorter<>(table.getModel());
```

```
    final ContainsFilter containsFilter = new ContainsFilter("");
    tableRowSorter.setRowFilter(containsFilter);
    table.setRowSorter(tableRowSorter);

    // Filtereingabe
    final JTextField filterTextField = new JTextField();
    filterTextField.addActionListener(new ActionListener()
    {
        @Override
        public void actionPerformed(final ActionEvent event)
        {
            // Filterung basierend auf Eingabe ausführen
            containsFilter.setFilterValue(filterTextField.getText());
            tableRowSorter.sort();
        }
    });

    frame.add(filterTextField, BorderLayout.NORTH);
    frame.add(new JScrollPane(table), BorderLayout.CENTER);

    frame.setSize(500, 270);
    frame.setVisible(true);
}
```

Listing 9.26 *Ausführbar als* **'JTABLESORTINGANDFILTERINGEXAMPLE'**

Führen wir das Programm JTABLESORTINGANDFILTERINGEXAMPLE aus, so erhalten wir eine Darstellung ähnlich zu Abbildung 9-43, wenn wir nach dem Begriff "`eMaL`" filtern. Diese Schreibweise wurde gewählt, um die Unabhängigkeit der Filterung von der konkreten Groß-/Kleinschreibung sowie das Auffinden von Teilstrings zu demonstrieren.

Hinweis: Vordefinierte Filter

Man muss nicht unbedingt eigene Filterklassen von der abstrakten Basisklasse `RowFilter<M, I>` ableiten, denn von Hause aus steht bereits einige vordefinierte Funktionalität zur Verfügung. Mit dieser kann man beispielsweise einen `RegexFilter`, `DateFilter` usw. sowie auch spezielle Verknüpfungsfilter z. B. `Or-Filter` und `AndFilter` als Instanzen statischer innerer Klassen erzeugen. Diese Klassen erlauben es, Werte gemäß regulären Ausdrücken oder nach Datum zu filtern bzw. mehrere Filter miteinander zu kombinieren. Das übersteigt jedoch den Detailgrad, auf dem wir die Klasse `RowFilter<M, I>` untersuchen wollen. Benutzen Sie als Startpunkt für eigene Experimente bitte die Onlinedokumentation des JDKs.

Veränderlichkeit einzelner Zellen und Editoren

Die `JTable` erlaubt das sogenannte ***Inplace Edit***, d. h. Werte direkt in Zellen zu modifizieren, allerdings nur, sofern dies durch das Tabellenmodell vorgesehen ist. Das für die Beispiele eingesetzte `AbstractTableModel` erlaubt das Editieren standardmäßig nicht, das `DefaultTableModel` dagegen schon. Die Funktionalität

des Editierens direkt innerhalb einer Zelle ist sehr praktisch, weil es zusätzliche Editierdialoge vermeidet. Daher wird man auch für eigene Modelle, die auf `AbstractTableModel` basieren, die Funktionalität nutzen wollen. Dazu ist lediglich die Methode `isCellEditable(int, int)` zu implementieren. Durch deren Rückgabewert wird für einzelne Zellen der Tabelle bestimmt, ob diese editierbar sind oder nicht. Sofern `isCellEditable(int, int)` den Wert `true` zurückgibt, wird von der Tabelle automatisch ein passender, vom Spaltentyp abhängiger Defaulteditor präsentiert. Der Rückgabewert der bereits kennengelernten Methode `getColumnClass(int)` bestimmt, welcher Defaulteditor verwendet wird:

- `Boolean.class` – Für boolesche Werte wird eine Checkbox genutzt.
- `Number.class` und `String.class` – Für diese Typen erhält man ein Textfeld als Eingabemöglichkeit, das für Zahlenwerte rechtsbündig ausgerichtet ist.
- `Object.class` – Es wird ein Textfeld angeboten, wobei das zu editierende Objekt per `toString()` in eine textuelle Repräsentation umgewandelt wird. Problematisch ist dabei sowohl das Editieren als auch die Rückkonvertierung in Objekte.
- `Date.class` – Für Datumswerte wird ein Textfeld genutzt, das leider für die deutsche Locale einige Probleme beim Editieren verursacht. Hierbei werden zur Darstellung, zum Parsing und zum Editieren verschiedene Datumsformate (vgl. Abschnitt 10.1.6) genutzt, wodurch der Defaulteditor nahezu unbrauchbar wird! Hier würde man sich ein `DatePicker`-Control wünschen, das es leider in Swing standardmäßig nicht gibt. JavaFX bietet es dagegen seit Version 8 im Standard an (vgl. Abschnitt 14.4.3).

Die in einem Editor vorgenommenen Änderungen werden über die Methode `setValueAt(Object, int, int)` im Datenmodell verarbeitet. Beachten Sie unbedingt, dass ein Aufruf von `setValueAt(Object, int, int)` zunächst ohne Wirkung bleibt, wenn wir die abstrakte Basisklasse `AbstractTableModel` nutzen: Dort ist die Methode leer implementiert. Daher fehlt zur Verarbeitung und zur Übernahme von Änderungen aus der Editorkomponente in unser Datenmodell noch eine eigene Implementierung der Methode `setValueAt(Object, int, int)`.

Nehmen wir dazu an, die Klasse `Person` wird dahingehend geändert, dass sie jetzt mit Ausnahme des Vornamens veränderlich ist und zudem ein boolesches Attribut `isVegetarian` erhält. Das Ganze kann man dann folgendermaßen in das `PersonTableModel` integrieren:

```
public class PersonTableModel extends AbstractTableModel
{
    private static final String[] COLUMN_NAMES = { "Firstname", "Name",
                                                   "Gender", "Vegetarian?" };

    private static final Class<?>[] COLUMN_CLASSES = { String.class,
                                    String.class, Gender.class, Boolean.class
```

```
};

    private static final int COLUMN_IDX_FIRSTNAME = 0;
    private static final int COLUMN_IDX_NAME = 1;
    private static final int COLUMN_IDX_GENDER = 2;
    private static final int COLUMN_IDX_IS_VEGETARIAN = 3;

    // …

    @Override
    public boolean isCellEditable(final int rowIndex, final int columnIndex)
    {
        return columnIndex > COLUMN_IDX_FIRSTNAME;
    }

    @Override
    public void setValueAt(final Object value, final int rowIndex,
                           final int columnIndex)
    {
        final Person person = persons.get(rowIndex);

        if (columnIndex == COLUMN_IDX_NAME)
            person.setLastname((String)value);
        else if (columnIndex == COLUMN_IDX_GENDER)
            person.setGender((Gender)value);
        else if (columnIndex == COLUMN_IDX_IS_VEGETARIAN)
            person.setVegetarian((Boolean)value);

        fireTableCellUpdated(rowIndex, columnIndex);
    }
}
```

Die Methodenimplementierungen bzw. Aufrufe zur Datenänderung sind relativ leicht verständlich. Jedoch sehen wir in der Methode `setValueAt(Object, int, int)` einen Aufruf von `fireTableCellUpdated(int, int)`, der uns noch nicht geläufig ist. Dieser Aufruf wird benötigt, um den View darüber zu informieren, dass sich der Wert einer Zelle geändert hat, und somit für eine aktuelle Darstellung zu sorgen.

Starten wir das Programm JTABLEEDITINGEXAMPLE, so können wir sowohl die Nachnamen als auch das boolesche Flag verändern: Für die Nachnamen wird durch einen Doppelklick in der Tabellenzelle ein Textfeld eingeblendet. Die booleschen Werte lassen sich direkt über die angezeigte Checkbox ändern. In Abbildung 9-44 ist das dargestellt. Demnach sind Joshua Bloch und Neil Gafter nun Vegetarier und einige Personen haben einen Zusatz im Nachnamen, der das Wort “Java” enthält. Nach genau diesem Text wurde die Tabelle gefiltert.

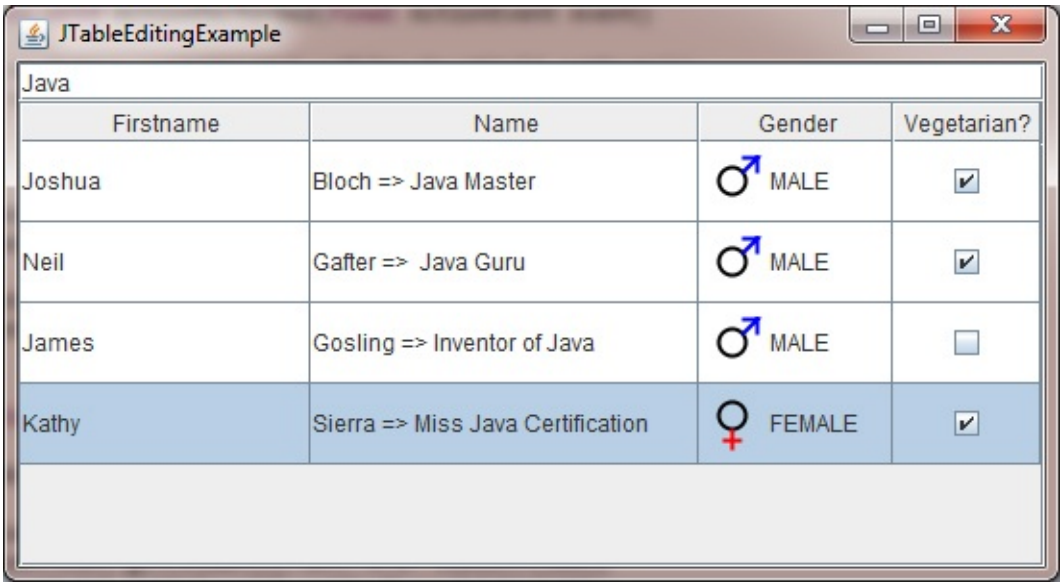


Abbildung 9-44 *Editierbarkeit von Zellen einer* `JTable`

Obwohl wir auch die Spalte für das Geschlecht als editierbar gekennzeichnet haben, tut sich nichts, wenn man dort einmal oder doppelt klickt. Offensichtlich lässt sich der Typ `Gender` nicht mithilfe einer der vordefinierten Editoren bearbeiten. Selbst wenn wir als Spaltentyp `Object` nutzen würden, wäre das Editieren der über `toString()` erzeugten textuellen Repräsentation kaum sinnvoll – ähnlich wie dies für die Darstellung mit Renderern gilt. Was kann man also tun? Schauen wir uns einmal an, welche Lösungshilfe die Swing-Bibliothek anbietet.

Bereitstellung eigener Editoren

Wünschenswert ist es, die zwei Geschlechter in einer Combobox zur Auswahl zu präsentieren. Praktischerweise wurde von den Entwicklern von Swing vorgesehen, beliebige Komponenten als Editor zu nutzen. Diese müssen dazu nur das Interface `javax.swing.table.TableCellEditor` implementieren. Dessen Methode `getTableCellEditorComponent()` legt fest, welches Bedienelement zur Eingabe verwendet werden soll. Häufig ist eine derart flexible Lösung aber gar nicht notwendig!

Einfacher geht es mithilfe der Klasse `DefaultCellEditor`, die das Interface `TableCellEditor` implementiert und es ermöglicht, alternativ ein Textfeld, eine Checkbox oder eine Combobox zum Editieren der Werte in den Zellen einzusetzen. Dazu kann das gewünschte Bedienelement an den Konstruktor der Klasse `DefaultCellEditor` übergeben werden. Im Hintergrund wird dann für die korrekte Verarbeitung gesorgt – unter anderem wird die Methode `setValueAt(Object, int, int)` aufgerufen, nachdem das Editieren beendet wurde.

Als Editor für die Spalte `Gender` verwenden wir einen `DefaultCellEditor`, dem wir eine vorkonfigurierte Combobox übergeben. Zudem wird durch Aufruf von `setClickCountToStart(int)` festgelegt, dass sich der Editor erst bei einem Doppelklick öffnet.[18] Abschließend wird der Editor bei der Tabelle für den Typ

`Gender` registriert. Wir ergänzen folgende Zeilen in der `main()`-Methode:

```
// JComboBox mit zwei Einträgen füllen
final JComboBox<Gender> genderComboBox = new JComboBox<>(Gender.values());
genderComboBox.setRenderer(new GenderListCellRenderer());

// Editor basierend auf Combobox erstellen (auf Doppelklick)
final DefaultCellEditor genderEditor = new DefaultCellEditor(genderComboBox);
genderEditor.setClickCountToStart(2);

// Editor für den Typ Gender.class registrieren
table.setDefaultEditor(Gender.class, genderEditor);
```

Listing 9.27 *Ausführbar als* **'JTABLEEDITINGEXAMPLEIMPROVED'**

Bei der Konstruktion der Combobox nutzen wir zwei praktische API-Details aus dem JDK. Zum einen kann man eine Combobox basierend auf einem Array gültiger Werte initialisieren. Zum anderen geben `enum`-Aufzählungen bei Aufruf von deren Methode `values()` ein Array aller in dem Aufzählungstyp definierten Werte zurück. Dadurch wird das Initialisieren der Combobox wie gezeigt ein Einzeiler. Außerdem legen wir noch einen Renderer für die Einträge fest – dessen Implementierung ist hier nicht gezeigt, geschieht aber analog zu den vorherigen Renderern.

Startet man das Programm JTABLEEDITINGEXAMPLEIMPROVED, sind nun auch die Werte in der Spalte `Gender` veränderlich (vgl. Abbildung 9-45).

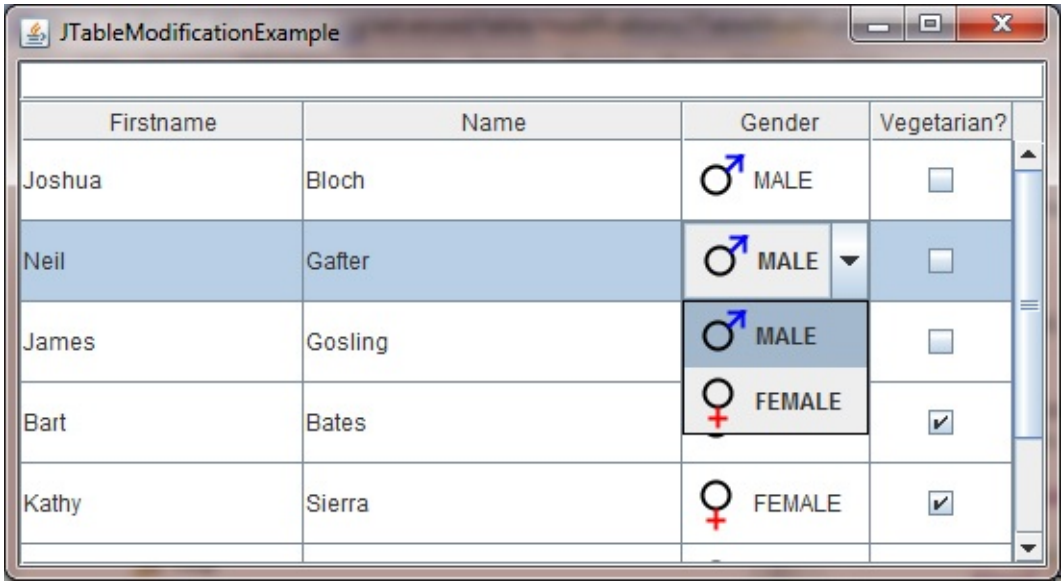


Abbildung 9-45 *Verbesserte Editierbarkeit von Zellen einer* `JTable`

Verarbeitung dynamischer Tabelleninhalte

Die Basisklasse `AbstractTableModel` bietet initial lediglich Lesezugriff auf die Daten. Sie ist allerdings dafür ausgelegt, bei der Verarbeitung von Datenänderungen zu helfen. Das haben wir bereits für das Editieren einzelner Zellen kennengelernt. Da dies alles automatisch über Swing im EDT abläuft, braucht man sich dabei keine Gedanken über Multithreading zu machen. Für extern ausgelöste Änderungen in der Zusammensetzung der Daten einer Tabelle gilt das natürlich nicht, und man muss

wieder die Single-Thread-Regel beachten.

Beispielhaft wollen wir das Datenmodell dahingehend erweitern, dass sich sowohl neue Einträge durch eine Methode `addEntry(Person)` hinzufügen als auch die gesamte Datenbasis durch eine Methode `setNewContent(List<Person>)` austauschen lassen. In beiden Fällen müssen Änderungen am Tabellenmodell durch die nachfolgend gezeigten vordefinierten Benachrichtigungsmethoden des `AbstractTableModel`s an interessierte Listener propagiert werden:

- `fireTableRowsInserted()` – Zeilen wurden hinzugefügt.
- `fireTableRowsUpdated()` – Zeilen wurden inhaltlich geändert.
- `fireTableRowsDeleted()` – Zeilen wurden gelöscht.
- `fireTableDataChanged()` – Viele/alle Zellen der Tabelle haben sich geändert.
- `fireTableStructureChanged()` – Die Struktur hat sich geändert, d. h. entweder die Anzahl der Spalten oder die Namen der Spalten.

Realisierung eines veränderlichen Tabellenmodells

Das Listing zeigt die Klasse `DynamicPersonTableModel`, die auf der Klasse `PersonTableModel` basiert und die zuvor angesprochenen Methoden zur Modifikation wie folgt implementiert:

```
public class DynamicPersonTableModel extends PersonTableModel
{
    public DynamicPersonTableModel(final List<Person> persons)
    {
        super(persons);
    }

    // Veränderlichkeit von Einträgen und des gesamten Inhalts
    public void addEntry(final Person newPerson)
    {
        getModifiablePersons().add(newPerson);
        // Spezifische Propagation der Änderungen
        final int newRowIndex = getModifiablePersons().size()-1;
        fireTableRowsInserted(newRowIndex, newRowIndex);
    }

    public void setNewContent(final List<Person> newPersons)
    {
        getModifiablePersons().clear();
        getModifiablePersons().addAll(newPersons);
        // Allgemeine Propagation der Änderungen
        fireTableDataChanged();
    }
}
```

Ein Detail fällt auf: Die Subklasse `DynamicPersonTableModel` besitzt zunächst keinen Zugriff auf das Attribut `persons` der Basisklasse `PersonTableModel`. Daher müssen wir diese um eine `protected` Zugriffsmethode

`getModifiablePersons()` erweitern. Dadurch besitzen wir in der Subklasse dann einen verändernden Zugriff auf die `List<Person>`.

Wie für `JList<E>` angesprochen, ist es ratsam, Änderungen durch Aufruf einer der oben vorgestellten Benachrichtigungsmethoden möglichst spezifisch an den View zu kommunizieren. Ein Aufruf von `fireTableDataChanged()` empfiehlt sich nur dann, wenn – wie im obigen Fall der Methode `setNewContent(List<Person>)` – der gesamte Tabelleninhalt ausgetauscht wird. In der Regel sollten Sie eine derart allgemeine Benachrichtigung vermeiden, weil ansonsten immer die gesamte Tabelle neu gezeichnet wird. Dadurch kann es zu deutlichen Performance-Einbußen kommen, die das GUI hakelig, wenig reaktiv und damit schlecht bedienbar machen.

Alternative: Austausch des Modells

Zwar kann man für Tabellen das Modell bei Änderungen immer vollständig neu aufbauen, d. h. ein neues `TableModel` erzeugen und dieses per `setModel(TableModel)` an die `JTable` übergeben. Dieses Vorgehen besitzt jedoch die bereits bekannten Nachteile, wie etwa Flackern, Zurücksetzen von Einstellungen (hier: Selektionen und die Sortierung). Aber nicht nur das! Wenn man für Tabellen einen `TableRowSorter` einsetzt und dann das Datenmodell austauscht, so muss man das Modell nicht nur für die `JTable`, sondern insbesondere auch für den `TableRowSorter` wie folgt neu setzen:

```
SwingUtilities.invokeAndWait(new Runnable()
{
    public void run()
    {
        final List<Person> persons = DataDefinitions.famousJavaPersons;
        final TableModel newTableModel = new DynamicPersonTableModel(persons);
        newTableModel.addTableModelListener(new
              SimpleTableModelModificationReporter());

        // Achtung: Spaltenbreiten und -anordnungen usw. gehen verloren
        table.setModel(newTableModel);
        // Wichtig für Konsistenz von Darstellung und Datenmodell
        tableRowsorter.setModel(newTableModel);

        // newTableModel.addEntry(new Person("Mr", "Exception", Gender.MALE));
    }
});
```

Wird der Aufruf von `setModel(TableModel)` am `TableRowSorter` nicht durchgeführt, so werden für dieses Beispiel nur zwei Zeilen angezeigt. Das scheint nicht ganz so schlimm zu sein, weil zunächst nur ein Darstellungsproblem ersichtlich ist. Wenn wir aber die im obigen Listing auskommentierte Zeile mit dem Aufruf von `addEntry(Person)` ausführen, so kommt es zu einer `IndexOutOfBoundsException` beim Einfügen des Eintrags '`Mr Exception`'.

Ich hoffe, Sie für die Problematik sensibilisiert zu haben und Ihnen so die ersten Gehversuche mit etwas komplexeren Tabellen zu erleichtern.

9.4.4 Die Klasse `JTree`

Dieser Abschnitt gibt eine kurze Einführung in die Darstellung von Baumstrukturen. In Swing nutzt man zu deren Visualisierung und Verarbeitung die Klasse `JTree` aus dem Package `javax.swing.tree`. Aus diesem Package stammen auch alle nachfolgend vorgestellten Klassen und Interfaces.

Bereits von Hause aus bietet die Klasse `JTree` vieles von dem, was man gewöhnlich zur Verwaltung hierarchischer Daten benötigt. Dazu gehört beispielsweise die Möglichkeit, Knoten mit deren Unterknoten auf- und zuzuklappen sowie Elemente entsprechend ihrer Hierarchieebene einzurücken.

Interfaces und Klassen im Zusammenspiel mit `JTree`

Ebenso wie die zuvor besprochenen Bedienelemente `JList<E>` und `JTable` delegiert die Klasse `JTree` das Zeichnen an einen Renderer vom Typ `TreeCellRenderer` und die Datenbereitstellung an eine Realisierung eines `TreeModels`. Die im `JTree` selektierten Einträge werden in einem `TreeSelectionModel` verwaltet. Abbildung 9-46 zeigt diese Interfaces und Klassen in einem Gesamtzusammenhang.

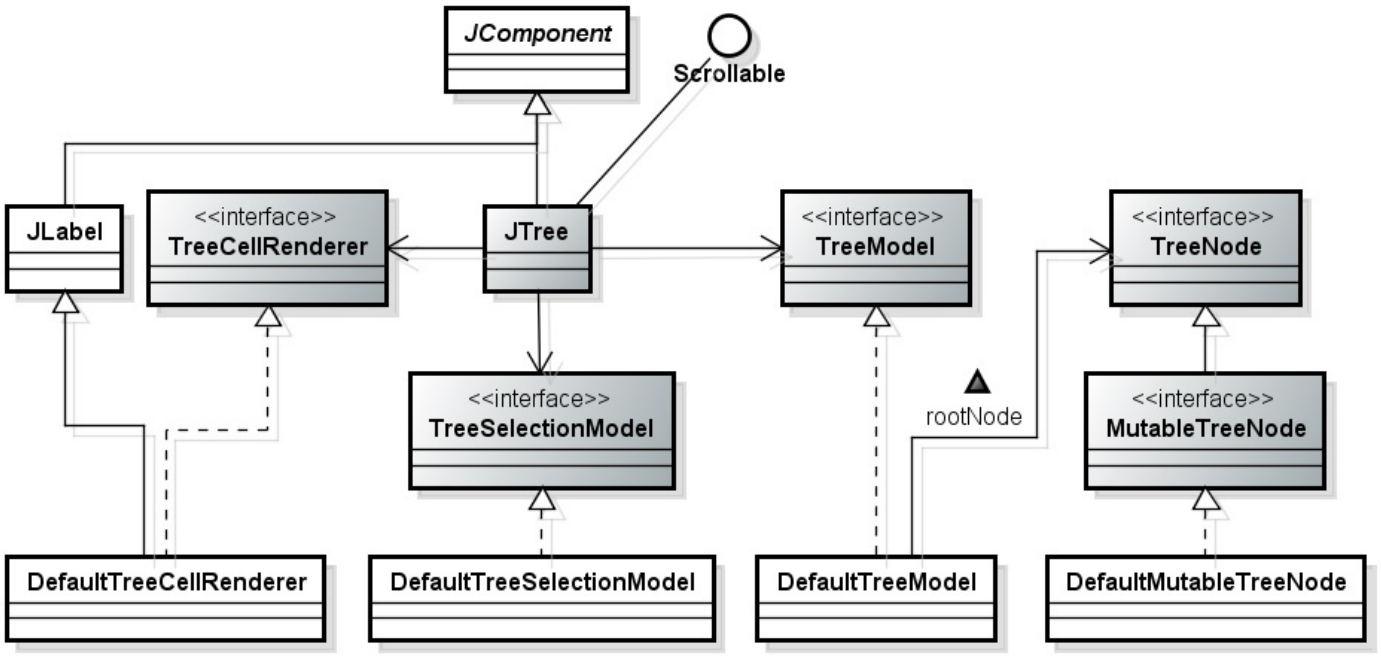


Abbildung 9-46 *Klassendiagramm* `JTree`

Schauen wir nun auf die einzelnen Bestandteile und beginnen mit Baumstrukturen, die die Grundlage für das Datenmodell eines `JTrees` bilden.

Baumstrukturen

In der Informatik besteht ein Baum aus einem Wurzelknoten (Root) und darunter liegenden Knoten (Nodes), die man als innere Knoten oder auch Kindknoten

bezeichnet. Diesen können wiederum beliebig viele Kindknoten zugeordnet sein. Spezielle Knoten, die keine weiteren Kindknoten besitzen, nennt man Blätter (Leafs). Diese informatische Sicht auf einen Baum wird in Swing durch das Interface `TreeNode` repräsentiert, das ausschließlich lesenden Zugriff bietet:

```
// Kommentierter Auszug aus dem JDK
public interface TreeNode
{
    // Zugriff auf die Eigenschaften dieses Knotens
    boolean isLeaf();
    boolean getAllowsChildren();

    // Zugriffe auf Elternknoten bzw. alle direkten Kindknoten
    TreeNode getParent();
    Enumeration children();


    // Zugriffe auf Kinder
    int getChildCount();
    TreeNode getChildAt(int childIndex);
    int getIndex(TreeNode node);
}
```

Als Erweiterung dieses Interface bietet das Interface `MutableTreeNode` unter anderem die Möglichkeit, Knoten hinzuzufügen und zu entfernen:

```
// Auszug aus dem JDK
public interface MutableTreeNode extends TreeNode
{
    void insert(MutableTreeNode child, int index);
    void remove(int index);
    void remove(MutableTreeNode node);
    // …
}
```

Eine konkrete Implementierung dieses Interface erfolgt im JDK durch die Klasse `DefaultMutableTreeNode`. Mit deren Hilfe lässt sich eine Baumstruktur durch Verknüpfung von Knoten aufbauen. Einen zweistufigen Baum kann man als eine Menge verbundener Knoten wie folgt konstruieren:

```
public static MutableTreeNode createExampleTreeStructure()
{
    // Wurzelknoten erstellen
    final DefaultMutableTreeNode rootNode = new
DefaultMutableTreeNode("Root");

    // Knoten für erste Ebene erstellen
    final MutableTreeNode firstChild = new DefaultMutableTreeNode("First");
    final MutableTreeNode secondChild = new DefaultMutableTreeNode("Second");
    final MutableTreeNode thirdChild = new DefaultMutableTreeNode("Third");

    // Erste Ebene mit Wurzelknoten verbinden - DefaultMutableTreeNode.add()
    rootNode.add(firstChild);
    rootNode.add(secondChild);
    rootNode.add(thirdChild);

    // Zweite Ebene erstellen und einfügen - MutableTreeNode.insert()
    secondChild.insert(new DefaultMutableTreeNode("2.1"), 0);
```

```
    secondChild.insert(new DefaultMutableTreeNode("2.2"), 1);

    return rootNode;
}
```

Die Baumstruktur selbst wäre relativ nutzlos ohne Daten, die einem Knoten zugeordnet sind. Daher kann jede Instanz der Klasse `DefaultMutableTreeNode` eine Nutzinformation in Form eines beliebigen benutzerdefinierten Objekts tragen. Vereinfachend werden hier Objekte vom Typ `String` bei der Konstruktion übergeben. Im Nachhinein kann man auf das Benutzerobjekt mit den Methoden `getUserObject()` bzw. `setUserObject(Object)` lesend bzw. schreibend zugreifen.

Im Listing sehen wir zwei Varianten, um Kindknoten hinzuzufügen: `add(MutableTreeNode)` und `insert(MutableTreeNode, int)`. Dabei sind folgende Besonderheiten erwähnenswert: Im Interface `MutableTreeNode` wird leider nur die Methode `insert(MutableTreeNode, int)` angeboten, der eine Position übergeben werden muss. Häufig möchte man Kindknoten einfach hinten anfügen. Angenehmer ist es daher, auf die Positionsangabe zu verzichten und einfach die Methode `add(Mutable-TreeNode)` zu nutzen. Diese steht jedoch nur in der Klasse `DefaultMutableTreeNode` bereit. Eine weitere Unschönheit besteht darin, dass die Methode `getUserObject()` auch nur in der Klasse `DefaultMutableTreeNode` vorhanden ist. Das ist insofern recht unpraktisch, als Schreibzugriffe mit `setUserObject(Object)` bereits im Interface `MutableTreeNode` möglich sind. Aufgrund dieser API-(Fehl)Konstruktion kann man häufig nicht ausschließlich – wie es wünschenswert wäre – gegen Interfaces bzw. Abstraktionen programmieren, sondern muss einen Down Cast auf den konkreten Typ `DefaultMutableTreeNode` vornehmen, um die nur dort angebotene Funktionalität aufrufen zu können.

Hinweis: Zyklenfreiheit

Wenn Sie etwas Experimentierfreude mitbringen, werden Sie sich fragen, was passieren würde, wenn Sie mutwillig einen Zyklus in den Baum einführen, indem sie einem beliebigen Kindknoten die Referenz auf die Wurzel übergeben, etwa wie folgt:

```
// Unterknoten mit Wurzelknoten verbinden
secondChild.insert(rootNode, 2);
```

Um solche Zyklen zu vermeiden,[a] wird beim Einfügen geprüft, ob versucht wird, eine Referenz auf einen Vorgängerknoten einzufügen. Ist das der Fall, so kommt es zu einer `IllegalArgumentException` und der Einfügevorgang wird abgebrochen. Auch beim Einfügen von Knoten wird eine Rückverzweigung vermieden: Es wird sichergestellt, dass ein Knoten immer nur genau einen Elternknoten besitzt.

[a] Weil aus einem Baum dann ein Graph würde, der Spezialbehandlungen erfordert.

Das Modell und die Klasse `TreeModel`

Wir haben gerade gesehen, wie eine Baumstruktur aus Knoten entsteht. Zwar könnte man denken, dass man damit bereits alles besitzt, um eine Visualisierung des Baums mit einem `JTree` vorzunehmen, doch das ist nicht so: Während der Typ `TreeNode` die Baumstruktur implementiert, wird durch Realisierungen des Interface `TreeModel` zusätzlich diejenige Funktionalität bereitgestellt, die ein `JTree` zur Darstellung des Baums benötigt. Werfen wir daher einen Blick auf die Methoden des Interface `TreeModel` aus dem JDK:

```
// Kommentierter Auszug aus dem JDK
public interface TreeModel
{
    // Zugriff auf Wurzel, Abfrage auf Blatt
    public Object getRoot();
    public boolean isLeaf(Object node);

    // Zugriff auf Kindknoten
    public int getChildCount(Object parent);
    public Object getChild(Object parent, int index);
    public int getIndexOfChild(Object parent, Object child);

    // Benachrichtigung bei Änderungen im Modell
    public void valueForPathChanged(TreePath path, Object newValue);
    public void addTreeModelListener(TreeModelListener listener);
    public void removeTreeModelListener(TreeModelListener listener);
}
```

Ebenso wie bei den anderen Bedienelementen erwähnt, dienen auch hier die meisten der obigen Methoden des Modells dazu, das Bedienelement `JTree` bei dessen Arbeit zu unterstützen, etwa um Zugriff auf die Wurzel oder Kindknoten zu erhalten bzw. deren Anzahl zu ermitteln. Darüber hinaus können sich Interessenten vom Typ `TreeModelListener` beim Modell registrieren, um über Veränderungen informiert zu werden. Der `JTree` nutzt dies, um den View zu aktualisieren, wenn beispielsweise Kindknoten hinzukommen, entfernt werden oder sich deren Daten ändern.

Hinweis: Besonderheiten beim `TreeModel`

Wenn Sie sehr aufmerksam sind, ist Ihnen bestimmt aufgefallen, dass alle Methoden eines `TreeModel`s überraschenderweise mit dem Typ `Object` statt mit dem Typ `TreeNode` arbeiten. Tatsächlich ist die zu einem `JTree` gehörende Modellklasse `TreeModel` unabhängig von `TreeNode`s aufgebaut. Es wird sogar nicht einmal eine Baumstruktur vorgegeben, sondern lediglich Methoden, die Zugriffe auf Kindknoten bieten. Das ist häufig mehr Fluch als Segen. Zwar kann man auf diese Weise alle möglichen Daten als Grundlage für ein `TreeModel` nutzen, allerdings geht dies auf Kosten der Typsicherheit und Benutzbarkeit: Für viele Anwendungsfälle würde sich die Handhabung vereinfachen, wenn die Klasse `TreeNode` genutzt würde.

Standarddatenmodell

Im JDK wird für das Interface `TreeModel` mit der Klasse `DefaultTreeModel` eine Standardimplementierung zur Verfügung gestellt, die Methoden zum Ändern der Baumstruktur anbietet und insbesondere auf die Verwaltung von den eben vorgestellten `DefaultMutableTreeNodes` ausgelegt ist: Das ist praktisch, weil man die gewünschte Baumstruktur in Form verknüpfter `TreeNodes` erstellen kann – ähnlich wie wir dies zuvor mit der Methode `createExampleTreeStructure()` kennengelernt haben – und dann den Wurzelknoten an den Konstruktor eines `DefaultTreeModel`s übergeben kann. Dieses Modell arbeitet als Adapter: Es sorgt in diesem Fall dafür, dass die Methoden des Interface `TreeModel` auf Zugriffe auf die durch `TreeNodes` beschriebene Baumstruktur abgebildet werden. Außerdem dient diese Modellinstanz als Eingabe für den Konstruktor eines `JTrees`. Folgendes Listing verdeutlicht den beschriebenen Ablauf:

```
public static void main(final String[] args)
{
    final JFrame frame = new JFrame("JTreeExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);

    // Erzeugen der Baumstruktur, des Modells und des Bedienelements
    final MutableTreeNode rootNode = createExampleTreeStructure();
    final TreeModel treeModel = new DefaultTreeModel(rootNode);
    final JTree tree = new JTree(treeModel);

    frame.add(new JScrollPane(tree), BorderLayout.CENTER);
    frame.setSize(300, 150);
    frame.setVisible(true);
}
```

Listing 9.28 *Ausführbar als* **'JTREEEXAMPLE'**

Das hier entwickelte Beispiel kann als Programm JTREEEXAMPLE gestartet werden. Eine Ausführung ist als Screenshot in nachfolgender Abbildung 9-47 gezeigt.

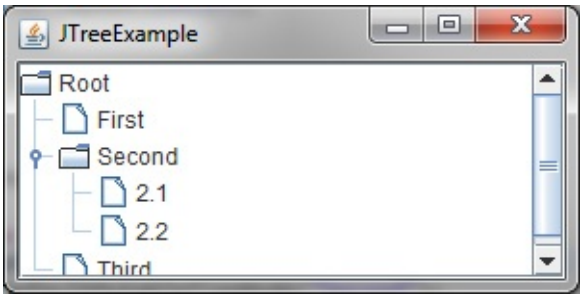


Abbildung 9-47 *Das Programm* `JTreeExample` *in Aktion*

Benachrichtigungen und Selektion

Ein `JTree` löst je nach Bedienhandlung ein oder mehrere Ereignisse aus. Sind für diese entsprechende Event Listener registriert, so wird man informiert, wenn Knoten geöffnet, geschlossen oder selektiert werden. Dazu dienen der `TreeExpansionListener`, der `TreeWillExpandListener` und der `TreeSelectionListener`, mit dem man über Selektionsänderungen durch einen

Aufruf der Methode `valueChanged(TreeSelectionEvent)` informiert wird.

Das folgende Listing demonstriert die Arbeitsweise. Dort wird die Klasse `SimpleTreeSelectionListener` implementiert. Diese erfüllt das Interface `TreeSelectionListener` und protokolliert die Selektionen im Baum. Dazu werden verschiedene Informationen aus dem `TreeSelectionEvent` ausgelesen und in einem `JLabel` dargestellt:

```
public final class SimpleTreeSelectionListener implements
TreeSelectionListener
{
    private final JLabel infoLabel;

    public SimpleTreeSelectionListener(final JLabel infoLabel)
    {
        this.infoLabel = infoLabel;
    }

    @Override
    public void valueChanged(final TreeSelectionEvent event)
    {
        final TreePath treePath = event.getPath();
        final Object pathComponent = treePath.getLastPathComponent();
        final String type = pathComponent.getClass().getSimpleName();

        infoLabel.setText("Path: " + treePath + " / Object: " + pathComponent
+
                          " / Type: " + type);
    }
}
```

Die Protokollierung der Selektion können Sie nachvollziehen, wenn Sie das Programm JTREESELECTIONEXAMPLE starten.

Baumpfade und die Klasse `TreePath`

Bei einem genaueren Blick auf die Methode `valueChanged(TreeSelectionEvent)` finden wir die bisher nicht vorgestellte Klasse `TreePath`. Diese modelliert einen Pfad durch den Baum: Darunter versteht man eine Abfolge aller Knoten ausgehend von der Wurzel zu einem gegebenen Knoten. Dieser Pfad besteht aus einem `Object[]` und nicht aus einem `TreeNode[]`, wie man dies stark vermuten könnte (und es häufig praktisch wäre). Das liegt an der Ausrichtung des `TreeModel`s auf den Typ `Object`, wie dies bereits im vorherigen Hinweis kritisch betrachtet wurde.

Interessanterweise gibt es im JDK keine Methode, um einen solchen `TreePath` für einen Knoten zu ermitteln – nutzt man ein `DefaultTreeModel` existiert dort eine ähnliche Methode `getPathToRoot(TreeNode)`, die einen `TreeNode[]` statt eines `TreePath` als Rückgabe liefert. Eine Methode `getTreePath(TreeNode)` lässt sich jedoch leicht selbst implementieren, etwa in einer Utility-Klasse `TreeUtils`:

```
public static TreePath getTreePath(final TreeNode treeNode)
{
```

```
    final List<TreeNode> treeNodes = new ArrayList<>();
    treeNodes.add(treeNode);

    TreeNode parentNode = treeNode.getParent();
    while (parentNode != null)
    {
        treeNodes.add(parentNode);
        parentNode = parentNode.getParent();
    }

    // Reihenfolge umdrehen: Wurzel -> Zielknoten!
    Collections.reverse(treeNodes);
    return new TreePath(treeNodes.toArray());
}
```

Renderer

Im vorangegangenen Beispiel wurden die Elemente des Baums alle korrekt und erwartungskonform angezeigt. Das liegt daran, dass wir Strings als Benutzerobjekte (Nutzdaten) verwenden, die sich durch den Default-Renderer, der bekanntlich auf einem `JLabel` basiert, darstellen lassen. Wie in Abschnitt 9.4.1 bei der Besprechung von Renderern für `Person`-Objekte als Nutzdaten für Listen gezeigt, lassen sich diese nicht sinnvoll mit dem Default-Renderer visualisieren, da hier eine textuelle Darstellung auf Basis von `toString()` erfolgt. Eine Abhilfe in Form von eigenen Renderern habe ich sowohl für Listen also auch für Tabellen bereits recht ausführlich vorgestellt, daher möchte ich hier auf diese Art von Renderern nicht mehr eingehen.

Stattdessen möchte ich einen Renderer erstellen, der neben den Nutzdaten auch einige Metainformationen über die Baumknoten liefert: Alle Knoten, die keine Blätter sind, werden fett darstellt. Zudem soll jeder Knoten die Anzahl der direkten Kindknoten sowie die Anzahl aller Unterknoten als ergänzende Information anzeigen. Letztgenannten Wert ermitteln wir über die später beschriebene Hilfsmethode `getTotalChildCount(DefaultMutableTreeNode)`. Wir implementieren Folgendes:

```
public class BoldTextTreeNodeRenderer extends DefaultTreeCellRenderer
{
    @Override
    public Component getTreeCellRendererComponent(final JTree tree,
                                final Object value, final boolean isSelected,
                                final boolean isExpanded, final boolean isLeaf,
                                final int row, final boolean hasFocus)
    {
        super.getTreeCellRendererComponent(tree, value, isSelected,
                                           isExpanded, isLeaf, row, hasFocus);
        // Für unser Modell gültig, da es DefaultMutableTreeNode nutzt
        if (value instanceof DefaultMutableTreeNode)
        {
            final DefaultMutableTreeNode node = (DefaultMutableTreeNode)
value;
            setMetaAttributes(tree, node);
        }

        return this;
```

```
    }

    private void setMetaAttributes(final JTree tree,
                                   final DefaultMutableTreeNode node)
    {
        final int childCount = node.getChildCount();
        if (childCount > 0)
        {
            setFont(getFont().deriveFont(Font.BOLD, 14));
            setText(getText() + " (" + childCount + "; " +
                                       getTotalChildCount(node) + ")");
        }
        else
        {
            setFont(tree.getFont());
        }
    }
}
```

Bei der Implementierung dieses Renderers nutzen wir, dass man durch Aufruf von `getChildCount()` der Klasse `DefaultMutableTreeNode` die Anzahl direkter Kindknoten ermitteln kann. Auch die Methode `getTotalChildCount(DefaultMutableTreeNode)` greift auf Standardfunktionalität des JDKs zurück: Die hier verwendete Methode `breadthFirstEnumeration()` liefert eine `Enumeration` über alle Knoten eines (Teil-)Baums (auch die Wurzel selbst bzw. den Knoten des Aufrufs). Ausgehend vom Startknoten läuft die Enumeration zunächst die erste Ebene der Knoten ab. Danach geschieht dies sukzessive für die weiteren Ebenen. Die Anzahl aller Knoten ermittelt man dann durch Zählen aller dabei traversierten Elemente wie folgt:

```
public static int getTotalChildCount(final DefaultMutableTreeNode parentNode)
{
    if (parentNode.getChildCount() == 0)
        return 0;

    int count = 0;
    final Enumeration nodeEnumeration = parentNode.breadthFirstEnumeration();
    while (nodeEnumeration.hasMoreElements())
    {
        final TreeNode node = (TreeNode) nodeEnumeration.nextElement();
        count++;
    }

    // Enumeration zählt auch immer den eigenen TreeNode mit!
    return count - 1;
}
```

Alle sukzessiv hinzugefügten Funktionalitäten sind im Progamm `JTreeRendererExample` enthalten, dieses wird hier aber nur ausschnittsweise aufgelistet. Abbildung 9-48 zeigt einen Start des zugehörigen Programms JTREERENDERER EXAMPLE.

```
public static void main(final String[] args)
{
    final JLabel infoLabel = new JLabel("No selection!");
    final JFrame frame = new JFrame("JTreeRendererExample");
    frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
```

```
    final MutableTreeNode rootNode =
TreeStructure.createExampleTreeStructure();
    final TreeModel treeModel = new DefaultTreeModel(rootNode);
    final JTree tree = new JTree(treeModel);

    tree.setCellRenderer(new BoldTextTreeNodeRenderer());
    tree.addTreeSelectionListener(new SimpleTreeSelectionListener(infoLabel));

    frame.add(new JScrollPane(tree), BorderLayout.CENTER);
    frame.add(infoLabel, BorderLayout.SOUTH);
    frame.setSize(450, 180);
    frame.setVisible(true);
}
```

Listing 9.29 *Ausführbar als* **'JTREERENDEREREXAMPLE'**

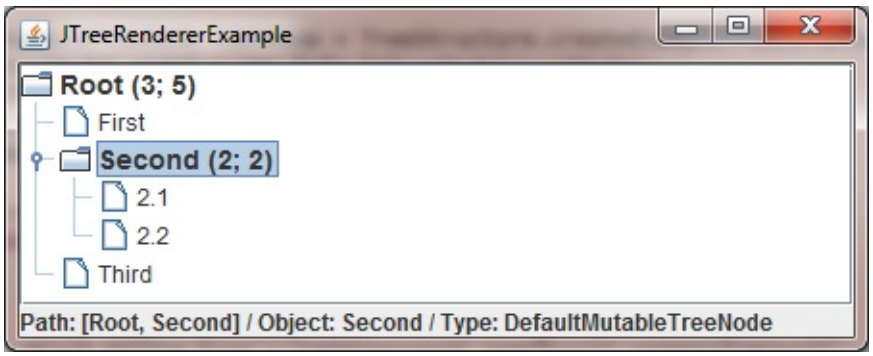


Abbildung 9-48 *Beispiel eines eigenen Renderers für einen* `JTree`

Veränderlichkeit der Daten in Bäumen

Die in einem `JTree` verwalteten Daten sind selten statischer Natur. Wie zuvor für Listen und Tabellen besprochen, möchte ich auch für Bäume auf die Dynamik von Daten und deren korrekte Verarbeitung eingehen.

Änderungen an der Baumstruktur

Es ist naheliegend – weil wir so auch die Baumstruktur aus `DefaultMutableTreeNodes` aufgebaut haben –, einem bestehenden Knoten neue Knoten hinzuzufügen:

```
// Versuch 1: Einfügen am TreeNode, dabei zwei Varianten möglich
firstChild.add(new DefaultMutableTreeNode("1.1"));
firstChild.insert(new DefaultMutableTreeNode("1.2"), 1);
```

Wenn wir die gezeigten Änderungen mit unserem Vorwissen über die Single-Thread-Regel für Swing durch einen Aufruf von `invokeLater(Runnable)` im EDT ausführen, so sehen wir jedoch keine Änderung in der Anzeige. Wie kommt das?

Denken wir darüber nach: Wir haben zwar die Daten korrekt im EDT verändert, aber weder das `TreeModel` noch der `JTree` haben davon Kenntnis. Für den `JTree` genügt bei einer Strukturänderung ein Aufruf von `repaint()`, um dies im GUI zu reflektieren – das steht im Gegensatz zu einer `JList<E>`, bei der ein Aufruf von

repaint() nicht ausreichend ist. Für den JTree ist das dadurch begründet, dass das DefaultTreeModel direkt auf die TreeNodes zugreift. Es ist kein guter Stil, auf ein Repaint zur rechten Zeit zu hoffen, weil dieses eventuell gar nicht oder erst später kommt.

Daher gilt: Wenn man schon Knoten manuell (am Modell vorbei) verändert, so sollte man dies dem Modell zumindest nachträglich mitteilen. Dazu ruft man die Methode reload(TreeNode) auf, die im DefaultTreeModel bereitsteht:

```
firstChild.add(new DefaultMutableTreeNode("1.3"));

// wichtig zu wissen, welches Modell
final DefaultTreeModel defaultModel = (DefaultTreeModel)tree.getModel();
defaultModel.reload(firstChild);

// Programmatisch aufklappen
tree.expandPath(TreeUtils.getTreePath(firstChild));
```

Dieser Listing-Ausschnitt zeigt eine nützliche Funktionalität, die der JTree bietet: Man kann einzelne Knoten programmgesteuert auf- und zuklappen. Hierbei kommt unsere zuvor erstellte Methode getTreePath(TreeNode) zum Einsatz.

Anstatt auf den DefaultMutableTreeNodes zu arbeiten, sollte man Methoden des DefaultTreeModels aufrufen, da somit automatisch Benachrichtigungen durch das Modell für den View erzeugt werden. Nachfolgend ist dies für das Einfügen und Entfernen von Knoten mithilfe von Methoden des Modells gezeigt:

```
// Zwei Knoten hinzufügen, einen Knoten entfernen
defaultModel.insertNodeInto(new DefaultMutableTreeNode("3.1"), thirdChild, 0);
defaultModel.insertNodeInto(new DefaultMutableTreeNode("3.2"), thirdChild, 1);
defaultModel.removeNodeFromParent(secondChild);
```

Das Ganze ist so zwar etwas unhandlicher, als mit den DefaultMutableTreeNodes zu arbeiten, dafür wird aber die Darstellung im View automatisch synchron mit dem Datenmodell gehalten. Dies geschieht dadurch, dass sich der JTree selbst als Listener für Änderungen am Modell registriert.

Änderungen am Benutzerobjekt

Als Letztes verbleibt noch, Änderungen an den Daten eines Knotens zu betrachten, also an dessen Benutzerobjekt. Wenn man an einem DefaultMutableTreeNode die Methode setUserObject(Object) aufruft, so geschieht diese Modifikation ohne Benachrichtigung des Modells – und ist damit ebenso problematisch wie das Hinzufügen von Knoten an einem DefaultMutableTreeNode. Auch hier wird die Änderung im GUI zunächst nicht reflektiert. Das geschieht wiederum erst durch einen Aufruf von repaint(). Allerdings kann es dabei Darstellungsprobleme geben, im Beispiel wird der Text abgeschnitten, weil der View noch vom Originalinhalt und dessen Dimensionen ausgeht.

Als Abhilfe kommt die Methode valueForPathChanged(TreePath, Object) aus dem TreeModel zum Einsatz, die für einen Knoten, bestimmt durch

dessen `TreePath`, das übergebene Benutzerobjekt setzt. In diesem Fall wird der View über Änderungen informiert und sorgt automatisch für eine korrekte Darstellung.

```
// Inhaltliche Änderung eines Knotens
firstChild.setUserObject("Changed First"); // // keine Auswirkung im Modell
frame.repaint(); // // Darstellungsprobleme durch abgeschnittenen Text

final TreePath treePath = TreeUtils.getTreePath(firstChild);
model.valueForPathChanged(treePath, "Changed First 2");
```

Ganz nebenbei sehen wir hier einen weiteren sinnvollen Einsatzzweck für die Hilfsmethode `getTreePath(TreeNode)`, denn wir benötigen hier keinen Knoten, sondern dessen Pfad!

Damit haben wir diverse Möglichkeiten zur Änderung hierarchischer Daten kennengelernt. Um das Ganze etwas konkreter und greifbarer zu machen, sind die gezeigten Anweisungen Teil eines Beispielprogramms, das Sie als Programm JTREEMODIFICATIONEXAMPLE ausführen können. In Abbildung 9-49 ist das Endergebnis der Menge der Änderungen gezeigt.

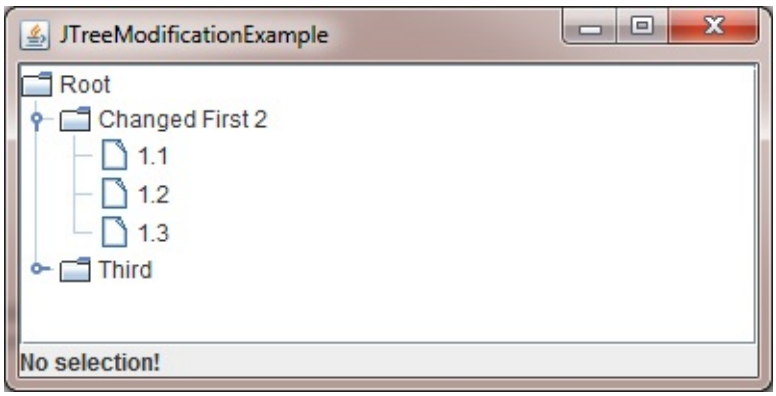


Abbildung 9-49 *Ausführung des Programms* JTREEMODIFICATIONEXAMPLE

Alternative: Austausch des Modells

Manchmal sollen Änderungen transaktional erfolgen und sind umfangreicher. Auch dann kann man sicherlich die Baumstruktur sukzessive ändern. Stattdessen bietet es sich aber an, die Baumstruktur immer vollständig neu aufzubauen, daraus ein neues `TreeModel` zu erzeugen und dieses per `setModel(TreeModel)` an den `JTree` zu übergeben. Dieses Vorgehen besitzt jedoch den Nachteil, dass sowohl die Selektionen als auch der »Aufklappzustand« aller Knoten zurückgesetzt wird, was vom Benutzer meistens als recht störend empfunden wird.

9.5 Weiterführende Literatur

Das Themengebiet der grafischen Benutzeroberflächen ist extrem umfangreich. In diesem Kapitel konnte ich daher nur auf einige wichtige Aspekte eingehen; diverse andere konnten hier leider nicht berücksichtigt werden. Wie im gesamten Buch war mir besonders daran gelegen, die Hintergründe und das Verständnis für die

Zusammenhänge zu vermitteln, statt akribisch das API zu dokumentieren. Hierfür finden Sie in der Dokumentation des JDKs viele hilfreiche Informationen. Im Speziellen konnte ich natürlich nicht jedes Detail der Bedienelemente erwähnen. Dazu empfehle ich die Literatur eines der beiden nachfolgend genannten Grundlagenbücher, die jeweils einen umfassenderen Überblick geben:

- »**Java Swing**« von Robert Eckstein, Marc Loy and Dave Wood [21]

  Dieses Buch beschreibt sehr detailliert das API der Swing-Bedienelemente. Man kann es recht gut als Nachschlagewerk benutzen. Was allerdings fehlt, ist die Darstellung für den Einsatz in der Praxis. Vor allem auf Swing und Multithreading wird nur sehr knapp und erst sehr spät eingegangen.

- »**Swing**« von Matthew Robinson und Pavel Vorobiew [76]

  Matthew Robinson und Pavel Vorobiew beschreiben Swing dagegen deutlich mehr an der Praxis orientiert als das zuvor beschriebene Buch und nutzen dazu ansprechende Beispiele.

In den obigen Büchern werden z. B. die Themen Java 2D und grafische Effekte nicht vorgestellt. Informationen dazu finden Sie in folgenden Büchern:

- »**Java 2D Graphics**« von Jonathan Knudsen [55]

  Um erste Schritte mit Java 2D zu machen, bietet sich die Lektüre dieses Buchs an. Alle wesentlichen Grundlagen werden dort in Form einfacher Beispiel erklärt. Dabei steht jedoch eher das API im Fokus, weniger grafische Effekte.

- »**Java 2D API Graphics**« von Vincent J. Hardy [37]

  Dieses Buch bietet eine umfassende, gut verständliche Einführung in Java 2D. Neben der ansprechenden Darstellung der Grundlagen werden auch diverse grafische Effekte beschrieben, mit denen man eigene Applikationen optisch aufwerten kann. Wie Sie das Ganze auf Ihr GUI ausdehnen, wird in »Filthy Rich Clients« beschrieben.

- »**Filthy Rich Clients**« von Chet Haase und Romain Guy [36]

  Wenn Sie Ihre GUIs mit optischen Effekten aufpolieren möchten, dann sollten Sie einen Blick in dieses Buch werfen. Die vorgestellten Lösungen werden inklusive der dafür benötigten Grundlagen verständlich dargestellt.

- »**Swing Hacks**« von Joshua Marianacci und Chris Adamson [61]

  Ebenso wie im zuvor genannten Buch stellt »Swing Hacks« verschiedene Techniken vor, um GUIs optisch und funktional aufzuwerten. Der Fokus liegt nicht auf der Darstellung aller Grundlagen – diese werden als bekannt vorausgesetzt. Stattdessen werden in kleinen Episoden in sich abgeschlossene Techniken vorgestellt.

- »**Core Swing: advanced programming**« von Kim Topley [86]

  Dieses Buch behandelt einige fortgeschrittene Themen der Swing-

Programmierung und geht insbesondere sehr detailliert auf alle möglichen GUI-Komponenten zur Texteingabe ein. Neben Renderern und Editoren für Tabellen werden auch Drag-and-Drop sowie das Swing-Undo-Package ausführlich behandelt.

Auf Informationen zur Gestaltung einer Benutzeroberfläche wie verwendete Farben, Formen und Anordnungen konnte hier leider auch nicht eingegangen werden. Es gibt dazu verschiedene Interface Design Guidelines, die zwar zum Teil kleine Abweichungen voneinander besitzen, deren Grundtenor jedoch einheitlich ist. Folgende Lektüre ist für den Java-Bereich empfehlenswert:

- »**Java Look and Feel Design Guidelines**« von Sun Microsystems [66]

  Wenn Sie tiefer in die Materie einer gelungenen Gestaltung einer Benutzeroberfläche einsteigen wollen und dabei die elementaren Dinge wie Platzierung von Bedienelementen und deren Abstände, zu nutzende Tastaturkürzel sowie Icons und deren Gestaltung, aber auch vieles mehr von Grund auf kennenlernen wollen, dann sollten Sie einen Blick in dieses Buch werfen.

10 Basiswissen Internationalisierung

Soll eine Anwendung in verschiedenen Ländern eingesetzt werden, so sind lokale Besonderheiten, wie verschiedene Darstellungen von Währungen, Uhrzeiten und Datumswerten, aber vor allem die Übersetzung und Anpassung der in der grafischen Oberfläche dargestellten Texte und Informationen, wichtige Themen. Gerade Letzteres ist für Benutzer ein offensichtlicher Aspekt der Internationalisierung. Bei der Realisierung spielen diverse weitere Dinge eine wichtige steuernde Rolle, etwa die (im Betriebssystem) gewählte Sprache und das Land. Die folgenden Abschnitte behandeln verschiedene Themen der Internationalisierung so weit, dass eine Umsetzung in eigenen Programmen möglich wird.

Abschnitt 10.1 gibt eine Einführung in das Thema Internationalisierung und stellt insbesondere die formatierte, landesspezifische Ein- und Ausgabe vor, etwa Darstellungen von Uhrzeit, Datum und Währungen. Die beschriebenen Funktionalitäten werden in Abschnitt 10.2 zu Programmbausteinen zur Internationalisierung ausgebaut.

10.1 Internationalisierung im Überblick

Nachfolgend werden verschiedene Grundlagen und Klassen beschrieben, die wir als wiederverwendbare Bausteine in unseren Applikationen benötigen. Abschnitt 10.1.1 beschreibt einige Grundlagen. Die Basis zur Verwaltung landes- oder sprachabhängiger Besonderheiten bildet die Klasse `java.util.Locale`, die in Abschnitt 10.1.2 vorgestellt wird. Zum Verwalten von textuellen Informationen ist die Klasse `java.util.PropertyResourceBundle` hilfreich. Sie wird in Abschnitt 10.1.3 beschrieben. Danach gibt Abschnitt 10.1.4 einen kurzen Überblick über die `Format`-Klassen aus dem Package `java.text`. In den darauffolgenden Abschnitten werden dann einige Spezialisierungen genauer vorgestellt. Im Einzelnen sind dies die Klassen `NumberFormat`, `DateFormat` sowie `MessageFormat` in den Abschnitten 10.1.5 bis 10.1.7. Im Anschluss daran stelle ich in Abschnitt 10.1.8 die Klasse `Collator` vor, die es erlaubt, beim Vergleich von Strings landes- oder sprachspezifische Besonderheiten zu beachten.

10.1.1 Grundlagen und Normen

Eine sprachabhängige Benutzeroberfläche fällt direkt beim Start eines Programms auf. Für eine gelungene Internationalisierung ist die Darstellung übersetzter Texte jedoch nur ein erster Schritt. Meistens sind weitere regionale Unterschiede in der Darstellung zu berücksichtigen, damit sich ein Benutzer wirklich heimisch fühlt. Am Beispiel der Schweiz wird das besonders deutlich. Eine Internationalisierung lediglich über die Angabe einer Sprache ist dort unmöglich, da zumindest vier verschiedene

Sprachen gesprochen werden – alle mit ihren lokalen Besonderheiten, etwa verschiedenen Formaten bei Zahlen und Währungen.

Um in einer Anwendung verschiedene Übersetzungen anbieten zu können, ist es erforderlich, Sprachen und Regionen beschreiben und unterscheiden zu können. Betrachten wir im Folgenden einige Details dazu.

Sprache

Verschiedene Sprachen lassen sich mithilfe der Norm ISO 639 unterscheiden. Dort werden Kennungen für Sprachen in Form von zwei Kleinbuchstaben definiert, etwa `de` für Deutsch, `en` für Englisch, `fr` für Französisch, `it` für Italienisch usw.

Land und Region

Zur Codierung von Ländern wird die Norm ISO 3166 verwendet. Diese beschreibt Länder über zwei Großbuchstaben. Einige Beispiele gebräuchlicher Kennungen sind: `DE` für Deutschland, `GB` für Großbritannien, `FR` für Frankreich, `IT` für Italien usw.

Zahlen

Die Formatierung von Zahlen ist regional unterschiedlich. Am auffälligsten sind die verschiedenen Dezimaltrennzeichen (oder Trennzeichen für Nachkommastellen). Je nach Region wird ein Punkt oder ein Komma verwendet. Auch das Tausendertrennzeichen kann unterschiedlich sein. In der Schweiz ist etwa ein Apostroph statt eines Punkts gebräuchlich.

Währungen

Die Darstellung von Währungen variiert von Region zu Region. Dabei ist sowohl die Position als auch das Währungskürzel selbst verschieden:

```
Deutschland:           12.345,67 €
USA:                   $12,345.67
Italien:               € 12.345,67
Italienische Schweiz:  SFr. 12'345.67
```

Wie man sieht, sind die Zahlenwerte außerdem unterschiedlich formatiert.

Datum und Uhrzeit

Es existieren weltweit einige Varianten bei der Darstellung von Datum und Uhrzeit. Eine Datumsangabe 01/03/05 kann man als 1. März 2005 (Tag / Monat / Jahr), als 3. Januar 2005 (Monat / Tag / Jahr) oder aber als 5. März 2001 (Jahr / Monat / Tag) deuten. Um zumindest bei der Jahreszahl Verwechslungen zu vermeiden, ist die Angabe einer vierstelligen Jahreszahl ratsam.

Die Norm ISO 8601 definiert Formate für numerische Datums- und Zeitangaben und schlägt folgende Vereinheitlichung vor:

```
Datum:              2010-08-26 (YYYY-MM-DD)
Uhrzeit:            20:08:16   (hh:mm:ss)
```

In der Regel wird eine Applikation eher den regionalen Besonderheiten und nicht der Norm folgen:

```
Deutschland:            26.08.2010  (DD.MM.YYYY)
USA:                    08/26/2010  (MM/DD/YYYY)
```

Achtung: In Java werden in der Klasse* `DateFormat` *von den gezeigten Kürzeln abweichende Kürzel verwendet! Tatsächlich führt die Verwendung eines großen `Y` sogar zu einem unerwarteten Verhalten – später mehr dazu.

10.1.2 Die Klasse `Locale`

Die Klasse `java.util.Locale` hilft bei der Beschreibung landes- oder sprachabhängiger Besonderheiten: Es wird unter anderem spezifiziert, welches Format ein Datum besitzt. Auch die Formatierung von Zahlen wird festgelegt. Jedes `Locale`-Objekt repräsentiert eine bestimmte Region (geografisch, kulturell usw.) inklusive der dort gesprochenen Sprache. Für einige gebräuchliche `Locale`-Objekte existieren vordefinierte Konstanten. Weitere `Locale`-Objekte lassen sich durch einen Konstruktoraufruf mit der Angabe von Sprache und Land erzeugen:

```
public Locale(String language, String country)
```

Die Angaben folgen den Normen ISO 639 und ISO 3166 und sind relativ intuitiv, wie dies folgendes Listing zeigt:

```
public final class LocaleExample
{
    public final class Locale LOCALE_US             = Locale.US;

    public final class Locale LOCALE_FRENCH_SWISS   = new Locale("fr", "CH");
    public final class Locale LOCALE_ITALIAN_SWISS  = new Locale("it", "CH");

    public final class Locale LOCALE_SPANISH        = new Locale("es");
    public final class Locale LOCALE_SPANISH_MEXICO = new Locale("es", "MX");
    // …
```

Die Angabe des Landes an den `Locale`-Konstruktor ist optional. Wird hier ein Leerstring übergeben oder der überladene Konstruktor mit nur einem Parameter für die Sprache aufgerufen, so repräsentiert das derart erzeugte `Locale`-Objekt jedoch lediglich eine Sprache *ohne* Landesinformation. Das sehen wir im Beispiel für die Konstante `LOCALE_SPANISH`.

Einstellungen zu Land und Sprache kann man über folgende Methoden der Klasse `Locale` ermitteln:

- `getLanguage()` – Liefert das Sprachkürzel, beispielsweise `de`, `fr`, `en` usw.
- `getCountry()` – Ermittelt das Land. Der Rückgabewert kann leer sein, wenn

es sich um eine reine »Sprach«-`Locale` handelt.

Neben den Kürzeln sind häufig die tatsächlichen Namen der Länder und die der dort gesprochenen Sprachen interessant. Eine menschenlesbare, textuelle Repräsentation liefern die zwei Methoden `getDisplayCountry()` und `getDisplayLanguage()`, für die jeweils eine überladene Variante mit einem `Locale`-Objekt als Parameter existiert. Die parameterlose Variante liefert eine textuelle Repräsentation für die momentan in der JVM gewählte Default-Locale – was in Deutschland meistens der vordefinierten `Locale.GERMANY` entspricht. Durch Aufruf der überladenen Methoden kann eine lesbare Darstellung in einer beliebigen anderen Sprache erfolgen, die durch ein übergebenes `Locale`-Objekt spezifiziert wird. Verdeutlichen wir uns dies anhand eines Beispiels. Nehmen wir folgende Aufrufe an:

```
public static void main(final String[] args)
{
    System.out.println(LOCALE_US.getDisplayCountry());
    System.out.println(LOCALE_US.getDisplayCountry(LOCALE_ITALIAN_SWISS));
    System.out.println(LOCALE_US.getDisplayLanguage());
    System.out.println(LOCALE_US.getDisplayLanguage(LOCALE_ITALIAN_SWISS));
}
```

Listing 10.1 *Ausführbar als* **'LOCALEEXAMPLE'**

Führt man das Programm LOCALEEXAMPLE aus, so werden Informationen zu Land und Sprache zu den zuvor definierten `Locale`-Objekten auf der Konsole ausgegeben:

```
Vereinigte Staaten von Amerika
Stati Uniti
Englisch
inglese
```

Zugriff auf alle verfügbaren `Locale`-Objekte

Alle verfügbaren `Locale`-Objekte kann man durch Aufruf der statischen Methode `getAvailableLocales()` der Klasse `Locale` ermitteln. Beachten Sie bitte, dass das zurückgelieferte Array von `Locale`-Objekten unsortiert ist!

Einsatz der Klasse

Zur Demonstration der bislang vorgestellten Methoden schreiben wir ein Beispielprogramm, das alle verfügbaren `Locale`-Objekte in einer Tabelle auflistet und einige Informationen sprachabhängig gemäß der Auswahl einer beliebigen Sprache darstellt. Dazu sind zunächst zwei Funktionalitäten zu entwickeln:

1. Die Liste der dargestellten Informationen soll alphabetisch sortiert werden. Von Hause aus ist die Klasse `Locale` nicht sortierbar, da sie das Interface `Comparable<T>` nicht erfüllt und damit auch keine natürliche Ordnung definiert. Zur Realisierung eines Sortierkriteriums muss man daher das Interface `Comparator<T>` nutzen (vgl. Abschnitt 5.1.8).

2. Zur Darstellung der Sprachinformationen sollen aus der Gesamtmenge der verfügbaren `Locale`-Objekte all diejenigen herausgefiltert werden, die lediglich Sprachinformationen anbieten.

Sortierung

Im praktischen Einsatz wäre eine Sortiermöglichkeit wünschenswert, die jedoch standardmäßig nicht existiert. Im Fall der Klasse `Locale` bleibt nur ein `Comparator<Locale>` als Alternative: Die bereits bekannten Methoden `getLanguage()` und `getCountry()` nutzen wir zur Definition eines Sortierkriteriums gemäß Sprache und Land. Folgendes Listing zeigt die Realisierung als anonyme Klasse, die innerhalb einer hier nicht gezeigten Utility-Klasse `LocaleUtils` erfolgt:

```
// Diamond Operator für anonyme Klassen nicht möglich
public static final Comparator<Locale> LOCALE_COMPARATOR =
                                        new Comparator<Locale>()
{
    public int compare(final Locale locale1a, final Locale locale2)
    {
        final int val =
locale1.getLanguage().compareTo(locale2.getLanguage());
        if (val == 0)
            return locale1.getCountry().compareTo(locale2.getCountry());

        return val;
    }
};
```

Filterung

Für die Darstellung von `Locale`-Objekten in einer Combobox sollen nur die `Locale`-Objekte genutzt werden, die reine Sprach-Locales darstellen, d. h., deren Land unbesetzt ist. In diesem Fall verwenden wir eine recht einfache Umsetzung. Da keine Erweiterungen (z. B. eine Kombination von Filterbedingungen) erfolgen sollen, ist dieser Ansatz vollkommen ausreichend. Folgendes Listing zeigt die Methode `getLanguageOnlyLocales()`, die ebenfalls in der Utility-Klasse `LocaleUtils` realisiert ist:

```
public static final List<Locale> getLanguageOnlyLocales()
{
    final List<Locale> sortedLocaleList = getSortedLocales();

    final List<Locale> filteredLocaleList = new ArrayList<>();
    for (final Locale currentLocale : sortedLocaleList)
    {
        if (currentLocale.getCountry().isEmpty())
            filteredLocaleList.add(currentLocale);
    }

    return filteredLocaleList;
}
```

Die Beispielapplikation

Wir kombinieren die vorgestellten Bausteine zu einem Beispielprogramm, das über eine Combobox eine Sprachauswahl ermöglicht und verschiedene Informationen zu `Locale`-Objekten in einer Tabelle anzeigt. In Abbildung 10-1 ist ein Screenshot gezeigt.

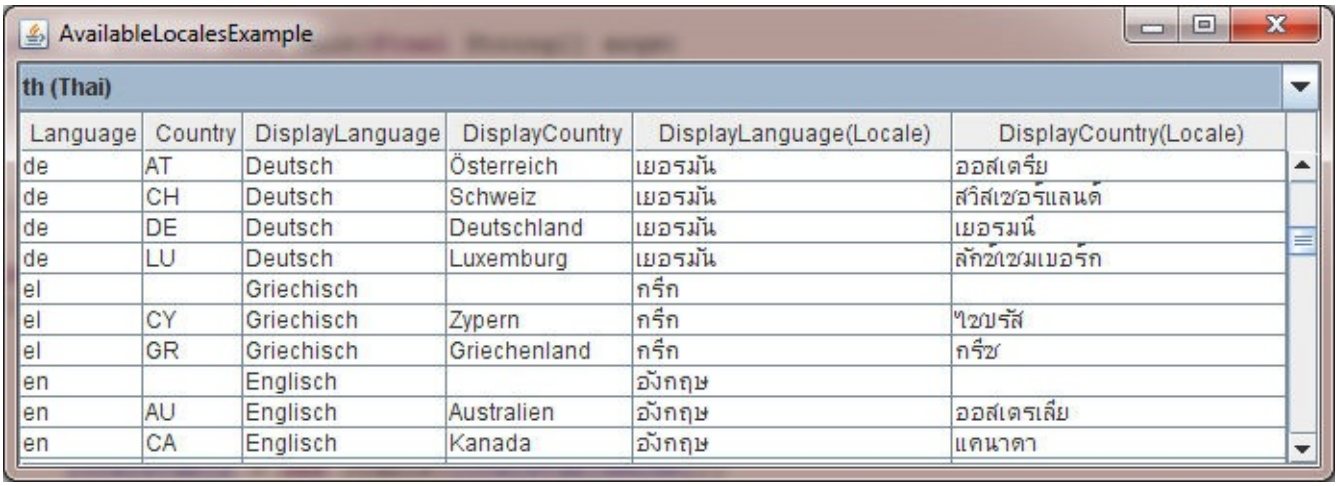


Abbildung 10-1 *Beispielapplikation* AVAILABLELOCALESEXAMPLE

Zentral im Fenster wird eine Tabelle dargestellt, die alle verfügbaren `Locale`-Objekte auflistet. Die einzelnen Spalten werden durch die uns bereits bekannten Methoden, unter anderem `getDisplayLanguage()`, mit Werten versorgt. Im oberen Teil des Fensters bietet eine Combobox Zugriff auf alle vordefinierten Sprach-Locales. Eine Auswahl aktiviert die entsprechende Locale. Dadurch werden in der Tabelle entsprechende, Locale-spezifische Informationen in den hinteren beiden Spalten angezeigt. Die im Beispiel gewählte Locale ist Thai. Daher kommt es zu den exotischen Schriftzeichen in den beiden letzten Spalten. Starten Sie das Programm AVAILABLELOCALESEXAMPLE, um eigene Experimente durchzuführen.

10.1.3 Die Klasse `PropertyResourceBundle`

Sollen in einer Applikation Texte entsprechend einer gewählten Sprache und Region angezeigt werden, sind dazu einige Vorarbeiten durchzuführen. Wünschenswert ist eine Trennung von dargestellten Informationen und dem eigentlichen nutzenden Sourcecode. Eine Repräsentation von Texten der Benutzeroberfläche als Magic Strings schließt sich daher von selbst aus. Statt die benötigten Übersetzungen direkt im Programm zu hinterlegen, ist es sinnvoll, im Sourcecode lediglich einige Konstanten zu definieren, die eindeutige Schlüssel zur Identifikation von Texten und Informationen darstellen. Die eigentlichen Übersetzungen werden in einer Datei[1] abgelegt und können über die Schlüssel abgerufen werden.

Definition verschiedener Sprachdateien

Die Klasse `java.util.PropertyResourceBundle` erweitert die abstrakte

Klasse `java.util.ResourceBundle` und nutzt die Klasse `Properties` zum Einlesen von Schlüssel-Wert-Paaren aus Dateien (vgl. Abschnitt 6.5.2). Betrachten wir zunächst mögliche Eingaben, bevor wir die Verarbeitung detaillierter kennenlernen. Deutsche Texte könnten wie folgt definiert sein:

```
txt_file=Datei
txt_new=Neu…
txt_open=Öffnen…
txt_properties=Einstellungen…
txt_quit=Beenden
```

Listing 10.2 **CONFIG/RESOURCES/PDFEDITOR_DE_DE.PROPERTIES**

Das englische Gegenstück dazu ist:

```
txt_file=File
txt_new=New
txt_open=Open…
txt_properties=Properties…
txt_quit=Quit
```

Listing 10.3 **CONFIG/RESOURCES/PDFEDITOR_EN_GB.PROPERTIES**

Einlesen der Texte

Das Einlesen der passenden Sprachdateien erledigt die statische Methode `getBundle(String, Locale)` der Klasse `PropertyResourceBundle`. Der Methode wird lediglich der Pfad und das Präfix der gewünschten Sprachdatei sowie ein `Locale`-Objekt übergeben:[2]

```
final String BUNDLE_BASENAME = "config.resources.PDFEditor";

final ResourceBundle germanBundle = PropertyResourceBundle.getBundle(
                                                BUNDLE_BASENAME, Locale.GERMANY);
final ResourceBundle englishBundle = PropertyResourceBundle.getBundle(
                                                BUNDLE_BASENAME, Locale.UK);
```

Die in der Locale genutzten Sprach- und Landesinformationen ergeben einen Dateinamen im Format 'Name_<Sprachkürzel>_<Länderkürzel>.properties' und die dadurch referenzierte Property-Datei wird eingelesen.[3]

Dies ist sehr praktisch: Verschiedene Sprachdateien lassen sich allein durch Variation des übergebenen `Locale`-Objekts ansprechen. Zur Unterstützung mehrerer Sprachen muss lediglich für jede gewünschte Sprache eine Property-Datei angelegt werden, wie dies zuvor für eine deutsche und englische Locale gezeigt wurde.

Zugriff auf die Texte

Der Zugriff auf die so hinterlegten Texte erfolgt über spezielle Zugriffsschlüssel, die in einer Konstantenklasse oder besser einer `enum`-Aufzählung wie folgt definiert werden:

```
public enum ResourceKeys
{
    txt_file, txt_new, txt_open, txt_properties, txt_close, txt_save, txt_quit
}
```

Solche Zugriffsschlüssel werden genutzt, um den jeweils zugehörigen sprachspezifischen Text in einem `PropertyResourceBundle` zu suchen. Wir definieren diese Funktionalität als Methode `getLangString( ResourceBundle, ResourceKeys)` in der folgenden Utility-Klasse `ResourceBundleUtils`:

```
public final class ResourceBundleUtils
{
    private static final String INDICATOR_MISSING_RESOURCE = "?";
    private static final String INDICATOR_MISSING_KEY = "??";

    public static String getLangString(final ResourceBundle resourceBundle,
                                       final ResourceKeys key)
    {
        if (resourceBundle != null)
        {
            try
            {
                return resourceBundle.getString(key.name());
            }
            catch (final MissingResourceException e)
            {
                return INDICATOR_MISSING_KEY + key;
            }
        }

        return INDICATOR_MISSING_RESOURCE + key;
    }
    // …
```

Beispiel

Die folgende `main()`-Methode verwendet die zuvor vorgestellten Methoden und liest zwei verschiedene Sprachdateien ein. Anschließend werden zwei Übersetzungen für die Konstanten `txt_file` und `txt_open` auf Deutsch und Englisch ausgegeben:

```
public static void main(final String[] args) throws MissingResourceException
{
   final String BUNDLE_BASENAME = "config.resources.PDFEditor";

   showTexts(PropertyResourceBundle.getBundle(BUNDLE_BASENAME,
Locale.GERMANY));
   showTexts(PropertyResourceBundle.getBundle(BUNDLE_BASENAME, Locale.UK));
}

private static void showTexts(final ResourceBundle resourceBundle)
{
   final String txt_file = ResourceBundleUtils.getLangString(resourceBundle,
                                                    ResourceKeys.txt_file);
   final String txt_open = ResourceBundleUtils.getLangString(resourceBundle,
                                                    ResourceKeys.txt_open);
   System.out.println("txt_file='" + txt_file + "' / " +
                      "txt_open='" + txt_open + "'");
```

```
}
```

Listing 10.4 *Ausführbar als* **'PROPERTYRESOURCEBUNDLEEXAMPLE'**

Es kommt zu folgender Ausgabe auf der Konsole:

```
txt_file='Datei' / txt_open='Öffnen…'
txt_file='File' / txt_open='Open…'
```

Ausblick

Diese Einführung hat einen Anwendungsfall nicht betrachtet: In der Regel sind nicht nur einzelne Begriffe, sondern Kombinationen von Wörtern oder ganze Sätze zu lokalisieren. Aufgrund unterschiedlicher Grammatikregeln verschiedener Sprachen kann man ganze Sätze nicht durch einfaches Verketten von übersetzten Teilstrings zusammenfügen. Stattdessen nutzt man die Möglichkeit, ganze Sätze in Property-Dateien abzulegen und an einigen Stellen Platzhalter einzufügen. Wie man dann mithilfe der Klasse `MessageFormat` auf diese Weise ganze Sätze sprachabhängig gestalten kann, zeigt Abschnitt 10.1.7.

Die hier eingeführte Basisfunktionalität werden wir später wieder aufgreifen und zu einer kleinen internationalisierten Applikation ausbauen.

10.1.4 Formatierte Ein- und Ausgabe

In Abschnitt 4.3.3 haben wir zur Formatierung von Zahlen, Datumsangaben usw. die Methode `format(String, Object…)` der Klasse `String` sowie die Methoden `format(String, Object…)` und `printf(String, Object…)` der Klasse `PrintStream` kennengelernt. Eine flexiblere und objektorientierte Lösung ist durch den Einsatz der im Folgenden vorgestellten `Format`-Klassen aus dem Package `java.text` möglich.

Überblick über die `Format`-Klassen

Zum Teil sind die Ausgaben von Programmen individuell zu formatieren oder auf die Bedürfnisse verschiedener Nationalitäten anzupassen. Die abstrakte Basisklasse `Format` und ihre Subklassen `DateFormat`, `MessageFormat` und `NumberFormat` stellen dazu notwendige Funktionalität bereit. Die Klasse `MessageFormat` kann zur Anpassung von Ausgaben weitere `Format`-Klassen verwenden. Abbildung 10-2 visualisiert die Klassenhierarchie.

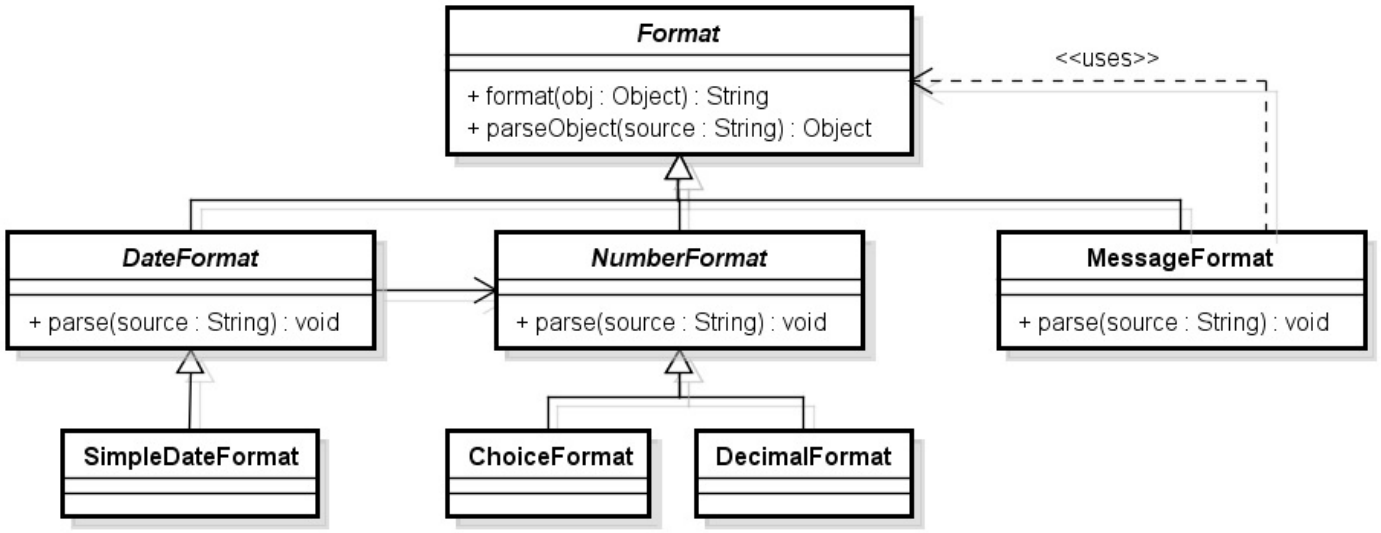


Abbildung 10-2 *Klassenhierarchie der* `Format`*-Klassen*

Durch Einsatz der `Format`-Klassen lassen sich nicht nur landes- bzw. sprachabhängige Ausgaben realisieren, sondern es können auch Zeichenketten wieder in Typen wie `Number` oder `Date` zurückgewandelt werden. Die Basisklasse `Format` bietet dazu folgende zwei Methoden:

```
public String format(Object obj)
public Object parseObject(String source) throws ParseException
```

Die Methode `format(Object)` konvertiert übergebene Objekte in einen String. Durch Aufruf der Methode `parseObject(String)` wird versucht, einen String in ein spezielles Objekt zu konvertieren. Die Umwandlung beliebiger Strings mit Datums- und Zeitinformationen kann jedoch fehlschlagen, wenn die Eingabe nicht dem erwarteten sprachabhängigen Format entspricht. In solchen Fällen wirft die Methode `parseObject(String)` eine `java.text.ParseException`.

Achtung: API-Merkwürdigkeit der `Format`-Klassen

In den Subklassen der Basisklasse `Format` wird zusätzlich eine öffentliche Methode `parse(String)` angeboten. Diese wiederum wird durch die Methode `parseObject(String)` aufgerufen. In Applikationen spielt letztgenannte Methode kaum eine Rolle: Stattdessen wird meistens direkt mit den `parse(String)`-Methoden der Subklassen gearbeitet.

Achtung: `format()`-Inkompatibilität mit den `Format`-Klassen

Weder die Methode `format(String, Object...)` der Klasse `String` noch die Methoden `format(String, Object...)` und `printf(String, Object...)` der Klasse `PrintStream` verwenden `Format`-Objekte, sondern nutzen die Klasse `Formatter`. Diese bietet eine weitgehende Kompatibilität zu `printf()`-Ausgaben in C/C++. Die eingesetzten Formatzeichen sind dabei jedoch unterschiedlich zu denen der `Format`-Klassen.

10.1.5 Zahlen und die Klasse `NumberFormat`

Bei der Verarbeitung von Zahlen, Prozentangaben und Währungen hilft die Klasse `NumberFormat`.

Formatierte Ausgabe

Zur Formatierung von Ausgaben können über verschiedene Fabrikmethoden Instanzen der Klasse `NumberFormat` erzeugt werden:

- **`getInstance()`** und **`getNumberInstance()`** – Beide stellen ein Format für Gleitkommazahlen bereit, das standardmäßig auf drei Stellen nach dem Komma rundet. Mit der Methode `setMaximumFractionDigits(int)` kann die Anzahl an Nachkommastellen eingestellt bzw. über `getMaximumFractionDigits()` abgefragt werden. Analog dazu existieren Methoden, um die minimale Anzahl an Nachkommastellen festzulegen bzw. zu ermitteln.
- **`getIntegerInstance()`** – Erzeugt ein Format, das den Nachkommateil abschneidet und die Zahl vorher rundet.
- **`getCurrencyInstance()`** – Erzeugt ein Format für Währungen, das auf zwei Nachkommastellen rundet.
- **`getPercentInstance()`** – Dient zum Darstellen von Prozentwerten. Dazu wird die übergebene Zahl auf zwei Stellen nach dem Komma gerundet und dann mit 100 multipliziert: Beispielsweise wird die Zahl 0.7271 als 73% dargestellt.

Um zusätzlich zu den voreingestellten Landesspezifika andere Formate bei der Ausgabe zu berücksichtigen, existieren jeweils überladene Methoden mit einem `Locale`- Objekt als Parameter. Das erinnert an die Realisierungen der Methoden `getDisplayLanguage()` und `getDisplayCountry()` in der Klasse `Locale`.

Beispiel

Das folgende Beispielprogramm NUMBERFORMATEXAMPLE nutzt die vorgestellten Fabrikmethoden, um den Einsatz verschiedener Formate zur Ausgabe von Zahlen und Währungen zu zeigen. Dabei werden die lokalen Unterschiede am Beispiel von Italien und der italienischen Schweiz demonstriert. Auch außereuropäische Besonderheiten werden mit den Locales für die USA und Hongkong gezeigt:

```
public static void main(final String[] args)
{
    final double value = 12345.67890;

    NumberFormat numFormat = NumberFormat.getInstance(Locale.GERMANY);
    System.out.println("getInstance de DE\t\t" + numFormat.format(value));

    numFormat = NumberFormat.getNumberInstance(Locale.US);
```

```
    System.out.println("getNumberInstance en US\t" + numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(Locale.GERMANY);
    System.out.println("getCurrencyInstance de DE\t" +
numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(Locale.US);
    System.out.println("getCurrencyInstance en US\t" +
numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(Locale.ITALY);
    System.out.println("getCurrencyInstance it IT\t" +
numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(LocaleExample.
          LOCALE_ITALIAN_SWISS);
    System.out.println("getCurrencyInstance it CH\t" +
numFormat.format(value));

    numFormat = NumberFormat.getCurrencyInstance(new Locale("zh", "HK"));
    System.out.println("getCurrencyInstance zh HK\t" +
numFormat.format(value));

    // Zahlenformat stammt aus Sprache, Währungszeichen sprachabhängig aus
Land
    numFormat = NumberFormat.getCurrencyInstance(new Locale("de", "HK"));
    System.out.println("getCurrencyInstance de HK\t" +
numFormat.format(value));
}
```

Listing 10.5 *Ausführbar als* **'NUMBERFORMATEXAMPLE'**

Dieses Beispiel verdeutlicht, dass die lokalen Besonderheiten (Währungsinformationen, Zahlenformate etc.) vom Benutzer nicht angegeben werden müssen. Es reicht vielmehr aus, ein `Locale`-Objekt an die entsprechenden Methoden der `Format`-Klassen zu übergeben. Für den Wert `12345.67890` kommt es zu folgenden Ausgaben:

```
getInstance de DE           12.345,679
getNumberInstance en US     12,345.679
getCurrencyInstance de DE   12.345,68 €
getCurrencyInstance en US   $12,345.68
getCurrencyInstance it IT   € 12.345,68
getCurrencyInstance it CH   SFr. 12'345.68
getCurrencyInstance zh HK   HK$12,345.68
getCurrencyInstance de HK   HKD 12.345,68
```

Man erkennt diverse Unterschiede in der Formatierung der Ausgabe. Eine Besonderheit wird bei der (künstlichen) deutschsprachigen Locale für Hongkong sichtbar: Die Sprache bestimmt das Format der Zahlen. Das Land bestimmt das bzw. die Währungszeichen, wobei diese sprachabhängig sind – im Beispiel also etwa `HK$` bzw. `HKD` für den Hongkong-Dollar.

All diese sprach- und landesabhängigen Spezialitäten von Hand zu realisieren, wäre sehr aufwendig und fehleranfällig. Insbesondere gilt das auch für das Parsing derartiger Informationen. Auch dabei unterstützt die Klasse `NumberFormat`.

Achtung: Zweifelhaftes OO-Design im JDK

Einige Fabrikmethoden der Klasse `NumberFormat` erzeugen Objekte vom spezielleren Typ `DecimalFormat`. Es ist ein fragwürdiges OO-Design, wenn eine Basisklasse ihre Subklassen kennt und diese sogar erzeugt. Korrekterweise hätten die Fabrikmethoden in der Klasse in `DecimalFormat` definiert werden sollen.

Parsen von Strings – Umwandlung in `Number`-Objekte

Die Unterstützung verschiedener Formate ist bei der Auswertung von Zeichenketten und der Umwandlung in Objekte noch deutlich wichtiger als bei der Ausgabe. Dieser Abschnitt geht genauer darauf ein.

Um textuell vorliegende Informationen in korrespondierende Objekte umzuwandeln, verwendet man die Methode `parse(String)`. Dabei ist zu beachten, dass je nach gewählter Locale eine Zahl mit Komma "`1,2`" oder Punkt "`1.2`" unterschiedlich interpretiert wird! Folgendes Beispielprogramm verdeutlicht dies anhand der Auswertung der Eingaben "`123,456,789`" und "`123.456.789`" für die Zahlenformate für Deutschland, Frankreich und die USA.

```
public static void main(final String[] args) throws ParseException
{
    final NumberFormat DE_FORMAT = NumberFormat.getInstance(Locale.GERMANY);
    final NumberFormat FR_FORMAT = NumberFormat.getInstance(Locale.FRANCE);
    final NumberFormat US_FORMAT = NumberFormat.getInstance(Locale.US);

    final String[] values = { "123,456,789", "123.456.789" };
    for (final String number : values)
    {
        System.out.println("Value " + number);
        System.out.println("NumberInstance DE  " + DE_FORMAT.parse(number));
        System.out.println("NumberInstance FR  " + FR_FORMAT.parse(number));
        System.out.println("NumberInstance US  " + US_FORMAT.parse(number));
    }
}
```

Listing 10.6 *Ausführbar als* **'NUMBERFORMATPARSEEXAMPLE'**

Führt man das Programm NUMBERFORMATPARSEEXAMPLE aus, so erscheinen folgende Ausgaben auf der Konsole:

```
Value 123,456,789
NumberInstance DE   123.456
NumberInstance FR   123.456
NumberInstance US   123456789
Value 123.456.789
NumberInstance DE   123456789
NumberInstance FR   123
NumberInstance US   123.456
```

Mehrfache Angaben eines Trennzeichens für die Nachkommastelle oder die Tausendertrennung werden jeweils nur einmal ausgewertet. Erwartete Unterschiede findet man bei der amerikanischen und den beiden europäischen Formaten. Unerwartet ist jedoch die unterschiedliche Auswertung der punktseparierten Zahl für die deutsche und die französische Locale.

10.1.6 Datumswerte und die Klasse `DateFormat`

Die Basisklasse `DateFormat` und ihre konkrete Subklasse `SimpleDateFormat` erleichtern die formatierte Ausgabe von Datumswerten. Die beiden Klassen ermöglichen zudem die Konvertierung von Strings, die Datums- und/oder Zeitinformationen enthalten, in ein Objekt vom Typ `Date`.

Formatierte Ausgabe

Zur Ausgabe von Datumswerten stellen folgende Fabrikmethoden verschiedene Arten von Datumsformaten bereit:

- **`getInstance()`** – Formatierung von Datums- und Zeitangaben
- **`getTimeInstance()`** – Formatierung von Zeitangaben
- **`getDateInstance()`** – Formatierung von Datumsangaben
- **`getDateTimeInstance()`** – Formatierung von Datums- und Zeitangaben

Es existieren überladene Versionen dieser Methoden. Dort kann durch die Angabe der Parameter `SHORT`, `MEDIUM`, `LONG` und `FULL` eine Steuerung des Detailgrads der auszugebenden Informationen erfolgen. Für `getDateTimeInstance()` lässt sich sogar der Detailgrad der Datums- und Zeitangabe separat spezifizieren.

Folgendes Beispielprogramm nutzt diese Methoden[4] und zeigt, wie man die Varianten auf elegante Art und Weise durch die Definition zweier Arrays beschreiben kann. Dadurch erspart man sich viel Sourcecode-Duplikation, die ansonsten zur Realisierung notwendig gewesen wäre.

```
public static void main(final String[] args)
{
   // Aktuelles Datum erzeugen
   final Date now = new Date();

   // Ausgabe mit toString()
   System.out.println("Date.toString()        " + now.toString());

   // Ausgabe mit DateFormat
   System.out.println("\nDateFormat");
   System.out.println("getInstance()          " + getInstance().format(now));
   System.out.println("getTimeInstance()      " +
getTimeInstance().format(now));
   System.out.println("getDateInstance()      " +
getDateInstance().format(now));
   System.out.println("getDateTimeInstance()  " + getDateTimeInstance().format(
         now));


   // Definition aller Varianten (int-Werte auf der Klasse DateFormat)
   final int[] styles = { SHORT, MEDIUM, LONG, FULL };
   // Achtung: Index gemäß Definition der DateFormat-Konstanten
   // Daher ist die Reihenfolge hier anders als in styles
```

```
    final String[] styleNames = { "FULL  ", "LONG  ", "MEDIUM", "SHORT " };

    // Zeige alle Varianten von DateFormat.getTimeInstance()
    System.out.println("\nDateFormat.getTimeInstance()");
    for (final int currentStyle : styles)
    {
        System.out.println(styleNames[currentStyle] + "\t\t\t" +
getTimeInstance(
                currentStyle).format(now));
    }

    // Zeige alle Varianten von DateFormat.getDateInstance()
    System.out.println("\ngetDateInstance()");
    for (final int currentStyle : styles)
    {
        System.out.println(styleNames[currentStyle] + "\t\t\t" +
getDateInstance(
                currentStyle).format(now));
    }
    // Zeige alle Varianten von DateFormat.getDateTimeInstance()
    System.out.println("\nDateFormat.getDateTimeInstance()");
    for (final int currentDateStyle : styles)
    {
        for (final int currentTimeStyle : styles)
        {
            System.out.println(styleNames[currentDateStyle] + " / " +
styleNames[
                    currentTimeStyle] + "\t\t" + getDateTimeInstance(
                    currentDateStyle, currentTimeStyle).format(now));
        }
    }
}

    private DateFormatExample()
    {
    }
}
```

Listing 10.7 *Ausführbar als* **'DATEFORMATEXAMPLE'**

Das Programm DATEFORMATEXAMPLE gibt in etwa folgendes aus (gekürzt):

```
Date.toString()                     Fri Aug 20 14:44:29 CEST 2010

DateFormat
getInstance()                       20.08.10 14:44
getTimeInstance()                   14:44:29
getDateInstance()                   20.08.2010
getDateTimeInstance()               20.08.2010 14:44:29
…
DateFormat.getDateTimeInstance()
..
FULL   / SHORT                      Freitag, 20. August 2010 14:44
FULL   / MEDIUM                     Freitag, 20. August 2010 14:44:29
FULL   / LONG                       Freitag, 20. August 2010 14:44:29 MESZ
FULL   / FULL                       Freitag, 20. August 2010 14:44 Uhr MESZ
```

Interessanterweise entspricht die Ausgabe der Methode **`toString()`** ***der Klasse*** **`Date`** ***keinem der vordefinierten Ausgabeformate!*** Somit kann man mit den Standardformaten demnach die von `Date` ausgegebenen Datumswerte nicht parsen.

Weiterhin überraschen die Ausgaben bei der Wahl von `FULL` für die Zeitangabe,

denn diese enthalten keine Sekundenangaben, aber den Text "Uhr".

Parsen von Strings – Umwandlung in `Date`-Objekte

Um textuell vorliegende Datumsinformationen in Objekte umzuwandeln, verwendet man die Methode `parse(String)` der Klasse `DateFormat`. Je nach gewählten Einstellungen werden verschiedene Datumsdarstellungen akzeptiert. Dies ist im folgenden Listing DATEFORMATPARSEEXAMPLE für die spezielle Datumsangabe "`32.12.2008 20:14:55 GMT+7`" gezeigt. Hier wird ein mittleres Datumsformat und ein kurzes Zeitformat zum Parsing benutzt – auf eine landesspezifische Anpassung durch Übergabe eines `Locale`-Objekts wird aus Gründen der Übersichtlichkeit verzichtet:

```
public static void main(final String[] args)
{
    final String dateString = "32.12.2008 20:14:55 GMT+7";

    final DateFormat format =
DateFormat.getDateTimeInstance(DateFormat.MEDIUM,
                                                    DateFormat.SHORT);
    format.setLenient(true);

    try
    {
        final Date date = format.parse(dateString);
        System.out.println("Original string:        " + dateString);
        System.out.println("Formatted parsed date:  " + format.format(date));
        System.out.println("Parsed date:            " + date);
    }
    catch (final ParseException ex)
    {
        System.out.println("ERROR: could not parse date '" + dateString +
"'");
    }
}
```

Listing 10.8 *Ausführbar als* **'DATEFORMATPARSEEXAMPLE'**

Der Standardparser ist großzügig: Werden keine Angaben zu Teilbestandteilen des Datums gemacht, so werden hier Defaultwerte angenommen. Zudem werden einige Werte auf andere Tage oder Uhrzeiten »gerundet«. Beispielsweise wird der 32. Dezember auf den ersten Januar des Folgejahres verschoben.

Die zu verarbeitende Eingabe enthält sowohl eine Zeitzone als auch Sekundenangaben. Da beides nicht Bestandteil des genutzten Datumsformats ist, werden diese Angaben beim Parsing ignoriert. Als Folge werden die Zeitzone auf den Wert '`CET`' und die Sekunden auf den Wert 0 gesetzt:

```
Original string:        32.12.2008 20:14:55 GMT+7
Formatted parsed date:  01.01.2009 20:14
Parsed date:            Thu Jan 01 20:14:00 CET 2009
```

Es ist in gewissem Rahmen möglich, die »Freundlichkeit« des Parsings zu beeinflussen. Durch den Aufruf von `format.setLenient(false)` wird die Eingabe des Werts `32` nicht mehr als Tagesangabe akzeptiert. Es kommt zu einer

ParseException.

Einsatz von SimpleDateFormat für Anpassungen

Die mit den Fabrikmethoden erzeugten DateFormat-Instanzen sind für einige Einsatzgebiete bereits ausreichend. Allerdings gibt es immer wieder Situationen, in denen man ein Datumsformat nach eigenen Bedürfnissen gestalten möchte oder muss. Mithilfe der Klasse SimpleDateFormat wird es möglich, eigene Datumsformate zu definieren, deren Beschreibung durch Angabe eines speziellen Musterstrings erfolgt. Die wichtigsten Formatzeichen sind im Muster 'dd. MMMM yyyy HH:mm:ss' enthalten. Dieses führt beispielsweise zu folgender Ausgabe von Datum und Uhrzeit:

```
20. Dezember 2009 15:54:54
```

Die Anzahl der Zeichen im Muster legt die Art der Ausgabe fest: Vier Zeichen werden als Long-Format interpretiert; drei oder weniger als Short-Format. Wie sich das auswirkt, ist in Tabelle 10-1 für Jahres- und Monatsangaben gezeigt.

Tabelle 10-1 *Einfluss der Musterlänge auf die Darstellung*

Jahreszahl	Monatsinformation
'y' -> '2009'	'M" -> '7' bzw. '12'
'yy' -> '09'	'MM' -> '07'
'yyy' -> '09'	'MMM' -> 'Jul'
'yyyy' -> '2009'	'MMMM' -> 'Dezember'

Überraschenderweise kommt es bei der Angabe eines einstelligen Jahresplatzhalters zu der Ausgabe einer vierstelligen Jahresangabe.

Tipp: Spezialfälle bei der Musterangabe im SimpleDateFormat

Fallstricke Die meisten in der Musterangabe der Klasse SimpleDateFormat verwendeten Formatzeichen sind intuitiv klar und aus den englischen Begriffen für den jeweils gewünschten Einsatzzweck herleitbar. Zu beachten ist, dass hier Groß- und Kleinschreibung eine semantische Bedeutung trägt. Das große 'M' steht für Monatsangaben, wohingegen das kleine 'm' für Minuten steht. Auch eine Verwechslung der Zeichen 'd' und 'D' kann beispielsweise dafür sorgen, dass nicht der Tag im Monat ('d'), sondern der im Jahr ('D') ausgegeben wird.

Es lauern noch weitere Fehlerquellen durch die Platzhalter für Stundenangaben: 0 –

23 ('H') oder 1 – 24 ('k') bzw. 0 – 11 (' K') oder 1 – 12 (' h'). Diese Platzhalter sind alles andere als intuitiv. Es ist daher ratsam, die API-Dokumentation zur Klasse `SimpleDateFormat` zu konsultieren, wenn man sich unsicher ist.

`SimpleDateFormat` und `Date.toString()` Mithilfe der obigen Informationen können wir ein `SimpleDateFormat` definieren, das die Ausgaben von `Date.toString()` lesen kann. Dazu verwenden wir die in der Klasse `Date` in der Methode `toString()` gefundene Formatangabe "`EEE MMM dd HH:mm:ss zzz yyyy`".

Dabei stehen "`EEE`" für den Wochentag und "`zzz`" für die Zeitzone. Damit das Parsing möglich wird, ist allerdings ein amerikanisches `Locale`-Objekt bei der Konstruktion zu übergeben. Die `toString()`-Methode der Klasse `Date` ist patriotisch und nutzt immer ein solches `Locale`-Objekt zur Ausgabe der Zeitzone. Korrekter wäre es gewesen, hier die Zeitzone der momentan in der JVM hinterlegten Default-Locale zu verwenden.

```
public static void main(final String[] args) throws ParseException
{
    // Neues Date-Objekt erzeugen
    final Date now = new Date();

    // In String wandeln und als Vergleichswert ausgeben
    final String dateString = now.toString();
    System.out.println("Now:  " + dateString);

    // Definition des passenden Datumsformats
    final String pattern = "EEE MMM dd HH:mm:ss zzz yyyy";
    final SimpleDateFormat format = new SimpleDateFormat(pattern, Locale.US);

    // Versuche die Ausgabe von Date.toString() zu parsen
    final Date parsed = format.parse(dateString);
    System.out.println("Parsed date:  " + parsed);

    // Prüfe, ob das Format auch bei der Ausgabe korrekt arbeitet
    System.out.println("Formatted date:  " + format.format(now));
}
```

Listing 10.9 *Ausführbar als* **'DATEFORMATPARSETOSTRINGEXAMPLE'**

10.1.7 Textmeldungen und die Klasse `MessageFormat`

Sind Textmeldungen formatiert auszugeben, hilft dabei die Klasse `MessageFormat`. Deren Objekte werden im Unterschied zu den bisher vorgestellten `Format`-Klassen nicht über Fabrikmethoden, sondern einfach über einen Konstruktoraufruf erzeugt. Diesem wird eine Vorlage der auszugebenden Nachricht übergeben:

```
MessageFormat(String pattern)
```

Im Parameter `pattern` können Platzhalter angegeben werden. Diese Platzhalter werden zur Laufzeit mit Werten ersetzt, wenn beim Aufruf der Methode

format(Object) ein Object[] mit Werten übergeben wird. Folgendes Listing zeigt ein einfaches Beispiel dreier Platzhalter:

```
public static void main(final String[] args) throws ParseException
{
    final String textTemplate = "Am {1} ist {0}. {2} Sekt bitte!";
    final MessageFormat messageFormat = new MessageFormat(textTemplate);

    // Date(int year, int month, int date, int hrs, int min)
    // ACHTUNG: auf das Jahr wird der Wert 1900 addiert => 109 == 2009
    final Object[] arguments = { "Karneval", new Date(109,10,11,11,11), 3 };
    final String formattedText = messageFormat.format(arguments);

    System.out.println("Eingabe: " + textTemplate);
    System.out.println("Formatiert: " + formattedText);
}
```

Listing 10.10 *Ausführbar als* **'MESSAGEFORMATEXAMPLE'**

Die Werte aus dem Array werden über Platzhalter wie {0}, {1} etc. in die Nachricht eingefügt. Die Nummern entsprechen den Indizes des Object-Arrays der Argumente. In der Textvorlage können diese Indizes mehrfach und in beliebiger Reihenfolge angegeben werden oder auch ungenutzt bleiben. Wird ein Index angegeben, für den es keinen Eintrag im Object-Array der Werte gibt, so erfolgt keine Ersetzung des Platzhalters.

Startet man das Programm MESSAGEFORMATEXAMPLE, so kommt es zu folgender Ausgabe:

```
Eingabe: Am {1} ist {0}. {2} Sekt bitte!
Formatiert: Am 11.11.09 11:11 ist Karneval. 3 Sekt bitte!
```

Die Klasse MessageFormat bietet auch eine statische Methode format(String, Object…) für die Ausgabe, wodurch man kein MessageFormat-Objekt konstruieren und dessen Objektmethode format(Object) aufrufen muss. Im eigenen Sourcecode ist daher folgende Variante kürzer und besser lesbar:

```
final String formattedString = MessageFormat.format(textTemplate, arguments);
```

Erweiterungen der Formatspezifikation

Platzhalter können nicht nur den Index des Arguments enthalten, sondern auch weitere Informationen zur Formatierung. Mit der Syntax

```
{ArgumentIndex, FormatType}
```

kann die Ausgabe angepasst werden. Dabei wird ein Formatierer des entsprechenden Formattyps number, date, time oder choice erzeugt. Für number entspricht dies beispielsweise NumberFormat.getInstance(getLocale()). Bei Bedarf kann sogar der Formatstil angegeben werden:

```
{ArgumentIndex, FormatType, FormatStyle}
```

Für das Format date sind hier die Werte short, medium, long und full vordefiniert. Korrespondierende Stile sind aus dem DateFormat bekannt. Für das Format number können die Stile integer, currency und percent gewählt

werden, die denen der Klasse `NumberFormat` entsprechen. Verdeutlichen wir uns dies anhand eines Beispiels:

```
public static void main(final String[] args) throws ParseException
{
    final Object[] arguments = { "MessageFormat", 2, 3.1415, new Date() };
    final String textTemplate = "Beispiel für die Klasse {0}. "
                              + "\nWährung: {1, number, currency} "
                              + "\nZahl: {2, number} "
                              + "\nDatum und Uhrzeit: {3, date} um {3,
time}";

    final String formattedText = MessageFormat.format(textTemplate,
arguments);

    System.out.println("Eingabe: " + textTemplate);
    System.out.println();
    System.out.println("Formatiert: " + formattedText);
}
```

Listing 10.11 *Ausführbar als* **'MESSAGEFORMATEXAMPLE2'**

Führt man das Programm MESSAGEFORMATEXAMPLE2 aus, so kommt es zu folgender, aufs Wesentliche gekürzten Ausgabe:

```
Formatiert: Beispiel für die Klasse MessageFormat.
Währung: 2,00 C
Zahl: 3,142
Datum und Uhrzeit: 18.08.2010 um 00:41:03
```

Fazit

Dieser Abschnitt hat einen kurzen Einstieg in das Thema formatierte Ein- und Ausgabe gegeben und soll als Anregung für eigene Experimente dienen. Zudem wurden Grundkenntnisse zur Internationalisierung aufgebaut, die das Verständnis der im Folgenden vorgestellten Themen erleichtern.

10.1.8 Stringvergleiche mit der Klasse `Collator`

Neben der Darstellung von Informationen mithilfe verschiedener `Format`-Klassen ist es insbesondere für Listen oder tabellarische Darstellungen von Interesse, Texte in einer bestimmten Reihenfolge anzuordnen. Der Vergleich von Strings scheint auf den ersten Blick eine einfache Aufgabe zu sein, da die Klasse `String` das Interface `Comparable<String>` erfüllt und damit auch eine Methode `compareTo(String)` bietet. Damit lassen sich Strings in eine gewünschte Reihenfolge bringen – man kann etwa eine Liste von Namen alphabetisch sortieren. Allerdings beruht der Vergleich der einzelnen Strings auf den Unicode-Werten der enthaltenen Zeichen. In einigen Anwendungsfällen reicht dies bereits aus. Es gibt aber viele Anwendungsfälle, in denen diese vorgegebene Art des Vergleichs nicht zweckmäßig ist – das gilt insbesondere dann, wenn landes- oder sprachspezifische Besonderheiten beachtet werden müssen. Schon ein Vergleich deutscher Wörter

gemäß `Comparable<String>` kann unerwartete Ergebnisse liefern. Das liegt daran, dass in der Unicode-Codierung Kleinbuchstaben hinter dem Großbuchstaben 'Z' und Umlaute hinter dem Kleinbuchstaben 'z' angeordnet sind.

Wenn Strings landes- bzw. sprachspezifisch entsprechend einer gewählten Locale zu vergleichen sind, wäre es aufwendig und fehleranfällig, diese Funktionalität selbst zu programmieren. Für sprachabhängige Vergleiche sollte man stattdessen die Klasse `java.text.Collator` einsetzen, die wir nun kennenlernen wollen.

Die Klasse `Collator`

Sprachspezifische Vergleiche von Strings kann man mithilfe der Klasse `Collator` auf elegante Weise realisieren. Eine Instanz davon erhält man über verschiedene Fabrikmethoden, die in der Klasse `Collator` definiert sind. Die Methode `getInstance()` liefert ein `Collator`-Objekt für die Default-Locale, wohingegen die Methode `getInstance(Locale)` ein `Collator`-Objekt für das als Parameter übergebene `Locale`-Objekt zurückgibt.

Die Klasse `Collator` implementiert das Interface `Comparator<Object>` und bietet daher eine Methode `compare(Object, Object)`.[5] Damit lassen sich zwei beliebige Strings miteinander vergleichen und ordnen. Insbesondere fällt damit der häufige Anwendungsfall des Sortierens von Strings leicht. Dazu können die Utility-Methoden `sort(T[], Comparator<? super T>)` und `sort(List<T>, Comparator<? super T>)` aus dem Collections-Framework genutzt werden. Details zum Interface `Comparator<T>` beschreibt Abschnitt 5.1.8.

Zur Verdeutlichung der Funktionalität und Arbeitsweise eines `Collators` soll ein Vergleich von Objekten einer Klasse `Person` realisiert werden.

Beispiel

Die im Listing gezeigte Klasse `Person` erfüllt das Interface `Comparable<Person>` und realisiert daher die Methode `compareTo(Person)`. Diese ermöglicht es, `Person`-Objekte miteinander zu vergleichen und diese zu ordnen. In diesem Beispiel basiert der Vergleich auf dem Attribut `lastname`. Die eigentliche Vergleichsfunktionalität wird hier durch einen `Collator` für `Locale.GERMANY` gemäß deutschen Regeln bereitgestellt:

```
public class Person implements Comparable<Person>
{
    private static final Collator collator =
                                  Collator.getInstance(Locale.GERMANY);

    private final String lastname;

    Person(final String lastname)
    {
        this.lastname = lastname;
    }
```

```
    public int compareTo(final Person compareObject)
    {
        return collator.compare(getLastName(), compareObject.getLastName());
    }
}
```

Vergleich im Detail steuern

Das einführende Beispiel zeigt einen einfachen Anwendungsfall eines `Collators`. Der Vergleich von Strings lässt sich bei Bedarf im Detail beeinflussen. Dazu kann man durch Aufruf der Methode `setStrength(int)` eine sogenannte »Strenge« vorgeben, die bestimmt, wann Zeichen als unterschiedlich gelten sollen. Dazu übergibt man folgende Konstanten:

- `Collator.PRIMARY` – Es werden lediglich verschiedene Buchstaben als unterschiedlich erkannt. Beim Vergleich werden daher Groß- bzw. Kleinschreibung sowie eventuell vorhandene Akzente und Umlaute außer Acht gelassen, d. h., ein 'Ä' wird beim Sortieren wie ein 'a' behandelt.[6]
- `Collator.SECONDARY` – Bei dieser Vergleichsart werden zusätzlich zur Einstellung `PRIMARY` auch Akzente und Umlaute beim Vergleich berücksichtigt. Unterschiede in der Groß- bzw. Kleinschreibung beeinflussen die Sortierung jedoch nicht.
- `Collator.TERTIARY` – Diese Vergleichsart ist »strenger« als die Einstellung `SECONDARY`, denn sie beachtet auch Unterschiede in der Groß- bzw. Kleinschreibung. Es gilt etwa 'a' < 'A'. ***Das ist die Standardeinstellung.***
- `Collator.IDENTICAL` – Hierbei werden alle Unicode-Zeichen als unterschiedlich angesehen. Das gilt auch für verschiedene nicht darstellbare Unicode-Zeichen, die bei den anderen Einstellungen zum Teil als Leerzeichen gewertet werden. Dadurch werden Zeichenketten, die ansonsten gleiche Zeichen enthalten, bei allen anderen Strengegraden als gleich gewertet, selbst wenn diese unterschiedliche nicht darstellbare Unicode-Zeichen enthalten.

Zur Veranschaulichung zeigt folgende Tabelle die Sortierreihenfolge bei einem Vergleich der Strings `Abc` und `Äbc` für Umlaute sowie der Strings `Hallo` und `hallo` für Groß- bzw. Kleinschreibung für verschiedene `Collator`-Strengegrade. Im Anschluss daran erstelle ich ein Beispielprogramm zur weiteren Verdeutlichung.

Tabelle 10-2 *Einflüsse der* `Collator`*-Strengegrade*

Sortierungsvariante	Reihenfolge der Strings
`Comparable<String>`	Abc < Hallo < hallo < Äbc
`PRIMARY`	(Abc = Äbc) < (Hallo = hallo)
`SECONDARY`	Abc < Äbc < (Hallo = hallo)
`TERTIARY`	Abc < Äbc < hallo < Hallo

Das folgende Listing demonstriert die Auswirkungen verschiedener Einstellungen für die »Strenge« auf den Vergleich von Strings. Dazu wird ein `String[]` mit Werten initialisiert, die besonders geeignet sind, die Auswirkungen unterschiedlicher Strengegrade beobachten zu können. Dabei werden auch spezielle Unicode-Zeichen genutzt, damit insbesondere die Einstellung `IDENTICAL` verständlich wird:

```
public final class CollatorStrengthExample
{
    // Diese Werte werden verglichen, inklusive spezieller Unicode-Zeichen
    private static final String[] EXAMPLE_VALUES = { "\u0001BC", "\u0002BC",
                                                     "ABC", "ÄBC", "abc",
                                                     "Maße", "Masse" };

    public static void main(final String[] args)
    {
        System.out.println("CollatorStrengtExample");
        System.out.println("Values: " + Arrays.toString(EXAMPLE_VALUES) +
"\n");

        final Collator collator = Collator.getInstance(Locale.GERMANY);

        for (final CollatorStrength strength : CollatorStrength.values())
        {
            // Achtung: Kopie des Arrays ist wichtig, da der sort-Befehl
            // ansonsten das Ausgangsarray ändern würde!
            sortByCollator(collator, strength, EXAMPLE_VALUES.clone());
        }
    }

    private static void sortByCollator(final Collator collator,
                                       final CollatorStrength
collatorStrength,
                                       final String[] exampleValues)
    {
        // Gemäß Strenge sortieren und ausgeben
        collator.setStrength(collatorStrength.collatorStrengthValue);
        Arrays.sort(exampleValues, collator);
        System.out.println("Collator-Strength: " + collatorStrength + " => " +
                           Arrays.toString(exampleValues));

        // Aufbereiten der Vergleichsordnung
        final String orderingInfo = buildOrderingInfo(collator,
exampleValues);
        System.out.println("Using ordering: [ " + orderingInfo + "]" + "\n");
```

```
    }

    // …

    enum CollatorStrength
    {
        PRIMARY(Collator.PRIMARY), SECONDARY(Collator.SECONDARY),
        TERTIARY(Collator.TERTIARY), IDENTICAL(Collator.IDENTICAL);

        final int collatorStrengthValue;

        CollatorStrength(final int collatorStrengthValue)
        {
            this.collatorStrengthValue = collatorStrengthValue;
        }
    }

    // …
```

Listing 10.12 *Ausführbar als* **'COLLATORSTRENGTHEXAMPLE'**

Startet man das Programm COLLATORSTRENGTHEXAMPLE, kommt es zu folgender, für die bessere Lesbarkeit leicht modifizierten Ausgabe, wobei die zwei besonderen Unicode-Zeichen nicht druckbar sind und daher hier als '?' dargestellt sind:

```
Values: [?BC, ?BC, ÄBC, ABC, abc, Maße, Masse]

Collator-Strength: PRIMARY => [ÄBC, ABC, abc, ?BC, ?BC, Maße, Masse]
Using ordering: [‘`?BC’ = ‘?BC’, ‘?BC’ > ‘ÄBC’, ‘ÄBC’ = ‘ABC’,
                 ‘ABC’ = ‘abc’, ‘abc’ < ‘Maße’, ‘Maße’ = ‘Masse’]

Collator-Strength: SECONDARY => [ABC, abc, ÄBC, ?BC, ?BC, Maße, Masse]
Using ordering: [‘?BC’ = ‘?BC’, ‘?BC’ > ‘ÄBC’, ‘ÄBC’ > ‘ABC’,
                 ‘ABC’ = ‘abc’, ‘abc’ < ‘Maße’, ‘Maße’ = ‘Masse’]

Collator-Strength: TERTIARY => [abc, ABC, ÄBC, ?BC, ?BC, Masse, Maße]
Using ordering: [‘?BC’ = ‘?BC’, ‘?BC’ > ‘ÄBC’, ‘ÄBC’ > ‘ABC’,
                 ‘ABC’ > ‘abc’, ‘abc’ < ‘Maße’, ‘Maße’ > ‘Masse’]

Collator-Strength: IDENTICAL => [abc, ABC, ÄBC, ?BC, ?BC, Masse, Maße]
Using ordering: [‘?BC’ < ‘?BC’, ‘?BC’ > ‘ÄBC’, ‘ÄBC’ > ‘ABC’,
                 ‘ABC’ > ‘abc’, ‘abc’ < ‘Maße’, ‘Maße’ > ‘Masse’]
```

Anhand der Ausgaben erkennen wir den Einfluss des gewählten Strengegrads sehr schön. Zum besseren Verständnis und zum leichteren Nachvollziehen werden die Ergebnisse der Vergleiche textuell dargestellt, etwa `'ABC' = 'abc'`. Diese Informationen werden durch die Methode `buildOrderingInfo(Collator, String[])` aufbereitet. Für den allgemeinen Teil der Darstellung des Vergleichs wird vom konkreten Typ `Collator` abstrahiert und eine generische Hilfsmethode `buildComparatorInfo(Comparator<T>, T, T)` für den allgemeineren Typ `Comparator<T>` definiert. Dadurch ist diese bei Bedarf auch in anderen Anwendungsfällen einsetzbar.

```
private static String buildOrderingInfo(final Collator collator,
                                         final String[] exampleValues)
{
    String orderingInfo = "";
    for (int i = 0; i < exampleValues.length; i += 2)
    {
```

```
        orderingInfo += buildComparatorInfo(collator, exampleValues[i],
                                                      exampleValues[i + 1]);

        // ', ' für alle Einträge hinzufügen, außer letztem
        if (i < exampleValues.length - 2)
            orderingInfo += ", ";
    }
    return orderingInfo;
}

public static <T> String buildComparatorInfo(final Comparator<T> comparator,
                                              final T obj1, final T obj2)
{
    final String order;
    if (comparator.compare(obj1, obj2) < 0)
        order = " < ";
    else if (comparator.compare(obj1, obj2) > 0)
        order = " > ";
    else
        order = " = ";

    return "'" + obj1 + "'" + order + "'" + obj2 + "'";
}
```

Das war ein eher künstliches Beispiel, das dazu diente, ein erstes Verständnis für die verschiedenen Strengegrade aufzubauen. In der Praxis wird man einen `Collator` dazu nutzen, Attribute einer Klasse zu vergleichen, wie wir dies bereits für die Klasse `Person` getan haben. Kommen wir zur Betrachtung eines Details zur Optimierung noch einmal auf diese zurück.

Optimierung des Vergleichs

Die durch `Collator`-Instanzen realisierten Vergleiche sind langsamer als »normale« Stringvergleiche, die auf dem Interface `Comparable<String>` und der Methode `compareTo(String)` basieren. Für Performance-kritische Applikationen kann man die Klasse `java.text.CollationKey` zur Optimierung der Ausführungsgeschwindigkeit nutzen. Diese Klasse stellt vereinfachend gesagt eine Ordnungszahl dar, die im Voraus berechnet wird. Im nachfolgenden Listing ist der Einsatz eines `CollationKeys` für das Attribut `lastname` einer Klasse `Person` gezeigt. Instanzen der Klasse `CollationKey` erzeugt man nicht per Konstruktor, sondern über die Methode `getCollationKey(String)` der Klasse `Collator`:

```
public class Person implements Comparable<Person>
{
    private static final Collator collator =
                                 Collator.getInstance(Locale.GERMANY);

    private final String lastname;

    // zur Performance-Optimierung
    private final CollationKey key;

    Person(final String lastname)
    {
        this.lastname = lastname;
```

```
        this.key = collator.getCollationKey(lastname);
    }

    @Override
    public int compareTo(final Person otherPerson)
    {
        return this.key.compareTo(otherPerson.key);
    }
}
```

Während der Konstruktion einer Instanz der Klasse `Person` entsteht zwar ein geringer Aufwand für die Erzeugung der `CollationKey`-Instanz, jedoch fällt dieser einmalig an und ist vergleichsweise gering zu dem Aufwand, der durch wiederholte »normale« `Collator`-Vergleiche ohne diese `CollationKey`-Optimierung entsteht.

Sind Vergleiche eher selten, so kann man zusätzlich Lazy Initialization (vgl. Abschnitt 22.3) einsetzen, sodass die Berechnungsaufwände für die `CollationKeys` nur anfallen, wenn tatsächlich Vergleiche erfolgen.

Hinweis: Vergleiche von `CollationKeys`

Instanzen der Klasse `CollationKey` dürfen nur miteinander verglichen werden, wenn sie vom selben `Collator`-Objekt stammen, ansonsten kommt es zu unerwarteten und möglicherweise auch falschen Vergleichsergebnissen.

10.2 Programmbausteine zur Internationalisierung

Nachdem wir im vorangegangenen Abschnitt bereits einige Grundlagen zur Internationalisierung und dazu nützliche Java-Klassen kennengelernt haben, stellt dieser Abschnitt komplexere Programmbausteine vor, die bei der Internationalisierung eigener Anwendungen helfen. Beginnen wir in Abschnitt 10.2.1 mit der Unterstützung mehrerer Datumsformate und betrachten nachfolgend in Abschnitt 10.2.2 die Übersetzbarkeit dargestellter Texte.

10.2.1 Unterstützung mehrerer Datumsformate

Häufig muss man mehrere Datumsformate unterstützen. Dies ist in der Regel bei der Eingabe von Datumswerten in einer Benutzeroberfläche der Fall. Hier ist es für einen Nutzer angenehm, Datums- oder Zeitangaben nicht *exakt* in dem Format eingeben zu müssen, in dem es vom Programm später gespeichert und weiterverarbeitet wird. Machen wir es konkret: Würde beispielsweise ein Programm Datumswerte mit vierstelligen Jahreszahlen und Sekundenangaben im Format '`dd.MM.yyyy HH:mm:ss`' verarbeiten, so ist ein derart starres Format doch unhandlich für einen Anwender. Zum Teil möchte dieser lediglich eine zweistellige Jahreszahl angeben. Auch die Uhrzeit soll häufig nicht sekundengenau spezifiziert werden. In beiden

Fällen können Eingaben verarbeitet werden, wenn das Parsing fehlertolerant arbeitet und für nicht angegebene Werte sinnvolle Standardwerte vorsieht.

Im Folgenden stelle ich vor, wie man durch den Einsatz einer Utility-Klasse fehlertolerant parsen kann. Zum einen befreit dies den Entwickler der Applikation von den Details des Parsings. Zum anderen erlaubt dies dem Benutzer mehr Flexibilität bei der Eingabe von Datumswerten. Die Applikation kann zudem mit der gewünschten und für die Anwendungsfälle geeigneten Datumsrepräsentation (`long`, `Date`, `Calendar` oder einer eigenen Realisierung) arbeiten, ohne dass ein Nutzer dieses Format kennen und bei einer Eingabe beachten muss.

Auswertung von Datumsformaten

Nehmen wir an, wir wollten Datumsangaben mit zwei- oder vierstelligen Jahresangaben sowie optionaler Angabe von Sekunden unterstützen. Eine erste Idee könnte sein, nacheinander ein Parsing mit verschiedenen Datumsformaten durchzuführen. Man definiert dazu verschiedene erlaubte Datumsformate vom Typ `SimpleDateFormat`, etwa:

```
final DateFormat df1 = new SimpleDateFormat("dd.MM.yy HH:mm:ss");
final DateFormat df2 = new SimpleDateFormat("dd.MM.yy HH:mm");
final DateFormat df3 = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
final DateFormat df4 = new SimpleDateFormat("dd.MM.yyyy HH:mm");
```

Definition einer Utility-Klasse

Zur Umwandlung von textuellen Eingaben in `Date`-Objekte erstellen wir eine Utility-Klasse `DateParseUtils`. Einen ersten Baustein bildet die im Folgenden vorgestellte Methode `parseDate(String, DateFormat…)`. Dieser werden sowohl eine Datumseingabe als auch eine variable Anzahl erlaubter Datumsformate als Parameter übergeben. Um Klienten den Aufruf zu erleichtern, bietet sich hier der Einsatz des Sprachfeatures Varargs an. Somit können ein einzelnes Datumsformat, eine kommaseparierte Aufzählung oder ein Array als Eingabe genutzt werden.

```
public final class DateParseUtils
{
    public static Date parseDate(final String dateAndTime,
                                 final DateFormat… supportedDateFormats)
                      throws ParseException
    {
        for (final DateFormat currentFormat : supportedDateFormats)
        {
            try
            {
                // Rückgabe des Formats bei erfolgreichem Parsing
                return currentFormat.parse(dateAndTime);
            }
            catch (final ParseException e)
            {
                // Ignorieren und mit nächstem Format versuchen
            }
```

```
        }

        // Kein Format erlaubt eine Umwandlung
        throw new ParseException(dateAndTime, 0);
    }

    // …
```

Für jedes `DateFormat`-Objekt wird mit dessen `parse(String)`-Methode versucht, die Eingabe in ein `Date`-Objekt umzuwandeln. Ist dies nicht möglich, so wird eine `ParseException` ausgelöst. Diese wird hier zwar abgefangen, aber nicht weiter behandelt, da die Parsingaufgabe an ein nachfolgendes `DateFormat`-Objekt übertragen wird. Dieser Vorgang wiederholt sich, bis entweder das Parsing erfolgreich war oder aber kein weiteres Datumsformat verbleibt. In diesem Fall wird kein gültiges `Date`-Objekt zurückgeliefert, sondern explizit eine `ParseException` ausgelöst.

Betrachten wir nun eine `main()`-Methode, die die eben definierte Hilfsmethode nutzt, um den Datumswert Silvester 2009 zu parsen:

```
private static final String SILVESTER = "31.12.2009";

public static void main(final String[] args)
{
    final DateFormat[] supportedFormats = { df1, df2, df3, df4 };

    try
    {
        final Date date = DateParseUtils.parseDate(SILVESTER,
supportedFormats);
        System.out.println("Parsed '" + SILVESTER + "' into date " + date);
    }
    catch (final ParseException ex)
    {
        System.out.println("Parsing failed: value='" + SILVESTER +"'");
    }
}
```

Listing 10.13 *Ausführbar als* **'MULTIPLEDATEFORMATPARSINGEXAMPLE1'**

Führt man das Programm MULTIPLEDATEFORMATPARSINGEXAMPLE1 aus, so wird kein Datumswert ausgegeben, sondern die Fehlermeldung »`Parsing failed: value='31.12.2009'`«. Es fehlt jedoch ein Hinweis auf die Fehlerursache.

Aufbereitung von aussagekräftigen Fehlermeldungen

Beim Auftreten von Parsingfehlern helfen informative Fehlermeldungen, mögliche Ursachen besser erkennen zu können. Würde jede Applikation selbst wieder die Aufbereitung von Fehlermeldungen implementieren, so wäre dies aufwendig. Es bietet sich an, die gerade erstellte Utility-Klasse `DateParseUtils` um genau diese Funktionalität zu erweitern.

Die im folgenden Listing gezeigte Methode `buildErrorMessage(String, DateFormat…)` bereitet eine aussagekräftige Fehlermeldung auf, die alle beim Parsing akzeptierten Datumsformate aufzählt:

```
public static String buildErrorMessage(final String dateAndTime,
                                        final DateFormat… supportedDateFormats)
{
    // Aufbereiten der Muster
    final String[] patterns = new String[supportedDateFormats.length];
    for (int i = 0; i < supportedDateFormats.length; i++)
    {
        if (supportedDateFormats[i] instanceof SimpleDateFormat)
        {
            patterns[i] = "'" + ((SimpleDateFormat)
                                 supportedDateFormats[i]).toPattern() + "'";
        }
        else
        {
            patterns[i] = "'" + supportedDateFormats[i] + "'";
        }
    }

    return "Parsing error: value='" + dateAndTime + "'\nSupported formats: " +
           Arrays.toString(patterns);
}
```

Erweitern wir die `main()`-Methode um einen Aufruf dieser Fehlerbehandlung:

```
public static void main(final String[] args)
{
    final DateFormat[] supportedFormats = { df1, df2, df3, df4 };

    try
    {
        final Date date = DateParseUtils.parseDate(SILVESTER,
supportedFormats);
        System.out.println("Parsed '" + SILVESTER + "' into date " + date);
    }
    catch (final ParseException ex)
    {
        System.out.println(DateParseUtils.buildErrorMessage(SILVESTER,
                                                            supportedFormats));
    }
}
```

Listing 10.14 *Ausführbar als* **'MULTIPLEDATEFORMATPARSINGEXAMPLE2'**

Führt man das Programm MULTIPLEDATEFORMATPARSINGEXAMPLE2 aus, so erhält

```
Parsing error: value='31.12.2009'
Supported formats: ['dd.MM.yy HH:mm:ss', 'dd.MM.yy HH:mm',
                    'dd.MM.yyyy HH:mm:ss', 'dd.MM.yyyy HH:mm']
```

Offensichtlich ist diese Reaktion auf einen Fehler beim Parsing sehr hilfreich, um die Fehlerursache zu erkennen: Es wird neben der Angabe eines Datums auch die Angabe einer Uhrzeit erwartet. Letztere erfolgt hier jedoch nicht und löst so den Fehler aus.

Erweiterung: Einlesen von Formatstrings aus einer Property-Datei

Die ersten Bausteine für eine flexible Datumsprüfung sind damit geschrieben. Wir betrachten nun eine sinnvolle Erweiterung. Oftmals wandeln sich Anforderungen im

Laufe der Zeit: Würde z. B. die Unterstützung eines weiteren Datumsformats gewünscht, so wäre es unpraktisch, dafür jeweils neue Konstanten im Sourcecode definieren zu müssen. Das würde außerdem eine Neukompilierung erfordern, um die neue Funktionalität in das Programm zu integrieren. Eine flexible Konfigurierbarkeit erreicht man, wenn eine Darstellung und Speicherung in Form von Schlüssel-Wert-Paaren in einer Datei erfolgt. Eine Menge gültiger Datumsformate könnte man in etwa wie folgt definieren:

```
dateformat1=dd.MM.yy HH:mm:ss
dateformat2=dd.MM.yy HH:mm
dateformat3=dd.MM.yyyy HH:mm:ss
dateformat4=dd.MM.yyyy HH:mm
```

Betrachten wir nun eine Möglichkeit, diese Werte einzulesen. Folgende Methode `readDateFormatsFromFile(String)` nutzt die bereits bekannte JDK-Bibliotheksklasse `Properties` (vgl. Abschnitt 6.5.2), um diese Aufgabe zu realisieren. Aus den eingelesenen Werten wird das Muster des Datumsformats extrahiert und zur Konstruktion eines `SimpleDateFormat`-Objekts verwendet:

```
public static DateFormat[] readDateFormatsFromFile(final String filename)
                           throws IOException, FileNotFoundException
{
    try (final InputStream is = new BufferedInputStream(
                                    new FileInputStream(filename)))
    {
        final Properties dateFormatProperties = new Properties();
        dateFormatProperties.load(is);

        final List<DateFormat> dateFormats = new ArrayList<>();
        for (final Object value : dateFormatProperties.values())
        {
            // Erzeugen von SimpleDateFormat-Objekten aus den Properties
            dateFormats.add(new SimpleDateFormat((String) value));
        }

        return dateFormats.toArray(new DateFormat[dateFormats.size()]);
    }
}
```

Wir erweitern die `main()`-Methode um das Einlesen von Datumsformaten aus einer Datei. Als Reaktion auf die zuvor erhaltene präzise Fehlermeldung geben wir zur Korrektur eine Uhrzeit im Datumsstring an:

```
public static void main(final String[] args)
{
   try
   {
       final DateFormat[] supportedFormats =
             DateParseUtils.readDateFormatsFromFile(DATE_FORMAT_PROPERTY_FILE);

       final String silvester = "31.12.2009 18:00";
       try
       {
           final Date date = DateParseUtils.parseDate(silvester,
                                                      supportedFormats);
           System.out.println("Parsed '" + silvester + "' into Date " + date);
       }
```

```
        catch (final ParseException ex)
        {
            System.out.println(DateParseUtils.
                               buildErrorMessage(silvester, supportedFormats));
        }
    }
    catch (final IOException ex)
    {
       System.out.println("No DateFormat-File: " + DATE_FORMAT_PROPERTY_FILE);
    }
}
```

Listing 10.15 *Ausführbar als* **'MULTIPLEDATAFORMATPARSINGEXAMPLE3'**

Als Ausgabe auf der Konsole erhalten wir nach dieser Korrektur folgende Ausgabe:

```
Parsed '31.12.2009 18:00' into Date Thu Dec 31 18:00:00 CET 2009
```

Fazit

Das war ein kurzer Einstieg in das Thema Auswertung mehrerer Datumsformate. Sie besitzen damit das notwendige Wissen, um eigene Erweiterungen zu realisieren. Folgen Sie dem Rat, nützliche Funktionalität in Utility-Klassen auszulagern. Ihre Applikationen werden dadurch schlanker und eleganter.

10.2.2 Nutzung mehrerer Sprachdateien

Eine internationalisierte Anwendung stellt alle Informationen sprach- und landesspezifisch, entsprechend einer gewählten Sprache und Region, dar. Damit dies möglich wird, sind einige Vorarbeiten in der Applikation selbst durchzuführen. Wünschenswert ist auf jeden Fall eine Trennung von dargestellten Informationen und dem eigentlichen Sourcecode.

Bereits bei der Vorstellung der Klasse `PropertyResourceBundle` in Abschnitt 10.1.3 wurde dargelegt, dass es nicht sinnvoll ist, sämtliche in einer Bedienoberfläche dargestellten Texte als Magic Strings zu verwalten. Empfehlenswert ist vielmehr die Definition von Konstanten, die eindeutige Schlüssel zur Identifikation von Texten und Informationen darstellen. Die eigentlichen Übersetzungen werden in einer externen Datenquelle abgelegt[7] und können über diese Schlüssel abgerufen werden. Dadurch ergeben sich unter anderem folgende Vorteile:

- Entwickler müssen sich nicht um die Übersetzung kümmern. Im Sourcecode werden die Texte über die eindeutigen Schlüssel ausgelesen.
- Zur Übersetzung sind lediglich die Sprachdateien zu ändern. Eine Anwendung kann also ohne Kenntnis der eigentlichen Implementierung übersetzt werden.
- Verfügbare Sprachen können durch einfaches Austauschen der Sprachdateien, in der die Übersetzungstexte definiert sind, geändert und erweitert werden.
- Die in der Anwendung genutzte Sprache kann konfigurierbar gemacht werden.

Ein Wechsel der dargestellten Sprache erfordert keine Anpassung im Sourcecode und auch kein erneutes Kompilieren. Es ist nur eine bestimmte Sprachdatei einzulesen.

Verwaltung der Sprachdateien mit der Klasse `ResourceManager`

Zur Kapselung des Zugriffs auf Sprachdateien und der Ermittlung zu einer Sprache passender Texte bietet sich die Definition einer Klasse `ResourceManager` an. Diese lädt und verwaltet die vorhandenen Sprachressourcen und bietet Zugriff auf die jeweiligen sprachspezifischen Texte. Dadurch wird die Unterstützung verschiedener Übersetzungen deutlich erleichtert.

Dieser `ResourceManager` kann im einfachsten Fall mit verschiedenen Sprachdateien initialisiert werden. Schöner wäre es, automatisch ein spezielles Verzeichnis nach allen vorhandenen Textdateien zu durchsuchen und einzulesen. Beginnen wir jedoch zunächst mit der einfacheren Variante.

Für eine nutzende Applikation sind hauptsächlich drei Methoden von Interesse:

1. **`createInstance()`** – Diese statische Methode erzeugt eine gebrauchsfertige Instanz der Klasse `ResourceManager`. Sowohl das Einlesen als auch das initiale Aktivieren einer Sprache werden in dieser Methode gekapselt. Dies vereinfacht die Handhabung in nutzenden Applikationen.
2. **`getLangString(String)`** – Diese Methode bietet Zugriff auf die eingelesenen Übersetzungen. Es werden immer die Texte der momentan gewählten Locale verwendet. Diese kann über die Methode `getCurrentLocale()` abgefragt und über die Methode `activateLocale(Locale)` verändert werden.
3. **`activateLocale(Locale)`** – Ein Aufruf dieser Methode aktiviert die angegebene Locale und schaltet auf die dort hinterlegte Sprache um.

Schauen wir nun auf die Implementierung:

```
public final class ResourceManager
{
    private static final Logger log = Logger.getLogger(ResourceManager.class);

    // ACHTUNG: Muss im Classpath zugänglich sein!
    private static final String BUNDLE_PATH = "resources.PDFEditor";

    // Verwaltung und Speicherung der ResourceBundles
    private final Map<Locale, ResourceBundle> availableResourceBundles =
                                              new HashMap<>();

    private ResourceBundle currentResourceBundle = null;
    private Locale currentLocale = null;

    private ResourceManager()
    {
    }
```

```
    // Erzeugung einer gebrauchfertigen Instanz
    public static ResourceManager createInstance()
    {
        final ResourceManager resourceManager = new ResourceManager();
        resourceManager.init();
        resourceManager.activateLocale(Locale.GERMANY);

        return resourceManager;
    }

    private void init()
    {
        loadAndAddResourceBundle(Locale.GERMANY);
        loadAndAddResourceBundle(Locale.UK);
        loadAndAddResourceBundle(Locale.FRANCE);
    }

    private void loadAndAddResourceBundle(final Locale locale)
    {
        try
        {
            final ResourceBundle resourceBundle =
                        PropertyResourceBundle.getBundle(BUNDLE_PATH,
locale);
            addResourceBundle(locale, resourceBundle);
        }
        catch (final MissingResourceException ex)
        {
            log.warn("Missing recource '" + BUNDLE_PATH + "' for locale: '" +
                     locale + "'", ex);
        }
    }

    private void addResourceBundle(final Locale locale,
                                   final ResourceBundle resourceBundle)
    {
        availableResourceBundles.put(locale, resourceBundle);
    }

    // Zugriff auf sprachabhängige Texte
    public String getLangString(final ResourceKeys key)
    {
        return ResourceBundleUtils.getLangString(currentResourceBundle, key);
    }

    // Umschaltung der Sprache
    public boolean activateLocale(final Locale locale)
    {
        if (supportsLocale(locale))
        {
            currentResourceBundle = availableResourceBundles.get(locale);
            currentLocale = locale;
            return true;
        }
        return false;
    }

    // Hilfsmethoden mit selbsterklärenden Namen
    public boolean supportsLocale(final Locale locale)
    {
```

```
        return availableResourceBundles.containsKey(locale);
    }

    public Locale getCurrentLocale()
    {
        return currentLocale;
    }

    public List<Locale> getAvailableLocales()
    {
        return new ArrayList<>(availableResourceBundles.keySet());
    }
}
```

Mithilfe der Methode `supportsLocale(Locale)` kann man feststellen, ob das als Parameter übergebene `Locale`-Objekt unterstützt wird. Eine Auswahl aller möglichen `Locale`-Objekte liefert ein Aufruf von `getAvailableLocales()`. Die momentan genutzte Locale wird durch einen Aufruf von `getCurrentLocale()` ermittelt.

Einsatz der Klasse `ResourceManager`

Das folgende Beispiel verwendet die Klasse `ResourceManager`, um ein sprachabhängiges Menü aufzubauen. Über drei Buttons kann die momentan genutzte Sprache umgeschaltet werden. Als Folge wird das Menü komplett neu aufgebaut.

Die Klasse `ResourceManagerExample` definiert eine innere Klasse `AppFrame`, die für die Darstellung des Fensters und der Buttons zuständig ist. Das GUI wird im Konstruktor aufgebaut, inklusive dreier Aktionen zur Sprachumschaltung. Dort wird die entsprechende Locale im `ResourceManager` aktiviert und anschließend das Menü durch Aufruf der Methode `refreshMenu()` sprachabhängig neu aufgebaut. Eine hier nicht gezeigte Methode `createAndAddMenus()` erzeugt die Menüeinträge. Wichtig in diesem Zusammenhang ist, Änderungen im GUI durch Aufrufe an die drei Methoden `invalidate()`, `validate()` und `repaint()` auszuführen, um Darstellungsfehler und Inkonsistenzen zu vermeiden:

```
public final class ResourceManagerExample
{
    private static final ResourceManager resourceManager = ResourceManager.
          createInstance();

    public static void main(final String[] args)
    {
        final JFrame appFrame = new AppFrame("ResourceManagerExample");
        appFrame.setVisible(true);
    }

    public static class AppFrame extends JFrame
    {
        private final JMenuBar menuBar = new JMenuBar();

        AppFrame(final String title)
        {
```

```
        super(title);
        setSize(400, 200);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);

        createAndAddMenus(menuBar);

        getContentPane().setLayout(new BorderLayout());
        getContentPane().add(menuBar, BorderLayout.NORTH);

        final JButton deButton = createButton(Locale.GERMANY);
        final JButton enButton = createButton(Locale.UK);
        final JButton frButton = createButton(Locale.FRANCE);
        final JPanel buttonPanel = new JPanel(new FlowLayout());
        buttonPanel.add(deButton);
        buttonPanel.add(enButton);
        buttonPanel.add(frButton);
        getContentPane().add(buttonPanel, BorderLayout.SOUTH);
    }

    private JButton createButton(final Locale locale)
    {
        final String name = locale.getDisplayLanguage();
        final JButton button = new JButton(new AbstractAction(name)
        {
            @Override
            public void actionPerformed(ActionEvent e)
            {
                resourceManager.activateLocale(locale);
                refreshMenu();
            }
        });
        return button;
    }

    private void refreshMenu()
    {
        menuBar.removeAll();
        createAndAddMenus(menuBar);
        // Diese drei Methoden bei GUI-Änderungen immer zusammen aufrufen
        invalidate();
        validate();
        repaint();
    }
    // …
```

Listing 10.16 *Ausführbar als* **'RESOURCEMANAGEREXAMPLE'**

Startet man das Programm RESOURCEMANAGEREXAMPLE, zeigt sich in etwa das in Abbildung 10-3 dargestellte Bild.

Abbildung 10-3 *Sprachauswahl im Programm* RESOURCEMANAGEREXAMPLE

Initialisierung vorhandener Sprachdateien

Bis hierher haben wir die Grundfunktionalität kennengelernt. Wünschenswert ist es, ein spezielles Verzeichnis nach Sprachdateien zu durchsuchen, diese einzulesen und dem Programm zur Verfügung zu stellen. Wir erweitern die Utility-Klasse `ResourceBundleUtils` um einen speziellen `FileFilter` wie folgt:

```
public final class ResourceBundleUtils
{
    public static class ResourceBundleFileFilter implements FileFilter
    {
        private final String bundlename;

        public ResourceBundleFileFilter(final String bundlename)
        {
            this.bundlename = bundlename;
        }

        @Override
        public boolean accept(final File pathname)
        {
            return pathname.getName().startsWith(bundlename) &&
                   pathname.getName().toLowerCase().endsWith(".properties");
        }
    };
    // …
```

Durch den Einsatz dieser `FileFilter`-Implementierung kann eine Ergebnismenge auf alle in einem Verzeichnis vorhandenen Property-Dateien eingeschränkt werden. Mit diesem Wissen wird die Klasse `ResourceManager` erweitert, sodass sie automatisch ein Verzeichnis mit hinterlegten Sprachdateien einliest. Dazu ändern wir lediglich die Implementierung der Methode `init()`:

```
private void init()
{
    final FileFilter fileFilter = new ResourceBundleUtils.
                                      ResourceBundleFileFilter("PDFEditor");
    final File[] propertyFiles = FileUtils.getAllMatchingFiles(bundleDir,
                                                               fileFilter);

    for (final File propertyFile : propertyFiles)
    {
        // JDK 7: ARM
        try (final InputStream is = new BufferedInputStream(
                                        new FileInputStream(propertyFile)))
        {
            final PropertyResourceBundle resourceBundle =
                                         new PropertyResourceBundle(is);

            final Locale locale =
ResourceBundleUtils.createLocaleFromBundleName
                  (propertyFile.getName());
            addResourceBundle(locale, resourceBundle);
        }
        catch (final IOException ex)
```

```
            {
                log.warn("Failed to load resource from file '" +
                         propertyFile.getAbsolutePath() + "'", ex);
            }
        }
    }
```

Das durch die Methode `getAllMatchingFiles(String, FileFilter)` zurückgelieferte `File[]` wird durchlaufen. Die enthaltenen Property-Dateien werden durch einen Konstruktoraufruf `PropertyResourceBundle(InputStream)` eingelesen. Dies kann nicht über die bereits bekannte Methode `getBundle(Locale)` der Klasse `PropertyResourceBundle` geschehen, da wir die `Locale`-Objekte nicht im Voraus kennen. Zur Speicherung benötigen wir jedoch ein `Locate`-Objekt. Wir wenden folgenden Trick an: Anhand des Dateinamens kann man die Sprach- und Länderkürzel auslesen. Voraussetzung ist, dass dieser gemäß der in Abschnitt 10.1.3 genannten Konvention im Format 'Name_<Sprachkürzel>_<Länderkürzel>.properties' vorliegt.

Basierend auf diesen Informationen kann durch einen Konstruktoraufruf ein `Locale`-Objekt erzeugt werden. Diese Funktionalität wird durch eine Methode `createLocaleFromBundleName(String)` gekapselt, die wiederum in der Utility-Klasse `ResourceBundleUtils` definiert wird. Wir erkennen wieder einmal, wie die entworfenen Programmbausteine ineinandergreifen und uns dadurch Arbeit abnehmen.

Unterstützung und Umschaltung aller vorhandenen Sprachdateien

Die erste Beispielapplikation hat das Umschalten der dargestellten Texte über Buttons erlaubt. Dabei wurden die möglichen Auswahlvarianten zum Kompilierzeitpunkt festgelegt. Wir erweitern die Applikation so, dass alle verfügbaren Sprachdateien beim Programmstart ermittelt und anschließend in einem Menü zur Auswahl bereitgestellt werden.

Im Menü soll eine menschenlesbare Darstellung der Sprachauswahl erfolgen. Dabei sollen die Texte der Sprachauswahl von der derzeit gewählten Spracheinstellung abhängig sein. Für eine derartige Funktionalität haben wir bereits die Klasse `Locale` und die Methoden `getDisplayLanguage(Locale)` und `getDisplayLocale(Locale)` kennengelernt. Wir nutzen dieses Wissen und definieren die folgende Klasse `MenuAction`. Diese Klasse soll darzustellende Texte entweder anhand eines `ResourceKeys`-Werts ermitteln oder aber anhand eines `Locale`-Objekts. Sie erlaubt sowohl eine sprachabhängige Darstellung normaler Menüeinträge durch Aufruf der bekannten Methode `getLangString(ResourceKeys)` als auch eine Locale-abhängige Darstellung einer Sprachauswahl:

```
public final class MenuAction extends AbstractAction
```

```
{
    // Menüeintrag basierend auf ResourceKeys-Konstante erzeugen
    public MenuAction(final ResourceKeys resourceKey)
    {
        putValue(Action.NAME, resourceManager.getLangString(resourceKey));
        putValue("OBJECT", resourceKey);
    }

    // Sprachmenüs basierend auf Locale erzeugen
    public MenuAction(final Locale menuLocale)
    {
        final Locale currentLocale = resourceManager.getCurrentLocale();
        final String language = menuLocale.getDisplayLanguage(currentLocale);
        final String country = menuLocale.getDisplayCountry(currentLocale);
        // Kein Querstrich, wenn keine Landesangabe erfolgt
        final String optionalCountryInfo = (country.isEmpty() ? "" :
                                            " / " + country);

        putValue(Action.NAME, language + optionalCountryInfo);
        putValue("OBJECT", menuLocale);
    }

    @Override
    public void actionPerformed(final ActionEvent e)
    {
        handleMenuAction(getValue("OBJECT"));
    }
}
```

Die bereits aus dem vorherigen Beispiel bekannte Klasse `AppFrame` wird lediglich in der Methode `createAndAddMenus(JComponent)` angepasst. Dort werden die einzelnen Menüeinträge mithilfe der Klasse `MenuAction` sprach- bzw. Locale-abhängig initialisiert. Zudem wird eine Methode `handleMenuAction(Object)` definiert, die die gewählten Menüeinträge auswertet und abhängig davon Aktionen auslöst. Für dieses Beispiel ist dies der Einfachheit halber die Darstellung einer Meldungsbox über `JOptionPane.showMessageDialog()` bzw. das Verlassen des Programms mit `System.exit(0)`. Für die Sprachmenüs wird das zugehörige `Locale`-Objekt bestimmt und eine Umschaltung in der Klasse `ResourceManager` vorgenommen. Damit die Änderungen im GUI sichtbar werden, wird die bekannte Methode `refreshMenu()` aufgerufen:

```
private void createAndAddMenus(final JComponent comp)
{
    // Menüeinträge sprachabhängig erzeugen
    final JMenu fileMenu = new JMenu(new MenuAction(ResourceKeys.txt_file));
    fileMenu.add(new MenuAction(ResourceKeys.txt_new));
    fileMenu.add(new MenuAction(ResourceKeys.txt_open));
    fileMenu.add(new JSeparator());

    final JMenu propertiesMenu = new JMenu(new MenuAction(ResourceKeys.
          txt_properties));
    fileMenu.add(propertiesMenu);

    // Vorhandene Locales ermitteln, dann sortieren
    final List<Locale> availableLocales =
resourceManager.getAvailableLocales();
    Collections.sort(availableLocales, LocaleUtils.LOCALE_COMPARATOR);
```

```
    // Entsprechende Menüeinträge bereitstellen
    for (final Locale currentLocale : availableLocales)
        propertiesMenu.add(new MenuAction(currentLocale));

    fileMenu.add(new JSeparator());
    fileMenu.add(new MenuAction(ResourceKeys.txt_quit));

    comp.add(fileMenu);
}

public void handleMenuAction(final Object object)
{
    // Menüverarbeitung
    if (object instanceof ResourceKeys)
    {
        final ResourceKeys resourceKey = (ResourceKeys) object;
        switch (resourceKey)
        {
            case txt_new:
            case txt_open:
            case txt_save:
                JOptionPane.showMessageDialog(null, "Not implemented yet!");
                break;
            case txt_quit:
                System.exit(0);
        }
    }

    // Sprachmenüs bearbeiten
    if (object instanceof Locale)
    {
        final Locale newLocale = (Locale) object;
        resourceManager.activateLocale(newLocale);
        refreshMenu();
    }
}
```

Listing 10.17 *Ausführbar als* **'RESOURCEMANAGERV2EXAMPLE'**

Startet man das angegebene Programm RESOURCEMANAGERV2EXAMPLE, zeigt sich in etwa das in Abbildung 10-4 dargestellte Bild.

Abbildung 10-4 *Sprachauswahl im Programm* RESOURCEMANAGERV2EXAMPLE

Erweiterung: Parametrierte Sprachbausteine

Ein häufiger Anwendungsfall ist es, gewisse Werte in die auszugebenden Texte zu

integrieren. Bei Warnmeldungen könnte dies etwa der Pfad einer nicht gefundenen Datei sein. Wir nutzen die in Abschnitt 10.1.7 eingeführte Klasse `MessageFormat`, um die Methode `getLangString(ResourceKeys, Object…)` folgendermaßen zu überladen. Dadurch wird die geschilderte Funktionalität realisiert:

```
public String getLangString(final ResourceKeys key, final Object… params)
{
    String str = getLangString(key);
    if (str != null)
    {
        // Parameter im String ersetzen
        try
        {
            str = MessageFormat.format(str, params);
        }
        catch (final IllegalArgumentException e)
        {
            // Möglich, wenn Klammern nicht geschlossen sind, hier ignorieren
        }
    }

    return str;
}
```

Fazit und Ausblick

In diesem Abschnitt wurden verschiedene nützliche Bausteine zur Internationalisierung vorgestellt. Dabei wurde zunächst Basisfunktionalität entworfen und diese mit bereits bekannten Programmbausteinen kombiniert. Dadurch konnte der Aufwand zur Internationalisierung in der Applikation selbst gering gehalten werden.

Es gibt diverse weitere Einflussfaktoren auf die Internationalisierung, die hier nicht weiter betrachtet wurden, da diese nicht im Fokus dieses Buchs liegen. Dazu gehören unter anderem Themen wie der Einfluss von Zeitzonen und die Schreibrichtung von Text.

Teil III

Die Neuerungen in Java 8

11 Lambda-Ausdrücke

Mit Lambda-Ausdrücken (kurz: ***Lambdas***, zum Teil auch ***Closures*** genannt) wurde ein neues und von vielen Entwicklern heiß ersehntes Sprachkonstrukt in Java eingeführt, das bereits in ähnlicher Form in verschiedenen anderen Programmiersprachen wie C#, Groovy und Scala erfolgreich genutzt wird. Der Einsatz von Lambdas erfordert zum Teil eine andere Denkweise und führt zu einem neuen Programmierstil, der dem Paradigma der ***funktionalen Programmierung*** folgt. Mithilfe von Lambdas lassen sich einige Lösungen auf sehr elegante Art und Weise formulieren. Insbesondere im Bereich von Frameworks und zur Parallelverarbeitung kann die Verwendung von Lambdas enorme Vorteile bringen. Diverse Funktionalitäten im Collections-Framework und an anderen Stellen des JDKs wurden auf Lambdas umgestellt. Bevor wir darauf zurückkommen, schauen wir uns zunächst einmal Lambdas an sich an.

Beispiel: Sortierung nach Länge und kommaseparierte Aufbereitung

Um die Vorteile von Lambdas und auch später von den sogenannten Bulk Operations on Collections besser nachvollziehen zu können, betrachten wir als praxisnahes Beispiel eine Liste von Namen. Diese Namen wollen wir nach deren Länge sortieren und die Längen danach kommasepariert ausgeben. Dazu würden wir bis einschließlich JDK 7 in etwa folgenden Sourcecode schreiben:

```
// Sortierung mit Comparator
final List<String> names = Arrays.asList("Andy", "Michael", "Max", "Stefan");
Collections.sort(names, new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
});

// Iteration und Ausgabe
final Iterator<String> it = names.iterator();
while (it.hasNext())
{
    System.out.print(it.next().length() + ", "); // 3, 4, 6, 7,
}
```

Beim Betrachten dieser Umsetzung kann man sich fragen, ob das nicht kürzer und einfacher gehen sollte? Die Antwort ist: Ja, mit JDK 8 kann man dazu Lambdas nutzen.

11.1 Einstieg in Lambdas

Das Sprachkonstrukt Lambda kommt aus der funktionalen Programmierung. Ein ***Lambda*** ist ein Behälter für Sourcecode ähnlich einer Methode, allerdings ohne Namen und ohne die explizite Angabe eines Rückgabetyps oder ausgelöster Exceptions. Vereinfacht ausgedrückt kann man einen Lambda am ehesten als anonyme Methode mit folgender Syntax und spezieller Kurzschreibweise auffassen:

```
(Parameter-Liste) -> { Ausdruck oder Anweisungen }
```

11.1.1 Lambdas am Beispiel

Ein paar recht einfache Beispiele für Lambdas sind die Addition von zwei Zahlen vom Typ `int`, die Multiplikation eines `long`-Werts mit dem Faktor 2 oder eine parameterlose Funktion zur Ausgabe eines Textes auf der Konsole. Diese Aktionen kann man als Lambdas wie folgt schreiben:

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
() -> { String msg = "Lambda"; System.out.println("Hello " + msg); }
```

Das sieht recht unspektakulär aus, und insbesondere wird klar, dass ein Lambda lediglich ein Stück ausführbarer Sourcecode ist, der

- keinen Namen besitzt, sondern lediglich Funktionalität, und dabei
- keine explizite Angabe eines Rückgabetyps und
- keine Deklaration von Exceptions erfordert und erlaubt.

Lambdas im Java-Typsystem

Wir haben bisher gesehen, dass sich einfache Berechnungen mithilfe von Lambdas ausdrücken lassen. Wie können wir diese aber nutzen und aufrufen? Versuchen wir zunächst, einen Lambda einer `java.lang.Object`-Referenz zuzuweisen, so wie wir es mit jedem anderen Objekt in Java auch tun können:

```
// Compile-Error: incompatible types: Object is not a functional interface
Object greeter = () -> { System.out.println("Hello Lambda"); };
```

Die gezeigte Zuweisung ist nicht erlaubt und führt zu einem Kompilierfehler. Die Fehlermeldung gibt einen Hinweis auf inkompatible Typen und verweist darauf, dass `Object` kein Functional Interface ist. Aber was ist denn ein Functional Interface?

Besonderheit: Lambdas im Java-Typsystem

Bis JDK 8 konnte in Java jede Referenz auf den Basistyp `Object` abgebildet werden. Mit Lambdas existiert nun ein Sprachelement, das nicht direkt dem Basistyp `Object` zugewiesen werden kann, sondern nur an Functional Interfaces.

11.1.2 Functional Interfaces und SAM-Typen

Ein ***Functional Interface*** ist eine neue Art von Typ, die mit JDK 8 eingeführt wurde, und repräsentiert ein Interface mit genau einer abstrakten Methode. Ein solches wird auch ***SAM-Typ*** genannt, wobei SAM für Single Abstract Method steht. Diese Art von Interfaces gibt es nicht erst seit Java 8 im JDK, sondern schon seit Langem und vielfach – wobei es früher für sie aber keine Bezeichnung gab. Vertreter der SAM-Typen und Functional Interfaces sind etwa `Runnable`, `Callable<V>`, `Comparator<T>`, `FileFilter`, `FilenameFilter`, `ActionListener`, `EventHandler` usw.

```
@FunctionalInterface
public interface Runnable
{
    public abstract void run();
}
```

```
@FunctionalInterface
public interface Comparator<T>
{
    int compare( T o1, T o2);
    boolean equals( Object obj);
}
```

Im Listing sehen wir die mit JDK 8 eingeführte Annotation `@FunctionalInterface` aus dem Package `java.lang`. Damit wird ein Interface explizit als Functional Interface gekennzeichnet. Die Angabe der Annotation ist optional: Jedes Interface mit genau nur einer abstrakten Methode (SAM-Typ) stellt auch ohne explizite Kennzeichnung ein Functional Interface dar. Sofern die Annotation angegeben wird, kann der Compiler eine Fehlermeldung produzieren, falls es mehrere abstrakte Methoden gibt.

Tipp: Besondere Methoden in Functional Interfaces

Wenn wir im obigen Listing genauer hinsehen, könnten wir uns fragen, wieso denn `java.util.Comparator<T>` ein Functional Interface ist, wo es doch zwei Methoden enthält und keine davon abstrakt ist, oder? Als Besonderheit gilt in Functional Interfaces folgende Ausnahme für die Definition von abstrakten Methoden: Alle im Typ `Object` definierten Methoden können zusätzlich zu der abstrakten Methode in einem Functional Interface angegeben werden.

Verbleibt noch die Frage, warum wir in der Definition des Interface `Comparator<T>` keine abstrakte Methode sehen. Mit ein wenig Java-Basiswissen oder nach einem Blick in die Java Language Specification (JLS) erinnern wir uns daran, dass alle Methoden in Interfaces automatisch `public` und `abstract` sind, auch wenn dies nicht explizit über Schlüsselwörter angegeben ist.

Basierend auf den Argumentationen ist die Methode `compare(T, T)` abstrakt und die Methode `equals(Object)` entstammt dem Basistyp `Object`. Sie darf damit zusätzlich im Interface zur abstrakten Methode aufgeführt werden.

Implementierung von Functional Interfaces

Herkömmlicherweise wird ein SAM-Typ bzw. Functional Interface durch eine anonyme innere Klasse implementiert. Seit JDK 8 kann man alternativ zu dessen Implementierung auch Lambdas nutzen. Voraussetzung dafür ist, dass das Lambda

die abstrakte Methode des Functional Interface erfüllen kann, d. h., dass die Anzahl der Parameter übereinstimmt sowie deren Typen und der Rückgabetyp kompatibel sind. Schauen wir zur Verdeutlichung zunächst auf ein allgemeines, etwas abstraktes Modell zur Transformation von bisherigen Realisierungen eines SAM-Typs mithilfe einer anonymen inneren Klasse in einen Lambda-Ausdruck:

```
// SAM-Typ als anonyme innere Klasse
new SAMTypeAnonymousClass()
{
    public void samTypeMethod(METHOD-PARAMETERS)
    {
        METHOD-BODY
    }
}

// SAM-Typ als Lambda
(METHOD-PARAMETERS) -> { METHOD-BODY }
```

Bei kurzen Methodenimplementierungen, wie sie für SAM-Typen häufig vorkommen, ist das Verhältnis von Nutzcode zu Boilerplate-Code (oder auch Noise genannt) bislang recht schlecht. Wenn man für derartige Realisierungen Lambdas einsetzt, so kann man mit einer Zeile das ausdrücken, was sonst fünf Zeilen oder mehr benötigt. Nachfolgend wird dies für die Interfaces `Runnable` und `Comparator<T>` verdeutlicht.

Beispiel 1: `Runnable`

Konkretisieren wir die allgemeine Transformation anhand eines `java.lang.Runnable`, das eine triviale Konsolenausgabe implementiert:

```
Runnable runnableAsNormalMethod = new Runnable()
{
    @Override
    public void run()
    {
        System.out.println("runnable as normal method");
    }
}
```

In diesem `Runnable` wird keine wirklich sinnvolle Funktionalität realisiert. Vielmehr dient dies nur der Verdeutlichung der Kurzschreibweise mit einem Lambda wie folgt:

```
Runnable runnableAsLambda = () -> System.out.println("runnable as lambda");
```

Beispiel 2: `Comparator<T>`

Die Vorteile von Lambdas lassen sich für das Functional Interface `Comparator<T>` prägnanter zeigen. Ich möchte kurz in Erinnerung rufen, dass mit einem Komparator ein Vergleich von zwei Instanzen vom Typ `T` realisiert wird. Dazu muss die abstrakte Methode `int compare(T, T)` passend realisiert werden und über ihren Rückgabewert die Reihenfolge der Werte ausdrücken (vgl. folgenden Praxishinweis). Wollte man zwei Strings nach deren Länge sortieren, so entsteht herkömmlicherweise recht viel Sourcecode:

```
// Hinweis: Diamond Operator ist nicht für anonyme innere Klassen möglich
Comparator<String> compareByLength = new Comparator<String>()
{
    @Override
    public int compare(final String str1, final String str2)
    {
        final int length1 = str1.length();
        final int length2 = str2.length();

        if (length1 < length2)
            return -1;
        if (length1 > length2)
            return 1;

        return 0;
    }
};
```

Mit JDK 7 wurde die Klasse `Integer` um eine Methode `compare(int, int)` erweitert, die einen komparatorkonformen Rückgabewert liefert und so die Implementierung deutlich vereinfacht und verkürzt:

```
Comparator<String> compareByLength =  new Comparator<String>()
{
    @Override
    public int  compare(final  String str1,  final  String str2)
    {
        return Integer.compare(str1.length(), str2.length());
    }
};
```

Wenn man Lambdas nutzt, lässt sich der Komparator knackig wie folgt schreiben:

```
Comparator<String> compareByLength =  (final  String  str1,  final  String
str2) ->
{
    return  Integer.compare(str1.length(), str2.length());
};
```

Hinweis: Realisierung von Komparatoren und Rückgabewerte

Der Rückgabewert der `compare(T, T)`-Methode bestimmt den Ausgang des Vergleichs der beiden Objekte: Ein Wert > 0 besagt, dass das erste Objekt größer ist, 0 beschreibt Gleichheit und < 0 signalisiert, dass das erste Objekt kleiner als das zweite ist. Einen `Comparator<T>` zu implementieren, ist nicht besonders schwierig, erfordert aber häufig einige Fallunterscheidungen und wird dadurch recht schnell unübersichtlich. Im obigen Beispiel sehen wir einige `if`-Anweisungen. Stattdessen scheint es einfacher, die beiden Werte voneinander zu subtrahieren. Zum Teil sieht man solche Lösungen. Dabei besteht jedoch die Gefahr von einem Überlauf des Wertebereichs des `int` und von Berechnungsfehlern. Außerdem spiegelt diese Art der Implementierung die Intention nicht gut wider und ist eher schlecht verständlich.

11.1.3 Type Inference und Kurzformen der Syntax

Die Syntax von Lambdas besitzt einige Besonderheiten, um den Sourcecode recht knapp formulieren zu können. Dabei nutzt man für Lambdas vor allem auch die sogenannte ***Type Inference***: Ähnlich wie beim Diamond Operator bei der Definition generischer Klassen ist es für Lambdas möglich, auf die Angabe der Typen für die Parameter im Sourcecode zu verzichten. Dazu ermittelt der Compiler die passenden Typen aus dem Einsatzkontext. Den vorherigen Komparator schreibt man ohne Typangabe wie folgt:

```
Comparator<String> compareByLength = (str1, str2) ->
{
    return Integer.compare(str1.length(), str2.length());
};
```

Eine weitere Verkürzung in der Schreibweise eines Lambdas kann man durch folgende Regeln erzielen: Falls das auszuführende Stück Sourcecode ein Ausdruck ist, können die geschweiften Klammern um die Anweisungen entfallen. Ebenfalls kann dann das Schlüsselwort `return` weggelassen werden und der Rückgabewert entspricht dem Ergebnis des Ausdrucks. Außerdem gilt: Existiert lediglich ein Eingabeparameter, so sind die runden Klammern um den Parameter optional. Damit ergibt sich für die Ausdrücke

```
(int x, int y) -> { return x + y; }
(long x) -> { return x * 2; }
```

folgende Kurzschreibweise:

```
(x, y) -> x + y
x -> x * 2
```

Neben dem offensichtlichen Vorteil einer recht kompakten Schreibweise, ist etwas anderes viel entscheidender: Lambdas können flexibler als streng typisierte Methoden genutzt werden. Für die gezeigten Berechnungen ist ein Einsatz überall dort möglich, wo für die Parameter die Operatoren + bzw. * definiert sind, also für die Typen `int`, `float`, `double` usw. Anders formuliert: ***Alles, was hergeleitet werden kann (und soll), darf in der Syntax weggelassen werden***. Als Beispiel betrachten wir folgende `ActionListener`-Implementierung, die schrittweise vereinfacht wird:

```
// Alter Stil
button.addActionListener(new ActionListener()
{
    @Override
    public void actionPerformed(final ActionEvent e)
    {
        System.out.println("button clicked (old way)");
    }
});
```

Diese herkömmliche Realisierung mithilfe einer anonymen inneren Klasse lässt sich als Lambda und mit Type Inference deutlich kürzer schreiben:

```
// Lambda-Variante mit Type Inference
button.addActionListener( (e) -> { System.out.println("button clicked!"); } );
```

Nutzt man zusätzlich die Regeln zur Schreibweisenabkürzung, so entsteht Folgendes:

```
// Lambda-Kurzschreibweise
button.addActionListener( e -> System.out.println("button clicked!") );
```

11.1.4 Lambdas als Parameter und als Rückgabewerte

Wir haben mittlerweile ein wenig Gespür für Lambdas gewonnen und wissen, dass man Lambdas anstelle einer anonymen inneren Klasse zur Realisierung eines SAM-Typs nutzen kann. Ebenso kann man Lambdas auch als Methodenparameter und als Rückgabe einer Methode verwenden, um Aufrufe lesbar zu gestalten.

Wir greifen das Beispiel aus der Einleitung wieder auf und betrachten das Sortieren einer Liste von Namen. Das können wir mit folgenden zwei Varianten eines Lambdas für das Interface `Comparator<T>` schreiben:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Max",
"Stefan");

   // Lambda als Methodenparameter
   Collections.sort(names, (str1, str2) -> Integer.compare(str1.length(),
                                                           str2.length()));

   // Alternative mit Lambda als Rückgabe einer Methode
   names.sort(compareByLength());
}

public static Comparator<String> compareByLength()
{
    return (str1, str2) -> Integer.compare(str1.length(), str2.length());
}
```

Wenn Sie im Listing genau hingeschaut haben, dann könnte Ihnen aufgefallen sein, dass bei der zweiten Variante gar nicht `Collections.sort()`, sondern `names.sort()` aufgerufen wird – also direkt auf einer Instanz von `List<String>`. Wie geht das denn? Diese Methode existiert doch gar nicht im Interface `java.util.List<T>`, oder etwa doch? Bis JDK 7 ist sie dort nicht vorhanden. Jedoch wurden mit JDK 8 das Interface `List<T>` und viele andere Interfaces erweitert. Darauf gehe ich gleich im Anschluss nach der Betrachtung einiger Details zu Lambdas und deren Verwendung ein.

11.1.5 Unterschiede: Lambdas vs. anonyme innere Klassen

Wir wissen zwar schon das eine oder andere über Lambdas, aber es gibt noch kleine Unterschiede zu anonymen inneren Klassen kennenzulernen: Nämlich bei der Bedeutung von `this` sowie dem Zugriff auf Variablen und der Erweiterbarkeit um

Methoden.

Bedeutung von `this`

Lambdas repräsentieren lediglich ein Stück Funktionalität und haben keine Bindung zu einem Objekt. Wenn man dort `this` referenziert, dann liegt der Bezugspunkt also außerhalb des Lambdas und demnach besitzt `this` innerhalb eines Lambdas die gleiche Bedeutung wie in den Zeilen direkt außerhalb davon. Für innere Klassen referenziert `this` dagegen die innere Klasse selbst. Das hat insbesondere Einfluss darauf, wie man auf Attribute der äußeren Klasse zugreifen kann.

Zugriff auf Variablen

Kommen wir zum Zugriff auf Variablen, die außerhalb des Lambdas bzw. der anonymen inneren Klasse definiert sind. Bis JDK 7 konnte man auf derartige Variablen nur dann zugreifen, wenn diese explizit `final` definiert waren. Mit JDK 8 wird das Ganze etwas gelockert: Es reicht nun sowohl für Lambdas als auch für anonyme innere Klassen, wenn die Variablen »effectively« `final` sind. Darunter versteht man, dass die Variablen nicht mehr explizit `final` deklariert werden müssen, sondern es genügt, wenn diese ihren Wert zur Programmlaufzeit nicht ändern. Dieser Sachverhalt wird vom Compiler geprüft und Verstöße werden als Fehler angemahnt.

Erweiterungen eines SAM-Typs

Innerhalb von anonymen inneren Klassen kann man beliebige weitere Methoden definieren. Für Lambdas ist das nicht möglich. Sie können zwar stellvertretend für SAMTypen genutzt werden, erlauben aber keine Erweiterung um Methoden, da sie eher anonymen Methoden als anonymen Klassen entsprechen.

Beispiel

Um die obigen Ausführungen zu verdeutlichen, schauen wir auf folgendes Beispiel:

```
public class LambdaVsInnerClassExample
{
   private String outerAttribute = "fromOutside";

   public static void main( final String[] args)
   {
      new LambdaVsInnerClassExample().executeMethodAndLambda();
   }

   private void executeMethodAndLambda()
   {
      // Nicht finale Variable war bis JDK 7 nicht referenzierbar
      /* final */ int effectivelyFinal = 4711;
```

```
        final Runnable asNormalMethod = new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println(this);

                // Nicht finale Variable war bis JDK 7 nicht referenzierbar
                System.out.println("effectivelyFinal = " + effectivelyFinal);
                // Spezielle Syntax zum Zugriff auf Attribute der äußeren Klasse
                System.out.println("outerAttribute = " +
                                   LambdaVsInnerClassExample.this.outerAttribute);
            }

            // In inneren Klassen kann man weitere Methoden definieren
            public String anotherMethod()
            {
                return "Anonymous Runnable";
            }
        };

        // Man kann keine weiteren Methoden in Lambdas definieren
        final Runnable asLambda = () ->
        {
            System.out.println(this);
            // Nicht finale Variable war bis JDK 7 nicht referenzierbar
            System.out.println("effectivelyFinal = " + effectivelyFinal);
            System.out.println("outerAttribute = " + outerAttribute);
        };

        asNormalMethod.run();
        asLambda.run();
    }
}
```

Das Programm LAMBDAVSINNERCLASSEXAMPLE produziert folgende Ausgaben:

```
// $1 => Innere Klasse
chapter2_lambdas.lambdas.LambdaVsInnerClassExample$1@2ff4acd0
effectivelyFinal = 4711
outerAttribute = fromOutside
chapter2_lambdas.lambdas.LambdaVsInnerClassExample@54bedef2
effectivelyFinal = 4711
outerAttribute = fromOutside
```

Zwar sieht man bei der Ausgabe nur einen kleine Abweichungen voneinander, jedoch zeigt der Sourcecode durchaus Unterschiede und Besonderheiten, die oben fett hervorgehoben sind.

Kommen wir kurz auf die Bedeutung von `this` zurück. Diese hat insbesondere Einfluss darauf, wie man auf Attribute der äußeren Klasse zugreifen kann. Im Beispiel muss man für die innere Klasse etwas umständlich `LambdaVsInnerClassExample.this.outerAttribute` schreiben. Für den Lambda nutzt man nur den Variablennamen, also hier `this.outerAttribute` oder kürzer einfach `outerAttribute`.

11.2 Defaultmethoden

Beim Entwurf von Lambdas und deren Integration in das JDK stellte sich heraus, dass für eine sinnvolle Nutzbarkeit auch die bestehenden Klassen und Interfaces erweitert werden mussten. Bis zur Einführung von JDK 8 war es allerdings nicht möglich, ein Interface nach seiner Veröffentlichung zu verändern, ohne dass dies Auswirkungen bei allen einsetzenden Klassen gehabt hätte. Vielmehr führte die Erweiterung eines Interface bis inklusive JDK 7 immer zu einem Kompatibilitätsproblem: Wenn eine Methode neu in ein Interface hinzugefügt wurde, musste diese in allen Klassen realisiert werden, die das Interface implementieren. Ansonsten kompilierten einige Klassen so lange nicht mehr, bis die Implementierung der neuen Methode im Nachhinein bereitgestellt wurde.

11.2.1 Interface-Erweiterungen

Um dieses Dilemma und insbesondere Inkompatibilitäten bei Interface-Erweiterungen zu vermeiden, ist es nun mit Java 8 möglich, im Sourcecode eines Interface eine sogenannte Defaultimplementierung vorzugeben. Dazu nutzt man das neue Sprachfeature der ***Defaultmethoden***. Das sind ***spezielle Implementierungen von Methoden, die in Interfaces definiert werden können***. Um sie von den normalen abstrakten Methoden in Interfaces zu unterscheiden, werden Defaultmethoden mit dem Schlüsselwort `default` eingeleitet. Die Defaultmethoden sind eine wichtige Neuerung, um Lambdas für bestehende Funktionalitäten des JDKs gewinnbringend nutzen zu können.

Die Defaultmethoden `sort()` und `forEach()`

Lassen Sie uns zwei Erweiterungen näher betrachten. Die erste ist der zuvor schon genutzte Aufruf von `sort()` direkt auf der Instanz einer `List<E>`. Als Zweites schauen wir auf eine Funktionalität, die eine Iteration über alle Elemente einer Collection und ein Bearbeiten jedes einzelnen Elements ermöglicht.

Beginnen wir mit der Erweiterung im Interface `List<E>`. Dort findet sich nun die Definition von `sort(Comparator<? super E>)` wie folgt (gekürzt):

```
public interface List<E> extends Collection<E>
{
    default void sort(Comparator<? super E> c)
    {
        Collections.sort(this, c);
    }
}
```

Das Sortieren ist zwar praktisch, aber eine eher spezielle Funktionalität. Gebräuchlicher und allgemeiner sind Iterationen über die Elemente einer Collection. Dazu steht nun die Defaultmethode `forEach(Consumer<? super T>)` im

Interface `java.lang.Iterable<T>` bereit, das die Basis von `java.util.Collection<E>` und `List<E>` bildet. Die Defaultmethode ist folgendermaßen implementiert (gekürzt):

```
public interface Iterable<T>
{
    default void forEach(Consumer<? super T> action)
    {
        Objects.requireNonNull(action);
        for (T t : this)
        {
            action.accept(t);
        }
    }
}
```

Werfen wir einen kurzen Blick auf das in der Signatur genutzte Functional Interface `java.util.function.Consumer<T>`. Dort ist die abstrakte Methode `accept(T)` deklariert, deren Implementierung die für ein Element auszuführende Funktionalität festlegt. Darüber hinaus ermöglicht die Defaultmethode `andThen(Consumer<? super T>)` die Hintereinanderausführung mehrerer `Consumer<T>`-Instanzen:

```
@FunctionalInterface
public interface Consumer<T>
{
    void accept(T t);

    default Consumer<T> andThen(Consumer<? super T> after)
    {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Neben dem im Listing gezeigten Functional Interface `Consumer<T>` wurde eine Vielzahl solcher Interfaces in JDK 8 integriert.

Hintergrundwissen: Functional Interfaces

Zum Einsatz von Lambdas sowie zur Ergänzung und Flexibilisierung wurde das JDK um diverse Functional Interfaces (also SAM-Typen) erweitert. Im Package `java.util.function` finden sich um die 40 Functional Interfaces, oftmals mit sprechendem Namen, unter anderem Folgende:

- `Consumer<T>` – Beschreibt eine Aktion auf einem Element vom Typ `T`. Dazu ist eine Methode `void accept(T)` definiert.
- `Predicate<T>` – Definiert eine Methode `boolean test(T)`. Diese berechnet für eine Eingabe vom Typ `T` einen booleschen Rückgabewert (z. B. `olderThan()`). Damit lassen sich sehr gut Filterbedingungen ausdrücken.
- `Function<T,R>` – Definiert eine Abbildungsfunktion in Form der Methode `R apply(T)`. Damit wird ein allgemeines Konzept von Transformationen beschrieben. Recht gebräuchlich ist beispielsweise die Extraktion eines

Attributs aus einem komplexeren Typ.

- `Supplier<T>` – Stellt ein Ergebnis vom Typ `T` bereit. Im Gegensatz zu `Function<T,R>` erhält ein `Supplier<T>` keine Eingabe. Im Interface ist die Methode `T get()` deklariert. Damit lassen sich Objekterzeugungen auf verschiedene Weise nachbilden.
- `BiFunction<T,U,R>` und `BiConsumer<T,U>` – Wie `Function<T,R>` bzw. `Consumer<T>`, jedoch mit jeweils zwei Eingabewerten vom Typ `T` und `U`.

Auch gibt es Varianten, die auf die Verarbeitung primitiver Typen spezialisiert sind. Für den Typ `int` sind das folgende: `IntConsumer`, `IntFunction<R>`, `IntPredicate` und `IntSupplier`. Gleiche gibt es für die Typen `long` und `double`. Die anderen primitiven Zahlentypen können mithilfe von Widening, einer Typerweiterung etwa von `byte` auf `int`, auf die drei unterstützten Typen abgebildet werden.

Lambdas und Defaultmethoden im Einsatz

Als Motivation zur Verwendung von Lambdas habe ich im einleitenden Beispiel gezeigt, wie man eine Liste von Namen ihrer Länge nach sortiert und die Längen dann kommasepariert ausgibt. Dazu waren über zehn Zeilen Sourcecode nötig. Mithilfe eines Lambdas als Realisierung des Functional Interface `Consumer<T>` kann man das Ganze kurz und knackig mit drei Zeilen realisieren – genau wie im Ausgangsbeispiel findet man auch hier noch den Schönheitsfehler: ein Komma nach dem letzten Element:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Max",
"Stefan");

   names.sort( (str1, str2) -> Integer.compare(str1.length(), str2.length())
);
   names.forEach(it -> System.out.print(it.length() + ", "));
}
```

Vielleicht fragen Sie sich, wofür `it` steht. Erinnern wir uns an die Transformation von anonymer innerer Klasse in einen Lambda. Demzufolge entspricht der im Lambda genutzte Parameter `it` dem Parameter der abstrakten Methode `accept(T)` im Functional Interface `Consumer<T>`. `it` ist eine gebräuchliche Abkürzung für Parameter beliebigen Typs bei Iterationen. In diesem Fall wäre `name` eine besser lesbare Alternative gewesen.

11.2.2 Vorgabe von Standardverhalten

Neben der Erweiterung eines Interface besteht ein weiterer Anwendungsfall von Defaultmethoden darin, für bereits existierende Methoden ein für viele

Einsatzzwecke passendes Standardverhalten vorgeben zu können.

Vor JDK 8 musste jeder Implementierer eines Interface für alle dort definierten Methoden eine eigene Realisierung bereitstellen. Das konnte störend sein, wenn für einzelne Methoden keine sinnvolle Implementierung vorgegeben werden konnte. Das war beispielsweise für eigene Implementierungen des Interface `Iterator<E>` und dessen Methode `remove()` häufig der Fall. Fast immer wurde diese so realisiert, dass dies eine `UnsupportedOperationException` ausgelöst. Nun ist dieses Standardverhalten im JDK durch die Implementierung der Defaultmethode `remove()` umgesetzt. Muss eine eigene Spezialisierungen von `Iterator<E>` erstellt werden, so muss man sich oftmals nicht mehr mit `remove()` beschäftigen, da diese die nachfolgende Implementierung besitzt, sondern kann sich ganz auf die Implementierung der Iteration fokussieren.

```
public interface Iterator<E>
{
    boolean hasNext();
    E next();

    default void remove()
    {
        throw new UnsupportedOperationException("remove");
    }
    // …
}
```

11.2.3 Erweiterte Möglichkeiten durch Defaultmethoden

Wie eben gesehen, lässt sich mithilfe von Defaultmethoden ein gewünschtes Standardverhalten vorgeben. In Spezialisierungen können natürlich eigene Realisierungen für Defaultmethoden bereitgestellt werden, in der man bei Bedarf eine Spezialbehandlung implementieren kann.

Das mag noch etwas merkwürdig klingen, wird aber sofort klar, wenn wir auf ein konkretes Beispiel schauen z. B. das Sortieren von Listen. Im Interface `List<E>` gibt es dazu die bereits vorgestellte Defaultmethode `sort(Comparator<? super E>)`. In der Klasse `ArrayList<E>` wird für die Defaultmethode darüber hinaus eine spezielle Implementierung vorgenommen, die eine performantere Sortierung realisiert. Betrachten wir das Ganze genauer.

Beispiel: Allgemeingültiges Sortieren von Listen

Die im Interface `List<E>` definierte Defaultmethode zum Sortieren nutzt die Utility-Klasse `Collections`, in der das Sortieren von Listen wie folgt realisiert wird:

```
// Auszug aus java.util.Collections
```

```
public static <T> sort(List<T> list, Comparator<? super T> c)
{
    // Schritt 1: Liste in ein temporäres Array übertragen
    Object[] a = list.toArray();
    // Schritt 2: Array sortieren
    Arrays.sort(a, (Comparator)c);
    // Schritt 3: Array zurück in die Liste übertragen
    ListIterator<T> i = list.listIterator();
    for (int j=0; j<a.length; j++)
    {
        i.next();
        i.set((T)a[j]);
    }
}
```

Anhand des Listings erkennen wir, dass im Wesentlichen drei Schritte ausgeführt werden: Zunächst wird die Liste in ein temporäres Array kopiert, dann wird dieses sortiert und anschließend wieder zurück in die Liste übertragen.

Es sind also zur eigentlichen Sortierung noch zusätzlich mehrere Kopier- bzw. Einfügeoperationen in und aus einem Array erforderlich. Diese Schritte verbrauchen Rechenzeit und auch zusätzlichen Speicher. Beides wirkt sich bei zunehmender Anzahl an gespeicherten Elementen negativ aus. Einen solchen Preis muss man jedoch oftmals zahlen, wenn man eine allgemeingültige Lösung realisiert. Der Grund dafür ist, dass diese Art von Lösung verschiedene Einschränkungen besitzt und nicht von Eigenschaften der Spezialisierungen profitieren kann. Betrachten wir mögliche Vorteile für die `ArrayList<E>` als Spezialisierung von `List<E>`.

Realisierung einer Spezialbehandlung

Im Spezialfall der `ArrayList<E>` kann man die zuvor für beliebige `List<E>` gezeigte allgemeingültige Sortierimplementierung performanter gestalten. Dabei nutzt man aus, dass die `ArrayList<E>` ihre Daten bereits in Form eines Arrays hält. Die Sortierung kann somit direkt auf der internen Datenstruktur erfolgen, wodurch sowohl die Umwandlung in ein temporäres Array als auch die Rückkonvertierung entfallen (können). Mit diesem Wissen ist das Sortieren im JDK wie folgt realisiert:

```
// Auszug aus java.util.ArrayList<E>
public void sort(Comparator<? super E> c)
{
    final int expectedModCount = modCount;
    Arrays.sort((E[]) elementData, 0, size, c);
    if (modCount != expectedModCount)
    {
        throw new ConcurrentModificationException();
    }
    modCount++;
}
```

Im Listing sehen wir mit dem Modifikationszähler in Form der Variablen `modCount` ein Detail der Fail-fast-Iteratoren aus dem Collections-Framework. Mithilfe dieses Zählers, der bei jeder Modifikation erhöht wird, können potenzielle Veränderungen erkannt und so mögliche Fehler durch nebenläufige Veränderungen entdeckt werden.

11.2.4 Spezialfall: Was passiert bei Konflikten?

Wenn nun die Definition von Defaultmethoden möglich ist, sollte man sich auch über potenzielle Konflikte Gedanken machen. Es ist durchaus üblich, dass eine Klasse zwei oder mehr Interfaces implementiert. Nehmen wir nun an, zwei Interfaces würden um eine gleichnamige Defaultmethode `sameMethod(int)` wie folgt erweitert:

```
interface Interface1
{
    default int sameMethod(final int x)
    {
        return 0;
    }
}

interface Interface2
{
    default int sameMethod(final int x)
    {
        return 4711;
    }
}

class ErroneousCombination implements Interface1, Interface2
{
}
```

Diese Implementierung führt zu einem Konflikt, weil der Compiler die zu nutzende Methode nicht mehr selbstständig wählen kann. Es kommt zu folgendem Kompilierfehler »`Duplicate default methods named sameMethod with the parameters (int) and (int) are inherited from the types Interface2 and Interface1`«. Um diesen Fehler zu beheben, muss vom Entwickler steuernd eingegriffen werden. Zwei mögliche Abhilfen stelle ich nun kurz vor.

Lösung 1

Eine Möglichkeit, um einen solchen Konflikt zu lösen, besteht darin, eine eigene Methodenimplementierung vorzugeben:

```
public class Correction1 implements Interface1, Interface2
{
    public int sameMethod(int x)
    {
        return 7;
    }
}
```

Lösung 2

Statt einer eigenen Realisierung kann alternativ ein Aufruf der Funktionalität einer

der Defaultmethoden gewünscht sein. Das kann mit folgender spezieller Syntax mit Angabe des Namens in Kombination mit `super` erfolgen:

```
public class Correction2 implements Interface1, Interface2
{
    public int sameMethod(int x)
    {
        return Interface1.super.sameMethod(x);
    }
}
```

11.2.5 Vorteile und Gefahren von Defaultmethoden

Defaultmethoden rufen bei mir gemischte Gefühle hervor, weshalb ich nochmals kurz deren Stärken und mögliche Fallstricke rekapitulieren möchte. Defaultmethoden …

- ermöglichen ***API-Erweiterungen*** unter der Beibehaltung von ***Rückwärtskompatibilität***,
- erlauben es, ein gewünschtes ***Standardverhalten vorzugeben***,
- kann man überschreiben. Damit wird es bei Bedarf möglich, ein durch das Basisinterface vorgegebenes Verhalten bzw. eine ***Spezialbehandlung*** zu modifizieren oder zu ersetzen.

Durch diese Eigenschaften stellen Defaultmethoden zweifellos eine wichtige Erweiterung von Java dar, ohne die eine Integration der neuen Funktionalitäten von Lambdas in das Collections-Framework und in andere Teile der JDK-Bibliotheken nur unzureichend möglich gewesen wäre – das werden wir später noch ausführlich bei der Besprechung der Streams in Abschnitt 12.3 erfahren. In dieser Hinsicht sind Defaultmethoden für API-Designer und Framework-Entwickler ein geeignetes Konstrukt, um eine sanfte, evolutionäre Entwicklung von APIs vornehmen zu können. Dies hat die Integration der neuen Funktionalitäten in das JDK 8 bereits gezeigt.

Andererseits öffnen Defaultmethoden Tür und Tor für eine laxe Programmierung mit zu wenig Fokus auf sauberes Design, da sich ja nachträglich immer noch Funktionalität in das System »hineinquetschen« lässt. Außerdem unterstützt Java durch Defaultmethoden nun auch Mehrfachvererbung: Glücklicherweise umfasst diese lediglich Verhalten und nicht Zustand. Letzteres hatte James Gosling aufgrund der damit möglichen und aus C++ bekannten Designprobleme bewusst aus Java herausgehalten.

Fallstrick mit JDK 8: Defaultmethoden und Zugriff auf Attribute

Spontan könnte man auf die Idee kommen, das neue Feature der Defaultmethoden dazu einzusetzen, um Zugriffsmethoden und sogar bereits Attribute bereitzustellen, etwa in Form folgender Implementierung:

```
public interface MisleadingDefaultMethods
{
    String name = "<Name>";     // public static final

    default void setName(final String newName)
    {
        this.name = newName;
    }

    default String getName()
    {
        return this.name;
    }
}
```

Zunächst sieht die Implementierung in Ordnung aus, jedoch kompiliert sie nicht. Entgegen der reinen Notation im Sourcecode, die man für die Definition eines Instanzattributs halten könnte, sind alle Attribute in Interfaces gemäß JLS implizit immer `public`, `static` und `final` und somit Klassenattribute. Ein Zugriff per `this` ist damit nicht möglich und löst Kompilierfehler aus. Selbst wenn man Zugriffe auf Klassenattribute realisieren möchte, so sind nur Lesezugriffe möglich, da die statischen Attribute laut JLS `final` sind.

11.2.6 Statische Methoden in Interfaces

Interfaces können mit Java 8 nicht nur Defaultmethoden, sondern auch statische Methoden enthalten. Damit wird es möglich, Hilfsmethoden direkt in Interfaces bereitzustellen und dafür keine separaten Utility-Klassen anbieten zu müssen. Diese Konstellation kennt man zum einen aus dem JDK und zum anderen wohl auch aus eigenen Projekten. Beispiele aus dem JDK sind etwa die Utility-Klasse `Paths` zum Interface `Path` aus dem Package `java.nio.file` oder die Kombination von der Utility-Klasse `Executors` und dem Interface `Executor` aus dem Package `java.util.concurrent`.

Im JDK 8 finden sich viele Beispiele, in denen man die Hilfsmethoden statt in einer eigenen Utility-Klasse direkt im Interface selbst implementiert hat. Dies gilt etwa für das Interface `Comparator<T>`. Nachfolgendes Listing zeigt nur auszugsweise einige statische Erzeugungsmethoden für spezielle Komparatoren, hier für eine umgedrehte Sortierung, die natürliche Ordnung sowie Sortierungen, die `null`-Werte vorne bzw. hinten einsortieren:

```
@FunctionalInterface
public interface Comparator<T>
{
    // …

    public static <T extends Comparable<? super T≫ Comparator<T>
reverseOrder()
    {
        return Collections.reverseOrder();
    }
```

```
    public static <T extends Comparable<? super T≫ Comparator<T>
naturalOrder()
    {
        return (Comparator<T>) Comparators.NaturalOrderComparator.INSTANCE;
    }

    public static <T> Comparator<T> nullsFirst(Comparator<? super T>
comparator)
    {
        return new Comparators.NullComparator<>(true, comparator);
    }

    public static <T> Comparator<T> nullsLast(Comparator<? super T>
comparator)
    {
        return new Comparators.NullComparator<>(false, comparator);
    }

    // …
}
```

Zwar lassen sich mithilfe statischer Methoden in Interfaces auf einfache Art gewisse Funktionalitäten bereitstellen, allerdings ***sollte man sich bewusst sein, dass man damit das Konzept des Interface zur Definition einer Schnittstelle immer mehr verwässert***. Aber nicht nur das! In diesem Beispiel erkennen wir, dass die eigentliche Schnittstellenbeschreibung nun Abhängigkeiten von diversen speziellen Klassen und Implementierungsdetails besitzt. Für diese Realisierung im JDK ist das wohl nicht so kritisch. ***Für eigene Interfaces sollte man sich jedoch sehr genau überlegen, ob man dort statische Methoden anbieten möchte und welche möglichen Auswirkungen und Abhängigkeiten sich dadurch ergeben***. Deklariert man dagegen lediglich Methoden in Interfaces, nimmt also eine reine Schnittstellenbeschreibung vor, so lassen sich diese Interfaces meistens viel leichter auch in andere Projekte integrieren, ohne eine (unerwartete) Menge von weiteren Abhängigkeiten anzuziehen.

11.3 Methodenreferenzen

Wir haben bisher gesehen, wie sich Lambdas gewinnbringend einsetzen lassen. Darüber hinaus kann der Einsatz der mit JDK 8 eingeführten Methodenreferenzen dazu beitragen, die Lesbarkeit des Sourcecodes zu erhöhen. Das Sprachfeature der Methodenreferenzen besitzt die Syntax `Klasse::Methodenname` und verweist auf …

- eine Methode – `System.out::println`, `Person::getName`, …
- einen Konstruktor – `ArrayList::new`, `Person[]::new`, …

Das wirkt recht unspektakulär. Eine Methodenreferenz kann aber zur Vereinfachung der Schreibweise anstelle eines Lambdas genutzt werden:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Max", "Andy", "Michael",
"Stefan");
```

```
    // Lambda
    names.forEach( it -> System.out.println(it) );

    // Methodenreferenz
    names.forEach( System.out::println );
}
```

Wie man sieht, verbessert sich die Lesbarkeit. Allerdings sollte man sich noch folgende Fragen zu der Ersetzung stellen: Methoden erhalten oftmals Parameter – wie auch im Listing. Wie werden diese für Methodenreferenzen übergeben? Wie ist die Reihenfolge bei mehreren? Die Antwort darauf ist, dass diese Informationen vom Compiler ermittelt und automatisch beim jeweiligen Methodenaufruf übergeben werden.

Ergänzend zu dieser Ausführung möchte ich an ein paar Beispielen zeigen, wie sich Methodenreferenzen auf Lambdas bzw. andersherum abbilden lassen. Dabei gibt es vier verschiedene Varianten, die in Tabelle 11-1 dargestellt sind.

Tabelle 11-1 *Methodenreferenzen*

Referenz auf …	Als Methodenreferenz	Als Lambda
Statische Methode	`String::valueOf`	`obj -> String.valueOf(obj)`
Instanzmethode eines Typs	`Object::toString`	`obj -> obj.toString()`
	`String::compareTo`	`(str1, str2) -> str1.compareTo(str2)`
Instanzmethode eines Objekts	`person::getName`	`() -> person.getName()`
Konstruktor	`ArrayList::new`	`() -> new ArrayList<>()`

Spezialfall

Wir haben eben gesehen, wie sich Methodenreferenzen und Lambdas ineinander überführen lassen. Dabei gilt: Ein Lambda ist immer dann durch eine Methodenreferenz ersetzbar, wenn neben dem Methodenaufruf keine weiteren Aktionen in dem Lambda erfolgen. Des Öfteren möchte man aber möglicherweise noch einige weitere Anweisungen ausführen, beispielsweise eine Konkatenation eines Kommas für eine kommaseparierte Ausgabe. Dazu muss man (zunächst) einen

Lambda nutzen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Max", "Andy", "Michael",
"Stefan");

   // names.sort((str1, str2) -> Integer.compare(str1.length(),
str2.length()));
   names.sort(MethodReferenceExample::stringLengthCompare);

   // Methodenreferenz nachfolgend nicht direkt nutzbar
   names.forEach(it -> System.out.print(it.length() + ", "));
}

public static Comparator<String> stringLengthCompare()
{
    return (str1, str2) -> Integer.compare(str1.length(), str2.length());
}

// Diese Methode wird referenziert
public static int stringLengthCompare(final String str1, final String str2)
{
    return Integer.compare(str1.length(), str2.length());
}
```

Bevor ich auf das Umwandeln eines Lambdas in eine Methodenreferenz eingehe, möchte ich noch darauf hinweisen, dass im obigen Beispiel bewusst zwei gleichnamige Methoden definiert sind, um zu demonstrieren, dass die Methodenreferenz die am besten passende Methode findet. Hier ist das die letztere Methode, die zwei Eingaben vom Typ `String` entgegennimmt und einen Rückgabe vom Typ `int` liefert, also die Methodensignatur `int compare(String o1, String o2)` aus dem Interface `Comparator<String>` erfüllt.

Im Beispiel gibt es einen weiteren Punkt, nämlich die Konvertierung des Lambdas zur Ausgabe in eine Methodenreferenz. Wir definieren eine eigene Methode `commaSeparatedPrint()` und rufen diese wie folgt auf:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Andy", "Michael", "Max",
"Stefan");

   names.sort(stringLengthCompare());
   names.forEach(MethodReferenceExample::commaSeparatedPrint);
}

public static void commaSeparatedPrint(final String str)
{
    System.out.print(str.length() + ", ");
}
// …
```

11.4 Fazit

In diesem Kapitel haben wir gelernt, dass Lambdas überall dort eingesetzt werden

können, wo vor JDK 8 eine anonyme innere Klasse zur Implementierung eines SAM-Typs benötigt wurde. Als Beispiele wurden verschiedene funktionale Interfaces mithilfe von Lambdas implementiert, wodurch weniger Sourcecode benötigt wurde. Darüber hinaus lässt sich mitunter Funktionalität einfacher beschreiben, insbesondere dann, wenn man auf Daten der umgebenden Klasse zugreifen möchte und dazu die `this`-Referenz benötigt. Mit anonymen inneren Klassen gibt es dabei syntaktische Besonderheiten: Die `this`-Referenz auf die äußere Klasse muss immer mit deren Klassennamen qualifiziert werden, was die Lesbarkeit erschwert. Aber auch Lambdas sind (vor allem am Anfang) nicht immer einfach zu lesen. Hier gilt wie bei jedem Feature: Man sollte es mit Bedacht und wo es sinnvoll ist einsetzen.

Alternativ zu Lambdas und oftmals eleganter ist es, die Funktionalität als Methode zu realisieren und diese über eine Methodenreferenz anzusprechen. Wenn Lambdas länger als etwa fünf bis zehn Zeilen werden, so sollte man sich überlegen, die Funktionalität in Form von Klassen oder Methoden zu realisieren, um deren potenzielle Wiederverwendbarkeit zu erhöhen. Die Argumentation für Lambdas gilt ebenso für anonyme innere Klassen. In beiden Fällen existiert sonst einiger Sourcecode, der nicht aus anderen Kontexten (anderen Klassen, Methoden, Lambdas) zugreifbar ist und somit auch nicht wiederverwendet werden kann.[1]

12 Bulk Operations on Collections

Neben der kürzeren Schreibweise für SAM-Typen lassen sich Lambdas insbesondere bei der Formulierung von Algorithmen und bei der Verarbeitung von Daten in Collections vorteilhaft einsetzen. Mit JDK 8 wurden zwei Varianten von Massenoperationen auf Collections (Bulk Operations on Collections) eingeführt. Zum einen sind dies Erweiterungen in den jeweiligen Interfaces, etwa `List<E>`. Zum anderen sind dies sogenannte Streams. Diese liefern die Schnittstelle zur funktionalen Programmierung und bieten eine effiziente Möglichkeit zur Parallelisierung, wodurch sich die Fähigkeiten von Multicore-Maschinen besser ausnutzen lassen. Vor JDK 8 musste dies explizit ausprogrammiert werden – weil das aufwendig war, wurden Daten in Collections oftmals sequenziell verarbeitet. Wurde doch Parallelität benötigt, so konnte man beispielsweise auf das mit JDK 7 eingeführte Fork-Join-Framework zurückgreifen. Allerdings möchte man sich als Applikationsentwickler eigentlich möglichst wenig mit den zugrunde liegenden Details auseinandersetzen müssen, insbesondere auch nicht mit einer geeigneten Zerlegung der Aufgabe in einzelne durch Fork-Join zu verarbeitende Tasks. In der Regel besteht der Wunsch, sich auf einer höheren, konzeptionellen Ebene mit dem Thema Parallelisierung beschäftigen zu können. Mit JDK 8 und Streams wird dies möglich.

Bevor wir uns genauer mit Streams beschäftigen, werfen wir in Abschnitt 12.1 als Vorbereitung einen Blick auf zwei Varianten der Iteration. In Abschnitt 12.2 schauen wir dann auf API-Erweiterungen im Collections-Framework. Anschließend behandeln wir Streams und im Speziellen das Filter-Map-Reduce-Framework in den Abschnitten 12.3 und 12.4. Abgerundet wird dieses Kapitel durch eine Betrachtung möglicher Fallstricke der funktionalen Programmierung mit Java 8 in Abschnitt 12.5.

12.1 Externe vs. interne Iteration

Mit ***Iteration*** bezeichnet man das ***Durchlaufen einer Collection***. Dies kann als externe oder als interne Iteration geschehen. Dabei bedeutet externe Iteration, dass die Collection das Durchlaufen unterstützt, etwa durch indizierte Zugriffe oder aber mithilfe eines `java.util.Iterator<E>`s. Hier wird der Vorgang der Iteration vom Aufrufer kontrolliert. Bei der internen Iteration wird der Vorgang des Durchlaufens dagegen durch die Collection-Klasse gekapselt und dort intern realisiert. Implementierungsdetails bleiben so verborgen, allerdings sind auch die Möglichkeiten zur Einflussnahme begrenzt. Schauen wir nachfolgend kurz auf einige Beispiele für die externe und interne Iteration.

12.1.1 Externe Iteration

Nehmen wir an, wir wollten alle Elemente einer Collection auf der Konsole ausgeben. Herkömmlicherweise könnte man dies wie folgt implementieren:

```
final List<String> names = Arrays.asList( "Andi", "Mike", "Ralph", "Stefan" );

// Klassische Variante mit Iterator …
final Iterator<String> it = names.iterator();
while (it.hasNext())
{
    System.out.println(it.next());
}

// … oder alternativ mit indiziertem Zugriff
for (int i = 0; i < names.size(); i++)
{
    System.out.println(names.get(i));
}

// JDK-5-Schreibweise mit "for-each"
for (final String name : names)
{
    System.out.println(name);
}
```

An diesem Beispiel erkennt man sehr schön die iterative und sequenzielle Abarbeitung sowohl für die Variante mit Iterator als auch für den danach gezeigten indizierten Zugriff. Die Variante mit der seit JDK 5 verfügbaren sogenannten for-each-Schleife[1] zeigt den sequenziellen Charakter weniger klar.[2] In allen drei Fällen spricht man von ***externer Iteration***, weil die ***Traversierung im Applikationscode programmiert*** wird.

12.1.2 Interne Iteration

Mit JDK 8 wurden die Klassen des Collections-Frameworks derart erweitert, dass sie verschiedene Verarbeitungsmethoden anbieten, die man bisher über `for`- oder `while`-Schleifen – wie oben im Listing – selbst programmieren musste. Erweiterungen sind die schon verwendeten Methoden `sort(Comparator<? super T>)` und `forEach(Consumer<? super T>)`. In beiden Fällen (und auch im Allgemeinen) übergibt man die in der internen Iteration auszuführende Funktionalität. Dazu sind verschiedene Callback-Interfaces definiert. Seit JDK 8 bieten sich zu deren Implementierung Lambdas und Defaultmethoden an:

```
// Interne Iteration in drei Varianten
names.forEach((String name) -> { System.out.println(name); });
names.forEach(name -> System.out.println(name) );
names.forEach(System.out::println);
```

Die im Listing gezeigte Form wird ***interne Iteration*** genannt. Hierbei muss die Iteration nicht vom Entwickler selbst programmiert werden, sondern diese ***wird im Framework realisiert***. Man übergibt – wie bereits gesehen – nur die auszuführende Aktion.

12.1.3 Externe vs. interne Iteration an einem Beispiel

Zwar kennen wir jetzt die Begriffe, jedoch sind die Unterschiede und Auswirkungen möglicherweise noch nicht wirklich greifbar. Daher möchte ich an einem Beispiel die Vorteile einer internen Iteration verdeutlichen.

Externe Iteration

Nehmen wir an, es wäre eine Hilfsmethode zu realisieren, die eine Aktion für die Objekte einer Liste ausführen soll, z. B. alle in einem Malprogramm selektierten Figuren etwas heller darzustellen. Als externe Iteration würde man dies etwa folgendermaßen realisieren:

```
public static void brightenExtern(final List<GraphicsFigure> selectedFigures)
{
    for (final GraphicsFigure figure : selectedFigures)
    {
        brighten(figure);
    }
}
```

Diese Lösung besitzt die zuvor angedeuteten Eigenschaften der vom Aufrufer kontrollierten und durchgeführten Iteration. Nachteilig ist in der Regel, dass diese sequenziell abläuft. Darüber hinaus wird die Funktionalität und die Iteration an sich miteinander gemischt – hier wiegt es nicht so schwer, da durch den Methodenaufruf recht gut für Klarheit gesorgt wird.

Interne Iteration

Schauen wir nun auf die korrespondierende interne Iteration:

```
public static void brightenIntern(final List<GraphicsFigure> selectedFigures)
{
    selectedFigures.forEach(figure ->
    {
        brighten(figure);
    });
}
```

Auf den ersten Blick sieht diese Implementierung mit Lambda kaum anders aus als die externe Variante. Dieser Eindruck täuscht, insbesondere weil ich hier bewusst eine ähnliche Formatierung des Sourcecodes gewählt habe. Rekapitulieren wir zunächst nochmals, dass hier die Iteration durch das Framework, also den Implementierer von `forEach(Consumer<? super T>)`, realisiert wird. Dies bietet vor allem den Vorteil, dass theoretisch eine Parallelverarbeitung erfolgen kann, solange die Aktionen für die einzelnen Elemente voneinander unabhängig sind. Ebenfalls kann bei interner Iteration bei Bedarf die Reihenfolge der Verarbeitung von der sequenziellen abweichen.

Der ganz entscheidende Unterschied ist aber, dass zur Berechnung ein Lambda genutzt wird. Durch Einsatz des SAM-Typs `Consumer<T>` kann man mit ein wenig Erfahrung die obige Methode mit minimalem Aufwand wie folgt verallgemeinern, sodass die Abarbeitung beliebiger `Consumer<T>` möglich wird:

```
public static void process(final Collection<GraphicsFigure> figures,
                           final Consumer<GraphicsFigure> consumer)
{
    figures.forEach(consumer);
}
```

Um die Funktionalität der Aufhellung zu realisieren, können wir der obigen Methode dann einen passenden `Consumer<GraphicsFigure>` wie folgt übergeben:

```
final Consumer<GraphicsFigure> brighten = figure -> brighten(figure);
process(figures, brighten);
```

Man benötigt wenig Fantasie, um zu erkennen, welche Vielfalt an Möglichkeiten sich daraus ergibt, wenn man andere Realisierungen von `Consumer<GraphicsFigure>` nutzt und somit beliebige Funktionalität ausführen kann. Wir beginnen zu erahnen, was funktionale Programmierung ausmacht: Man kann Funktionalität in Form von Sourcecode als Parameter übergeben (»Code as Data«) und an beliebiger Stelle ausführen. Wenn wir noch ein wenig darüber nachdenken, erkennen wir, dass die Methode `process()` überflüssig ist und man nur einen Lambda und die Iteration benötigt:

```
final Consumer<GraphicsFigure> brighten = figure -> brighten(figure);
figures.forEach(brighten);
```

Diese Herleitung mag dem Geübten klar sein, für Einsteiger dient sie vor allem dem Warmwerden mit der neuen Denkweise und der neuen Art zu programmieren.

Hinweis: Preconditions und Parameterprüfung mit interner Iteration

Nicht nur in der eigentlichen Programmlogik, sondern auch für Zustandsprüfungen kann man von interner Iteration profitieren. Zur Demonstration schauen wir uns eine typische Realisierung einer Methode an, die Aktionen für diejenigen Elemente einer Collection ausführt, die eine bestimmte Bedingung erfüllen:

```
public static void doAction(final List<Person> persons)
{
    // Zustandsprüfung
    Objects.requireNonNull(persons, "list of persons must not be null");

    final Iterator<Person> it = persons.iterator();
    while (it.hasNext())
    {
        final Person person = it.next();
        // Sicherheitsprüfung und Logik
        if (person != null && accept(person))
        {
            performAction(person);
            // …
```

Wir konzentrieren uns hier auf die Prüfung gültiger Eingaben: Oftmals wird zwar – wie auch hier – sichergestellt, dass der oder die Übergabeparameter ungleich `null` sind. Seit JDK 7 nutzt man dazu sinnvollerweise die Methode `Objects.requireNonNull()`. Aufwendiger gestaltet es sich häufig, die einzelnen Elemente auf Gültigkeit bzw. zumindest auf ungleich `null` zu testen. Im obigen Listing wird dies in Kombination mit der eigentlichen Anwendungslogik realisiert. Das spart zwar Schreibaufwand und ist etwas performanter, jedoch vermischt man hier Zustandsprüfung und Anwendungslogik. Dadurch lässt sich häufig die Funktionalität schwieriger lesen und extrahieren. Erschwerend kommt hinzu, dass man fehlerhafte Eingaben verschleiert und diese Fehlersituation stillschweigend behandelt. Das sollte man für ein klares Design – wenn möglich – vermeiden. Eine zusätzliche externe Iteration zur Konsistenzprüfung wirkt allerdings auch nicht sonderlich elegant:

```
// Prüflogik mit externer Iteration
for (final Person person : persons)
{
    Objects.requireNonNull(person);
}
```

Wenn wir aber eine interne Iteration mit `forEach(Consumer<? super T>)` mit Methodenreferenzen kombinieren, lässt sich die Zustandsprüfung wie folgt klarer realisieren und fügt sich dadurch nahtlos in die anderen Prüfungen ein:

```
// Prüflogik mit interner Iteration und Methodenreferenz
persons.forEach(Objects::requireNonNull)
```

Auf diese Weise kann man mit Zustandsprüfungen für die Einhaltung von Preconditions sorgen, wodurch sich nachfolgende Programmteile auf eine korrekte Initialisierung verlassen können. Das Ganze hat allerdings seinen Preis: Die Iteration über die Elemente erfolgt nun zweimal, wodurch sich die Performance verschlechtert. Das ist zu bedenken, wenn man performancekritische Abschnitte überarbeiten muss.

12.2 Collections-Erweiterungen

Neben der bereits eingesetzten Iteration mit `forEach(Consumer<? super T>)` existieren diverse weitere Beispiele für diese Art der internen Iteration im JDK. Einige wichtige finden wir im Collections-Framework, die wir im Anschluss betrachten, wobei wir zunächst dafür grundlegende funktionale Interfaces kennenlernen werden.

12.2.1 Das Interface `Predicate<T>`

Das funktionale Interface `java.util.function.Predicate<T>` erlaubt es, sogenannte ***Prädikate*** zu formulieren. Das sind boolesche Bedingungen, die durch

Aufruf der im Interface definierten Methode `boolean test(T)` ausgewertet werden. Darüber hinaus sind ein paar andere Methoden im Interface `Predicate<T>` wie folgt definiert (gekürzt):

```
@FunctionalInterface
public interface Predicate<T>
{
   boolean test(T t);

   default Predicate<T> and(Predicate<? super T> other) { … }
   default Predicate<T> negate() { … }
   default Predicate<T> or(Predicate<? super T> other) { … }
}
```

Auf die gezeigten Defaultmethoden gehe ich ein, nachdem ich einige Beispiele für Implementierungen des Interface selbst und der `test(T)` -Methode gegeben habe.

Im nachfolgenden Listing sind mithilfe von Lambdas und Methodenreferenzen einfache Prüfungen auf den Wert `null`, einen Leerstring oder ein Mindestalter von 18 Jahren kurz und knackig formuliert:

```
public static void main(final String[] args)
{
   // Prädikate formulieren
   final Predicate<String> isNull = str -> str == null;
   final Predicate<String> isEmpty = String::isEmpty;
   final Predicate<Person> isAdult = person -> person.getAge() >= 18;

   System.out.println("isNull(""):      " + isNull.test(""));
   System.out.println("isEmpty(""):     " + isEmpty.test(""));
   System.out.println("isEmpty('Pia'):  " + isEmpty.test("Pia"));
   System.out.println("isAdult(Pia):    " + isAdult.test(new Person("Pia",
55)));
}
```

Führt man das Programm FIRSTPREDICATESEXAMPLE aus, so werden die Prädikate erwartungsgemäß wie folgt ausgewertet:

```
isNull(""):      false
isEmpty(""):     true
isEmpty('Pia'): false
isAdult(Pia):    true
```

Komplexere Bedingungen mit Prädikaten formulieren

Zwar kann man mit einfachen Prädikaten schon einige Anwendungsfälle abdecken. Zur Realisierung komplexerer Abfragen wird man jedoch verschiedene Bedingungen miteinander kombinieren wollen. Um boolesche Verknüpfungen auszuführen, bietet sich oftmals der Einsatz der folgenden drei Defaultmethoden an:

- `negate()` – Negiert die Bedingung.
- `and(Predicate<? super T>)` – Verknüpft die aktuelle Bedingung mit einer anderen Bedingung mit logischem UND.
- `or(Predicate<? super T>)` – Verknüpft die aktuelle Bedingung mit einer

anderen Bedingung mit logischem ODER.

Mit diesem Wissen bauen wir unser Beispiel ein wenig aus und richten den Fokus dabei auf die Kombination von Prädikaten.

```
public static void main(final String[] args)
{
   final List<Person> persons = createDemoData();

   // Einfache Prädikate formulieren
   final Predicate<Person> isAdult = person -> person.getAge() >= 18;
   final Predicate<Person> isMale = person -> person.getGender() ==
Gender.MALE;

   // Negation einzelner Prädikate
   final Predicate<Person> isYoung = isAdult.negate();
   final Predicate<Person> isFemale = isMale.negate();

   // Kombination von Prädikaten mit AND
   final Predicate<Person> boys = isMale.and(isYoung);
   final Predicate<Person> women = isFemale.and(isAdult);

   // Kombination von Prädikaten mit OR
   final Predicate<Person> boysOrWomen = boys.or(women);

   removeAll(persons, boysOrWomen);
   System.out.println(persons);
}
```

Im Listing sehen wir den Aufruf der erst im nächsten Abschnitt gezeigten Methode `removeAll(List<E>, Predicate<E>)`, deren Funktionalität intuitiv verständlich ist und Elemente entfernt, die der übergebenen Bedingung entsprechen. In diesem Beispiel sind dies alle männlichen Personen unter 18 sowie alle erwachsenen Frauen.

12.2.2 Die Methode `Collection.removeIf()`

Wie im Beispiel schon angedeutet, lassen sich Prädikate für das Löschen von Elementen, die einer Bedingung genügen, einsetzen. Herkömmlicherweise lässt sich dies mit externer Iteration recht mühsam mithilfe eines Iterators in etwa wie folgt lösen:

```
// Löschen von Elementen mit externer Iteration
private static <E> void removeAll(final List<E> list,
                                  final Predicate<? super E> condition)
{
    final Iterator<E> it = list.iterator();
    while (it.hasNext())
    {
        final E element = it.next();
        if (condition.test(element))
        {
            it.remove();
        }
    }
```

```
}
```

Praktischerweise wurden in JDK 8 im Interface `Collection<E>` verschiedene Methoden hinzugefügt. Dort findet man z. B. die Methode `removeIf(Predicate<T>)`, die funktional analog zur eben selbst realisierten Methode `removeAll(List<E>, Predicate<E>)` arbeitet, dabei jedoch eine interne Iteration nutzt. Die genannte Methode wollen wir wieder am Beispiel der Liste von Personen kennenlernen. Aus dieser sollen diejenigen Einträge herausgelöscht werden, die einen Leereintrag darstellen. Das realisieren wir mit der Methode `removeIf(Predicate<T>)`, der als Eingabe vom Typ `Predicate<T>` eine Methodenreferenz auf `String::isEmpty` dient:

```
public static void main(final String[] args)
{
   final List<String> names = createDemoNames();

   // Löschaktionen ausführen
   names.removeIf(String::isEmpty)
   System.out.println(names);
}

private static List<String> createDemoNames()
{
   final List<String> names = new ArrayList<>();
   names.add("Max");
   names.add("");                  // Leereintrag
   names.add("Andy");
   names.add("Michael");
   names.add("   ");                // potenziell auch ein "Leereintrag"
   names.add("Stefan");
   return names;
}
```

Starten wir das Programm REMOVEIFEXAMPLE, so wird die beschriebene Löschoperation ausgeführt und es kommt zu folgender Ausgabe:

```
[Max, Andy, Michael,    , Stefan]
```

Wir sehen, dass der Whitespace-Eintrag in der Liste verblieben ist. Eine minimal komplexere Bedingung hilft, auch diesen Eintrag zu entfernen:

```
names.removeIf(str -> str.trim().isEmpty());
```

Die gezeigte Umsetzung birgt jedoch die Gefahr von `NullPointerExceptions`, wenn die Eingabewerte auch den Wert `null` enthalten können. Statt den Lambda etwas komplexer zu gestalten, wollen wir nachfolgend als Alternative und zur Korrektur die mit JDK 8 im Interface `List<E>` neu eingeführte Methode `replaceAll(UnaryOperator<T>)` verwenden. In der Signatur sehen wir den bisher unbekannten Typ `UnaryOperator<T>`, den wir zunächst kurz anschauen, bevor wir dann auf das Löschen von Einträgen zurückkommen.

12.2.3 Das Interface `UnaryOperator<T>`

Im funktionalen Interface `java.util.function.UnaryOperator<T>` selbst ist lediglich die statische Methode `identity()` definiert. Entscheidender ist, dass es über sein Basisinterface `Function<T,R>` die Methode `apply(T)` anbietet. Diese bildet ein Element vom Typ `T` auf ein Element vom Typ `R` ab. Für den `UnaryOperator<T>` sind die Typen `T` und `R` gleich (Interfaces sind nachfolgend auf das Wesentliche gekürzt):

```
@FunctionalInterface
public interface UnaryOperator<T> extends Function<T, T>
{
    // Statische Methoden seit JDK 8 in Interfaces erlaubt
    static <T> UnaryOperator<T> identity()
    {
        return t -> t;
    }
}

@FunctionalInterface
public interface Function<T, R>
{
    R apply(T t);
    // …
}
```

Das Ganze ist noch etwas abstrakt, daher schauen wir uns verschiedene Realisierungen von `UnaryOperator<String>` an: Man könnte etwa alle mit »`M`« startenden Namen speziell markieren und mit Großbuchstaben schreiben (`markTextWithM`). Für die Praxis eher relevante Beispiele bestehen in dem gezeigten Trimmen (`trimmer`) und der Abbildung von `null`-Werten auf gewünschte Defaultwerte (`mapNullToEmpty`). Die korrespondierenden `UnaryOperator<String>`s realisieren wir wie folgt:

```
public static void main(final String[] args)
{
   // Mark
   final UnaryOperator<String> markTextWithM = str -> str.startsWith("M") ?
                                        "≫" + str.toUpperCase() + "≪" : str;

   printResult("Mark 1", "unchanged", markTextWithM);
   printResult("Mark 2", "Michael", markTextWithM);

   // Trim
   final UnaryOperator<String> trimmer = String::trim;
   printResult("Trim 1", "no_trim", trimmer);
   printResult("Trim 2", "  trim me   ", trimmer);

   // Map
   final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" : str;
   printResult("Map same", "same", mapNullToEmpty);
   printResult("Map null", null, mapNullToEmpty);
}

private static void printResult(final String text, final String value,
                                final UnaryOperator<String> op)
{
   System.out.println(text + ": '" + value + "' -> '" + op.apply(value) +
"'");
```

```
}
```

Das Programm UNARYOPERATOREXAMPLE produziert folgende Ausgaben:

```
Mark 1: 'unchanged' -> 'unchanged'
Mark 2: 'Michael' -> '≫MICHAEL≪'
Trim 1: 'no_trim' -> 'no_trim'
Trim 2: '  trim me   ' -> 'trim me'
Map same: 'same' -> 'same'
Map null: 'null' -> ''
```

12.2.4 Die Methode `List.replaceAll()`

Auch das Interface `List<E>` wurde mit JDK 8 erweitert. Nachfolgend betrachten wir die Methode `replaceAll(UnaryOperator<T>)`. Diese ermöglicht es, für alle Elemente einer `Collection<E>` eine Aktion auszuführen: Jedes Element wird durch den Rückgabewert der Implementierung der Methode `apply(T)` des funktionalen Interface `UnaryOperator<T>` ersetzt. Durch die Anweisungen der Realisierung wird auch die Entscheidung getroffen, welche Elemente wie bearbeitet werden sollen. Im Speziellen müssen nicht immer alle Elemente tatsächlich auch verändert werden. Das »`All`« im Namen bezieht sich also lediglich darauf, dass die übergebene Aktion für alle Elemente der Liste ausgeführt wird.

Nach diesen zuvor eher theoretischen Details kommen wir auf ein konkretes Anwendungsbeispiel: Nehmen wir an, wir würden entweder von einer externen Datenquelle oder dem GUI eine Liste von Eingabewerten erhalten. Oftmals entsprechen solche Eingaben nicht den Erwartungen und verstoßen gegen Regeln, etwa sind Einträge leer, bestehen nur aus Leerzeichen oder enthalten diese am Anfang oder Ende. All dies erschwert die weitere Bearbeitung. Die Grundlagen für eine Korrekturfunktionalität, die derartige Werte umwandelt oder herausfiltert, haben wir bereits kennengelernt. Wir müssen das Ganze nur noch geeignet kombinieren:

```
public static void main(final String[] args)
{
    final List<String> names = createDemoNames();

    // Spezialbehandlung von null-Werten
    final UnaryOperator<String> mapNullToEmpty = str -> str == null ? "" :
str;
    names.replaceAll(mapNullToEmpty);

    // Leerzeichen abschneiden
    names.replaceAll(String::trim);

    // Leereinträge herausfiltern
    names.removeIf(String::isEmpty);

    System.out.println(inputs);
}
```

Mit dieser Erweiterung werden nicht nur potenzielle `null`-Einträge entfernt, sondern auch diejenigen Einträge, die Leerstrings oder lediglich Leerzeichen

enthalten. Außerdem werden Leerzeichen am Anfang und Ende von Einträgen gelöscht.

12.3 Streams

Wir haben bisher verschiedene spezialisierte Formen von Bulk Operations kennengelernt. Deutlich mehr Möglichkeiten bietet das in JDK 8 neu eingeführte Konzept der ***Streams***. Dabei spielt das Interface `java.util.stream.Stream<T>` eine Schlüsselrolle. Streams sind eine Abstraktion für ***Folgen von Verarbeitungsschritten auf Daten***. Darüber hinaus ähneln Streams sowohl Collections als auch Iteratoren, wobei Streams keine Speicherung der Daten vornehmen und nur einmal traversiert werden können. Als weitere und beste Analogie kann die Abarbeitung als Pipeline oder Fließband betrachtet werden. Dabei unterscheidet man zwischen folgenden drei Typen von Operationen: ***Create*** (Erzeugung), ***Intermediate*** (Berechnung) und ***Terminal*** (Ergebnisbereitstellung). Nachfolgend ist dies schematisch dargestellt:

$$\underbrace{Quelle \Rightarrow STREAM}_{Create} \Rightarrow \underbrace{OP_1 \Rightarrow OP_2 \Rightarrow ... \Rightarrow OP_n}_{Intermediate} \Rightarrow \underbrace{Ergebnis}_{Terminal}$$

Einführendes Beispiel

Das folgende Listing zeigt die Operationen, ohne auf Details einzugehen. Das geschieht in den folgenden Abschnitten. Hier geht es nur darum, einen ersten Eindruck für Streams und die Verarbeitung zu bekommen. Dazu schauen wir auf die Filterung einer Liste von Personen auf alle Erwachsenen. Diese werden als `List<Person>` zurückgegeben:

```
List<Person> adults = persons.stream().                    // Create
                              filter(Person::isAdult).     // Intermediate
                              collect(Collectors.toList()); // Terminal
```

Neben all diesen (noch unbekannten) Implementierungsneuerungen erkennt man sehr schön, dass sich Konzepte und das »Was« viel klarer erkennen lassen und nicht das »Wie« (die Details der Implementierung der Funktionalität) im Vordergrund steht.

12.3.1 Streams erzeugen — Create Operations

Nach dem ersten Beispiel zu Streams wollen wir unsere Kenntnisse vertiefen. In den folgenden Abschnitten stelle ich Varianten zur Erzeugung von Streams vor.

Streams für Arrays und Collections

Zunächst sollten wir uns fragen, wie wir an ein `Stream`-Objekt kommen. In den vorangegangenen Beispielen haben wir schon gesehen, dass dies für Arrays oder Collections leicht möglich ist, Für beide ist eine `stream()`-Methode definiert, die man folgendermaßen nutzen kann:

```
final String[] namesData = { "Karl", "Ralph", "Andi", "Andy", "Mike" };
final List<String> names = Arrays.asList(namesData);

final Stream<String> streamFromArray = Arrays.stream(namesData);
final Stream<String> streamFromList = names.stream();
```

Als Besonderheit können Collections eine sequenzielle sowie eine parallele Variante eines Streams liefern:

```
final Stream<String> sequentialStream = names.stream();
final Stream<String> parallelStream = names.parallelStream();
```

Für Arrays bietet die Utility-Klasse `java.util.Arrays` nur Zugriff auf eine sequenzielle Variante. Um hier eine Parallelverarbeitung zu erreichen, kann man die Methode `parallel()` auf dem Stream aufrufen, die das Umschalten auf Parallelverarbeitung vornimmt. Für das obige Array könnte man somit Folgendes schreiben:

```
final Stream<String> parallelArrayStream =
Arrays.stream(namesData).parallel();
```

Streams für vordefinierte Wertebereiche

Teilweise soll über Streams ein fixer, vordefinierter Wertebereich abgebildet und bearbeitet werden. Für diese Fälle existieren spezielle Methoden, etwa `of()`, `range()` und `chars()`:

```
final Stream<String> names = Stream.of("Tim", "Andy", "Mike");    // String
final Stream<Integer> integers = Stream.of(1, 4, 7, 7, 9, 7, 2); // Integer

final IntStream values = IntStream.range(0, 100);                 // int
final IntStream chars = "This is a test".chars();                 // int
```

Im Listing kommt neben dem generischen Interface `Stream<T>` auch das für den primitiven Datentyp `int` spezifische Interface `java.util.stream.IntStream` zum Einsatz. Die Verarbeitung erfolgt in dieser Art von Streams mit Werten primitiver Typen und nicht wie bei `Stream<Integer>` mit Werten, die gegebenenfalls zunächst per Auto-Boxing in ein `Integer`-Objekt umgewandelt werden mussten.

Hinweis: API-Ergänzungen zu Streams im JDK 8

Im vorangegangenen Beispiel sehen wir die statischen Methoden `of(T...)` und `range(int, int)` aus dem Interface `Stream<T>` bzw. `IntStream`. Wieso können denn dort statische Methoden definiert sein? Erinnern wir uns an Abschnitt 11.2.6: Im Zuge von JDK 8 und der Erweiterung um Defaultmethoden können Interfaces nun auch statische Methoden bereitstellen. Insgesamt verschwimmt der

Unterschied zwischen abstrakten Klassen und Interfaces immer mehr.

Diverse Klassen und Interfaces des JDKs bieten nun Methoden an, die Streams zurückliefern. Oben ist dies für das Interface `java.lang.CharSequence` und die Methode `chars()` gezeigt. Diese ist dort in Form einer Defaultmethode realisiert und steht damit den Spezialisierungen (`String`, `StringBuffer` und `StringBuilder`) direkt zur Verfügung.

Besonderheit 1: Streams für primitive Typen

Im vorherigen Beispiel wurde schon angedeutet, dass es mit der Klasse `IntStream` eine auf primitive Typen ausgerichtete Variante von Streams gibt. Der Grund dafür ist, dass die Verarbeitung primitiver Werte ein recht gebräuchlicher Anwendungsfall ist und Berechnungen damit schneller als auf den korrespondierenden Wrapper-Klassen ablaufen. Daher wurde das JDK um besondere Streams erweitert, die auf die Verarbeitung der primitiven Typen `int`, `long` und `double` spezialisiert sind. Das sind die Klassen `IntStream`, `LongStream` und `DoubleStream` aus dem Package `java.util.stream`. Neben spezialisierten, etwas performanteren Berechnungen und Konvertierungen untereinander kann man die Streams auch per `boxed()` bzw. `mapToObj()` wieder in einen Stream von Wrapper-Instanzen bzw. beliebigen Objekten umwandeln. Diese Aktionen sind im nachfolgenden Listing gezeigt. Dort entsteht aus einer Liste von Strings ein korrespondierender Stream, der dann in einen Stream gewandelt wird, der die Stringlänge enthält. Dieser `IntStream` wird durch Aufruf von `asLongStream()` in einen auf den Typ `long` spezialisierten `LongStream` konvertiert. Mit `boxed()` wird daraus ein `Stream<Long>` usw.:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Mike", "Stefan", "Nikolaos");
   Stream<String> values = names.stream().                    // ->
Stream<String>
                            mapToInt(String::length).         // -> IntStream
                            asLongStream().                   // -> LongStream
                            boxed().                          // -> Stream<Long>
                            mapToDouble(x -> x * .75).        // -> DoubleStream
                            mapToObj(val -> "Val: " + val); // ->
Stream<String>

   values.forEach(System.out::println);
}
```

Führen wir das Programm PRIMITIVESSTREAMEXAMPLE aus, so wird die Liste der Namen in eine Liste von Stringlängen vom Typ `int` konvertiert. Nach ein paar Umwandlungen multiplizieren wir diese Werte mit dem Faktor 0.75, um `double`-Werte zu erhalten. Schließlich nutzen wir `mapToObj()`, wodurch ein `Stream<String>` entsteht. Es kommt zu folgenden Ausgaben:

```
Val: 3.0
Val: 4.5
Val: 6.0
```

Frage: Was ist mit dem Support für die anderen primitiven Typen?

Da es neben `int`, `long` und `double` eine Menge weiterer primitiver Typen gibt, kann man sich natürlich fragen, wieso es nur für die drei genannten Typen spezielle Streams gibt. Der Grund ist der, dass man mit diesen dreien alle Anwendungsfälle adäquat abdecken kann. Die anderen primitiven Zahlentypen können mithilfe des automatisch stattfindenden Widening auf die für Streams bereitgestellten Typen abgebildet werden. Unter Widening versteht man, dass z. B. Werte vom Typ `byte` vom Compiler automatisch auf solche vom Typ `short` und diese wiederum auf `int` konvertiert werden können. Es gilt folgende »Kette«: `byte` → `short` → `int` → `long` → `float` → `double`. Basierend darauf sind die drei Spezialisierungen für `int`, `long` und `double` für alle anderen primitiven Zahlentypen ausreichend, weil es immer eine passende Widening-Konvertierung gibt.

Besonderheit 2: Unendliche Streams

Gerade im mathematischen Kontext möchte man mitunter eine unendliche Folge modellieren. Eine solche kann aufgrund ihrer Unendlichkeit und des damit verbundenen unendlichen Platzbedarfs praktisch gar nicht vollständig existieren, sondern muss Stück für Stück berechnet werden.

Für primitive Typen kann man dazu die Methode `iterate(int, IntUnaryOperator)` nutzen, der man einen Startwert und eine Implementierung des funktionalen Interface `java.util.function.IntUnaryOperator` übergibt. Dabei handelt es sich um eine auf `int`-Werte spezialisierte Form des in Abschnitt 12.2.3 beschriebenen `UnaryOperator<T>`.

Zur Generierung von Folgen, die Werte beliebiger Referenztypen enthalten, kann man die Methode `generate(Supplier<T>)` aus dem Interface `Stream<T>` einsetzen. Das Functional Interface `java.util.function.Supplier<T>` realisieren wir mit einer Methodenreferenz auf die Methode `getAndIncrement()` der Klasse `AtomicInteger` aus dem Package `java.util.concurrent.atomic`.

```
public static void main(final String[] args)
{
   final IntStream iteratingValues = IntStream.iterate(0, x -> x + 1);

   final AtomicInteger ai = new AtomicInteger(0);
   final Stream<Integer> generatedValues =
Stream.generate(ai::getAndIncrement);

   final int[] firstTen = iteratingValues.limit(10).toArray();
   final Object[] secondTen = generatedValues.limit(10).toArray();

   System.out.println(Arrays.toString(firstTen));
   System.out.println(Arrays.toString(secondTen));
   System.out.println("Element type: " +
```

```
secondTen[0].getClass().getTypeName());
}
```

Starten wir das obige Programm INFINITESTREAMSEXAMPLE, so werden folgende Wertefolgen produziert:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
Element type: java.lang.Integer
```

Anhand des Listings und der Ausgaben lassen sich die zwei Dinge lernen: Erstens müssen unendliche Streams mithilfe eines Aufrufs von `limit(long)` beschränkt werden, wenn man diese sinnvoll verarbeiten möchte – das wird später noch etwas genauer beschrieben. Zweitens arbeitet der `IntStream` direkt auf `int`-Werten, wodurch im Gegensatz zu einem `Stream<Integer>` kein Auto-Boxing benötigt wird. Dass dem so ist, sieht man daran, dass der primitive Stream einem `int[]` zugewiesen werden kann. Der `Stream<Integer>` wird bei der Konvertierung in ein Array mit `toArray()` in ein `Object[]` umgewandelt, das aber `Integer`-Instanzen enthält, wie es die letzte Zeile der Ausgabe bestätigt.

12.3.2 Intermediate und Terminal Operations im Überblick

Nachdem wir kurz gesehen haben, wie wir ein Objekt vom Typ `Stream<T>` entweder aus einer Collection oder aber einer anderen Quelle erhalten können, schauen wir nun überblicksartig darauf, was man mit Streams anfangen kann, bevor wir das Ganze in den nachfolgenden Abschnitten vertiefen.

Gebräuchliche Anwendungsfälle für den Einsatz von Streams sind etwa das Filtern, das Transformieren und das Sortieren von Werten. Dazu nutzt man sogenannte ***Intermediate Operations***. Diese beschreiben ***Verarbeitungsschritte***, die sich einfach hintereinander schalten lassen. Das Besondere daran ist, dass zunächst keine Berechnungen erfolgen, sondern lediglich die Abläufe beschrieben werden. Dabei unterscheidet man zudem zwischen ***zustandslosen*** und ***zustandsbehafteten*** Varianten. Filtern ist eine zustandslose Aktion. Damit ist gemeint, dass für jedes Element des Streams unabhängig von den anderen diese Aktion ausführbar ist. Dadurch lassen sich zustandslose Operationen auch hervorragend parallelisieren. Sortieren ist dagegen eine zustandsbehaftete Aktion, die die Kenntnis der anderen Elemente im Stream (oder zumindest eines Teils davon) erfordert. Da Streams keine (oder für zustandsbehaftete Operationen meistens nur eine Untermenge der) Daten zwischenspeichern, verbrauchen Streams im Gegensatz zu Collections in der Regel deutlich weniger Speicher. Somit hat die Konstruktion von Streams oftmals wenig Einfluss auf Speicherbedarf und Ausführungszeit.

Irgendwann sollen die ***Bearbeitungsergebnisse*** zusammengefasst, auf der Konsole ausgegeben oder anderweitig verarbeitet werden. Dazu dienen ***Terminal Operations***. Diese lösen erst tatsächlich die Ausführung der durch die Intermediate Operations

beschriebenen Verarbeitungsschritte aus.

Abschließend möchte ich mit den ***Short-circuiting Operations*** ein weitere Variante vorstellen. Short-circuiting Operations zeichnen sich dadurch aus, dass sie ihre Berechnungen nicht immer vollständig für alle Elemente eines Streams ausführen müssen. Beispiele sind die Suche nach einem beliebigen Treffer oder danach, ob es überhaupt ein Element gibt, das einer gewünschten Bedingung genügt. Insbesondere bei Parallelverarbeitung können Short-circuiting Operations die Berechnung deutlich beschleunigen, da nach einem Treffer keine weiteren Berechnungen gestartet werden müssen. Short-circuiting Operations existieren sowohl für Intermediate Operations als auch und vor allem für Terminal Operations. In den nachfolgenden Auflistungen sind die Namen von Short-circuiting Operations jeweils kursiv dargestellt.

Intermediate Operations

Bei den zustandslosen Intermediate Operations sind folgende wichtig:

- `filter()` – Filtert alle Elemente aus dem Stream heraus, die nicht dem übergebenen `Predicate<T>` genügen.
- `map()` – Transformiert Elemente mithilfe einer `Function<T, R>` vom Typ `T` auf solche mit dem Typ `R`. Im Speziellen können die Typen auch gleich sein.
- `flatMap()` – Bildet verschachtelte Streams als einen flachen Stream ab.
- `peek()` – Führt eine Aktion für jedes Element des Streams aus. Dies kann für Debuggingzwecke sehr nützlich sein.

Darüber hinaus sollte man folgende zustandsbehaftete Intermediate Operations kennen:

- `distinct()` – Entfernt alle gemäß der Methode `equals(Object)` als Duplikate erkannte Elemente aus einem Stream.
- `sorted()` – Sortiert die Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- *`limit()`* – Begrenzt die maximale Anzahl der Elemente eines Streams auf einen bestimmten Wert. Dies ist eine Short-circuiting Operation.
- `skip()` – Überspringt die ersten *n* Elemente eines Streams.

Terminal Operations

Neben den umfangreichen Intermediate Operations wird eine noch imposantere Zahl an Terminal Operations geboten, unter anderem folgende:

- `forEach()` – Führt eine Aktion für jedes Element des Streams aus.
- `toArray()` – Überträgt die Elemente aus dem Stream in ein Array.

- `collect()` – Überträgt die Elemente aus dem Stream in eine Collection.
- `reduce()` – Verbindet die Elemente eines Streams. Ein Beispiel ist die kommaseparierte Konkatenation von Strings. Alternativ kann man aber auch Summationen, Multiplikationen usw. ausführen, um einen Ergebniswert zu berechnen.
- `min()` – Ermittelt das Minimum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `max()` – Ermittelt das Maximum der Elemente eines Streams gemäß einem Sortierkriterium basierend auf einem `Comparator<T>`.
- `count()` – Zählt die Anzahl an Elementen in einem Stream.
- *`anyMatch()`* – Prüft, ob es mindestens ein Element gibt, das die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.
- *`allMatch()`* – Prüft, ob alle Elemente die Bedingung eines `Predicate<T>` erfüllen. Dies ist eine Short-circuiting Operation, die allerdings abbricht, wenn sie das erste Gegenbeispiel gefunden hat.
- *`noneMatch()`* – Prüft, ob keines der Elemente die Bedingung eines `Predicate<T>` erfüllt. Dies ist eine Short-circuiting Operation.
- *`findFirst()`* – Liefert das erste Element des Streams, falls es ein solches gibt. Dies ist eine Short-circuiting Operation.
- *`findAny()`* – Liefert ein beliebiges Element, falls es ein solches gibt. Dies ist eine Short-circuiting Operation und kann manchmal günstiger sein als `findFirst()`, wenn es wirklich nur darum geht, einen beliebigen Treffer zu erhalten.

12.3.3 Zustandslose Intermediate Operations

In diesem Abschnitt betrachten wir verschiedene zustandslose Intermediate Operations. Dabei beginnen wir mit der Filterung von Werten und kommen dann zur Extraktion bzw. Abbildung. Im Anschluss lernen wir eine spezielle Mapping-Funktionalität kennen. Danach sehen Sie wir uns das Filtern und die Extraktion im praktischen Einsatz an. Abschließend schauen wir auf die Inspektion von Verarbeitungsschritten. Dies erleichtert es beim Auftreten von Problemen, mögliche Fehlersituationen besser nachvollziehen zu können.

Die Methode `filter()` – Filterung

Das Filtern ist eine gebräuchliche Funktionalität, die bisher leider nicht durch das JDK bereitgestellt wurde. Mit JDK 8 ist dies glücklicherweise sehr einfach möglich.

Betrachten wir dazu ein Beispiel einer Liste von `Person`-Objekten. Aus dieser

wollen wir mithilfe von `filter(Predicate<Person>)` diejenigen ermitteln, die erwachsen sind, indem wir die Methodenreferenz `Person::isAdult` wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Micha", 43));
    persons.add(new Person("Barbara", 40));
    persons.add(new Person("Yannis", 5));

    // final Predicate<Person> isAdult = person -> person.getAge() >= 18;
    final Stream<Person> adults = persons.stream().filter(Person::isAdult);

    adults.forEach(System.out::println);
}
```

Die Bedingung `isAdult` kann man – wie im Kommentar angedeutet – als Lambda schreiben oder man verwendet eine besser lesbare Methodenreferenz, die auf folgende Methode `isAdult()` in der Klasse `Person` verweist:

```
public class Person
{
    private int age;

    // …

    public boolean isAdult()
    {
        return age >= 18;
    }
}
```

Mehrstufige Filterung

In der Praxis soll oftmals eine mehrstufige Filterung nach verschiedenen Kriterien erfolgen. Mit der Pipeline- oder Fließbandanalogie im Hinterkopf kann man dazu mehrere Filter hintereinander schalten, wie dies folgendes Listing für drei Filterbedingungen zeigt:

```
final Stream<Person> allAdultMikes = persons.stream().
                                filter(Person::isAdult).
                                filter(person -> person.getName().equals("Mike").
                                filter(mike -> mike.livesIn("Zürich"));
```

Auf diese Weise filtern wir zunächst alle Erwachsenen und dann all diejenigen, deren Name »Mike« ist. Aus dieser Ergebnismenge werden wiederum diejenigen herausgefiltert, die wohnhaft in Zürich sind.

Meinung: Namensgebung von Lambda-Parametern

Wie Sie vielleicht bemerkt haben, verwende ich für die Parameter in Lambdas bevorzugt sprechende Namen oder aber Standards wie `it`. Meiner Meinung nach gilt auch hier, dass man so lesbar wie möglich programmieren sollte. Nur weil man funktional programmiert, heißt das nicht, dass man wieder auf

Namensverkümmerungen wie `a`, `p`, `x` zurückgreifen muss. Natürlich gibt es auch Fälle, in denen Kürzel mit einem Buchstaben ihren Wert haben. Das gilt immer dann, wenn im Lambda eine beliebige Berechnung erfolgt, etwa `x -> x + 1`. Dabei trägt der Parameter keine oder nur wenig semantische Bedeutung – meistens, weil eine »echte« mathematische Funktion beschrieben wird.

Die Methode `map()` – Mapping von Daten, Extraktion von Werten

Neben der Filterung ist die Konvertierung oder Extraktion von Werten eine typische Intermediate Operation. Hierbei soll eine Menge von Eingabedaten in ein anderes Format überführt oder abgebildet werden. So könnte etwa aus einer Liste von Personen jeweils das Attribut Name, Vorname oder Alter extrahiert werden. Dabei findet eine Abbildung oder ein Mapping von einem Typ auf einen anderen statt: Im Beispiel wird aus dem Typ `Person` ein Attribut herausgelesen und auf denjenigen Typ des gewünschten Attributs, z. B. `String`, abgebildet. Dazu kann man Spezialisierungen des Interface `Function<T,R>` nutzen und dort die Methode `apply(T)` entsprechend implementieren. Das haben wir bereits im Zusammenhang mit dem Interface `UnaryOperator<T>` in Abschnitt 12.2.3 kurz besprochen. Hier wird die Definition des Interface `Function<T,R>` zum Einstieg wiederholt:

```
interface Function<T,R>
{
    R apply(T t);
}
```

Nehmen wir an, es wäre der Name aus einem `Person`-Objekt zu extrahieren. Dies implementieren wir mithilfe eines Lambdas oder mit einer Methodenreferenz wie folgt:

```
// Lambda
final Function<Person, String> nameExtractor_V1 = person -> person.getName();

// Alternativ mit Methodenreferenz
final Function<Person, String> nameExtractor_V2 = Person::getName;
```

Extraktion am Beispiel

Wir haben nun genug Vorwissen gesammelt und machen uns damit daran, die Extraktion des Namens bzw. des Alters für eine Liste von Personen folgendermaßen auszuprogrammieren:

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Barbara", 40));
    persons.add(new Person("Yannis", 5));

    // Mapping auf Name mit Lambda
    final Stream<Person> adults = persons.stream().filter(Person::isAdult);
    final Stream<String> namesStream = adults.map(person -> person.getName());
```

```
    // Mapping auf Alter mit Methodenreferenz
    final Stream<Integer> agesStream = persons.stream().map(Person::getAge).
                                                filter(age -> age >= 18);

    namesStream.forEach(System.out::println);
    agesStream.forEach(System.out::println);
}
```

Führen wir das obige Programm ATTRIBUTEEXTRACTIONEXAMPLE aus, so erhalten wir folgende Ausgaben:

```
Barbara
40
```

Die Methode `flatMap()` – spezielle Arten von Mappings

Während sowohl die `filter()`- als auch die `map()`-Methode recht intuitiv zu benutzen sind, wirkt die `flatMap()`-Methode zunächst nicht ganz so eingängig. Rekapitulieren wir kurz: `filter()` und `map()` werden auf einem Stream angewendet und liefern einen neuen Stream zurück – bei `map()` gegebenenfalls einen Stream eines anderen Typs, etwa wenn man von `Person`-Objekten auf deren Attribut `name` abbildet.

Bei derartigen Abbildungen gibt es einen Spezialfall zu beachten, nämlich den verschachtelter Streams. Das lässt sich am besten an einem Beispiel verdeutlichen. Gegeben sei eine Menge von Sätzen.[3] Für diese soll die Häufigkeit der Vorkommen einzelner Wörter gezählt werden. Bevor wir uns später an die Implementierung dieser Funktionalität machen, wollen wir uns zunächst dem Stream-im-Stream-Problem widmen, das aus der Beschreibung möglicherweise noch nicht ganz deutlich geworden ist. Schauen wir folgende Ausgangslage an:

```
final List<String> sentences = Arrays.asList( "This is the first line.",
                                               "The second line of this text.",
                                               "Third line contains some
text.",
                                               "Last line and goodbye:",
                                               "End of text!");

final Stream<String> asStream = sentences.stream();
```

Um aus diesen Sätzen einzelne Wörter zu extrahieren, könnte man die Methode `String.split(String)` nutzen, die ein `String[]` liefert. Wenn wir diese Rückgabe mithilfe von `map()` verarbeiten wollen, so erfordert der Kontrakt für `map()`, dass ein Stream zurückgeliefert wird. Dadurch erhält man aber einen Stream von Streams:

```
Stream<Stream<String>> words = asStream.map(line -> Stream.of(line.split(" ")));
```

Das Ergebnis wäre dann ein verschachtelter Stream, in unserem Beispiel etwa folgendermaßen (nachfolgend symbolisieren die Zeichen < > die Begrenzer eines Streams):

```
≪This, is, the, first, line.>, <The, second, line, … <End, of, text!≫
```

Diese Form erschwert es aber, über die einzelnen Werte zu iterieren. Zur Auswertung wünscht man sich, dass die geschachtelten Streams »flach geklopft« werden und als Ergebnis ein `Stream<String>` entsteht. Das ist recht einfach möglich, indem man die Methode `flatMap()` folgendermaßen nutzt:

```
Stream<String> words = asStream.flatMap(line -> Stream.of(line.split(" ")));
```

Beispiel: Die Intermediate Operations in Aktion

Wir haben nun drei verschiedene Intermediate Operations kennengelernt. Um das Wissen darüber weiter zu vertiefen, wollen wir diese in einem Beispiel einsetzen. Wir führen hier die Auswertung einer Menge von Sätzen, um die Häufigkeiten von Wörter festzustellen, fort. Damit das Beispiel etwas reizvoller und realitätsnäher wird, bestehen weitere Anforderungen. Zunächst einmal sollen die Daten aus einer Datei eingelesen werden. Zudem sollen alle Wörter mit drei oder weniger Buchstaben nicht gezählt werden. Dazu formulieren wir ein `Predicate<String>` namens `isShortWord`. Darüber hinaus sollen verschiedene Füllwörter bei der Zählung der Häufigkeiten nicht betrachtet werden. Die Wörter »this«, »these« und »them« wollen wir ignorieren. Auch dazu formulieren wir ein korrespondierendes `Predicate<String>` namens `isIgnorableWord`. Durch Negation erhalten wir dann das Prädikat `isSignificantWord`. Die Filterung können wir mithilfe der gerade genannten Prädikate formulieren:

```
public static void main(final String[] args) throws IOException
{
    final Path exampleFile = Paths.get("src/main/resources/" +
                                       "jdk8/streams/Example.txt");

    // Datei einlesen neu in JDK 8: Siehe dazu Kapitel 6
    final List<String> contents = Files.readAllLines(exampleFile);

    // Daraus einen Stream von Worten machen
    final Stream<String> words = contents.stream().
                                 flatMap(line -> Stream.of(line.split(" ")));

    // Prädikate für kurze Wörter
    final Predicate<String> isShortWord = word -> word.length() <= 3;

    final Predicate<String> notIsShortWord = isShortWord.negate();

    // Prädikate für spezielle und zu ignorierende Wörter
    final List<String> ignoreableWords = Arrays.asList("this", "these",
"them");
    final Predicate<String> isIgnorableWord = word ->
    {
        return ignoreableWords.contains(word.toLowerCase());
    };
    final Predicate<String> isSignificantWord = isIgnorableWord.negate();

    // Filterung basierend auf den Prädikaten
    final Stream<String> filteredContents = words.map(String::trim).
                                            filter(notIsShortWord).
                                            filter(isSignificantWord);
```

```
    filteredContents.forEach(it -> System.out.print(it + ", "));
}
```

Das Programm FLATMAPEXAMPLE gibt die gefilterten Ergebnisse wie folgt aus:

```
first, line., second, line, text., Third, line, contains, some, text.,
Last, line, goodbye:, text!,
```

An der Ausgabe sind zwei Schwächen zu erkennen: Zum einen sind noch Satzzeichen wie Punkt, Doppelpunkt und Ausrufezeichen Bestandteil der Wörter und zum anderen sind die Wörter nicht sortiert. Letzteres wird im Rahmen der zustandsbehafteten Intermediate Operations genauer besprochen, hier verwenden wir lediglich die intuitiv verständliche Methode `sorted()`, um die Ausgabe zu sortieren. Darüber hinaus nutzen wir ein Mapping, um die Satzendezeichen zu entfernen. Statt des `forEach()`-Aufrufs ergänzen wir folgende Zeilen:

```
final Function<String, String> removePunctationMarks = str ->
{
    if (str.endsWith(".") || str.endsWith(":") || str.endsWith("!"))
    {
        return str.substring(0, str.length()-1);
    }
    return str;
};

final Stream<String> mapped = filteredContents.map(removePunctationMarks);
final Stream<String> sorted = mapped.sorted(String.CASE_INSENSITIVE_ORDER);

sorted.forEach(it -> System.out.print(it + ", "));
```

Nach dieser Korrektur liefert das Programm FLATMAPEXAMPLE2 folgende Ausgabe:

```
contains, first, goodbye, Last, line, line, line, line, second, some,
text, text, text, Third,
```

Wir sind schon fast am Ziel der Gruppierung und Häufigkeitsbestimmung. Weiterhin könnten wird das abschließende Komma bemängeln. Diese Thematik werden wir bei der Besprechung der Terminal Operations wieder aufgreifen. Vorher betrachten wir eine Besonderheit bei Intermediate Operations, nämlich die Inspektion von Verarbeitungsschritten.

Die Methode `peek()` – Inspektion von Verarbeitungsschritten

Für den Fall, dass Intermediate Operations komplexer werden, möchte man sich eventuell auch einmal Zwischenergebnisse ausgeben lassen und danach die Verarbeitung fortsetzen. Auf ein Problem stößt man, wenn man einen bereits mit `forEach(Consumer<? super T>)` ausgegebenen oder irgendwie mit einer Terminal Operation verarbeiteten Stream weiter bearbeiten möchte, wie dies in folgendem Beispiel gezeigt ist:

```
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe, um die Filterung zu überprüfen
adults.forEach(System.out::println);
```

```
// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> mikes = adults.filter( person ->
                                            person.getName().equals( "Mike" ));
```

Führt man diese Schritte aus, so kommt es aber anstelle einer Weiterverarbeitung zu einer `IllegalStateException` mit dem Hinweis `stream has already been operated upon or closed`. Daran erkennt man die bereits erwähnte Eigenschaft von Streams, die Daten nur einmal bereitstellen zu können.[4]

Weil die Inspektion von Zwischenzuständen aber eine sehr wünschenswerte Funktionalität ist, stellt das Stream-API hierfür eine Möglichkeit bereit. Mithilfe der Intermediate Operation in Form der Methode `peek(Consumer<? super T>)` kann man beliebige Aktionen, etwa Konsolenausgaben, durchführen. Dies gleicht der Verarbeitung mit `forEach(Consumer<? super T>)`. Im Gegensatz dazu erhält man aber beim Aufruf von `peek(Consumer<? super T>)` wiederum einen Stream, mit dem man weiter arbeiten kann. Obiges Beispiel kann man zur Realisierung einer Inspektion folgendermaßen abändern:

```
final Stream<Person> adults = persons.stream().filter(Person::isAdult);

// Ausgabe mit peek(), um die Filterung zu überprüfen
final Stream<Person> adultsPeek = adults.peek(System.out::println);

// Weitere Filterung auf dem Stream vornehmen
final Stream<Person> mikes = adultsPeek.filter(person ->
                                               person.getName().equals("Mike"));
```

Nachfolgend zeige ich eine Variante, die nach jedem Verarbeitungsschritt der Pipeline eine Ausgabe vornimmt – im Listing jeweils fett markiert:

```
public static void main(final String[] args)
{
   final List<Person> persons = createDemoData();

   final Stream<Person> stream = persons.stream();
   final Stream<String> allMikes = stream.peek(System.out::println).
                                          filter(Person::isAdult).
                                          peek(System.out::println).
                                          map(Person::getName).

                                          peek(System.out::println).
                                          filter(name ->
name.startsWith("Mi")).
                                          peek(System.out::println).
                                          map(String::toUpperCase);
   // Löst die Verarbeitung aus
   System.out.println("Protokollierung jedes Schritts – Filter 'Mi':");
   allMikes.forEach(System.out::println);
}

private static List<Person> createDemoData()
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Michael", 43));
    persons.add(new Person("Max", 5));
    persons.add(new Person("Moritz", 7));
    persons.add(new Person("Merten", 38));
```

```
    persons.add(new Person("Micha", 42));
    return persons;
}
```

Nach dem Start des Programms StreamPeekExample werden die Filtervorgänge auf der Konsole protokolliert:

```
Protokollierung jedes Schritts – Filter 'Mi':
Person [name = Michael / age = 43]
Person [name = Michael / age = 43]
Michael
Michael
MICHAEL
Person [name = Max / age = 5]
Person [name = Moritz / age = 7]
Person [name = Merten / age = 38]
Person [name = Merten / age = 38]
Merten
Person [name = Micha / age = 42]
Person [name = Micha / age = 42]
Micha
Micha
MICHA
```

Anhand der Ausgabe erkennt man, dass die Verarbeitung nicht je Verarbeitungsschritt komplett für den Stream auf einmal, sondern elementweise geschieht: Die Daten werden Element für Element bearbeitet und durchlaufen jeweils einzeln die Stufen der Verarbeitungskette. Diese Art der Ausführung bildet die Grundlage für die Parallelisierbarkeit von Aktionen. Darüber hinaus wird der bereits erwähnte fundamentale Unterschied von Streams zu sonstigen Verarbeitungen klar, nämlich, dass Intermediate Operations wirklich erst dann abgearbeitet werden, wenn eine Terminal Operation eine Berechnung auslöst. Das können Sie prüfen, wenn Sie die letzte Zeile mit `forEach(Consumer<? super T>)` auskommentieren. Dann werden Sie keine Ausgaben sehen.

Im vorherigen Beispiel habe ich die elementweise Verarbeitung dargestellt. Diese bildet die Grundlage für eine mögliche Parallelisierbarkeit. Anhand von Grafiken möchte ich die Abläufe verdeutlichen. Dazu kann man sich zunächst vorstellen, dass die Elemente eines Streams hintereinander aufgereiht sind und dann die Aktionen ausgeführt werden. Schematisch ist dies in Abbildung 12-1 dargestellt: Die Elemente werden nacheinander durch die Pipeline geschickt und jede Pipeline-Stufe wird protokolliert:

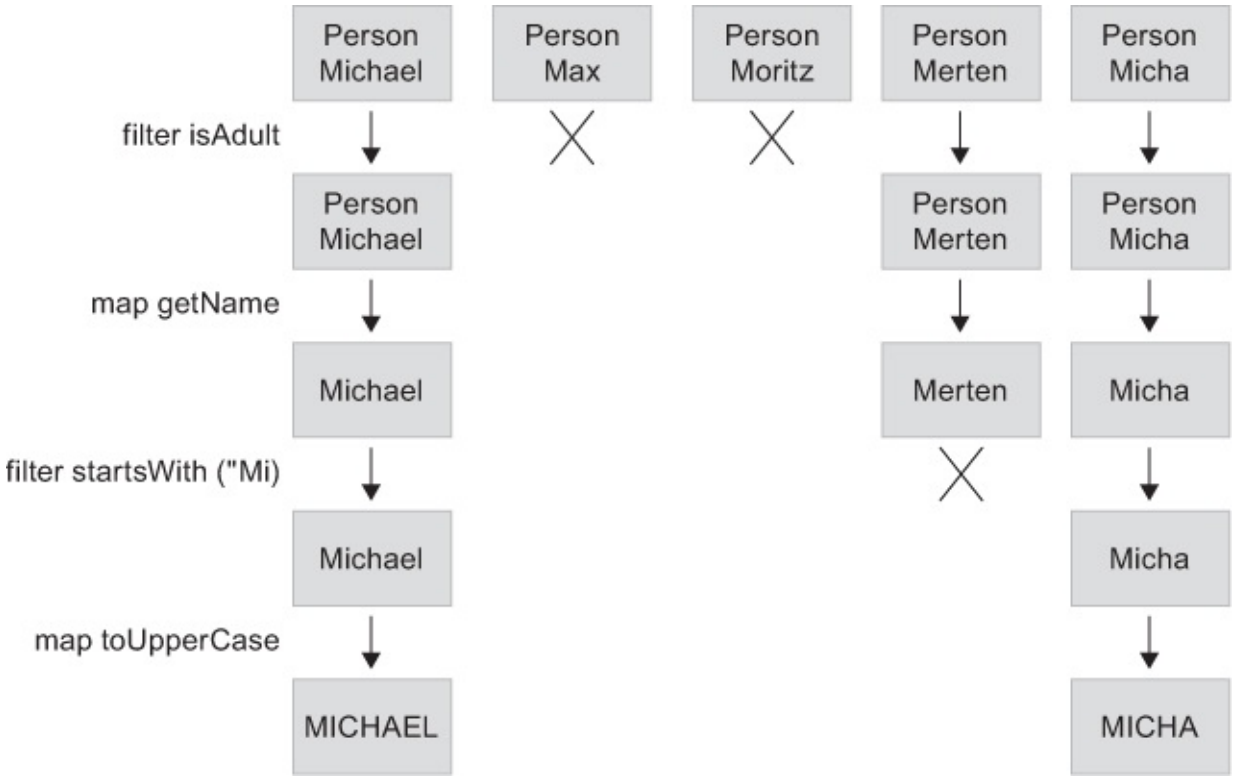


Abbildung 12-1 *Pipeline von Intermediate Operations*

Man erkennt sehr schön die elementweise Pipeline-Verarbeitung. Zur Parallelisierung kann man einfach einige Elemente zu Gruppen zusammenfassen, die man jeweils parallel durch eigene Threads verarbeiten kann. Voneinander unabhängige, zustandslose Operationen kann man maximal parallelisieren. Auch für die im Anschluss betrachteten zustandsbehafteten Operationen kann man eine recht effiziente Parallelisierung erreichen. In Abbildung 12-2 ist dies für das Summieren bzw. Sortieren gezeigt. Hier kann man jeweils Teilbereiche für sich verarbeiten und diese danach vereinigen.

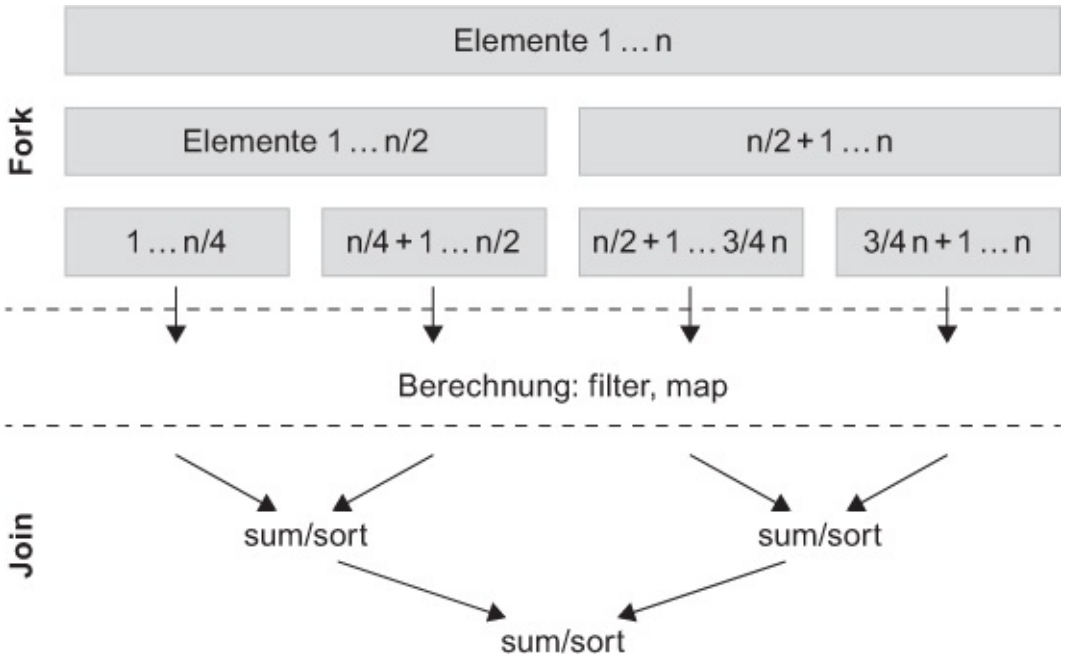


Abbildung 12-2 *Schematische Darstellung der parallelen Sortierung*

12.3.4 Zustandsbehaftete Intermediate Operations

Mit der Vorwegnahme der Sortieroperation leiten wir nun zum Thema zustandsbehafteter Intermediate Operations über. Wir lernen davon verschiedene Varianten kennen: Das Sortieren und Herausfiltern doppelter Einträge zeige ich zuerst. Danach gehe ich auf das Beschränken der Ausgabe auf eine gewisse Anzahl von Elementen ein: Dabei kann sowohl der Startwert als auch die Anzahl der gewünschten, im Stream zu verbleibenden Elemente festgelegt werden.

Die Methoden `distinct()` und `sorted()` – Ausgaben sortieren, Duplikation entfernen

Das Sortieren und das Herausfiltern doppelter Einträge ist im folgenden Listing mithilfe von `sorted()` und `distinct()` realisiert. Dabei betrachten wir zunächst die Ausführung jeder Methode für sich und danach in Kombination:

```
public static void main(final String[] args)
{
    final Stream<Integer> distinct = createIntStream().distinct();
    final Stream<Integer> sorted= createIntStream().sorted();
    final Stream<Integer> sortedAndDistinct = createIntStream().sorted().
                                                                distinct();
    printResult("distinct:          ", distinct);
    printResult("sorted:            ", sorted);
    printResult("sortedAndDistinct: ", sortedAndDistinct);
}

private static Stream<Integer> createIntStream()
{
    return Stream.of(7, 1, 4, 3, 7, 2, 6, 5, 7, 9, 8);
}

private static void printResult(final String hint,
                                final Stream<Integer> stream)
{
    final List<Integer> result = stream.collect(Collectors.toList());
    System.out.println(hint +  result);
}
```

Führt man das Programm SORTEDANDDISTINCTEXAMPLE aus, so werden doppelte Elemente entfernt und die Zahlen sortiert. Man erhält die erwartete Ausgabe:

```
distinct:          [7, 1, 4, 3, 2, 6, 5, 9, 8]
sorted:            [1, 2, 3, 4, 5, 6, 7, 7, 7, 8, 9]
sortedAndDistinct: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Wenn Sie im Listing ganz genau hingeschaut haben, dann ist Ihnen vielleicht der Aufruf von `collect(Collectors.toList())` aufgefallen. Dabei handelt es sich wie bei `forEach(Consumer<? super T>)` ebenfalls um eine Terminal Operation. Ein Aufruf von `collect(Collectors.toList())` überträgt die Daten aus einem Stream in eine Liste. Zum Thema Terminal Operations erfahren Sie gleich mehr.

Die Methoden `limit()` und `skip()` – Ausgaben beschränken

Die bereits gezeigte Filterung mit `filter(Predicate<? super T>)` erlaubt es, den Datenbestand einzuschränken. Eine Variante davon ist es, die Ergebnismenge auf *n* Elemente zu beschränken. Dazu kann man einen Aufruf von `limit(long)` nutzen. In Kombination mit `skip(long)` zum Überspringen von *n* Datensätzen kann man sogenanntes ***Paging*** realisieren – eine Aufbereitung von *n* Ergebnissen auf einer Seite, wie man dies etwa von der Präsentation von Suchergebnissen im Internet kennt. Man kann die Ausgabe folgendermaßen auf 25 Sucheinträge ab dem 75. Eintrag beschränken:

```
searchResults.skip(75).limit(25);
```

Es gibt aber noch weitere Anwendungsfälle für `limit(long)` und `skip(long)`, nämlich im Zusammenhang mit unendlichen Streams, etwa für eine Folge von `int`-Werten:

```
final IntStream iteratingValues = IntStream.iterate(0, x -> x + 1);

iteratingValues.skip(50).limit(12); // => 50,51,52,53,54,55,56,57,58,59,60,61
```

Im Beispiel sehen wir die Begrenzung des unendlichen Streams auf bestimmte Datensätze – hier die 12 Einträge, die auf die ersten 50 Einträge folgen.

12.3.5 Terminal Operations

Bisher haben wir mithilfe von Streams verschiedene Berechnungen ausgeführt und dabei häufig die Terminal Operation `forEach(Consumer<? super T>)` genutzt, um Konsolenausgaben zu produzieren. Betrachten wir Terminal Operations nun ein wenig allgemeiner. Erinnern wir uns zunächst nochmals daran, dass diese zur Abarbeitung der Pipeline führen und dadurch ein Ergebnis produzieren.

Die Methode `forEach()` – Verarbeitung oder Konsolenausgaben

Nachfolgend ist zunächst der Vollständigkeit halber die Aufbereitung von Konsolenausgaben mit `forEach(Consumer<? super T>)` gezeigt:

```
streamFromArray.forEach(System.out::println);
streamFromValues.sorted().distinct().forEach(System.out::println);
```

Die Methode `toArray()` – Streams in Arrays übertragen

Statt Berechnungsergebnisse direkt auf der Konsole auszugeben, ist es meistens sinnvoller, diese etwa in ein Array oder eine Collection zu überführen. Dazu

existieren die beiden Methoden `toArray()` und `toCollection()`.

Man könnte sich vorstellen, eine ganz bestimmte Menge von Aktionen ausführen zu müssen, etwa die Aktion »Erzeuge sieben Zufallszahlen von 0 bis 100« und die Berechnungsergebnisse müssten in einem Array bereitgestellt werden. Dazu kann man die Methode `toArray()` aufrufen. Diese liefert normalerweise ein `Object[]` – für Streams primitiver Typen werden aber Arrays von diesen Typen erzeugt, hier ein `int[]`:

```
public static void main(final String[] args)
{
    // Zufallszahlen von 0 bis 100
    final Random random = new Random();
    final Supplier<Float> randomSupplier = () -> random.nextFloat() * 100;

    final Object[] randomNumbers = Stream.generate(randomSupplier).
                                          limit(7).toArray();
    System.out.println(Arrays.toString(randomNumbers));
    System.out.println("Element type: " + randomNumbers[0].getClass());

    final int[] intRandoms = Stream.generate(randomSupplier).
                             limit(7).mapToInt(val ->
val.intValue()).toArray();
    System.out.println(Arrays.toString(intRandoms));
}
```

Das Programm StreamToArrayExample produziert in etwa folgende Ausgaben:

```
[82.59304, 92.31408, 33.92508, 44.195183, 59.675797, 16.191816, 99.11495]
Element type: class java.lang.Float
[91, 12, 44, 48, 44, 69, 58]
```

Die Methode `collect()` – Streams in Collections übertragen

Mithilfe von `forEach(Consumer<? super T>)` kann man bekanntermaßen über die Berechnungsergebnisse iterieren, z. B. um diese auszugeben. Viele Anwendungsfälle erfordern es, die Daten aus dem `Stream<E>` in einer `Collection<E>` zu speichern. Mithilfe von `java.util.stream.Collector`-Instanzen kann man die Daten auslesen und z. B. in eine Liste übertragen. Ein solcher Collector besteht aus vier Komponenten, einem `Supplier<A>` zum Bereitstellen eines Startwerts, einem `BiConsumer<A, T>` für Berechnungen, einem `BinaryOperator<A>` zum Zusammenführen von Zwischenergebnissen und einer `Function<A, R>` zum Berechnen des Ergebnisses. Wollte man dies selbst implementieren, würde das ziemlich kompliziert und fehlerträchtig. Praktischerweise existieren in der Utility-Klasse `java.util.stream.Collectors` schon diverse vordefinierte Methoden, die passende `Collector`-Instanzen zurückliefern, die die Komplexität stark reduzieren, wie dies nachfolgend gezeigt ist:

```
final List<Integer> ages = agesStream.collect(Collectors.toList());
final List<String> names = namesStream.collect(Collectors.
                                               toCollection(ArrayList::new));
```

Im Listing sehen wir den für viele Anwendungsfälle praktischen Aufruf von

`toList()`. Benötigt man mehr Kontrolle über den Typ der Ergebnisdatenstruktur, so kann man die Methode `toCollection()` aufrufen, der man die Referenz auf den Konstruktor der gewünschten Collection übergibt, wie dies im zweiten Aufruf dargestellt ist.

Die Methoden `count()`, `sum()`, `min()`, `max()` und `average()`

Neben Konsolenausgaben oder der Übertragung in eine Collection stellen Berechnungen auf den Daten gebräuchliche Terminal Operations dar. Für Zahlenwerte sind etwa Berechnungen wie Minimum, Maximum, Summe oder Durchschnitt typisch. Aber auch die Berechnung der Anzahl an Elementen im Stream gehört dazu.

Nachfolgend ermitteln wir mit `count()` zunächst, wie viele Personen mit dem Buchstaben »T« im Namen starten. Dann berechnen wir mithilfe der Methode `sum()` die Summe aus allen Altersangaben dieser Personen. Das Ergebnis ist vom Typ `int`. Ebenso können wir durch Aufruf von `min()` bzw. `max()` das Minimum bzw. Maximum, also hier das Alter der jüngsten bzw. ältesten Person ermitteln (im Listing nicht gezeigt). Wenn wir das Durchschnittsalter berechnen wollen, können wir dazu die Methode `average()` nutzen. Dabei gibt es zwei Dinge zu lernen. Zum einen ist das Ergebnis keine Ganzzahl mehr und zum anderen existiert möglicherweise kein Durchschnitt, nämlich dann, wenn die Quelle für den Stream leer ist. Um diesen Sachverhalt ausdrücken zu können, verwenden wir ganz nebenbei noch die Klasse `OptionalDouble` – ähnliche Klassen gibt es auch für andere primitive Typen und mit `Optional<T>` für Referenztypen. Ein solches `Optional<T>` bzw. die Varianten für primitive Typen modellieren optionale Werte, die möglicherweise keinen Wert repräsentieren. Details dazu beschreibt Abschnitt 15.2.

Zur Demonstration einiger Berechnungen entsteht folgendes Programm

```
public static void main(final String[] args)
{
    final List<Person> persons = new ArrayList<>();
    persons.add(new Person("Ten", 10));
    persons.add(new Person("Twenty", 20));
    persons.add(new Person("Thirty", 30));
    persons.add(new Person("Forty", 40));

    // Anzahl an Personen, deren Name mit 'T' startet
    final int count = persons.stream().filter(person -> person.getName().
                                        startsWith("T")).count();
    System.out.println("count:  " + count);

    // Summe berechnen
    final int sum = persons.stream().filter(person -> person.getName().
                                    startsWith("T")).
                                    mapToInt(Person::getAge).sum();
    System.out.println("sum:  " + sum);
```

```
    // Durchschnitt berechnen
    final OptionalDouble avg = persons.stream().
                                       filter(person -> person.getName().
                                       contains("X")).
                                       mapToInt(Person::getAge).average();
    System.out.println("avg: " + avg);
}
```

Starten wir das Programm CALCULATIONEXAMPLE, so werden mit `count()` drei Personen gezählt. Die Methode `sum()` liefert für die Altersangaben die Summe sechzig. Die zuletzt gezeigte Berechnung des Durchschnitts der Altersangaben (`average()`) für Personen mit einem »X« im Namen kann allerdings keinen Wert liefern, da keine Einträge dafür existieren. Dies wird durch `OptionalDouble.empty()` modelliert. Somit kommt es zu folgenden Ausgaben:

```
count: 3
sum: 60
avg: OptionalDouble.empty
```

Hintergrundwissen: Verarbeitungsmethoden in Wrapper-Klassen

Damit sich verschiedene Stream-Operationen leichter formulieren lassen, wurden die Wrapper-Klassen `Integer`, `Long` und `Double` um verschiedene Methoden erweitert, besonders hervorzuheben sind hier `min()`, `max()` und `sum()`.

Die Methoden `allMatch()`, `anyMatch()`, `noneMatch()`

Recht gebräuchliche Anwendungsfälle beim Verarbeiten von Daten sind Prüfungen, ob alle, einige Elemente oder aber kein Element eine gewisse Bedingung erfüllen, beispielsweise ob alle Personen älter als 18 Jahre sind, Namen mit einem gewissen Buchstaben starten usw. Zur Umsetzung bietet das `Stream<T>`-Interface die drei Methoden

- `allMatch(Predicate<? super T>)`,
- `anyMatch(Predicate<? super T>)` und
- `noneMatch(Predicate<? super T>)`,

die anhand des übergebenen `Predicate<T>` ein Ergebnis vom Typ `boolean` berechnen. Im Listing zeige ich drei Varianten von Tests, die auf den Startbuchstaben »`T`« prüfen:

```
public static void main(final String[] args)
{
   final List<String> names = Arrays.asList("Tim", "Tom", "Micha");

   final Predicate<String> startWithT = str -> str.startsWith("T");

   final boolean allStartWithT = names.stream().allMatch(startWithT);
   final boolean anyStartWithT = names.stream().anyMatch(startWithT);
```

```
    final boolean noneStartWithT = names.stream().noneMatch(startWithT);

    System.out.println("allStartWithT:   " + allStartWithT);
    System.out.println("anyStartWithT:   " + anyStartWithT);
    System.out.println("noneStartWithT:  " + noneStartWithT);
}
```

Die Ausführung des Programms MATCHEXAMPLE produziert folgende Ausgaben, die die obigen Aussagen zur Arbeitsweise verdeutlichen:

```
allStartWithT:   false
anyStartWithT:   true
noneStartWithT:  false
```

Die Methoden `findFirst()` und `findAny()`

Auch das Suchen ist ein so verbreiteter Anwendungsfall, dass er seit JDK 8 im Interface `Stream<T>` durch die Methoden `findFirst()` bzw. `findAny()` bereitgestellt wird:

```
final Optional<Person> optionalFirst = filteredPersons.findFirst();
final Optional<Person> optionalAny = filteredPersons.findAny();
```

Sinnvollerweise wird man zunächst eine Filterung vornehmen: `findFirst()` liefert dann das erste Element des Streams, falls es ein solches gibt. `findAny()` liefert ein beliebiges Element, falls es ein solches gibt. In beiden Fällen wird `Optional.empty` zurückgeliefert, sofern kein passendes Element bereitgestellt werden kann.

Beispiel zu `anyMatch()` und `findFirst()`

Nehmen wir an, wir wollten in einer Liste von Personen prüfen, ob eine Person mit dem gewünschten Namen enthalten ist bzw. wir möchten jeweils den ersten der gefundenen Einträge ermitteln. Vor JDK 8 könnte man eine entsprechende `containsPersonWithName()` bzw. `findPersonByName()`-Methode wie folgt schreiben:

```
boolean containsPersonWithName(final List<Person> persons, final String
desired)
{
    return findPersonByName(person, desired) != null;
}

Person findPersonByName(final List<Person> persons, final String desired)
{
    for (final Person person : persons)
    {
        if (person.getName().equals(desired))
        {
            return person;
        }
    }

    return null;
}
```

Wenn wir Lambdas in Kombination mit Streams nutzen, so schreiben wir kürzer Folgendes:

```
// Namensfilter definieren
final Predicate<Person> nameFilter = person ->
person.getName().equals(desired);

// containsPersonWithName()
final boolean personFound = persons.stream().anyMatch(nameFilter);

// findPersonByName()
final Optional<Person> searchedPerson = persons.stream().filter(nameFilter).
                                                         findFirst();
```

Die Filterbedingung vom Typ `Predicate<T>` realisieren wir als Lambda. Diesen `nameFilter` nutzen wir als Eingabe für einen Aufruf von `anyMatch(Predicate<T>)` zum Ermitteln, ob mindestens ein Element im Stream die übergebene Bedingung erfüllt. Außerdem wird der `nameFilter` an eine Kombination aus `filter(Predicate<T>)` und `findFirst()` übergeben, wodurch eine Filterung erfolgt und der erste Eintrag der Treffermenge ermittelt wird. Diese kann durchaus leer sein. Gewöhnlich wird so etwas durch die Rückgabe von `null` oder Null-Objekten gemäß dem gleichnamigen Entwurfsmuster implementiert. Wie bereits für optionale Rückgaben gesehen, bietet sich für Objekte die mit JDK 8 neu eingeführte generische Klasse `java.util.Optional<T>` an, um optionale Werte in Form eines Objekts zu modellieren. Existiert kein gültiger Wert, so wird dies durch `Optional.empty()` ausgedrückt. Auf die Klasse `Optional<T>` gehe ich in Abschnitt 15.2 genauer ein.

Die Methode `reduce()` – Elemente zusammenfassen

Wir haben schon verschiedenste Terminal Operations kennengelernt, beispielsweise solche, die aus einer Menge von Elementen einen akkumulierten Wert wie die Summe oder den Durchschnitt oder einen booleschen Wert berechnet haben.

Nun wollen wir die Methode `reduce(BinaryOperator<T>)` betrachten. Durch deren Einsatz kann man Elemente des gleichen Typs `T` miteinander kombinieren. Das kann eine beliebige Operation sein: Zunächst schauen wir uns dies für Strings und deren Konkatenation an, danach für `Integer`-Objekte und die Addition sowie Multiplikation:

```
public static void main(final String[] args)
{
  final Stream<String> names = Stream.of("Mike", "Tom", "Peter", "Chris");
  final Stream<Integer> integers = Stream.of(1, 2, 3, 4, 5);
  final Stream<Integer> empty = Stream.of();

  final Optional<String> stringConcat = names.reduce((s1,s2) -> s1 + ", " +
s2);
  final Optional<Integer> multiplication = integers.reduce((s1,s2) -> s1 *
s2);
  final Optional<Integer> addition = empty.reduce((s1,s2) -> s1 + s2);
```

```
  System.out.println("stringConcat:    " + stringConcat);
  System.out.println("multiplication:  " + multiplication);
  System.out.println("addition:        " + addition);
}
```

Startet man das obige Programm REDUCEEXAMPLE, so kommt es zu nachfolgend gezeigten Ausgaben. Diese verdeutlichen die Konkatenation bzw. Berechnung, wobei die von einem `Optional<T>` gespeicherten Werte in dessen Stringausgabe in eckigen Klammern angegeben sind:

```
stringConcat:    Optional[Mike, Tom, Peter, Chris]
multiplication:  Optional[120]
addition:        Optional.empty
```

Die Methoden `joining()`, `groupingBy()` und `partitioningBy()`

Neben der Ausgabe auf der Konsole oder der Umwandlung der Daten eines Streams in eine Collection sind weitere Transformationen wünschenswert, etwa das Verknüpfen von Strings sowie die Gruppierung oder Partitionierung von Daten. Dazu bietet die Utility-Klasse `Collectors` verschiedene Hilfsmethoden, die man gewinnbringend in Kombination mit der Methode `collect()` nutzen kann. Nachfolgend wollen wir einen kurzen Blick auf folgende Methoden werfen:

- `joining()` – Fasst Einträge vom Typ `String` zusammen. Das ist nützlich, um etwa eine kommaseparierte Auflistung zu realisieren.
- `groupingBy()` – Nimmt eine Gruppierung anhand eines Kriteriums vor.
- `counting()` – Zählt die Vorkommen in Kombination mit `groupingBy()`.
- `partitioningBy()` – Unterteilt die Eingabedaten basierend auf einer Realisierung eines `Predicate<T>` in zwei Partitionen.

Den Einsatz der obigen Methoden zeigt das folgende Listing, wobei hier zur besseren Lesbarkeit der Berechnungen statische Imports genutzt werden. Als Beispieldaten verwenden wir verschiedene Namen, die wir unter anderem nach Länge gruppieren und die Vorkommen wie folgt zählen:

```
import static java.util.stream.Collectors.counting;
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.Collectors.joining;
import static java.util.stream.Collectors.partitioningBy;

// …

final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi", "Mike",
                                         "Florian", "Michael", "Sebastian");

final String joined = names.stream().sorted().collect(joining(", "));

Map<Integer, List<String≫ grouped =
                            names.stream().collect(groupingBy(String::length));

Map<Integer, Long> counting =
```

```
                    names.stream().collect(groupingBy(String::length,
                                                      counting())));

Map<Boolean, List<String≫ partitions =
                              names.stream().filter(str -> str.contains("i")).
                              collect(partitioningBy(str -> str.length() > 4));

// …
```

Das Programm COLLECTORSSPECIALEXAMPLE produziert folgende Ausgaben, Anhand derer sich die zuvor kurz beschriebene Arbeitsweise der Methoden erschließt:

```
joined:      Andi, Florian, Michael, Mike, Ralph, Sebastian, Stefan
grouped:     {4=[Andi, Mike], 5=[Ralph], 6=[Stefan], 7=[Florian, Michael],
              9=[Sebastian]}
counting:    {4=2, 5=1, 6=1, 7=2, 9=1}
partitions: {false=[Andi, Mike], true=[Florian, Michael, Sebastian]}
```

Beispiel: Worthäufigkeitshistogramm

Bei der Besprechung der zustandslosen Intermediate Operations hatte ich im Rahmen der Ausführungen zur Methode `flatMap()` gezeigt, wie man aus einer Eingabe, bestehend aus einer Menge von Sätzen, die Häufigkeit bestimmter Wörter ermitteln kann. Dort waren wir so weit gekommen, alle Wörter sortiert hintereinander ausgeben zu können. Mit den gerade vorgestellten Methoden zur Gruppierung können wir nun ein Worthäufigkeitshistogramm aufbereiten. Zur Erinnerung sind einige der relevanten Zeilen nochmals abgebildet und um die Gruppierung und die case-insensitive Sortierung der Schlüssel ergänzt:

```
public static void main(final String[] args) throws IOException
{
   final Path exampleFile = Paths.get("src/chxx_jdk8/misc/Example.txt");

   // Datei einlesen neu in JDK 8: Siehe dazu Kapitel 6
   final List<String> contents = Files.readAllLines(exampleFile);

   // …

   // Filterung basierend auf den Prädikaten
   final Stream<String> filteredContents = words.map(String::trim).
                                                 filter(notIsShortWord).
                                                 filter(isSignificantWord);

   // Mappings zum Satzzeichen entfernen
   final Function<String, String> removePunctationMarks = str ->
   {
      if (str.endsWith(".") || str.endsWith(":") || str.endsWith("!"))
      {
         return str.substring(0, str.length()-1);
      }
      return str;
   };

   final Stream<String> mapped = filteredContents.map(removePunctationMarks);

   // Gruppierung
   Function<String, String> identity = Function.identity(); // str -> str;
```

```
    Map<String, Long> grouped = mapped.collect(groupingBy(identity,
counting()));

    // Sortierung der Schlüssel mithilfe einer TreeMap<K,V>
    Map<String, Long> sorted = new TreeMap<>(String.CASE_INSENSITIVE_ORDER);
    sorted.putAll(grouped);

    System.out.println(sorted);
}
```

Nach dem Start des Programms FLATMAPANDGROUPINGEXAMPLE erhält man folgende Ausgabe, die ein Worthäufigkeitshistogramm repräsentiert:

```
{contains=1, first=1, goodbye=1, Last=1, line=4, second=1, some=1,
text=3, Third=1}
```

12.3.6 Wissenswertes zur Parallelverarbeitung

Eingangs erwähnte ich, dass Streams als sequenzielle oder parallele Variante erzeugt werden können. Besonders interessant ist, dass man sich zu Beginn der Verarbeitung nicht darauf festlegen muss, wie alle Schritte abgearbeitet werden sollen. Es ist vielmehr möglich, beliebig zwischen paralleler und sequenzieller Abarbeitung hin und her zu schalten, wie es das nachfolgende Beispiel zeigt:

```
final String adults = persons.parallelStream().filter(Person::isAdult).
                              sequential().map(Person::getName).
                              collect(Collectors.joining(", "));
```

An diesem Beispiel erahnt man die Möglichkeiten zur Parallelisierung von Berechnungen, die sich durch die neuen Sprachfeatures ergeben. Besonders praktisch ist, dass die dahinter stehende Komplexität für den Entwickler verborgen bleibt. Man muss sich somit nicht um Details der Multithreading-Verarbeitung kümmern, sondern beschreibt die Abläufe auf einer höheren Abstraktionsebene.

Besonderheit bei `parallelStream()` und `forEach()`

Wenn Sie eine Verarbeitung parallel mit `parallelStream()` ausgeführt haben, dann entsprechen die Ausgaben über `forEach(Consumer<? super T>)` teilweise nicht dem, was Sie erwarten: Die Reihenfolge der Ausgabe ist oftmals ungeordnet.

Für das Sortieren kann man den geschilderten Sachverhalt gut nachvollziehen. Schauen wir zur Verdeutlichung auf eine Liste von Namen:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi",
"Mike");
    names.parallelStream().sorted().forEach(System.out::println);
}
```

Das Programm WRONGPARALLELFOREACHEXAMPLE produziert manchmal

ungeordnete Ausgaben ähnlich zu folgender, obwohl explizit `sorted()` zur Sortierung aufgerufen wurde.

```
Ralph
Andi
Stefan
Mike
```

Führt man das Programm mehrmals aus, so erkennt man die zufällige Reihenfolge. Um die korrekte Reihenfolge bei Parallelverarbeitung sicherzustellen, muss man Aufrufe der Methode `forEachOrdered(Consumer<? super T>)` wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("Stefan", "Ralph", "Andi",
"Mike");
    names.parallelStream().sorted().forEachOrdered(System.out::println);
}
```

Anmerkung: Sinnhaftigkeit der »Umschalterei«

Eine Umschaltung von Parallelverarbeitung auf eine anschließend sequenzielle Weiterverarbeitung kann möglicherweise ungünstig sein: Die zuvor durch die Parallelverarbeitung erzielten Performance-Vorteile können so teilweise wieder zunichtegemacht werden, weil am Ende der Parallelverarbeitung eine Abstimmung (Synchronisation) benötigt wird. Ein wiederholtes Umschalten zwischen paralleler und sequenzieller Abarbeitung ist daher in der Regel wenig sinnvoll.

12.4 Filter-Map-Reduce

Wir haben uns mittlerweile so viel Grundlagenwissen zu Streams erarbeitet, dass uns nun das Verständnis der mächtigen neuen Filter-Map-Reduce-Funktionalität – einer speziellen Untermenge des Stream-APIs – recht leicht fallen sollte.

Aufgabenstellung: Filtere eine Liste und extrahiere Daten

Nehmen wir an, unsere Aufgabe bestünde darin, eine Liste von Personen zu filtern, dabei alle im Juli Geborenen zu ermitteln und deren Namen kommasepariert auszugeben. Gegeben sei dazu folgende `List<Person>` als Eingabe:[5]

```
private static final List<Person> persons = Arrays.asList(
                        new Person("Stefan", LocalDate.of(1971, MAY, 12)),
                        new Person("Micha", LocalDate.of(1971, FEBRUARY, 7)),
                        new Person("Andi Bubolz", LocalDate.of(1968, JULY,
17)),
                        new Person("Andi Steffen", LocalDate.of(1970, JULY,
17)),
                        new Person("Merten", LocalDate.of(1975, JUNE, 16)));
```

Die Aufgabe lässt sich in folgende drei Schritte untergliedern:

1. Filtere auf alle im Juli Geborenen

2. Extrahiere ein Attribut, im Beispiel den Namen

3. Bereite eine kommaseparierte Liste auf

Bevor wir uns die mit JDK 8 bereitgestellte Filter-Map-Reduce-Funktionalität anschauen, werfen wir einen Blick darauf, wie man so etwas mit JDK 7 realisiert hätte.

12.4.1 Herkömmliche Realisierung

Der herkömmliche Ansatz bis einschließlich JDK 7 besteht darin, die Funktionalität selbst zu programmieren. Der Übersichtlichkeit halber werden die einzelnen Schritte in Form kurzer Methoden realisiert. Zum besseren Verständnis beginnen wir mit der Implementierung einer `main()`-Methode, die folgende drei Schritte ausführt:

```
public static void main(final String[] args)
{
    // Schritt 1: Filtere
    final List<Person> bornInJuly = filterByMonth(persons, Month.JULY);

    // Schritt 2: Extrahiere
    final List<String> names = extractNameAttribute(bornInJuly);

    // Schritt 3: Bereite Ergebnis auf
    final String result = joinStrings(names, ", ");

    System.out.println(result);
}
```

Filtere auf alle im Juli Geborenen

Wir konstruieren eine Ergebnisliste `filteredPersons` und fügen dort diejenigen `Person`-Objekte hinzu, die das Kriterium „Geboren im Juli" erfüllen:

```
static List<Person> filterByMonth(final List<Person> persons, final Month
month)
{
    final List<Person> filteredPersons = new ArrayList<>();
    for (final Person person : persons)
    {
        if (person.getBirthday().getMonth().equals(month))
        {
            filteredPersons.add(person);
        }
    }
    return filteredPersons;
}
```

Extrahiere ein Attribut

Als zweiten Schritt nehmen wir eine ***Extraktion*** von Daten vor (man spricht auch von

Projektion). Die Namen der Personen werden ausgelesen und als `List<String>` bereitgestellt. Die korrespondierende Implementierung ist in folgendem Listing gezeigt:

```
static List<String> extractNameAttribute(final List<Person> persons)
{
    final List<String> names = new ArrayList<>();
    for (final Person person : persons)
    {
        names.add(person.getName());
    }
    return names;
}
```

Bereite eine kommaseparierte Liste auf

Zur Aufbereitung der Ausgabe durchlaufen wir die als Parameter übergebene Liste und fügen jedes Element gefolgt von einem Komma (mit Ausnahme des letzten) in ein `StringBuilder`-Objekt per `append()` ein:

```
static String joinStrings(final List<String> names, final String delimiter)
{
    final StringBuilder sb = new StringBuilder();

    final Iterator<String> it = names.iterator();
    while (it.hasNext())
    {
        sb.append(it.next());
        if (it.hasNext())
            sb.append(delimiter);
    }

    return sb.toString();
}
```

Führen wir die realisierte Funktionalität – wie zuvor in `main()` gezeigt – aus, so werden die beiden im Juli geborenen Personen ausgegeben:

```
Andi Bubolz, Andi Steffen
```

Wenn man die Lösung betrachtet, so fällt negativ auf, dass diese recht lang ist. Erst bei etwas genauerem Überlegen bemerkt man einen weiteren Nachteil: Die Abarbeitung erfolgt sequenziell und die Laufzeit erhöht sich linear zu der Anzahl gespeicherter Personen. Als Alternative kann man die gesamte Funktionalität innerhalb nur einer Methode und ineinander verschränkt realisieren. Das wird zwar minimal schneller, jedoch deutlich unübersichtlicher – insbesondere, wenn die Abfragen komplexer werden. Auch kann man Erweiterungen kaum realisieren und die fehlende Kombinierbarkeit widerspricht dem Gedanken der Orthogonalität. Schauen wir uns jetzt eine mit Java 8 mögliche Variante an.

12.4.2 Filter-Map-Reduce mit JDK 8

Mit JDK 8 wird eine Filter-Map-Reduce-Funktionalität in Java bereitgestellt, die stark vom Einsatz von Lambdas profitiert. Dabei stehen die drei Begriffe Filter, Map und Reduce für die bereits im vorangegangenen Beispiel kennengelernten Aktionen:

- **Filter** – Aus einer Ausgangsmenge von Objekten werden diejenigen herausgefiltert, die den benötigten Anforderungen entsprechen.
- **Map** – Mit Mapping oder Projektion ist der Vorgang gemeint, der aus einem Objekt gewisse Informationen ableitet und diese in einer gewünschten Form aufbereitet. Map beschreibt eine Projektion, d. h. eine Transformation eines Elements in eine andere Repräsentation (wobei die Anzahl Elemente gleich bleibt).
- **Reduce** – Schlussendlich sollen die Berechnungsergebnisse verarbeitet werden, etwa auf der Konsole ausgegeben oder als Ergebnismenge in einer Collection aufbereitet werden. Reduce beschreibt somit das Zusammenfassen zu einem Resultat.

Diese Schritte sind in der nachfolgenden Abbildung visualisiert:

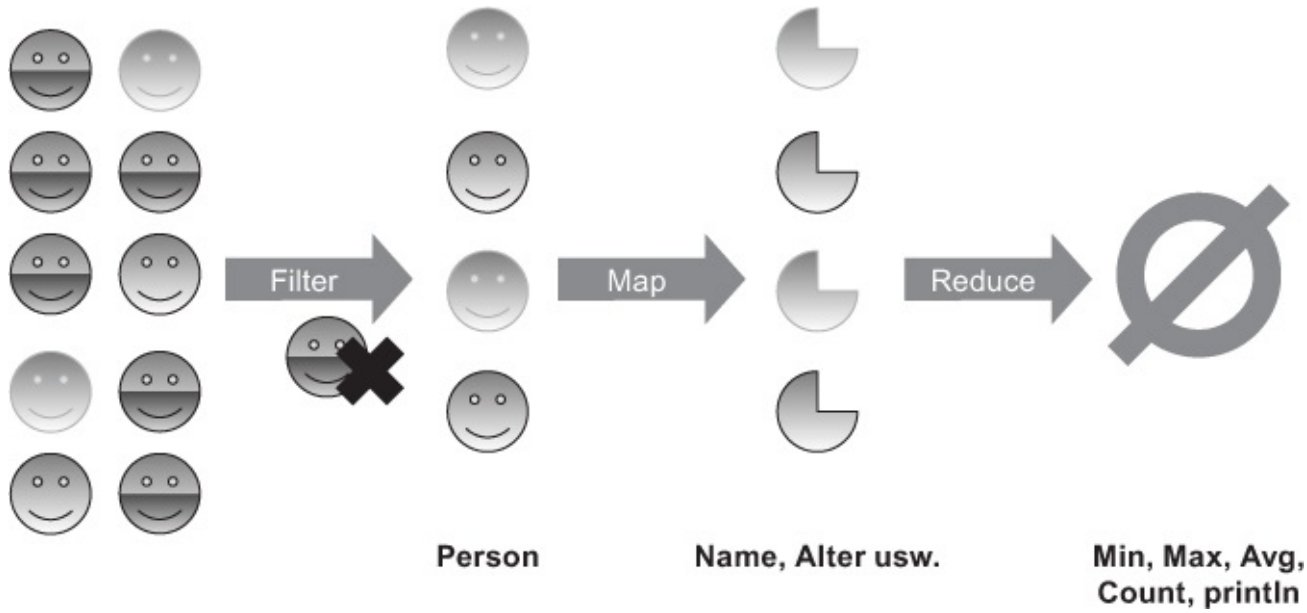


Abbildung 12-3 *Symbolische Darstellung von Filter-Map-Reduce*

Filter-Map-Reduce im Einsatz

Wir machen uns jetzt daran, die drei Verarbeitungsschritte zur Filterung mithilfe des Filter-Map-Reduce-Frameworks und von Lambdas zu implementieren. Die benötigten Grundlagen haben wir uns zuvor bei der Besprechung der Intermediate und Terminal Operations erarbeitet. Dieses Wissen muss man nur noch geeignet kombinieren:

```
final String bornInJuly = persons.stream().
                              filter( person ->
person.getBirthday().getMonth().
                                                    equals( Month.JULY)).
                              map( Person::getName).
                              reduce( "", stringCombiner);
```

Die beiden Schritte Filter und Map sind intuitiv verständlich, beim Reduce-Schritt bedienen wir uns eines Tricks. Hierbei wird die nachfolgend gezeigte Realisierung des funktionalen Interface `BinaryOperator<T>` namens `stringCombiner` genutzt. Dort wird aus zwei Eingaben vom Typ `T` ein Ergebnis vom Typ `T` berechnet. Die Kombination zweier Strings können wir wie folgt implementieren:

```
final BinaryOperator<String> stringCombiner = (str1, str2) ->
{
    if (str1.isEmpty())
    {
        return str2;
    }
    return str1 + ", " + str2;
};
```

Beim Betrachten dieses Lambdas erinnern Sie sich vielleicht an meinen Tipp, Lambdas lediglich für solche Dinge zu nutzen, deren Implementierung nur wenige Zeilen Sourcecode umfasst. Hier ist das schon grenzwertig. Natürlich könnte man hier alternativ einen ternären Ausdruck nutzen, aber auch der ist schon etwas unleserlich:

```
final BinaryOperator<String> stringCombiner = (str1, str2) ->
{
    return str1.isEmpty() ? str2 : str1 + ", " + str2;
};
```

Zusammenfassen von Werten mit `Collectors`

Das Zusammenfassen von Einträgen ist ein gebräuchlicher Anwendungsfall. Daher bietet das JDK 8 eine spezielle Form einer Reduce-Methode namens `collect()`, die wir bereits kurz im Rahmen der Terminal Operations in Abschnitt 12.3.5 besprochen haben. Vorgefertigte Implementierungen der dort benötigten `Collector`-Instanzen finden sich in der Utility-Klasse `Collectors`. Einige davon haben wir schon zuvor kennengelernt. Wir wählen hier `joining(String)` zur Kombination von Strings wie folgt:

```
final String bornInJuly = persons.stream().
                          filter(person ->
                                 person.birthday.getMonth() ==
Month.JULY).
                          map(person -> person.name).
                          collect(Collectors.joining(", "));
```

Man erkennt sehr schön, dass die gewählte Form mehr am zu lösenden Problem ausgerichtet ist und nicht ein spezieller Algorithmus zum Filtern programmiert wird. Außerdem versteckt `joining()` die Spezialbehandlung der korrekten Aufbereitung einer kommaseparierten Ausgabe.

Hinweis: Die Methode `String.join()`

Wenn tatsächlich nur textuelle Werte miteinander verknüpft werden sollen, dann kann man die in der Klasse `String` mit JDK 8 neu eingeführte Methode

`join(CharSequence delimiter, CharSequence… elements)` nutzen:[a]

```
final String stringConcat = String.join(", ", names);
```

Etwas unglücklich empfinde ich die Signatur, in der der Delimiter vor den zu verknüpfenden Elementen angegeben werden muss. Das liegt aber einfach daran, dass in Java Varargs nur für den letzten Parameter in einer Signatur auftreten dürfen.

[a] Wir haben zuvor mit `joinStrings()` eine ähnliche Methode entworfen.

Aufbereitung/Verknüpfung nicht textueller Werte

Gerade haben wir gesehen, wie einfach die kommaseparierte Aufbereitung von textuellen Werten mithilfe von Lambdas und den Neuerungen im Stream-API geschrieben werden kann.

Etwas mehr Aufwand muss man betreiben, wenn man Zahlen oder Objekte auf diese Weise miteinander verknüpfen möchte. Nehmen wir an, wir wollten die Altersangaben der Personen kommasepariert aufbereiten. Dazu müssen wir die Objekte, hier vom Typ `Integer`, in Strings wandeln, bevor wir sie mit `joining()` verknüpfen können. Die Umwandlung kann explizit durch einen Aufruf von `toString()` erfolgen oder aber implizit durch die Notation `"" + value`. Im folgenden Listing sind beide Varianten gezeigt:

```
// Explizite Umwandlung / Mapping mit toString
final String joined1 = persons.stream().mapToInt(Person::getAge)
                                       .mapToObj(Integer::toString)
                                       .collect(joining(", "));

// Implizite Umwandlung / Mapping durch "" + value
final String joined2 = persons.stream().map( person -> "" + person.getAge() )
                                       .collect(joining(", "));
```

12.5 Fallstricke bei Lambdas und funktionaler Programmierung

In den vorangegangenen Abschnitten haben wir gesehen, wie viel Positives durch den Einsatz von Lambdas in Kombination mit den Bulk Operations on Collections und der Filter-Map-Reduce-Funktionalität erzielbar ist.

Nachfolgend sollen aber zwei mögliche Probleme kurz thematisiert werden: Zum einen ist mit Streams und dem Filter-Map-Reduce-Framework die Lernkurve steiler geworden und Java ist an einigen Stellen recht komplex, da eine Vielzahl von Interfaces neu eingeführt wurde, deren Realisierung nicht immer ganz so intuitiv ist. Zum anderen stellt die funktionale Programmierung auch Ansprüche an die Fähigkeiten der Entwickler, das Paradigma richtig umzusetzen.

12.5.1 Java-Fehlermeldungen werden zu komplex

Wenn man anstelle von Lambdas versucht, die neuen Funktionalitäten mithilfe anonymer innerer Klassen zu nutzen, stößt man mitunter recht schnell auf Probleme und man erhält kaum verständliche Fehlermeldungen. Wir sehen uns hier zur Demonstration lediglich ein kurzes Beispiel an. Danach kann sich jeder Entwickler seine eigene Meinung darüber bilden.

Nachfolgend betrachten wir wieder einige Personen und realisieren ein Mapping von einem `Person`-Objekt auf das Attribut `age` als `Function<Person, Integer>`. Wir versuchen dann, das Durchschnittsalter mit einer (versehentlich bzw. im folgenden Beispiel zu Demonstrationszwecken) nicht ganz passenden `ToIntFunction<Double>` zu berechnen:

```
final List<Person> persons = new ArrayList<>();

// …

// Mapping auf Alter
final Stream<Integer> agesStream = persons.stream().
                                   map(new Function<Person, Integer>()
{
    @Override
    public Integer apply(final Person person)
    {
        return person.getAge();
    }
});

// Durchschnittsberechnung
final int averageAge = agesStream.collect(Collectors.
                                  averagingInt(new ToIntFunction<Double>()
{
    @Override
    public int applyAsInt(final Double value)
    {
        return value.intValue();
    }
});
```

Wenn man das Programm kompiliert (per javac oder in Netbeans), so bekommt man eine Fehlermeldung, die man erst einmal verdauen und vor allem verstehen muss. Das gestaltet sich allein schon aufgrund der schieren Länge als eine Herausforderung:

```
no suitable method found for collect(Collector<Double,CAP#1,Double>)
    method Stream.<R#1>collect(Supplier<R#1>,BiConsumer<R#1,? super Integer>,
          BiConsumer<R#1,R#1>) is not applicable
      (cannot infer type-variable(s) R#1
        (actual and formal argument lists differ in length))
    method Stream.<R#2,A>collect(Collector<? super Integer,A,R#2>) is not
          applicable
      (inferred type does not conform to upper bound(s)
        inferred: Integer
        upper bound(s): Double,Object)
where R#1,T,R#2,A are type-variables:
  R#1 extends Object declared in method
```

```
<R#1>collect( Supplier<R#1>, BiConsumer<R
        #1, ? super T>, BiConsumer<R#1, R#1>)
  T extends Object declared in interface Stream
  R#2 extends Object declared in method <R#2, A>collect( Collector<? super T, A, R
        #2>)
  A extends Object declared in method <R#2, A>collect( Collector<? super
T, A, R#2>)
where CAP#1 is a fresh type-variable:
  CAP#1 extends Object from capture of ?
```

Die von Eclipse produzierte Fehlermeldung ist etwas kürzer:

```
The method collect( Collector<? super Integer, A, R>) in the type Stream<Integer>
is not applicable for the arguments ( Collector<Double, capture#1-of ?, Double>)
```

Wie schon bei Intermediate Operations und Berechnungen besprochen, existieren bei der Berechnung von Durchschnittswerten zwei Besonderheiten. Zum einen kann für den Fall eines leeren Streams kein Durchschnitt ermittelt werden. Zum anderen lässt sich der Durchschnitt von `Integer`- oder `Long`-Werten nur durch einen Gleitkommatyp, etwa `Double`, darstellen. Hier hätte man einfach einen anderen Collector sowie einen anderen Ergebnistyp wählen müssen, was aber nur Experten anhand der Fehlermeldung erahnen können. Das Ganze ließe sich wie folgt korrigieren:

```
final Double averageAge = agesStream.collect( Collectors.averagingDouble(
                                                new ToDoubleFunction<Integer>()
{
    @Override
    public double applyAsDouble( final Integer value)
    {
        return value.doubleValue();
    }
}
```

Das gezeigte Beispiel müsste auch noch in seiner Funktionalität korrigiert werden, denn möglicherweise lässt sich bei einer leeren Eingabemenge kein Durchschnitt berechnen. Diese Thematik wollen wir hier nicht weiter behandeln, da es in diesem Abschnitt lediglich um die Darstellung der recht verwirrenden und komplexen Fehlermeldungen ging. In Abschnitt 15.2 gehe ich aber im Detail auf die Klasse `Optional<T>` zur Darstellung optionaler Werte sowie korrespondierende Klassen zur direkten Verarbeitung primitiver Typen ein.

12.5.2 Fallstrick: Imperative Lösung 1:1 funktional umsetzen

Jedes Programmierparadigma hat seine spezifischen Stärken und Schwächen. Wenn man aber zwei Paradigmen miteinander mischt, so besteht auch immer die Gefahr, die Schwächen zu kombinieren. Das nachfolgende prägnante Beispiel wurde leicht modifiziert und stammt ursprünglich aus einem Blog[6] von Johannes Weigend, der mir die Erlaubnis gegeben hat, es hier präsentieren zu dürfen.

Schauen wir auf ein Programmstück, das imperativ eine Menge von Personen

durchgeht und für alle Personen namens »TheOne« die Methode `action(Person)` aufruft.

```
for (int i = 0; i < persons.size(); i++)
{
    final Person person = persons.get(i);
    if (person.getName().equals("TheOne"))
    {
        action(person);
    }
}
```

Die gewünschte Funktionalität ist wie prädestiniert für den Einsatz von Filter-Map-Reduce. Wenn man nun versucht, den Algorithmus mithilfe funktionaler Konstrukte imperativ nachzuprogrammieren, könnte dabei etwa Folgendes herauskommen:

```
IntStream.range(0, persons.size()).
          mapToObj(persons::get).
          filter(person -> person.getName().equals("TheOne")).
          forEach(person -> action(person));
```

Man sieht, dass hier nicht das zu lösende Problem im Mittelpunkt steht, sondern eine 1:1-Umsetzung des Algorithmus: Die `for`-Schleife wird durch `IntStream.range()` und das indizierte Auslesen durch `mapToObj()` abgebildet. Ein solches Vorgehen ist wenig sinnvoll. Schauen wir uns die Korrektur an: Mit dem bisher aufgebauten Wissen können Sie das sicher schon selbst. Wahrscheinlich würden Sie durch Nutzung des funktionalen Stils die obige Funktionalität wie folgt realisieren:

```
persons.stream().filter(person -> person.getName().equals("TheOne")).
                 forEach(person -> action(person));
```

12.6 Fazit

Dieses Kapitel hat einen fundierten Überblick über die Erweiterungen im Zusammenhang mit Bulk Operations on Collections gegeben. Dabei wurden insbesondere die deutlichen Erleichterungen bei der Programmierung anhand kleiner praxisnaher Beispiele hervorgehoben.

Auch wenn ich zum Abschluss noch auf zwei mögliche Probleme beim Einsatz von Lambdas, dem Filter-Map-Reduce-Framework sowie der funktionalen Programmierung eingegangen bin, so überwiegen doch die positiven Seiten.

Neben verschiedenen praktischen Erweiterungen im Collections-Framework sind die neu eingeführten Streams besonders erwähnenswert. Damit ist es endlich möglich, viele Aufgabenstellungen näher am zu lösenden Problem zu beschreiben, als dies mit imperativer Ausformulierung eines Algorithmus möglich ist. Außerdem lassen sich Algorithmen bei Bedarf parallel ausführen. Dadurch kann man von den Multicores in heutigen Prozessoren profitieren, ohne die Parallelisierung aufwendig mit dem Fork-Join-Framework oder in mühevoller Kleinarbeit und fehlerträchtig selbst mit Threads umzusetzen.

13 JSR-310: Date And Time API

In diesem Kapitel stelle ich das im Rahmen von JSR-310 erarbeitete neue Date And Time API vor. Zunächst erläutere ich, wieso es notwendig wurde, einen dritten Wurf zur Datumsverarbeitung zu entwickeln, bevor ich dann auf das neue API eingehe.

13.1 Datumsverarbeitung vor JSR-310

Die Verarbeitung von Datumswerten und Zeitangaben scheint einfach, ist es aber nicht. Tatsächlich ist es sogar ziemlich kompliziert, weil verschiedene Dinge zu beachten sind, etwa der Einfluss von Zeitzonen, Schaltjahren sowie Sommer- und Winterzeit.

Beispielsweise kann man durch Aufruf der Methode `System.currentTimeMillis()` eine Zeitangabe in Form eines `long`-Werts erhalten, der die Anzahl der seit dem 1.1.1970 vergangenen Millisekunden darstellt. Häufig benötigt man aber eine objektorientierte Sichtweise und eine bessere Abstraktion. Das gilt etwa, wenn man Berechnungen anstellen möchte wie »Gehe einen Monat in die Vergangenheit«. So etwas in Millisekunden zu berechnen, ist kompliziert, da gegebenenfalls Schaltjahre und verschiedene Längen von Monaten zu berücksichtigen sind. Mit der Klasse `java.util.Date` lassen sich derartige Berechnungen nur unzureichend abbilden, weil nur eine minimale Abstraktion eines `long` geboten wird. Zudem lauern in der Klasse `Date` einige Fallstricke, weshalb eine Vielzahl der dort definierten Methoden als `deprecated` gekennzeichnet wurden und nicht mehr verwendet werden sollten. Warum dies der Fall ist, möchte ich an einem einfachen Konstruktoraufruf der Klasse `Date` zeigen, der mit dem 7. Februar 1971 den Geburtstag des Autors erzeugt. Intuitiv würde man dies etwa wie folgt realisieren, wobei hier schon die amerikanische Reihenfolge erst Monat, dann Tag genutzt wird:

```
public static void main(final String[] args)
{
    // Geburtstag des Autors: 7.2.1971
    final int year = 1971;
    final int month = 2;
    final int day = 7;

    System.out.println(new Date(year, month, day));
    System.out.println(new Date(year - 1900, month - 1, day));
}
```

Führt man das Programm DATECTORPROBLEMSEXAMPLE aus, so landet man im März des Jahres 3871! Nur die gezeigten Korrekturwerte führen zum korrekten Datum:

```
Tue Mar 07 00:00:00 CET 3871
Sun Feb 07 00:00:00 CET 1971
```

Selbst diese scheinbar einfache Konstruktion eines Datums ist also gespickt mit Fallstricken: Der Konstruktor addiert auf den Jahreswert das Offset 1900, erwartet Monatsangaben 0-basiert und verarbeitet Tagesangaben 1-basiert!

Als Alternative bzw. Ergänzung zur Klasse `Date` gibt es im JDK im Package `java.util` die abstrakte Klasse `Calendar`, deren konkrete Realisierungen wie `GregorianCalendar` weniger Fallstricke bergen und eine bessere Abstraktion bieten (Konstanten für Monate, Addition von Zeitwerten usw.). Damit werden die Verarbeitung und vor allem Berechnungen erleichtert. Allerdings ist einiges noch immer ziemlich kompliziert, und zwar insbesondere dann, wenn man statt mit Datum und Uhrzeit in Kombination nur mit Zeitangaben oder Datumswerten rechnen möchte. Für diese Fälle beginnen die Spezialisierungen von `Calendar` recht unhandlich zu werden.

Weitere Probleme des alten Datum-APIs sind:

- Die Klassen `Date`, `Calendar`, `SimpleDateFormat` und `DateFormatter` sind veränderlich. Im Zusammenhang mit Multithreading kann es dadurch schnell zu Inkonsistenzen kommen.
- Von der Klasse `Date` aus dem Package `java.util` sind die Klassen `Date`, `Time` und `Timestamp` aus dem Package `java.sql` abgeleitet. Diese Subklassen stellen aber semantisch keine wirklichen Subtypen dar, weil sie jeweils nur einen speziellen Aspekt eines Datumswerts beschreiben.
- Es existieren verschiedene Probleme bei der Ausgabe, etwa die fehlende Möglichkeit, `Calendar`-Objekte formatiert auszugeben.

Beispiel: API-Merkwürdigkeiten bei Berechnungen

Ein besonders kurioses Problem, über das ich erst neulich (wieder) gestolpert bin, möchte ich Ihnen nicht vorenthalten. Die einfache Aufgabe besteht darin, aus zwei Zeitwerten, die als `Date`-Objekt gegeben sind, deren zeitliche Differenz zu berechnen und im Format HH:mm:ss auszugeben.

```
public static void main(final String[] args) throws ParseException
{
    // Unterschied 1 Stunde, 10 Minuten und 20 Sekunden
    final String startTimeAsString = "10:10:10";
    final String endTimeAsString  = "11:20:30";

    // Umwandlung in Date-Objekte
    final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
    final Date startTime = dateFormat.parse(startTimeAsString);
    final Date endTime = dateFormat.parse(endTimeAsString);

    // Berechne Differenz basierend auf Millisekunden
    final long durationInMs = endTime.getTime() - startTime.getTime();
    System.out.println("duration in seconds = " + TimeUnit.MILLISECONDS.
                                                  toSeconds(durationInMs));

    final String duration1 = dateFormat.format( new Date(durationInMs));
```

```
    System.out.println("duration 1 = " + duration1);

    // DateFormat muss Zeitzone gesetzt bekommen
    dateFormat.setTimeZone(TimeZone.getTimeZone("GMT"));
    final String duration2 = dateFormat.format(new Date(durationInMs));
    System.out.println("duration 2 = " + duration2);
}
```

Startet man das Programm OLDAPIDURATIONCALCULATIONEXAMPLE, so wird die Differenz von 1 Stunde, 10 Minuten und 20 Sekunden korrekt als 4220 Sekunden berechnet. Wenn man diese jedoch mithilfe eines `SimpleDateFormats` ausgeben möchte, erhält man eine Zeitangabe von 2 Stunden 10 Minuten und 20 Sekunden. Erst dann, wenn man im `SimpleDateFormat` die Zeitzone setzt, erhält man die korrekte Ausgabe – darauf muss man erst einmal kommen!

```
duration in seconds = 4220
duration 1 = 02:10:20
duration 2 = 01:10:20
```

Dieses Beispiel illustriert die Merkwürdigkeiten im alten Date-API und zeigt, warum eine Neuentwicklung notwendig wurde.

Somit ist die Verarbeitung von Datumswerten in Java bislang recht umständlich, und weder die Klasse `Date` noch die abstrakte Klasse `Calendar` bieten gelungene APIs zur Datumsverwaltung. Zwar sind diese Probleme schon seit Längerem bekannt, aber seit JDK 1.2 gab es keine Neuerungen mehr, obwohl bereits seit 2007 in Form des JSR-310 an einer Neuentwicklung der Datumsverarbeitung gearbeitet wird. Aufgrund der Unzulänglichkeiten der bisher verfügbaren Implementierungen zur Datumsverarbeitung haben viele größere Projekte vermutlich zunächst eigene kleinere Hilfsmethoden oder Utility-Klassen geschrieben. Das war wohl auch der Grund, dass die Bibliothek Joda-Time entwickelt wurde. Sie hat sich mittlerweile als De-facto-Standard etabliert, weil sich durch deren Nutzung die Datumsverarbeitung stark vereinfacht. Da in Joda-Time alle Klassen immutable und damit Thread-sicher sind, ist deren Einsatz auch im Multithreading-Kontext unkritisch.

Die im Rahmen von JSR-310 entwickelten Klassen adressieren die Probleme mit den bisherigen Datums-APIs des JDKs und nutzen vor allem Ideen aus der Bibliothek Joda-Time, deren Schöpfer Stephen Colebourne eine führende Rolle bei der Entwicklung von JSR-310 innehatte. Die Zielsetzung von JSR-310 ist, alles besser und einfacher nutzbar zu machen und ein gelungenes, hilfreiches API zur Verwaltung und zur Manipulation von Datums- und Zeitwerten bereitzustellen. Diese Ergänzung findet mit Java 8 Einzug ins JDK und wird nachfolgend in Form kleinerer Beispiele vorgestellt.

13.2 Überblick über die neu eingeführten Klassen

Das durch JSR-310 realisierte neue Date And Time API fügt dem JDK einige Funktionalität in verschiedenen Packages unter `java.time` hinzu. Dabei unterscheidet man im wesentlichen zwei Konzepte: Zum einen ist dies die

kontinuierliche oder Maschinenzeit, bei der durch die Klasse `java.time.Instant` ein spezieller Zeitpunkt repräsentiert wird. Das ist näherungsweise mit der Intention der Klasse `Date` vergleichbar. Im Unterschied dazu wird jedoch eine Auflösung im Bereich von Nanosekunden geboten. Der Referenzzeitpunkt ist, wie bei `Date`, der 1. Januar 1970. Zum anderen existieren Datumsklassen, die eher an menschlichen Denkweisen ausgerichtet sind: Die Klassen `LocalDate` und `LocalTime` aus dem Package `java.time` repräsentieren Datumswerte ohne Zeitzonen in Form eines Datums bzw. einer Zeit. Beide modellieren jeweils nur die durch den Klassennamen beschriebene Zeitkomponente, also Datum oder Zeit.

Wir werden nachfolgend diverse neue Klassen anhand kurzer Beispiele kennenlernen. Dabei werden zur Objektkonstruktion in der Regel folgende Varianten genutzt:

- `now()` – Datums-/Zeitwert basierend auf dem aktuellen Zeitpunkt
- `of()`-Methoden – teilweise spezielle Methoden `ofDays()`,`ofMonths()`
- `parse()` – Parsing von textuellen Angaben

13.2.1 Die Klasse `Instant`

Eine Instanz vom Typ `java.time.Instant` repräsentiert einen Zeitpunkt in Nanosekunden in Bezug auf den Referenzzeitpunkt 1.1.1970 00:00:00 Uhr. Die Zeit schreitet dabei linear voran: Diese Modellierung vereinfacht die Verarbeitung durch Computer, da keine Spezialfälle zu betrachten sind.

Im nachfolgenden Beispiel modellieren wir Abfahrts- und Ankunftszeiten einer Reise mit der Dauer von 5 Stunden (etwa mit der Bahn), die zum jetzigen Zeitpunkt beginnt. Diesen Startzeitpunkt ermitteln wir durch den Aufruf von `now()`. Die resultierende Ankunftszeit wird als `expectedArrivalTime` berechnet. Außerdem nehmen wir eine Verspätung von 7 Minuten an. Auf zweierlei Art wird daraus durch das folgende Programm INSTANTEXAMPLE die reale Ankunftszeit wie folgt berechnet:

```
public static void main(final String[] args)
{
   // Abfahrt jetzt und Reisedauer 5 Stunden
   final Instant departureTime = Instant.now();
   final Instant expectedArrivalTime = departureTime.plus(5,
ChronoUnit.HOURS);

   // Verspätung von 7 Minuten auf zwei Arten berechnen
   final Instant realArrival = expectedArrivalTime.plus(7,
ChronoUnit.MINUTES);
   final Instant realArrival2 =
expectedArrivalTime.plus(Duration.ofMinutes(7));

   System.out.println(departureTime);         // 2014-03-22T13:54:50.818Z
   System.out.println(expectedArrivalTime);   // 2014-03-22T18:54:50.818Z
   System.out.println(realArrival);           // 2014-03-22T19:01:50.818Z
   System.out.println(realArrival2);          // 2014-03-22T19:01:50.818Z
}
```

13.2.2 Die Aufzählung ChronoUnit

Im vorherigen Beispiel haben wir vorgreifend die Aufzählung java.time.temporal.ChronoUnit und die Klasse java.time.Duration genutzt, um Zeitdauern zu spezifizieren. ChronoUnit ist eine Aufzählung, die all diejenigen Zeiteinheiten definiert, mit denen im Date And Time API gerechnet werden kann. In ChronoUnit findet man Definitionen unter anderem für Minuten, Stunden, Wochen usw. Tatsächlich sind dort Konstanten für Nanosekunden bis hin zu Jahrtausenden sowie Äras und eine spezielle FOREVER-Konstante definiert.

Greifen wir das obige Beispiel auf: Wir nutzen wieder Instanzen von ChronoUnit, um die Zeitdauer in verschiedenen Varianten (Stunden und Minuten) darzustellen. Eine wichtige Eigenschaft ist, dass man mithilfe der Methode between(Temporal, Temporal) die Differenz zwischen zwei Zeitpunkten bestimmen kann. Das dort als Parametertyp genutzte Interface java.time.temporal.Temporal ist die Basis für verschiedenste Klassen aus dem neuen Date And Time API, die Zeitpunkte modellieren wie Instant, LocalTime usw. Die Differenz für zwei Instant-Objekte wird durch Aufruf von between(Temporal, Temporal) ermittelt:

```
public static void main(final String[] args)
{
    // Abfahrt jetzt und Reisedauer 5 Stunden
    final Instant departureTime = Instant.now();
    final Instant arrivalTime = departureTime.plus(5, ChronoUnit.HOURS);

    System.out.println("departure now:     " + departureTime);
    System.out.println("arrival now + 5h:  " + arrivalTime);

    // Berechnungen durchführen: Differenz bilden
    final long inBetweenHours = ChronoUnit.HOURS.between(departureTime,
                                                         arrivalTime);
    final long inBetweenMinutes = ChronoUnit.MINUTES.between(departureTime,
                                                             arrivalTime);

    System.out.println("inBetweenHours:    " + inBetweenHours);
    System.out.println("inBetweenMinutes:  " + inBetweenMinutes);
}
```

Führt man das obige Programm ChronoUnitExample aus, so erhält man in etwa folgende Ausgaben auf der Konsole, die sehr schön die Berechnungen von 5 Stunden in die Zukunft sowie die Differenzbildung zwischen zwei Zeitpunkten in verschiedenen Zeiteinheiten (Stunden und Minuten) zeigen:

```
departure now:     2014-02-19T22:13:50.691Z
arrival now + 5h:  2014-02-20T03:13:50.691Z
inBetweenHours:    5
inBetweenMinutes:  300
```

13.2.3 Die Klasse Duration

Die Klasse `java.time.Duration` erlaubt es, eine Zeitdauer in Nanosekunden exakt festzulegen, etwa um Differenzen zwischen zwei `Instant`-Objekten auszudrücken. Instanzen der Klasse `Duration` können durch Aufruf verschiedener Methoden konstruiert werden, wobei die Konstruktion aus übergebenen Werten verschiedener Zeiteinheiten[1] oder aber aus der Differenz zweier `Instant`-Objekte möglich ist:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Duration durationFromSecs = Duration.ofSeconds(15);
    final Duration durationFromMinutes = Duration.ofMinutes(30);
    final Duration durationFromHours = Duration.ofHours(45);
    final Duration durationFromDays = Duration.ofDays(60);

    System.out.println("From Secs:     " + durationFromSecs);
    System.out.println("From Minutes:  " + durationFromMinutes);
    System.out.println("From Hours:    " + durationFromHours);
    System.out.println("From Days:     " + durationFromDays);

    // Berechnungen
    final Instant now = Instant.now();
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant myBirthday2015 = Instant.parse("2015-02-07T00:00:00Z");
    final Duration duration1 = Duration.between(now, silvester2013);
    final Duration duration2 = Duration.between(now, myBirthday2015);

    System.out.println(now + " - " + silvester2013 + ": " + duration1);
    System.out.println(now + " - " + myBirthday2015 + ": " + duration2);
}
```

Führen wir das Programm DURATIONEXAMPLE aus, so kommt es zu folgenden Ausgaben, wobei im Speziellen folgende Dinge von Interesse sind: Zum einen werden Zeitdifferenzen maximal in der Zeiteinheit von Stunden abgebildet, wodurch für 60 Tage der Wert 1440 Stunden zustande kommt. Zum anderen wird ein Sprung in die Vergangenheit bzw. in die Zukunft gezeigt:

```
From Secs:     PT15S
From Minutes:  PT30M
From Hours:    PT45H
From Days:     PT1440H
2014-03-23T11:22:54.648Z - 2013-12-31T00:00:00Z: PT-1979H-22M-54.648S
2014-03-23T11:22:54.648Z - 2015-02-07T00:00:00Z: PT7692H37M5.352S
```

Weil das Ganze recht intuitiv erscheint, es aber Fallstricke gibt, wollen wir das Thema Berechnungen noch ein wenig genauer betrachten. Neben der zuvor gezeigten Differenzberechnung mit `between(Temporal, Temporal)` ist auch eine Addition einer durch eine `Duration` definierten Zeitspanne zu einem `Instant`-Objekt möglich. Man erhält als Ergebnis wiederum ein `Instant`-Objekt.

Die Addition von Zeitspannen betrachten wir an folgendem Beispiel. Ausgehend vom Heiligabend, dem 24.12.2013, soll eine Woche in die Zukunft zum Silvester-Tag 2013 gesprungen werden. Anschließend führen wir einen größeren Zeitsprung zum 18.3.2014, dem Releasedatum von JDK 8, aus. Wir schreiben dazu folgendes Programm:

```
public static void main(final String[] args)
{
    // Erzeugung
    final Instant christmas2013 = Instant.parse("2013-12-24T00:00:00Z");
    final Instant silvester2013 = Instant.parse("2013-12-31T00:00:00Z");
    final Instant jdk8Release = Instant.parse("2014-03-18T00:00:00Z");

    // Vergleichswerte errechnen
    System.out.println("Christmas -> Silvester:      " +
                       Duration.between(christmas2013, silvester2013));
    System.out.println("Silvester -> JDK 8 Release:  " +
                       Duration.between(silvester2013, jdk8Release));

    // Berechnungen
    final Instant calcSilvester_1 = christmas2013.plus(Duration.ofDays(7));
    final Instant calcSilvester_2 = christmas2013.plus(7, ChronoUnit.DAYS);

    System.out.println(calcSilvester_1);
    System.out.println(calcSilvester_2);
}
```

Führt man das obige Programm DURATIONCALCULATIONEXAMPLE aus, so kommt es zu folgenden Ausgaben:

```
Christmas -> Silvester:      PT168H
Silvester -> JDK 8 Release:  PT1848H
2013-12-31T00:00:00Z
2013-12-31T00:00:00Z
```

Nach diesen einfachen Differenzberechnungen zwischen zwei Datumswerten könnten wir beispielsweise einige Wochen oder Monate in die Vergangenheit oder Zukunft springen wollen. Dazu wären Methoden wie `ofWeeks(long)` bzw. `ofMonths(long)` praktisch. Diese existieren jedoch für `Instants` nicht. Schauen wir uns das genauer an: Während die fehlende Bereitstellung einer Methode von `ofWeeks(long)` sich noch recht gut durch eigene Berechnungen und der Nutzung von `ofDays(long)` realisieren lässt, wird dies für Monate ohne `ofMonths(long)` schwieriger. Das wirkt umständlich und man entdeckt möglicherweise die Methode `plus(long, TemporalUnit)`. Diese scheint für unsere Berechnungen sehr praktisch zu sein, um Wochen oder Monate in die Zukunft zu springen. Setzen wir diese Methode einfach einmal ein:

```
public static void main(final String[] args)
{
    // Achtung: kein Duration ofWeeks(long) oder ofMonths(long)
    final Instant calcSilvester_3 = christmas2013.plus(1, ChronoUnit.WEEKS);
    final Instant calcJdk8Release = silvester2013.plus(3, ChronoUnit.MONTHS).
                                                  plus(Duration.ofDays(18));

    System.out.println(calcSilvester_3);
    System.out.println(calcJdk8Release);
}
```

Wenn Sie das obige Programm DURATIONSPECIALEXAMPLE ausführen, werden jedoch statt der gewünschten Berechnungen Exceptions folgender Form ausgelöst:

```
Exception in thread "main"
java.time.temporal.UnsupportedTemporalTypeException:
      Unsupported unit: Weeks
```

```
    at java.time.Instant.plus(Instant.java:867)
```

Zur Definition einer `Duration` können keine Zeiteinheiten genutzt werden, die sich nicht präzise durch Stunden, Minuten usw. ausdrücken lassen. Man könnte sich fragen: Wir haben doch aber eine `Duration` für die gewünschten Zeiträume basierend auf `Instant`s berechnen können. Wieso war das möglich? Die Antwort ist ganz einfach: Weil wir hier fixe Werte vorliegen haben und somit die Differenz dazwischen eindeutig zu bestimmen war. Andersherum gilt das jedoch nicht: Die abstrakte Angabe von einer Woche oder einem Monat besitzt nämlich kein exaktes Äquivalent in Form einer fixen Zeitspanne. Hier steht die Modellierung in Maschinenzeit in Konflikt mit der komplexeren Wirklichkeit. Später werden wir als Abhilfe die Klasse `Period` kennenlernen.

13.2.4 Die Klassen `LocalDate`, `LocalTime` und `LocalDateTime`

Die Klasse `java.time.LocalDate` repräsentiert eine reine Datumsangabe, also ohne Zeitinformationen. Ein `LocalDate` ist eine Kombination aus Jahr, Monat und Tag. Mit der Klasse `java.time.LocalTime` wird eine Zeitangabe ohne Datumsangabe modelliert, z. B. 18:00 h. Die Klasse `java.time.LocalDateTime` ist eine Kombination aus beiden. Folgendes Programm zeigt die Klassen im Einsatz:

```
public static void main(final String[] args)
{
    final LocalDate michasBirthday = LocalDate.of(1971, Month.FEBRUARY, 7);
    final LocalDate barbarasBirthday = michasBirthday.plusYears(2).
                                                      plusMonths(1).
                                                      plusDays(17);
    final LocalDate lastDayInFebruary = michasBirthday.with(TemporalAdjusters.
                                                            lastDayOfMonth());

    System.out.println("michasBirthday:     " + michasBirthday);
    System.out.println("barbarasBirthday:   " + barbarasBirthday);
    System.out.println("lastDayInFebruary:  " + lastDayInFebruary);

    final LocalTime atTen = LocalTime.of(10,00,00);
    final LocalTime tenFifteen = atTen.plusMinutes(15);
    final LocalTime breakfastTime = tenFifteen.minusHours(2);

    System.out.println("atTen:              " + atTen);
    System.out.println("tenFifteen:         " + tenFifteen);
    System.out.println("breakfastTime:      " + breakfastTime);

    final LocalDateTime jdk8Release = LocalDateTime.of(2014,  3, 18, 8, 30);
    System.out.println("jdk8Release:        " + jdk8Release);
    System.out.printf("jdk8Release:   %s.%s.%s\n", jdk8Release.getDayOfMonth(),
                                                   jdk8Release.getMonthValue(),
                                                   jdk8Release.getYear());
}
```

Im Listing sehen wir verschiedene Berechnungen mithilfe von `plusXYZ()`- sowie `minusXYZ()`-Methoden. Darüber hinaus haben wir die Utility-Klasse

TemporalAdjusters genutzt, in der unterschiedlichste Hilfsmethoden definiert sind. Dies ist etwa die Methode lastDayOfMonth() zur Berechnung des letzten Tags im Monat. Damit berechnen wir im Beispiel den letzten Tag des Monats Februar im Jahr 1971. Führt man das Programm LOCALDATEANDTIMEEXAMPLE aus, so kommt es zu folgenden Konsolenausgaben:

```
michasBirthday:     1971-02-07
barbarasBirthday:   1973-03-24
lastDayInFebruary:  1971-02-28
atTen:              10:00
tenFifteen:         10:15
breakfastTime:      08:15
jdk8Release:        2014-03-18T08:30
jdk8Release:   18.3.2014
```

13.2.5 Die Aufzählungen DayOfWeek und Month

Sowohl java.time.DayOfWeek als auch java.time.Month sind Aufzählungen, deren Einsatz einerseits den Sourcecode lesbarer macht und andererseits auch einfache Fehler vermeidet, weil man typsichere Konstanten verwendet. Bei Nutzung der alten APIs konnten durch nicht konsistente 0- und 1-basierte Angaben Probleme entstehen. Dass sich der Einsatz von Konstanten vorteilhaft auswirken kann, haben wir im vorangegangenen Beispiel für die Angabe des Monats Februar bereits ansatzweise kennengelernt. Im Calendar-API wusste man – ohne Blick in das Javadoc des APIs – nie so genau, ob Februar nun dem Wert 1 oder 2 entsprach. Unter anderem haben wir mit diesem Problem eingangs (schmerzhaft) Erfahrung gemacht, als wir ein Datum als Date modellieren wollten.

Neben der Typsicherheit bieten die neuen Aufzählungstypen den Vorteil, dass man Berechnungen mit ihnen durchführen kann. Nachfolgend demonstriere ich dies, indem ich zu einem Sonntag 5 Tage hinzu addiere und zum Februar 13 Monate:

```
public static void main(final String[] args)
{
    final DayOfWeek sunday = DayOfWeek.SUNDAY;
    final Month february = Month.FEBRUARY;

    System.out.println(sunday.plus(5));
    System.out.println(february.plus(13));
}
```

Wie erwartet, landet man an einem Freitag bzw. im März, wenn man das Progamm MONTHANDDAYOFWEEKEXAMPLE ausführt:

```
FRIDAY
MARCH
```

13.2.6 Die Klassen YearMonth, MonthDay und Year

Die schon zuvor beschriebenen `Instant`-Objekte sind zur Modellierung von wiederkehrenden Datumswerten, etwa Geburtstagen oder anderen Jahrestagen, nicht besonders gut geeignet. Das liegt daran, dass man hierzu »unvollständige Zeitangaben« benötigt, etwa Datumsangaben ohne Uhrzeit, Zeitangaben ohne Datum oder Datumsangaben ohne Jahresangaben. Wieso ist das wünschenswert?

Wie eingangs erwähnt, hat die Darstellung von Zeitangaben in Millisekunden, die sehr hilfreich für die Verarbeitung mit Computern ist, recht wenig mit der menschlichen Denkweise und Wahrnehmung von Zeit zu tun. Menschen denken in den Konzepten von Zeitabschnitten oder wiederkehrenden Datumsangaben, etwa 24.12. für Heiligabend, 31.12. für Silvester usw., und auch in Uhrzeiten ohne Bezug zu einem Datum, etwa 18.00 h Feierabend, oder als Kombination: Dienstags und Donnerstags 19 Uhr Karate-Training.[2] Wollten wir so etwas mit dem bisher existierenden API ausdrücken, wäre das recht schwierig. Schauen wir nun auf die neuen Möglichkeiten. Statt einer vollständigen Angabe aus Jahr, Monat und Tag kann man sich auch Kombinationen aus Jahr und Monat, Monat und Tag sowie einfach nur Jahr vorstellen, um gewisse Datumsangaben zu modellieren. Für diese Zwecke wurden im Package `java.time` die Klassen `YearMonth`, `MonthDay` sowie `Year` eingeführt, die wir im folgenden Listing nutzen:

```
public static void main(final String[] args)
{
    // YearMonth: Demonstration jeweils mit und ohne Konstanten
    final YearMonth yearMonth = YearMonth.of(2014, 2);
    final YearMonth february2014 = YearMonth.of(2014, Month.FEBRUARY);

    // MonthDay: Achtung, ISO-Format mit der Reihenfolge: Monat, Tag
    final int dayOfBirthday = 7;
    final MonthDay monthDay1 = MonthDay.of(2, dayOfBirthday);
    final MonthDay monthDay2 = MonthDay.of(Month.FEBRUARY, dayOfBirthday);

    // Year
    final Year year = Year.of(2012);

    System.out.println("YearMonth: " + february2014);
    System.out.println("MonthDay:  " + monthDay2);
    System.out.println("Year:      " + year + " / isLeap? " + year.isLeap());
}
```

Das Progamm YEARANDMOREEXAMPLE produziert folgende Konsolenausgaben:

```
YearMonth: 2014-02
MonthDay:  –02-07
Year:      2012 / isLeap? true
```

Anhand der Ausgaben sieht man verschiedene Notationsformen und insbesondere auch, dass man von der objektorientierten Umsetzung profitiert: Somit lässt sich etwa per Aufruf von `isLeap()` prüfen, ob ein Jahr ein Schaltjahr ist.

13.2.7 Die Klasse `Period`

Ähnlich wie die Klasse `Duration` modelliert die Klasse `java.time.Period` einen Zeitabschnitt. Beispiele sind etwa »2 Monate« oder »3 Tage«. Diese Art der Darstellung ist oftmals einfacher zu handhaben als eine korrespondierende Repräsentation in Nanosekunden oder Millisekunden. Konstruieren wir ein paar Instanzen von `Period`:

```
public static void main(final String[] args)
{
    // Erzeuge ein Period-Objekt mit 1 Jahr, 6 Monaten und 3 Tagen
    final Period oneYear_sixMonths_ThreeDays =
Period.ofYears(1).withMonths(6).
                                                        withDays(3);

    // Chaining von of() arbeitet anders, als man es eventuell erwartet!
    // Ergibt ein Period-Objekt mit 3 Tagen statt 2 Monate, 1 Woche und 3
Tagen
    final Period twoMonths_OneWeek_ThreeDays = Period.ofMonths(2).ofWeeks(1).
                                                         ofDays(3);

    final Period twoMonths_TenDays = Period.ofMonths(2).withDays(10);
    final Period sevenWeeks = Period.ofWeeks(7);
    final Period threeDays = Period.ofDays(3);

    System.out.println("1 year 6 months …:   " + oneYear_sixMonths_ThreeDays);
    System.out.println("Surprise just 3 days:  " +
twoMonths_OneWeek_ThreeDays);
    System.out.println("2 months 10 days:      " + twoMonths_TenDays);
    System.out.println("sevenWeeks:            " + sevenWeeks);
    System.out.println("threeDays:             " + threeDays);
}
```

Startet man das Programm PERIODEXAMPLE, so wird Folgendes ausgegeben:

```
1 year 6 months …:   P1Y6M3D
Surprise just 3 days:  P3D
2 month 10 days:       P2M10D
sevenWeeks:            P49D
```

Anhand des Beispiels und dessen Ausgaben lernen wir verschiedene Besonderheiten der Klasse `Period` kennen. Zwar lassen sich Aufrufe von `of()` hintereinander ausführen, es gewinnt aber der zuletzt aufgerufene.[3] Man kann also auf diese Weise keine Zeiträume kombinieren, sondern legt einen initialen Zeitraum fest. Sollen weitere Zeitabschnitte hinzugefügt werden, so muss man dafür verschiedene `with()`-Methoden nutzen. Dabei wird ein Implementierungsdetail sichtbar. Die Klasse `Period` verwaltet drei Einzelwerte, nämlich für Jahre, Monate und Tage, aber eben nicht für Wochen. Daher gibt es keine Methode `withWeeks()`, sondern nur eine `ofWeeks()`, die intern eine Umrechnung in Tage vornimmt.

Nachdem wir nun einen ersten Eindruck gewonnen haben, schauen wir, wie einfach und lesbar Berechnungen mit dem neuen Date And Time API gestaltet werden können. Ausgehend vom 7.2.1971 10:11 springen wir 31 Tage sowie einen Monat in die Zukunft. Außerdem nutzen wir die aktuelle Uhrzeit und addieren fünf Minuten sowie alternativ sieben Stunden hinzu. Das Ganze realisieren wir wie folgt:

```
public static void main(final String[] args)
{
```

```
    final LocalDateTime start = LocalDateTime.of(1971, 2, 7, 10, 11);

    final Period thirtyOneDays = Period.ofDays(31);
    System.out.println("7.2.1971 + 31 Tage:  " + start.plus(thirtyOneDays));

    final Period oneMonth = Period.ofMonths(1);
    System.out.println("7.2.1971 + 1 Monat:  " + start.plus(oneMonth));

    final LocalTime now = LocalTime.now();
    System.out.println("now:             " + now);

    final LocalTime fiveMinutesLater = now.plus(5, ChronoUnit.MINUTES);
    System.out.println("now + 5 min:     " + fiveMinutesLater);

    final LocalTime sevenHoursLater = now.plusHours(7);
    System.out.println("now + 7 hours:   " + sevenHoursLater);
}
```

Das Programm PERIODCALCULATIONEXAMPLE erzeugt z. B. folgende Ausgaben:

```
7.2.1971 + 31 Tage:  1971-03-10T10:11
7.2.1971 + 1 Monat:  1971-03-07T10:11
now:             20:38:29.937
now + 5 min:     20:43:29.937
now + 7 hours:   03:38:29.937
```

Hintergrundwissen: Warum gibt es `Duration` und `Period`?

Zunächst mag die Definition von Zeitabschnitten durch zweierlei Klassen verwundern – doch die Intention der beiden Klassen ist unterschiedlich. Während `Duration` einen Zeitabschnitt in Form von Sekunden modelliert, ist die Klasse `Period` eher zur Modellierung von Zeitabschnitten im Bereich von Tagen, Monaten oder gar Jahren gedacht.

Aufgrund ihrer Ausrichtung auf Sekunden modelliert die Klasse `Duration` etwa einen Tag als exakt 24 Stunden, also 24 * 60 * 60 = 86.400 Sekunden. Die Klasse `Period` arbeitet dagegen auf eher konzeptioneller Ebene mit Tagen und Monaten unabhängig von der exakten Länge in Sekunden.

Den daraus entstehenden fundamentalen Unterschied zwischen beiden Modellierungen kann man sich am besten im Zusammenhang mit Winter- und Sommerzeit klarmachen: Es gibt Tage, die 23 Stunden lang sind, und solche, die eine Dauer von 25 Stunden besitzen. Wird die Länge eines Tags jedoch fix als 24 Stunden angenommen und dieser Wert wiederum in Form einer Zeitspanne in Sekunden repräsentiert, so kommt es zu Berechnungsfehlern, wenn an kürzeren oder längeren Tagen ein Tag in die Zukunft oder Vergangenheit »gesprungen« werden soll: Man bewegt sich somit entweder eine Stunde zu wenig oder zu viel in die Zukunft. Nutzt man die Klasse `Period`, muss man sich um diese Details nicht kümmern, da das »Konzept Tag« und nicht dessen Pendant in Sekunden (eine simple ganze Zahl) zum Einsatz kommt.

13.2.8 Die Klasse `Clock`

In einigen technischen Anwendungsfällen benötigt man Zugriff auf Millisekundenangaben. Früher hat man dann Aufrufe von `System.currentTimeMillis()` genutzt, um die aktuelle Zeit in Millisekunden seit dem 1.1.1970 zu ermitteln. Nun verwendet man die Klasse `java.time.Clock` und schreibt etwa Folgendes:

```
public static void main( final String[] args)
{
    // Basis UTC
    final Clock clockUTC = Clock.systemUTC();
    System.out.println(clockUTC);
    printClockMillis(clockUTC);

    // Basis Default-Zeitzone
    final Clock clockDefaultZone = Clock.systemDefaultZone();
    System.out.println(clockDefaultZone);
    printClockMillis(clockDefaultZone);
}

private static void printClockMillis(final Clock clock)
{
    final long currentTime = clock.millis();
    System.out.println(currentTime);
}
```

Führen wir das Programm CLOCKEXAMPLE aus, so lassen sich an den Ausgaben die unterschiedlichen Zeitzonen erkennen:

```
SystemClock[Z]
1395495448297
SystemClock[Europe/Berlin]
1395495448365
```

13.2.9 Die Klasse `ZonedDateTime`

Neben der bereits vorgestellten Klasse `LocalDateTime` zur Repräsentation von Datum und Uhrzeit ohne Zeitzonenbezug existiert eine korrespondierende Klasse `java.time.ZonedDateTime`. Diese besitzt eine zugeordnete Zeitzone und berücksichtigt bei Berechnungen nicht nur die Zeitzone, sondern auch die Auswirkungen von Winter- und Sommerzeit. Um die aktuelle Zeit als `ZonedDateTime` zu ermitteln, kann man die Methode `now()` nutzen. Es existieren weitere Methoden, etwa um die Zeitzone und andere Werte abzufragen bzw. Instanzen von `ZonedDateTime` mit geänderter Wertebelegung durch Aufruf von `withXYZ()`-Methoden zu erzeugen. Interessant und etwas schade ist, dass man beim Aufruf von `withMonth(int)` einen `int`-Wert und keine Monatskonstante übergeben muss. Zur besseren Lesbarkeit empfiehlt sich, die Konstanten zu verwenden und auf deren `int`-Wert per `getValue()` zuzugreifen.

Nachfolgend sind verschiedene Beispiele für Berechnungen mit der Klasse `ZonedDateTime` gezeigt:

```
public static void main(final String[] args)
{
```

```
    // Aktuelle Zeit als ZonedDateTime-Objekt ermitteln
    final ZonedDateTime now = ZonedDateTime.now();

    // Die Uhrzeit ändern und in neuem Objekt speichern
    final ZonedDateTime nowButChangedTime = now.withHour(11).withMinute(44);

    // Neues Objekt mit verändertem Datum erzeugen
    final ZonedDateTime dateAndTime = nowButChangedTime.withYear(2008).
                                                        withMonth(9).
                                                        withDayOfMonth(29);

    // Einsatz einer Monatskonstanten
    final ZonedDateTime dateAndTime2 = nowButChangedTime.withYear(2008).
                                       withMonth( Month.SEPTEMBER.getValue()).
                                       withDayOfMonth(29);

    System.out.println("now:           " + now);
    System.out.println("-> 11:44:      " + nowButChangedTime);
    System.out.println("-> 29.9.2008:  " + dateAndTime);
    System.out.println("-> 29.9.2008:  " + dateAndTime2);
}
```

Führt man das Programm ZONEDDATETIMEEXAMPLE aus, so kommt es zu den nachfolgenden Ausgaben. Diese zeigen insbesondere den Einfluss von Winter- und Sommerzeit, wodurch im September 2008 die Abweichung von +02:00 angegeben wird:

```
now:           2014-03-20T23:15:01.488+01:00[Europe/Berlin]
-> 11:44:      2014-03-20T11:44:01.488+01:00[Europe/Berlin]
-> 29.9.2008:  2008-09-29T11:44:01.488+02:00[Europe/Berlin]
-> 29.9.2008:  2008-09-29T11:44:01.488+02:00[Europe/Berlin]
```

13.2.10 Beispiel: Berechnung einer Zeitdifferenz

Eingangs zeigte ich, dass die Berechnung der Differenz zweier Zeitwerte schon einige recht merkwürdige Besonderheiten des alten APIs offenbart. Insbesondere arbeitet man immer auf einer niedrigen Abstraktionsebene, in der die Klasse `Date` lediglich eine verzuckerte Version eines `long`-Werts ist. Auch das Setzen einer Zeitzone zum Erzielen von korrekten Berechnungsergebnissen ist wenig intuitiv und wirkt etwas befremdlich.

Wenn wir nun das Beispiel aufgreifen, so bietet sich hier der Einsatz der Klasse `LocalTime` geradezu an, weil diese einen Zeitpunkt modelliert. Mithilfe von `parse()`-Methoden kann man die in Strings gegebenen Zeitangaben leicht in `LocalDate`-Objekte umwandeln. Auch die Zeitdifferenz lässt sich einfach mithilfe eines Aufrufs von `between(Temporal, Temporal)` in Form eines `Duration`-Objekts ermitteln, von dem man dann die Dauer in Sekunden per `getSeconds()` erfragt. Aus diesem konstruiert man wiederum mit `ofSecondOfDay(long)` ein korrespondierendes `LocalTime`-Objekt, das man mit einem passend mit dem Muster "`HH:mm:ss`" konfigurierten `DateTimeFormatter` ausgibt:

```
public static void main(final String[] args) throws ParseException
{
```

```
    // Unterschied 1 Stunde, 10 Minuten und 20 Sekunden
    final String startTimeAsString = "10:10:10";
    final String endTimeAsString  = "11:20:30";

    // Umwandlung in LocalTime-Objekte
    final LocalTime startTime = LocalTime.parse (startTimeAsString);
    final LocalTime endTime = LocalTime.parse(endTimeAsString);

    // Berechne  Differenz als Duration und in Sekunden
    final Duration duration = Duration.between (startTime, endTime);
    final long durationInSecs = duration.getSeconds();

    System.out.println("duration = " + duration + " / secs = " +
durationInSecs);

    // Umwandlung in LocalTime und Ausgabe
    final DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern
("HH:mm:ss");
    final LocalTime asLocalTime = LocalTime.ofSecondOfDay(durationInSecs);
    System.out.println("durationInHHmmss = " + dateFormat.format(asLocalTime));
}
```

Führt man das obige Programm NEWAPIDURATIONCALCULATIONEXAMPLE aus, so erkennt man, dass man die Berechnungen mit dem neuen Date And Time API einfach und überraschungsfrei durchführen kann:

```
duration = PT1H10M20S in secs = 4220
durationInHHmmss = 01:10:20
```

13.2.11 Interoperabilität mit Legacy-Code

Zum Abschluss unserer Entdeckungsreise mit dem neuen Date And Time API wollen wir noch erkunden, wie man bestehenden Sourcecode stückweise migrieren kann. Insbesondere ist von Interesse, welche Klassen sich von der Idee her im alten JDK und im neuen Date And Time API entsprechen und wie man zwischen Instanzen der alten und neuen Klassen hin und her konvertieren kann. Wie eingangs schon erwähnt, besteht eine Analogie zwischen den Klassen `Date` und `Instant`. Die Klasse `GregorianCalendar` kann man am ehesten mit der Klasse `ZonedDateTime` vergleichen. Folgende Aufzählung nennt passende Konvertierungsmethoden:

- `Date.from(Instant)`
- `Date.toInstant()`
- `Calendar.toInstant()`
- `GregorianCalendar.toZonedDateTime()`
- `GregorianCalendar.from(ZonedDateTime)`

Im nachfolgenden Listing zeige ich einige der Methoden im Einsatz und darüber hinaus noch die beiden Methoden `ofInstant(Instant, ZoneId)`, die sowohl in der Klasse `LocalDateTime` und auch `ZonedDateTime` existieren:

```
public static void main(final String[] args)
{
    //  Berechnungen basierend auf Date
    final Date now = new Date();
    final Instant nowAsInstant = now.toInstant();

    final ZoneId systemZone = ZoneId.systemDefault();
    final LocalDateTime localDateTime = LocalDateTime.ofInstant(nowAsInstant,
                                                                systemZone);
    final ZoneId zoneCalifornia = ZoneId.of("America/Los_Angeles");
    final ZonedDateTime zonedDateTime = ZonedDateTime.ofInstant(nowAsInstant,
                                                                zoneCalifornia);

    System.out.println("LocalDateTime:  " + localDateTime);
    System.out.println("ZonedDateTime:  " + zonedDateTime);

    //  Berechnungen basierend auf Calender
    final GregorianCalendar nowAsCalendar = new GregorianCalendar();
    final ZonedDateTime nowAszonedDateTime = nowAsCalendar.toZonedDateTime();

    final Instant instant = nowAszonedDateTime.toInstant();
    System.out.println("Instant:        " + instant);
}
```

Das Programm LEGACYEXAMPLE erzeugt in etwa folgende Ausgaben:

```
LocalDateTime:  2014-05-22T21:08:36.089
ZonedDateTime:  2014-05-22T12:08:36.089-07:00[America/Los_Angeles]
Instant:        2014-05-22T19:08:36.172Z
```

Im obigen Listing sehen wir neben der Konvertierung in und aus Klassen des alten APIs auch noch die Verarbeitung von Zeitzonen mithilfe der Klasse `java.time.ZoneId`. Etwas unschön ist, dass man hier Strings als IDs nutzt. Praktischerweise erhält man die gültigen Zeitzonen-IDs durch Aufruf von `ZoneId.getAvailableZoneIds()`.

13.3 Fazit

Wir haben uns nun einen ersten Überblick über das neue Date And Time API zur Datumsverarbeitung verschafft, das jetzt in JDK 8 enthalten ist. Wie bereits erwähnt, sind beim Entwurf die Erfahrungen mit der Bibliothek Joda-Time eingeflossen. Viele Ideen wurden daraus entnommen und weiterentwickelt. Insgesamt macht die Arbeit mit dem neuen API durchaus Freude.

Beim Kennenlernen der Klassen anhand einfacher Beispiele haben wir gesehen, dass sich viele Aufgabenstellungen unter Zuhilfenahme des neuen Date And Time APIs einfach realisieren lassen. Es wurde auch deutlich, dass dort eine Vielzahl neuer, für eine spezifische Aufgabe spezialisierter Klassen existiert, die intuitiv zu verwenden sind. Das zeigte sich insbesondere am abschließenden Beispiel der Berechnung einer Zeitdauer: Zu Beginn dieses Kapitels haben wir eine Variante mit den herkömmlichen Klassen implementiert und daran die resultierende Komplexität sowie einige Fallstricke kennengelernt.

14 Einstieg JavaFX 8

In diesem Kapitel stelle ich Ihnen mit JavaFX das aktuellste und modernste GUI-Framework von Java vor, das Swing als Oberflächentechnologie ablösen soll. Dieser Schritt ist notwendig, weil Java in den letzten Jahren im Desktop-Bereich deutlich an Einfluss an die aufstrebende Konkurrenz aus dem Microsoft-Lager (.NET/WPF) aber auch an immer populärer werdende Webapplikationen verloren hat. JavaFX tritt nun an, die GUI-Programmierung zu erleichtern und attraktive GUIs entwickeln zu können.

Weil derzeit die Literatur zu JavaFX doch noch recht überschaubar ist, ganz besonders bei deutschsprachigen Titeln, beschreibe ich Ihnen zunächst die Grundlagen von JavaFX in Version 2.2. Das ist die Version, in der JavaFX im JDK 7 ausgeliefert wird. Abschnitt 14.1 beginnt mit einer Einführung in JavaFX. Danach schauen wir uns in Abschnitt 14.2 die deklarative Gestaltung von GUIs an, wodurch eine gute Trennung von Design und Funktionalität möglich wird. In Abschnitt 14.3 lernen wir die Vorzüge von JavaFX in Form von Effekten und Animation u. v. m. kennen. Anhand meiner Ausführungen erhalten Sie einen fundierten Überblick und eine gute Basis sowohl für eigene Experimente als auch zum Nachvollziehen der Beschreibungen zu den Neuerungen von JavaFX 8 in Abschnitt 14.4.

Weil ich nicht jedes Detail beschreiben kann, gehe ich davon aus, dass Sie sich schon etwas mit GUIs und Swing beschäftigt haben, um dem Text inhaltlich gut folgen zu können. Einen Einstieg in die Grafikprogrammierung mit Swing finden Sie in Kapitel 9.

14.1 Einführung – JavaFX im Überblick

In diesem Abschnitt gebe ich Ihnen einen kurzen Überblick und Einstieg in JavaFX. Wir lernen zunächst einige Grundbegriffe und dann eine einfache Form von Action Handling kennen. Anschließend schauen wir uns das Layoutmanagement an.

14.1.1 Motivation für JavaFX und Historisches

Neben .NET-Anwendungen bieten mittlerweile auch viele Webanwendungen häufig moderne, ansprechende GUIs mit hohem Bedienkomfort. Diesbezüglich wurden Webanwendungen früher aufgrund ihrer rudimentären Interaktivität und insbesondere wegen des fehlenden Komforts von Desktop-Entwicklern oftmals belächelt. Mittlerweile hat sich die Situation aber geändert: Webanwendungen haben beträchtliche Fortschritte gemacht und einige klassische Desktop-Applikationen vom Bedienkomfort her mitunter sogar überholt. Erschwerend kommt hinzu, dass in Webanwendungen grafische Effekte und Animationen bereits zum guten Ton

gehören, wodurch bei vielen Entwicklern und vor allem Nutzern der Wunsch entsteht, Derartiges auch zur Verbesserung der Benutzbarkeit in Desktop-Anwendungen einsetzen zu können.

Aus dem Gesagten wird klar, dass sich eine ernsthafte Konkurrenz zu Java-Desktop-GUIs entwickelt hat. Der schwindende Einfluss von Java im GUI-Bereich war wohl ein wichtiger Grund, weshalb JavaFX entwickelt wurde. Es wurde zunächst als Skriptsprache entworfen, die die grafische Gestaltung ansprechender GUIs z. B. durch Effekte sowie Animationen unterstützt. Die Programmierung mithilfe von Skriptcode stellte jedoch für viele Swing-Entwickler eine gewisse Hürde dar, weil zuerst die Skriptsprache erlernt werden musste und außerdem keine gute Integration in das Java-API stattfand. In Form der Skriptsprache hat sich JavaFX nie wirklich durchgesetzt. Nach diesem erfolglosen Versuch wurde es zunächst ruhig um JavaFX, bis dann Ende 2011 auf der Java One, einer bedeutenden Java-Konferenz in San Francisco, die Version 2 von JavaFX vorgestellt wurde. Mit JavaFX 2 findet eine Abkehr von der Skriptsprache statt und es wird eine Integration in das Java-API vorgenommen. Nachdem JavaFX längere Zeit eher ein Schattendasein geführt hat, wird es nun von Oracle stark gefördert sowie aktiv weiterentwickelt. Aktuell ist JavaFX in Version 8. Diese enthält diverse Neuerungen und ist Bestandteil von JDK 8.

14.1.2 Grundsätzliche Konzepte

In diesem Abschnitt lernen wir wichtige Grundlagen von JavaFX anhand einer Hello-World-Applikation kennen. Es soll ein Text in einem Fenster ausgeben werden. Bevor wir jedoch mit der Implementierung der Anwendung beginnen, möchte ich auf folgende zentrale Hauptbestandteile einer JavaFX-Applikation eingehen:

- **Stage** — Die sogenannte Stage vom Typ `javafx.stage.Stage` bildet die »Bühne« oder den Rahmen für eine JavaFX-Applikation und stellt die Verbindung zum genutzten Betriebssystem dar – vergleichbar mit einem `JFrame` in Swing.
- **Scene** — Eine sogenannte Scene ist vom Typ `javafx.scene.Scene`. Sie entspricht grob der Containerkomponente `ContentPane` in Swing, und ist dasjenige Element eines JavaFX-GUIs, in dem alle Bedienelemente platziert werden (eventuell indirekt durch Verschachtelungen ähnlich zu den Containern in Swing).
- **Scenegraph** und **Nodes** — Ähnlich wie bei AWT, SWT oder Swing ist auch bei JavaFX die Benutzeroberfläche hierarchisch organisiert: Der Inhalt einer `Scene` ist ein Baum, bestehend aus Knoten mit dem Basistyp `javafx.scene.Node`. Man spricht bei dem Baum auch vom ***Scenegraph***. Die Anordnung der Bedienelemente (`Nodes`) wird durch spezielle Container ähnlich zu den `LayoutManagern` in Swing bestimmt. Diese Container besitzen selbst wieder den Basistyp `Node` und sind Bestandteil des Scenegraphs.

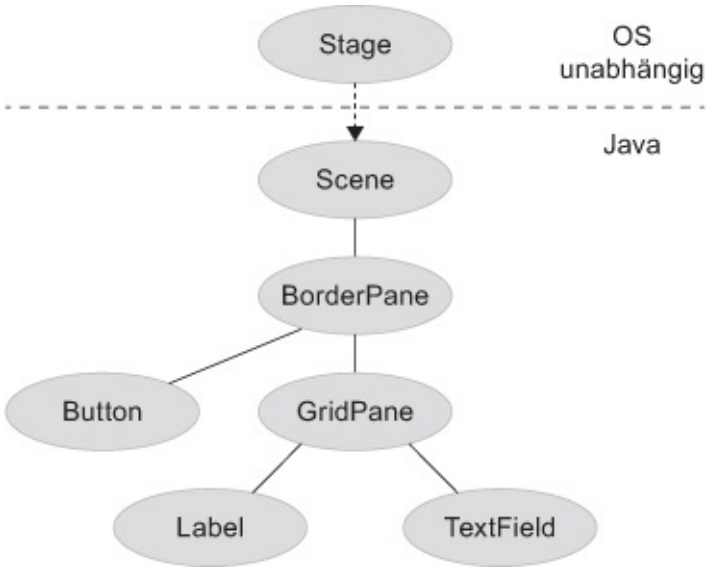


Abbildung 14-1 *Verbindung von Stage, Scene und Node*

Um einen ersten Eindruck davon zu gewinnen, wie man JavaFX-Applikationen erstellt, realisieren wir nun die Hello-World-Applikation. Praktischerweise gibt es im JDK die Basisklasse `javafx.application.Application`, die diverse Funktionalitäten bereitstellt, sodass lediglich noch die abstrakte Methode `start(Stage)` implementiert werden muss. Dort wird das GUI konstruiert. Als Container nutzen wir den Typ `javafx.scene.layout.StackPane`, in der die Bedienelemente wie in einem Kartenstapel übereinander angeordnet werden. Zur Darstellung eines Textes fügen wir dort ein Label vom Typ `javafx.scene.control.Label` folgendermaßen ein:

```
import javafx.application.Application;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class FirstJavaFxExample extends Application
{
    @Override
    public void start(final Stage stage) throws Exception
    {
        final StackPane stackPane = new StackPane();

        // Label erzeugen und hinzufügen
        final Node labelNode = new Label("Hello JavaFX World!");
        stackPane.getChildren().add(labelNode);

        // Stage, Scene und Stackpane verbinden
        stage.setScene(new Scene(stackPane, 250, 75));
        // Titel und Resizable-Eigenschaft setzen
        stage.setTitle("FirstJavaFxExample");
        stage.setResizable(true);
        // Positionierung und Sichtbarkeit
        stage.centerOnScreen();
        stage.show();
    }
    // …
}
```

Weil wir im Listing diverse unbekannte Klassen sehen und die Aktionen zum Aufbau des GUIs in JavaFX-Applikationen in etwa immer einem ähnlichen Muster folgen, werden hier einmalig die `import`-Anweisungen gezeigt und zudem einführend die obigen Programmzeilen detaillierter beschrieben. Das `Label` ist eine Spezialisierung einer `Node`. Eine Instanz davon wird der `StackPane` hinzugefügt, indem über `getChildren()` die `Nodes` im Container ermittelt und dann über `add(Node)` erweitert werden. Anschließend dient die `StackPane` als Eingabe für die Konstruktion einer `Scene`, die dann der `Stage` per Methodenaufruf von `setScene(Scene)` zugewiesen wird. Für die `Stage` werden noch verschiedene Eigenschaften wie Titel und Größenveränderlichkeit gesetzt. Abschließend wird durch Aufrufe von `centerOnScreen()` sowie `show()` ein Fenster mit dem zuvor konstruierten Inhalt zentriert auf dem Bildschirm dargestellt. Führen wir die Applikation FIRSTJAVAFXEXAMPLE aus, so erscheint ein Fenster mit einem Text, etwa wie in Abbildung 14-2.

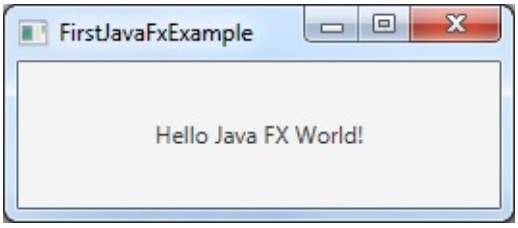


Abbildung 14-2 *Erste JavaFX-Applikation*

Zwei Dinge möchte ich noch explizit erwähnen:

1. Im Gegensatz zu Swing-Applikationen muss man als Anwender das Fenster bzw. die Stage nicht selbst erstellen, etwa per Konstruktoraufruf `new JFrame()`, sondern die `Stage` wird vom JavaFX automatisch erzeugt und an die Methode `start(Stage)` übergeben. Dadurch erreicht man eine gute Plattformunabängigkeit, weil der Inhalt der `Scene` und alle dort dargestellten Elemente den Basistyp `Node` besitzen und so eine Abstraktion von den tatsächlichen Bedienelementen des Betriebssystems ermöglichen.
2. Wiederum im Gegensatz zu Swing nimmt JavaFX viele Schritte beim Starten der Applikation ab und sorgt auf diese Weise für einen Thread-sicheren Start. Dabei wird die Methode `start(Stage)` automatisch aufgerufen, wenn wir die Methode `launch()` ausführen. Dies geschieht in der `main()`-Methode wie folgt:

```
public static void main(final String[] args)
{
    launch(args);
}
```

Erweiterung des Hello-World-Beispiels um Action Handling

Das vorherige Beispiel hat einen einführenden Überblick über die grundlegenden Zusammenhänge beim Entwurf einer JavaFX-Applikation gegeben. Nun wollen wir Benutzerinteraktionen unterstützen und einen ersten Eindruck von der Verarbeitung bekommen. Dazu ändern wir das Beispiel folgendermaßen leicht ab: Statt eines

Labels nutzen wir einen javafx.scene.control.Button, den wir in einer javafx.scene.layout.FlowPane platzieren. Zur Interaktion registrieren wir einen javafx.event.EventHandler<javafx.event.ActionEvent>. Als Reaktion auf das Drücken des Buttons erzeugen wir neue Labels, die wir der FlowPane dynamisch hinzufügen. Das Beschriebene realisieren wir wie folgt:

```
@Override
public void start(final Stage stage)
{
    final Button button = new Button();
    button.setText("Add 'Hello World' Label");

    final FlowPane pane = new FlowPane();
    pane.getChildren().add(button);

    // EventHandler registrieren
    button.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(final ActionEvent event)
        {
            pane.getChildren().add(new Label("- Hello World! -"));
        }
    });

    stage.setScene(new Scene(pane, 400, 100));
    stage.setTitle("JavaFxActionHandlingExample");
    stage.setResizable(true);
    stage.centerOnScreen();
    stage.show();
}
```

An diesem Beispiel wird neben dem Action Handling ein dynamisch veränderliches Layout gezeigt. Das Programm JAVAFXACTIONHANDLINGEXAMPLE produziert nach ein paar Klicks auf den Button eine Ausgabe ähnlich zu der in Abbildung 14-3.

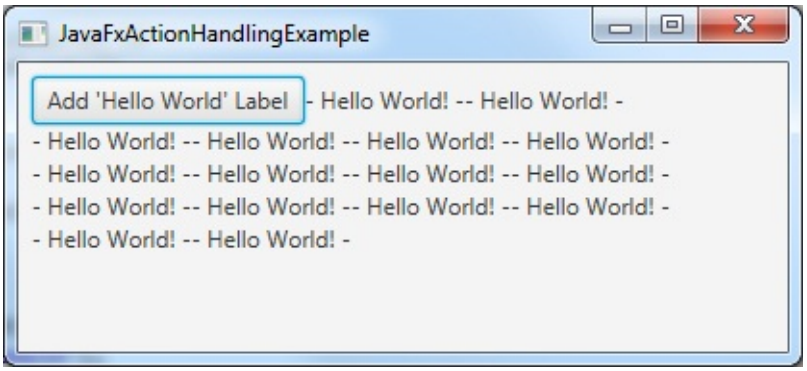


Abbildung 14-3 *JavaFX und Action Handling*

Relativ schnell wird diese Applikation langweilig und wir drücken das Schließkreuz. Ganz natürlich wird die Applikation dadurch beendet. Tatsächlich könnte uns das als praktisches Feature auffallen, denn in Swing wurde die Applikation nur dann beendet, wenn man eine passende Default-On-Close-Operation oder einen speziellen java.awt.event.WindowListener registriert hatte.

Wenn Sie bereits Erfahrung mit Swing haben, ist Ihnen bestimmt positiv aufgefallen, dass JavaFX das Layout der Applikation automatisch aktualisiert und dies

nicht wie bei Swing einige Aktionen erfordert. Das führt uns zum Thema Layoutmanagement.

14.1.3 Layoutmanagement

In den bisherigen Beispielen haben wir bereits zwei verschiedene Layout-Containerkomponenten von JavaFX kennengelernt, die folgende Ausrichtungen bereitstellen:

- `StackPane` – Die `StackPane` platziert die einzelnen `Nodes` wie in einem Stapel Karten übereinander. Damit ist es auf einfache Weise möglich, `Nodes` miteinander zu kombinieren. Sofern die einzelnen `Nodes` unterschiedlich groß oder teilweise transparent sind, kann man einen Überlagerungseffekt erzielen.
- `FlowPane` – Die `FlowPane` erinnert an das `FlowLayout` aus Swing. Hier werden `Nodes` sukzessive dargestellt und je nach Verlaufsrichtung horizontal oder vertikal umbrochen. Dies geschieht auf Grundlage der Breite bzw. Höhe der `FlowPane`.

In JavaFX ist eine Vielzahl weiterer Layouts definiert, unter anderem folgende:

- `BorderPane` – Die `javafx.scene.layout.BorderPane` ist sehr ähnlich zum `java.awt.BorderLayout` in Swing. Analog dazu bietet auch die `BorderPane` die bekannten fünf Bereiche, in denen `Nodes` platziert werden können (oder die auch unbelegt bleiben können). Diese Anordnung eignet sich für gebräuchliche Layouts mit einer Toolbar oder einer Menüzeile oben sowie einer Statuszeile unten. Auf der linken Seite kann eine Navigation dargestellt werden sowie zusätzliche Informationen auf der rechten Seite. Die Hauptinformation ist mittig platziert.
- `GridPane` – Mithilfe der `javafx.scene.layout.GridPane` lassen sich Anordnungen an einem Raster realisieren, ähnlich wie mit dem aus Swing bekannten `java.awt.GridLayout` oder dem komplexeren `java.awt.GridBagLayout` respektive dem `FormLayout` aus der GUI-Bibliothek JGoodies (verfügbar unter `http://www.jgoodies.com/`). In einer `GridPane` können `Nodes` beliebigen Zellen im Raster zugeordnet werden und sich auch über den Bereich mehrerer Zellen erstrecken. Mithilfe von `GridPanes` lassen sich sehr gut Eingabemasken realisieren, die eine in Spalten und Zeilen ausgerichtete Darstellung von Bedienelementen erfordern.
- `HBox` und `VBox` – Die `javafx.scene.layout.HBox` ist ein einfaches Layout: Die `Nodes` werden horizontal in einer Zeile ausgerichtet. Für die `javafx.scene.layout.VBox` erfolgt die Ausrichtung in der Vertikalen, ähnlich einer Spalte.

Layouts am Beispiel

Im folgenden Beispiel verwende ich exemplarisch die zuletzt genannten und zuvor noch unbenutzten Layouts, um eine recht typische Oberfläche zu gestalten, die eine Toolbar oben, eine Navigationsleiste links und verschiedene Textfelder zentral anbietet.

Bereits bei nur etwas komplexeren Programmen lohnt es sich, dem Design und der Strukturierung im Vorfeld einige Gedanken zu widmen. Würde man stattdessen direkt loslegen und das GUI vollständig innerhalb der `start(Stage)`-Methode aufbauen, so wäre diese selbst für das Beispiel schon recht lang und etwas unübersichtlich. Das gilt in zunehmendem Maße, wenn das zu konstruierende GUI komplexer wird. Dann bietet sich oftmals an, die einzelnen Bestandteile des GUIs mithilfe von eigenen Methoden zu konstruieren, wie dies nachfolgend durch die drei `createXYZ()`-Methoden und die entsprechende Platzierung der erzeugten Container in einer `BorderPane` gezeigt ist:

```
@Override
public void start(final Stage primaryStage)
{
    final BorderPane borderPane = new BorderPane();
    borderPane.setTop(createToolbarPane());
    borderPane.setCenter(createInputPane());
    borderPane.setLeft(createNavigationPane());

    primaryStage.setTitle(LayoutCombinationExample.class.getSimpleName());
    primaryStage.setScene(new Scene(borderPane, 350, 200));
    primaryStage.show();
}

private Pane createToolbarPane()
{
    final HBox hbox = new HBox(5);
    hbox.getChildren().addAll(new Text("TOP"), new Button("HBox1"),
                                               new Button("HBox2"));
    return hbox;
}

private Pane createInputPane()
{
    final GridPane gridPane = new GridPane();
    final Label label1 = new Label("Label1");
    final TextField textfield1 = new TextField();
    final Label label2 = new Label("Label2");
    final TextField textfield2 = new TextField();
    final Button button = new Button("Button");

    gridPane.add(label1, 0, 0);
    gridPane.add(textfield1, 1, 0);
    gridPane.add(label2, 0, 1);
    gridPane.add(textfield2, 1, 1);
    gridPane.add(button, 1, 2);
    return gridPane;
}

private Pane createNavigationPane()
{
    final VBox vbox = new VBox(5);
    vbox.getChildren().addAll(new Text("LEFT"), new Button("VBox1"),
```

```
                                                    new Button( "VBox2" ));
    return vbox;
}
```

Im Listing sehen wir verschiedene Besonderheiten und Methodenaufrufe, die wir noch nicht kennen. Da hier lediglich die Erstellung des Layouts mithilfe von Methoden von größerem Interesse ist, gehe ich auf die anderen Details später ein.

Starten Sie das Programm LAYOUTCOMBINATIONEXAMPLE, so kommt es zu folgender Ausgabe, die die Arbeitsweise der genannten Layouts verdeutlicht.

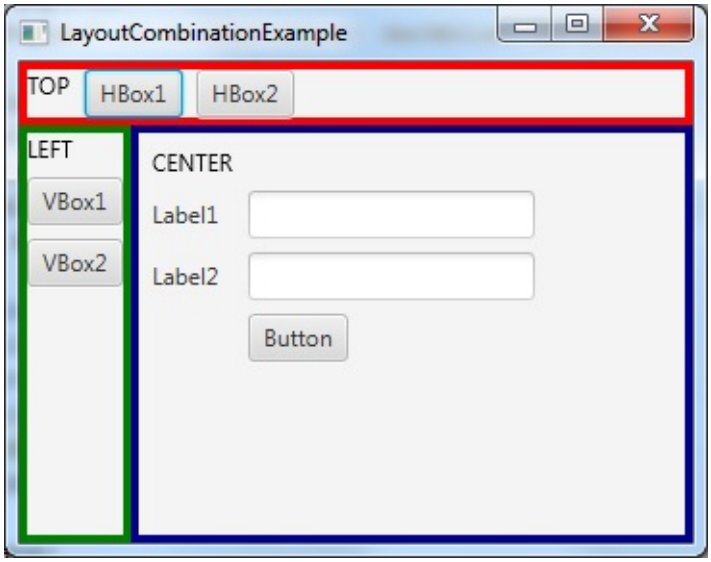


Abbildung 14-4 *Kombination von Layouts*

Hinweis: Komponentenbildung – Gestaltung komplexerer Layouts

Um grafische Elemente zu verschachteln und komplexere Strukturen aus Basisbausteinen zusammenzusetzen, kann man die einzelnen Bestandteile des GUIs mithilfe von Methoden erzeugen. Spielt der Aspekt Komponentenbildung und Wiederverwendbarkeit eine Rolle, so kann man statt Methoden besser eigenständige Klassen nutzen.

Man kann die beiden Ansätze sogar gewinnbringend kombinieren. Nachfolgend verdeutliche ich dies für die Toolbar. Diese ist nun in Form einer Klasse `ToolbarPane` implementiert und wird mithilfe der Fabrikmethode `createToolbarPane()` erzeugt. Somit ändert sich für die nutzende (obige) Applikation nichts an ihrem strukturellen Aufbau. Das Beschriebene kann man wie folgt umsetzen:

```
private Pane createToolbarPane()
{
    return new ToolbarPane();
}

static class ToolbarPane extends Pane
{
    public ToolbarPane()
    {
```

```
        final HBox hbox = new HBox(5);
        hbox.getChildren().addAll(new Text("TOP"),
                                  new Button("HBox1"), new Button("HBox2"));

        this.getChildren().add(hbox);
    }
}
```

Die HBox am Beispiel

Weil die `HBox` ein einfach zu verstehendes Layout realisiert, kann ich bei dessen Nutzung auf ein paar praktische Besonderheiten von JavaFX aufmerksam machen.

Dazu schauen wir auf folgendes Beispiel, in dem in einer `HBox` drei Bedienelemente, nämlich ein `Label`, ein `TextField` und ein `Button` mit auf 24pt vergrößerter Schriftart wie folgt hinzugefügt werden:

```
@Override
public void start(final Stage primaryStage)
{
    final Label label = new Label("Label");
    final TextField textfield = new TextField();
    final Button button = new Button("Button");
    button.setFont(Font.font(24));

    final HBox root = new HBox();
    root.getChildren().addAll(label, textfield, button);

    primaryStage.setTitle(FirstHBoxExample.class.getSimpleName());
    primaryStage.setScene(new Scene(root, 250, 70));
    primaryStage.show();
}
```

Führen wir das Programm FIRSTHBOXEXAMPLE aus, so erhalten wir eine Ausgabe wie in Abbildung 14-5.

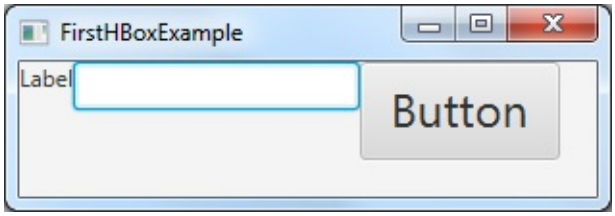


Abbildung 14-5 *Erstes Beispiel einer* `HBox`

Man erkennt sehr schön, dass die Bedienelemente horizontal, also innerhalb einer Zeile, angeordnet werden, wie wir dies von der `HBox` erwarten. Allerdings fallen gleich mehrere Dinge negativ auf:

- Die Bedienelemente werden ohne jeglichen Abstand direkt aneinander gezeichnet.
- Die Bedienelemente sind an ihrer oberen Kante ausgerichtet, was unruhig wirkt.

Besonderheiten von Layouts

Die beiden zuvor aufgelisteten kleineren Probleme im Layout inklusive möglicher Abhilfen wollen wir nachfolgend ein wenig genauer betrachten. Mit den gewonnenen Erkenntnissen bauen wir das obige Beispiel aus.

Besonderheit 1: Abstände

Häufig ist es sinnvoll, zwischen den Bedienelementen einen gewissen Abstand vorzugeben, anstatt sie direkt aneinander zu zeichnen. Dazu kann man für die meisten Layouts die Methoden `setHgap(double)` und `setVgap(double)` nutzen. Für die Klassen `HBox` und `VBox` werden diese Methoden jedoch nicht angeboten – nur jeweils eine davon wäre für die `HBox` bzw. die `VBox` passend. Daher lassen sich die Abstände für diese beiden Layouts spezifisch über Konstruktorparameter festlegen. Außerdem existiert alternativ dazu die Methode `setSpacing(double)`.

Neben Abständen zwischen Bedienelementen bietet es sich an, auch insgesamt um alle Bedienelemente einen Abstand zum Fensterrand bzw. zur umgebenden Containerkomponente vorzugeben. Dazu nutzt man die Klasse `javafx.geometry.Insets` und einem Aufruf von `setPadding(Insets)`:

```
// Abstand 10 Pixel zwischen Bedienelementen
final FlowPane pane = new FlowPane();
pane.setHgap(10);
pane.setVgap(10);

// Spezialbehandlung HBox und VBox
final HBox hbox = new HBox(10);
final VBox vbox = new VBox(10);

// 7 Pixel Abstand vom Rand
pane.setPadding(new Insets(7,7,7,7));
hbox.setPadding(new Insets(7,7,7,7));
vbox.setPadding(new Insets(7,7,7,7));
```

Besonderheit 2: Ausrichtung an der Basislinie

Die Darstellung ungleich hoher Bedienelemente direkt hintereinander ist optisch recht unharmonisch. Deutlich besser ist es, die Bedienelemente an einer virtuellen Linie auszurichten, die sich durch die in den Bedienelementen dargestellten Texte ergibt. Das spezifiziert man in JavaFX ganz einfach folgendermaßen:

```
// Linksbündige Ausrichtung an der Basislinie
hbox.setAlignment(Pos.BASELINE_LEFT);
```

Hier kommt die Aufzählung `javafx.geometry.Pos` zum Einsatz, in der verschiedene Positionierungen für X- und Y-Ausrichtung definiert sind, unter anderem etwa `TOP_LEFT`, `CENTER`, `BOTTOM_RIGHT` oder aber das oben verwendete `BASELINE_LEFT`, das eine linksbündige Ausrichtung auf der Höhe der Basislinie vornimmt.

Abhilfen im Einsatz

Mit den gerade gewonnenen Erkenntnissen wollen wir die angesprochenen Probleme lösen. Zunächst geben wir bei der Konstruktion der `HBox` einen Abstand vor, der zwischen den einzelnen Bedienelementen eingehalten werden soll. Zudem setzen wir einen Rand über `setPadding(Insets)`. Darüber hinaus sorgen wir durch Angabe von `Pos.BASELINE_LEFT` im Aufruf von `setAlignment(Pos)` dafür, dass die Bedienelemente an der Basislinie ausgerichtet sind:

```
@Override
public void start(final Stage primaryStage)
{
    final Label label = new Label("Label");
    final TextField textfield = new TextField();
    final Button button = new Button("Button");
    button.setFont(Font.font(24));

    // Besonderheit 1a: Abstand 10 Pixel zwischen Bedienelementen
    final HBox root = new HBox(10);
    // Besonderheit 1b: 7 Pixel Abstand vom Rand
    root.setPadding(new Insets(7,7,7,7));
    // Besonderheit 2: Ausrichtug an der Basislinie
    root.setAlignment(Pos.BASELINE_LEFT);

    root.getChildren().addAll(label, textfield, button);

    primaryStage.setTitle(HBoxWithAlignmentsExample.class.getSimpleName());
    primaryStage.setScene(new Scene(root, 300, 70));
    primaryStage.show();
}
```

Wenn wir das Programm HBoxWithAlignmentsExample starten, dann sehen wir die Verbesserung im Layout: Die Bedienelemente sind an einer Basislinie ausgerichtet. Beim Betrachten von Abbildung 14-6 erkennen wir auch, dass die Größe des Fensters nicht mehr ausreicht, um die Texte in den Bedienelementen vollständig darzustellen. Dann sorgt JavaFX dafür, dass die Texte durch Auslassungszeichen (eine sogenannte ***Ellipsis***, meistens als drei Punkte (…) dargestellt) abgekürzt werden. Insbesondere bei einem größenveränderlichen Fenster ist dies eine wünschenswerte Eigenschaft, die man nun out-of-the-Box mitgeliefert bekommt und besser noch, die sich zudem noch vielfältig konfigurieren lässt, wie wir dies im nachfolgenden Absatz kennenlernen werden.

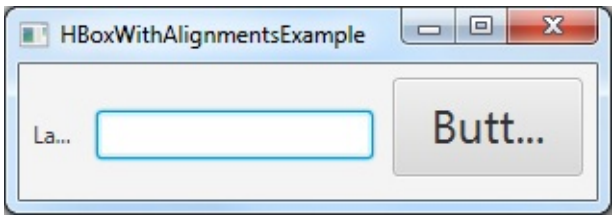


Abbildung 14-6 *Beispiel eines Layouts mit* `HBox` *mit Ausrichtung*

Besonderheit bei Größenveränderungen

Wir haben zuvor erkannt, dass JavaFX bereits diverse kleinere Verbesserungen und Bequemlichkeitsfunktionalitäten (Convenience) bereitstellt. Diese helfen unter anderem dabei, auf Größenveränderungen eines Fensters zu reagieren.

Nachfolgend möchte ich auf weitere Convenience-Funktionalitäten von JavaFX eingehen. Zunächst einmal auf die Konfigurierbarkeit der automatischen Verkürzung eines Textes durch Darstellung einer Ellipsis. Dabei lässt sich festlegen, an welcher Position die Ellipsis dargestellt werden soll. Zudem kann man auch die zur Abkürzung genutzte Zeichenfolge abändern. Des Weiteren kann man bestimmen, wie und welche Bedienelemente bei Größenanpassungen des Containers in ihrer Größe verändert werden sollen und welche nicht. Das wird über die Konstanten `ALWAYS`, `SOMETIMES` und `NEVER` aus der Aufzählung `javafx.scene.layout.Priority` gesteuert.

Mit diesem Wissen wollen wir das vorherige Beispiel anpassen: Wir legen für das `Label` und den `Button` durch Aufruf von `setTextOverrun(OverrunStyle)` das Verhalten beim Kürzen sowie mit `setEllipsisString(String)` die Zeichenfolge der Ellipsis fest. Für das `TextField` bestimmen wir durch Aufruf von `setHgrow(Priority)` mit der Priorität `ALWAYS`, dass Größenänderungen der `HBox`-Containerkomponente immer auch zu Größenänderungen des Textfelds führen. Zunächst sorgt die `HBox` aber dafür, dass alle enthaltenen Elemente ihre gewünschte Größe erhalten. Der darüber hinaus zur Verfügung stehende Platz wird dann an das `TextField` vergeben. All dies implementieren wir folgendermaßen:

```
@Override
public void start(final Stage primaryStage)
{
    final Label label = new Label("Label");
    label.setTextOverrun(OverrunStyle.ELLIPSIS);  // Standard
    final TextField textfield = new TextField();
    final Button button = new Button("This is a button");
    button.setFont(Font.font(24));

    // Setzen des Strings "##~##" als Ellipsis-Abkürzung
    button.setEllipsisString("##~##");
    button.setTextOverrun(OverrunStyle.CENTER_ELLIPSIS);

    final HBox root = new HBox(10);
    root.setPadding(new Insets(7,7,7,7));
    root.setAlignment(Pos.BASELINE_LEFT);
    root.getChildren().addAll(label, textfield, button);

    // Größenveränderung
    HBox.setHgrow(textfield, Priority.ALWAYS);

    primaryStage.setTitle(ResizableHBoxExample.class.getSimpleName());
    primaryStage.setScene(new Scene(root, 390, 120));
    primaryStage.show();
}
```

Nach dem Start des Programms RESIZABLEHBOXEXAMPLE kommt es in etwa zu einer Darstellung wie in Abbildung 14-7. Verändern Sie ein wenig die Größe des Fensters und beobachten Sie die Auswirkungen der zuletzt durchgeführten Erweiterungen.

Abbildung 14-7 *Größenveränderliches Layout mit einer* `HBox`

Die GridPane am Beispiel

Für professionelle Anwendungen benötigt man recht häufig eine Platzierung von Bedienelementen, die anhand eines Rasters erfolgt. Innerhalb der einzelnen Rasterzellen sollen Bedienelemente individuell ausgerichtet werden können (links-, rechtsbündig oder zentriert). Während in Swing das `GridLayout` nicht so viel Flexibilität bietet und man mit dem `GridBagLayout` häufig bezüglich der Konfigurationsangaben kämpfen musste, gestaltet sich die Arbeit mit dem JavaFX-Layoutcontainer `GridPane` deutlich angenehmer, weil die Zuordnung zum Raster auf einfache Weise erfolgt. Außerdem kann man bei Bedarf, etwa zu Debugging-Zwecken, Rasterlinien einblenden.

Mithilfe einer `GridPane` wollen wir nun einen recht einfachen Login-Dialog gestalten. Dort werden in zwei Zeilen jeweils in einer eigenen Spalte ein `Label` gefolgt von einem `TextField` platziert. In einer dritten Zeile wird in der zweiten Spalte ein Login-Button angeordnet. Zur Demonstration unterschiedlicher Ausrichtungen wählen wir für die `Label`s einmal links- und einmal rechtsbündig. Der `Button` wird auch rechtsbündig ausgerichtet. Um das praktische Feature der Gitterlinien demonstrieren zu können, nutzen wir eine `Checkbox` und einen `EventHandler<ActionEvent>`, über den diese Hilfslinien ein- bzw. ausgeschaltet werden können. Die beschriebenen Funktionalitäten realisieren wir folgendermaßen:

```
@Override
public void start(final Stage primaryStage) throws Exception
{
    final GridPane gridPane = new GridPane();
    gridPane.setPadding(new Insets(10, 10, 10, 10));
    gridPane.setHgap(7);
    gridPane.setVgap(7);

    final Label lblName = new Label("Name:");
    final TextField tfName = new TextField();

    final Label lblPassword = new Label("Password:");
    final PasswordField pfPassword = new PasswordField();

    final Button btnLogin = new Button("Login");

    // Bereitstellung von Gitterlinien
    final CheckBox checkBoxShowGridLines = new CheckBox("Show Gridlines");
    checkBoxShowGridLines.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(final ActionEvent event)
        {
```

```
            gridPane.setGridLinesVisible(checkBoxShowGridLines.isSelected());
        }
    });

    // Zuordnung zum Grid (Node, X-Position, Y-Position)
    gridPane.add(lblName, 0, 0);
    gridPane.add(tfName, 1, 0);
    gridPane.add(lblPassword, 0, 1);
    gridPane.add(pfPassword, 1, 1);
    gridPane.add(btnLogin, 1, 2);
    gridPane.add(checkBoxShowGridLines, 0, 5);

    // Layoutbesonderheiten
    GridPane.setHalignment(lblName, HPos.LEFT);
    GridPane.setHalignment(lblPassword, HPos.RIGHT);
    GridPane.setHalignment(btnLogin, HPos.RIGHT);

    primaryStage.setScene(new Scene(gridPane, 300, 150));
    primaryStage.setTitle("GridPaneExample");
    primaryStage.show();
}
```

Abbildung 14-8 zeigt die Ausgabe des Programms GRIDPANEEXAMPLE mit und ohne aktivierte Gitterlinien.

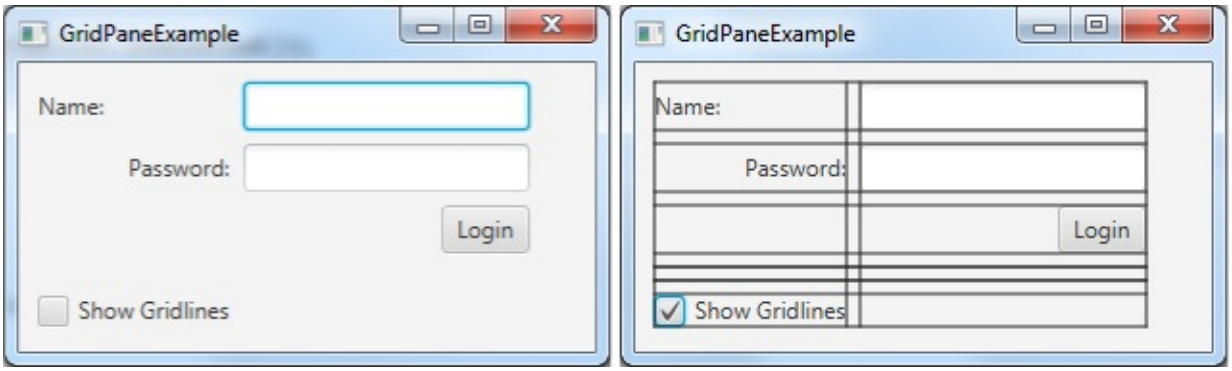


Abbildung 14-8 *Beispiel eines Layouts mit einer* `GridPane`

Durch die in der Abbildung gezeigten Gitterlinien erkennt man die Positionierung durch spezielle Abstandsspalten/-zeilen. Das ist eine mögliche Form der Nutzung. Alternativ dazu kann man die Abstände auch über die bereits kennengelernten Vorgaben über `Insets` erzielen. Bei komplexeren Eingabemasken kann das durchaus der bessere Weg sein, da das Gitter möglicherweise nicht die benötigte Flexibilität erlaubt.

Abschließend möchte ich noch kurz auf einige Dinge eingehen, die Ihnen vielleicht schon beim Betrachten des Listings als Fragen in den Sinn gekommen sind.

Frage: Sollte man Präfixe für Bedienelemente nutzen?

In diesem Beispiel werden Präfixe für Bedienelemente verwendet. Wann sollte man diese verwenden? Eine allgemeingültige Antwort auf diese Frage gibt es sicher nicht. Zum Teil bietet es sich an, für Bedienelemente verschiedene Kürzel wie `lbl` für `Label` oder `tf` für `TextField` zu nutzen, etwa um das Label `lblName` deutlich von dem korrespondierenden Textfeld `tfName` unterscheiden zu können. Es gibt

noch eine andere Form der Namensgebung: Man kann eine Suffix-Notation nutzen. Damit ergeben sich teilweise lesbare Namen wie `loginButton`. Aber für `nameLabel` und `nameTextfield` stößt auch diese Notation an ihre Grenzen.

Manchmal empfinde ich Präfixe bzw. Suffixe als hilfreich, manchmal als störend. Wichtig ist vor allem, dass der Rest des Namens aussagekräftig ist.

Merkwürdigkeit: Statische Positionierungsmethoden

Im Listing sehen wir die Verwendung statischer Methoden (`GridPane.setHalignment()`), um Attribute zur Ausrichtung zu setzen. Das wirkt eher unnatürlich, ist aber laut Cay S. Horstmanns empfehlenswerten Buchs »Java SE 8 for the Really Impatient« [44] der deklarativen Konstruktion von GUIs mithilfe von FXML (JavaFX Markup Language) und der Reihenfolge von Methodenaufrufen und Initialisierungen geschuldet.

Frage: Gibt es nicht ein GUI-Design-Tool?

Je komplexer die zu erstellenden Layouts werden, desto mehr tendiert der Sourcecode dazu, unübersichtlich und auch schlechter wartbar zu werden. Schnell kommt der Wunsch nach einem GUI-Design-Tool auf. Praktischerweise bietet Oracle das Tool SCENEBUILDER an. Es kann unter `http://www.oracle.com/technetwork/java/javafx/tools/index.htm` `ssSourceSiteId=ocomen` heruntergeladen werden. Die mit dem SceneBuilder erstellten Layouts produzieren als Ergebnis keinen Sourcecode, sondern die Oberfläche wird deklarativ mithilfe von FXML beschrieben, was wir im nächsten Unterkapitel kennenlernen werden.

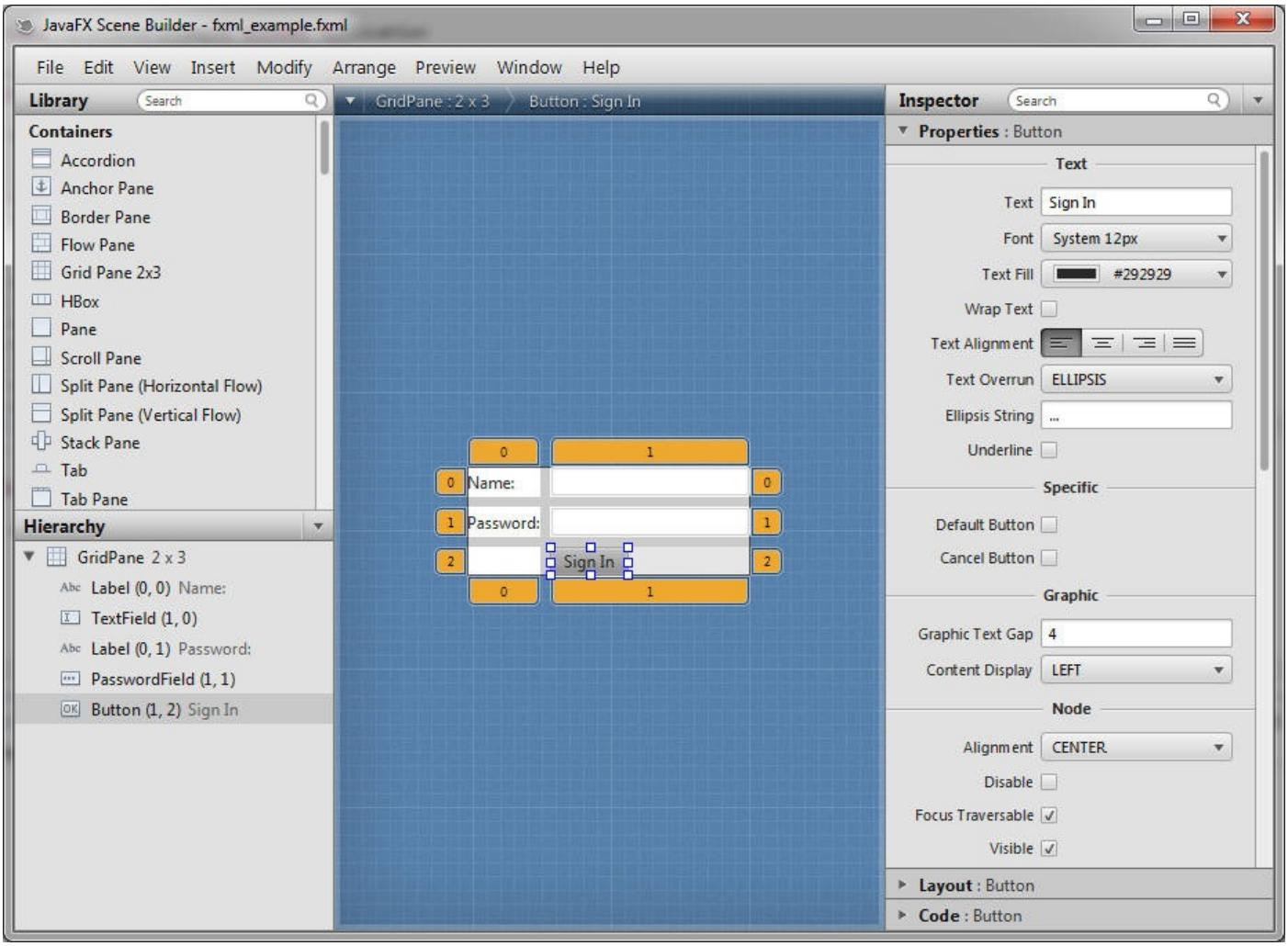


Abbildung 14-9 *Das Tool SceneBuilder*

14.2 Deklarativer Aufbau des GUIs

Ein GUI bzw. die Anordnung und der Zusammenhang der Bedienelemente wird in AWT, SWT und Swing gewöhnlich im Sourcecode ausprogrammiert, indem explizit Bedienelemente und Container erzeugt und miteinander verbunden werden. In den vorangegangenen Beispielen haben wir dies genauso mit JavaFX gemacht.

Die Erfahrung zeigt, dass diese Art der GUI-Erzeugung recht schnell zu schlecht wartbarem Spaghetticode führt – insbesondere dann, wenn Layouts komplexer werden. Eine mögliche Abhilfe besteht darin, Hilfsmethoden zu definieren, die die einzelnen Bestandteile des GUIs konstruieren, wie ich es bei der Vorstellung des Layoutmanagements gezeigt habe.

14.2.1 Deklarative Beschreibung von GUIs

Als Alternative zum programmatischen Zusammenbau kann man GUIs deklarativ beschreiben, wodurch sich häufig die zugrunde liegende hierarchische Struktur besser erkennen lässt, als dies für die Konstruktion im Sourcecode der Fall ist.

Für Swing gab es keinen Standard für die deklarative Beschreibung von GUIs. Mit JavaFX ändert sich dies. Mithilfe der XML-basierten FXML (JavaFX Markup Language) ist eine deklarative Beschreibung möglich. Dazu wird die Darstellung des GUIs (View) in Form von FXML definiert und das Verhalten (Controller) sowie das Datenmodell (Model) als Java-Klassen programmiert. Dieses Vorgehen folgt dem Model-View-Controller-Ansatz und bietet den Vorteil einer besseren Trennung von Darstellung und Modelldaten[1] und kann daher leichter verständlich sowie wartbar sein. Zudem wird es durch die Trennung (zumindest theoretisch) möglich, den Entwurf des GUIs von Designern durchführen zu lassen und nicht von Entwicklern. Letztere können sich dann ganz auf ihre Stärken in der Programmierung der GUI-Funktionalität konzentrieren.

Es gibt sogar noch einen weiteren Vorteil der deklarativen Gestaltung: Wenn man Programme für verschiedene Ausgabegeräte (Desktop, Tablet oder Handy) schreibt, so kann man für die jeweiligen Gerätegattungen eigene FXML-Dateien bereitstellen, die ein optimal passendes Layout definieren. Das führt oftmals zu ansprechenderen Ergebnissen, als wenn man versucht, eine allgemeingültige Größenanpassbarkeit ins Layout zu integrieren.

14.2.2 Hello-World-Beispiel mit FXML

Schauen wir nun, wie wir FXML für das Beispiel des Login-Dialogs nutzen können. Beginnen wir mit der Darstellung des GUIs. Nachfolgend ist dessen Definition als FXML-Datei `fxml_example.fxml` gezeigt — wobei ich einige Vereinfachungen am Layout vorgenommen habe, um für diese Einführung das Verständnis zu erleichtern.

```
<?xml version="1.0" encoding="UTF-8"?>

<?import java.lang.*?>
<?import java.net.*?>
<?import javafx.geometry.*?>
<?import javafx.scene.control.*?>
<?import javafx.scene.layout.*?>
<?import javafx.scene.text.*?>

<GridPane alignment="CENTER" hgap="7.0" vgap="7.0"
      xmlns:fx="http://javafx.com/fxml/1"
      xmlns="http://javafx.com/javafx/2.2"
      fx:controller="javafx.fxml.FXMLExampleController">

  <children>
     <Label text="Name:" GridPane.columnIndex="0" GridPane.rowIndex="0" />
     <TextField fx:id="nameField" GridPane.columnIndex="1"
                                  GridPane.rowIndex="0" />

     <Label text="Password:" GridPane.columnIndex="0" GridPane.rowIndex="1" />
     <PasswordField fx:id="pwdField" GridPane.columnIndex="1"
                                     GridPane.rowIndex="1" />

     <Button onAction="#handleSubmitButtonAction" text="Login"
```

```
            GridPane.columnIndex="1" GridPane.rowIndex="2" />
  </children>
</GridPane>
```

Im Listing sehen wir, dass es zu den Bedienelementen aus JavaFX korrespondierende Elemente in FXML gibt, etwa `TextField` oder `Label`, aber auch `GridPane`. Diese den Elementnamen entsprechenden Klassennamen sind im Listing bei ihrem ersten Auftreten fett markiert.

Wenn wir FXML zur Definition des GUIs nutzen, entfällt im Java-Code selbst logischerweise der Aufbau des GUIs. Stattdessen nutzen wir die Klasse `FXMLLoader` und deren Methode `load(URL)`, die aus der gezeigten deklarativen Beschreibung des GUIs in der FXML-Datei korrespondierende Container und Bedienelemente erstellt. Die Root-Komponente wird in Form eines Containers vom Typ `javafx.scene.Parent` zurückgeliefert. Mit diesem Wissen implementieren wir die `start(Stage)`-Methode folgendermaßen:

```
@Override
public void start(final Stage stage) throws Exception
{
    final Parent root = FXMLLoader.load(getClass().
                                        getResource("fxml_example.fxml"));

    stage.setScene(new Scene(root, 300, 150));
    stage.setTitle("FirstFxmlExample");
    stage.show();
}
```

Führen wir das Programm FIRSTFXMLEXAMPLE aus, so erhalten wir in etwa eine Ausgabe wie in Abbildung 14-10 gezeigt, die abgesehen von Ausrichtungen und der Checkbox zur Aktivierung von Gitterlinien dem zuvor programmatisch realisierten Layout eines Login-Dialogs stark ähnelt.

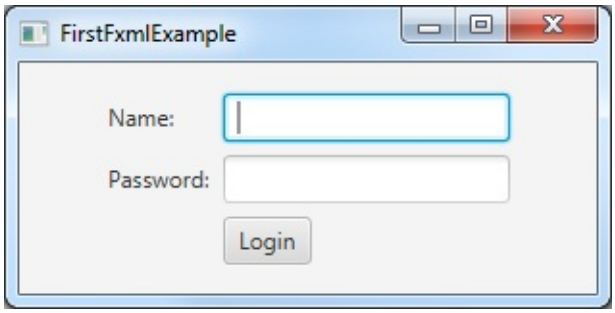


Abbildung 14-10 *Beispiel für GUI-Design und FXML*

Interessanterweise lässt sich das per FXML erstellte GUI schon bedienen und beim Drücken des Buttons werden die für Name und Passwort eingegebenen Werte auf der Konsole ausgegeben. Wir sollten uns fragen, wieso das Action Handling denn eigentlich funktioniert. Schauen wir kurz auf die dafür verantwortlichen, fett markierten Zeilen im FXML:

```
<GridPane alignment="CENTER" hgap="7.0" vgap="7.0"
      xmlns:fx="http://javafx.com/fxml/1"
      xmlns="http://javafx.com/javafx/2.2"
      fx:controller="javafx.fxml.FXMLExampleController">

  …
```

```
  <Button onAction="#handleSubmitButtonAction" text="Login"
          GridPane.columnIndex="1" GridPane.rowIndex="2" />

</GridPane>
```

Im Listing verweisen die Angaben `fx:controller` auf eine Klasse und `onAction` auf eine `EventHandler`-Methode. Beides lernen wir nun kennen.

Controller

Für dieses Beispiel nutzen wir die nachfolgende Implementierung eines Controllers. Dieser realisiert die vom obigen Programm ausgeführte Funktionalität:

```
public class FXMLExampleController
{
    @FXML
    private TextField nameField;
    @FXML
    private PasswordField pwdField;

    @FXML
    protected void handleSubmitButtonAction(final ActionEvent event)
    {
        System.out.println("Login button pressed! name: " +
nameField.getText()
                           + " pwd: " + pwdField.getText());
    }
}
```

Im Listing des Controllers sehen wir, dass der Klassenname mit der Angabe in `fx:controller` übereinstimmt. Ähnliches gilt für die Angabe des korrespondierenden Methodennamens in `onAction` im FXML. Dort wird lediglich das `#`-Zeichen entfernt und nach einer gleichnamigen Methode im Controller gesucht. Das wird mithilfe der Annotation `@FXML` möglich. Diese nutzt man auch zur Verbindung von den durch FXML beschriebenen Bedienelementen zu den im Controller definierten Attributen: Ist ein Attribut mit `@FXML` annotiert, so wird in der FXML-Datei nach einem passenden Pendant gesucht. Hierzu wird der Name des Attributs mit der in FXML angegebenen `fx:id` abgeglichen. Schauen wir auf die dafür relevanten Zeilen in FXML:

```
<TextField fx:id="nameField" GridPane.columnIndex="1" GridPane.rowIndex="0"/>
…
<PasswordField fx:id="pwdField" GridPane.columnIndex="1"
GridPane.rowIndex="1"/>
```

14.2.3 Diskussion: Design und Funktionalität strikt trennen

Die Trennung von Zuständigkeiten ist in der Regel etwas Erstrebenswertes. Wenn man FXML nutzt, so kann das GUI mithilfe eines GUI-Design-Tools entworfen

werden (am besten von UI-Experten) und die Entwickler können sich um die Realisierung von Funktionalität kümmern.

Wenn man Design und Funktionalität strikt voneinander trennen möchte, so dürfen konsequenterweise deren Verknüpfungen nicht in FXML verdrahtet werden, sondern müssen nachträglich programmatisch realisiert werden. Demnach darf in FXML streng genommen nicht einmal die Controllerklasse angegeben werden. Auf jeden Fall sind aber die Angaben von Methoden zum Action Handling zu vermeiden, wenn man das Ziel hat, Design und Funktionalität möglichst gut gegeneinander abzuschirmen.

Notwendige Anpassungen in FXML

Die Trennung von GUI-Design in FXML und Controller erfordert ein paar Änderungen. Die bisher automatisch ablaufende Verknüpfung zwischen FXML und Controllerklasse sowie der Methode zum Event Handling muss nun programmatisch vorgenommen werden. Als Erstes ist in FXML die Angabe von `fx: controller` zu entfernen und als Zweites benötigen wir für den Button statt der Angabe von `onAction` eine ID:

```
<GridPane alignment="CENTER" hgap="7.0" vgap="7.0"
      xmlns:fx="http://javafx.com/fxml/1"
      xmlns="http://javafx.com/javafx/2.2">

  …

  <Button fx:id="loginButton" text="Login"
          GridPane.columnIndex="1" GridPane.rowIndex="2" />

</GridPane>
```

Damit ist das FXML frei von Verweisen auf den Controller und die Programmlogik. Nun muss der Controller derart angepasst werden, dass die Verknüpfung erstellt wird.

Notwendige Anpassungen im Controller

Zunächst führen wir ein weiteres Attribut ein. Darüber hinaus lassen wir den Controller das Interface `Initializable` erfüllen, was eine `initialize()`-Methode deklariert, die wir passend implementieren:

```
public class FXMLExampleSpecialController implements Initializable
{
    @FXML
    private PasswordField passwordField;
    @FXML
    private TextField nameField;
    @FXML
    private Button loginButton;

    @Override
    public void initialize(final URL location, final ResourceBundle resources)
```

```
    {
        loginButton.setOnAction(this::handleSubmitButtonAction);
    }

    protected void handleSubmitButtonAction(ActionEvent event)
    {
        System.out.println("Login button pressed!  name:  " +
                           nameField.getText() + "  pwd:  " +
pwdField.getText());
    }
}
```

Notwendige Anpassungen in der Applikation

Schließlich muss noch eine kleine Anpassung in der Applikation selbst erfolgen, um den eigenen Controller explizit zu setzen:

```
@Override
public void start(final Stage stage) throws Exception
{
  final URL fxmlUrl =
getClass().getResource("fxml_example_no_controller.fxml");
  final FXMLLoader fxmlLoader = new FXMLLoader(fxmlUrl);
  fxmlLoader.setController(new FXMLExampleSpecialController());
  final Parent root = fxmlLoader.load();

  // Fehler: würde den Controller wieder überschreiben
  // final Parent root = FXMLLoader.load(getClass().
  //                            getResource("fxml_example_no_controller.fxml"));

  stage.setScene(new Scene(root, 450, 175));
  stage.setTitle("FXMLExampleWithSpecialController");
  stage.show();
}
```

Bitte beachten Sie, das ein versehentlicher Aufruf der statischen Methode `load()` – wie oben im Kommentar angedeutet –, die zuvor gemachte Angabe übersteuern würde.

Fazit

Wie man sieht, muss man recht wenig Aufwand treiben, um eine saubere Trennung zwischen Design in FXML und Funktionalität im Java-Code sicherzustellen. Es ist nun an Ihnen, sich dafür oder dagegen zu entscheiden. Zumindest besitzen Sie jetzt das dazu notwendige Handwerkszeug.

Hinweis: Komponentenbildung – Gestaltung komplexerer Layouts

Bei der Vorstellung des Layoutmanagements bin ich darauf eingegangen, dass man Bestandteile des GUIs mithilfe von Methoden oder durch eigenständige Klassen realisieren sollte. Letzteres ist insbesondere für Wiederverwendbarkeit empfehlenswert.

Wenn man den Aufbaus des GUIs mithilfe von FXML beschreibt, gilt das Gesagte ähnlich. Um grafische Elemente zu verschachteln und komplexere Strukturen aus Basisbausteinen zusammenzusetzen, kann man die Informationen in FXML in mehrere Dateien aufspalten und diese durch die Anweisung `fx:include` in die Haupt-FXML-Datei inkludieren. Damit lässt sich eine FXML-Datei aus anderen Dateien hierarchisch zusammenbauen. Die Controller können ebenfalls passend pro Subkomponente bereitgestellt werden. Damit wird eine GUI-Komponenten-orientierte Arbeitsweise erleichtert, die es erlaubt, wiederverwendbare Bausteine zu erzeugen. Für eine ausführlichere Darstellung verweise ich auf die am Ende dieses Buchs angegebene Literatur.

14.3 Rich-Client Experience

Die bisher beschriebenen Funktionalitäten unterscheiden sich kaum von denen herkömmlicher GUI-Frameworks. Warum wurde also JavaFX entwickelt und was unterscheidet es von bzw. zeichnet es gegenüber anderen GUI-Frameworks aus?

Wenn Sie schon einmal selbst ansprechende GUIs realisiert haben, die visuelle Anpassungen an Bedienelementen und möglicherweise sogar eigene Controls oder Effekte und Animationen zur Verbesserung der User Experience enthalten sollten, dann wissen Sie aus leidvoller Erfahrung, wie schnell Dinge recht komplex und manchmal sogar frustrierend werden können. In JavaFX wird vieles deutlich einfacher. Werfen wir zunächst einen einführenden Blick auf die in JavaFX genutzten Cascading Style Sheets (CSS), mit denen man starken Einfluss auf die grafische Gestaltung nehmen kann. Danach schauen wir kurz auf Effekte und Animationen.

14.3.1 Gestaltung mit CSS

Das bisher entwickelte, noch wenig funktionale GUI wirkt doch recht blass und unscheinbar. Mit ein wenig CSS lässt sich das ändern, weil auf diese Weise GUI-Elemente mit einem gewünschten Aussehen versehen werden können.

Bedienelemente mit CSS optisch gestalten

Ich zeige exemplarisch für zwei Buttons, wie man Bedienelemente von JavaFX mit CSS optisch gestalten (und aufpolieren) kann. Dabei wird deutlich, dass man mit CSS sehr weitreichende Gestaltungsmöglichkeiten besitzt. Wir verändern das Aussehen mit folgenden CSS-Angaben:

- `-fx-text-fill` – Legt die Schriftfarbe fest.
- `-fx-background-color` – Legt die Hintergrundfarbe fest.
- `-fx-font-family` – Legt die Schriftart fest.

- `-fx-font-size` – Bestimmt die Schriftgröße.
- `-fx-font-weight` – Wählt zwischen Normal- und Fettschrift.
- `-fx-effect` – Legt einen Effekt fest.
- `linear-gradient` – Definiert einen linearen Gradienten mit den angegebenen Farbwerten und Zwischenabstufungen.
- `radial-gradient` – Beschreibt einen radialen Gradienten mit den angegebenen Farben sowie einen definierten Mittelpunkt.

Neben denjenigen aus der Aufzählung existiert eine Vielzahl von CSS-Angaben, die (größtenteils) das Präfix `-fx` tragen, wodurch gekennzeichnet wird, dass es sich um JavaFX-spezifische Erweiterungen von CSS handelt.

CSS an Beispielen

Im nachfolgenden Listing werden wir ausnutzen, dass man das gewünschte CSS per `setStyle(String)` einem Bedienelement zuweisen kann. Schon vorab möchte ich darauf hinweisen, dass dieses Vorgehen nur für kleinere Programme oder wie hier zum ersten Kennenlernen und Ausprobieren eingesetzt werden sollte. Ansonsten kommt es recht schnell zu unübersichtlichem Sourcecode und verstößt gegen die Trennung von Zuständigkeiten, weil Funktionalität und Design vermischt werden. Eine Abhilfe lernen wir im Anschluss an dieses Beispiel kennen.

Nach diesem Hinweis kommen wir zu den CSS-Stilmitteln zurück und gestalten damit die Bedienoberfläche wie folgt:

```
public class FirstCssExample extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception
    {
        final Button loginButton = new Button("Login Button");
        loginButton.setStyle("-fx-text-fill: silver; -fx-font-size: 18pt;" +
                             "-fx-font-weight: bold; " +
                             "-fx-background-color: " +
                             "radial-gradient(center 25% 25%, radius 50%, "
+
                             "reflect, dodgerblue,darkblue
75%,dodgerblue);");

        final Button fancyButton = new Button("Fancy Login");
        fancyButton.setStyle("-fx-font-weight: bold;"
          + "-fx-font-family: "Dialog"; -fx-font-size: 36pt;"
          + "-fx-effect: dropshadow( three-pass-box , black, 5, 0.2 , 2 , 3);"
          + "-fx-text-fill: linear-gradient(to left, darkviolet 15%,yellow
45%,"
          + " red 75%,firebrick 85%);"

          + "-fx-background-color: linear-gradient(darkblue 10%, #ABCDEF 65%,"
          + "dodgerblue 90%)");

        final FlowPane flowPane = new FlowPane();
        flowPane.setHgap(7);
```

```
        flowPane.setVgap(7);
        flowPane.setPadding(new Insets(7,7,7,7));
        flowPane.getChildren().addAll(loginbutton, fancyButton);

        primaryStage.setScene(new Scene(flowPane, 550, 110));
        primaryStage.setTitle(this.getClass().getSimpleName());
        primaryStage.show();
    }
    // …
}
```

Welche Auswirkungen die Angaben haben, wird durch Abbildung 14-11 recht gut nachvollziehbar oder durch den Start der Applikation FIRSTCSSEXAMPLE.

Abbildung 14-11 *JavaFX mit CSS*

Für die beiden Buttons mag das realisierte Design gerade noch fancy aussehen. Es ist aber optisch schon leicht überfrachtet.

Wenn Sie miterleben möchten, wie sich der CSS-Effekt sogar dynamisch auf eingegebenen Text auswirkt, so starten Sie bitte das Programm FXMLEXAMPLEWITHCSS, das das FXML-Beispiel des recht faden Login-Dialog mit CSS sehr bunt gestaltet.

Abbildung 14-12 *Login-Dialog mit CSS*

Besonders am letzten Beispiel erkennt man die optische Überfrachtung des von mir stammenden Informatiker-Designs – im Normalfall sollten Sie das Design daher besser ausgewiesenen CSS-Profis überlassen.

CSS in eine eigene Datei auslagern

Wenn wir uns vorstellen, dass man die Bedienelemente durch Angaben im Sourcecode mit CSS-Informationen versehen wollte, kann man sich ausmalen, wie unleserlich es wird, wenn der Sourcecode mit CSS durchmischt ist. Was kann man

dagegen tun?

Generell bietet es sich an, die CSS-Angaben in eine eigene Datei auszulagern. Das gilt umso mehr, je umfangreicher die Styling-Informationen werden. Neben mehr Klarheit im Sourcecode besitzt diese Separation von Styling und Sourcecode noch einen weiteren Vorteil: Die grafische Gestaltung kann beliebig, auch dynamisch gewechselt werden – indem kurzerhand das gesamte CSS gewechselt wird oder spezielle Stile angegeben werden. Gleich dazu mehr.

Wir wollen die CSS-Angaben für die beiden Buttons in eine Datei `buttons.css` auslagern. Um verschiedene Varianten des Stylings mit CSS zu demonstrieren, füge ich dem Beispiel einen weiteren Button hinzu und nutze nachfolgend

- den allgemeinen Selektor `.button`, der das Styling für alle (nicht durch andere Stylings übersteuerten) Buttons in JavaFX bestimmt,
- einen Gruppenselektor `.customloginbutton`, der das Styling für eine Menge von Bedienelementen, hier Login-Buttons, definiert,
- eine spezielle ID `#fancybutton`, mit der sich das Aussehen von Bedienelementen festlegen lässt, denen diese CSS-ID zugewiesen ist.

Mit diesem Vorwissen erstellen wir die CSS-Datei und verwenden dabei weitestgehend die zuvor in `setStyle(String)` gemachten Angaben wie folgt:

```
.button
{
    -fx-text-fill: firebrick; -fx-font-size: 18pt; -fx-font-weight: bold;
}

.customloginButton
{
   -fx-text-fill: silver; -fx-font-size: 18pt; -fx-font-weight: bold;
   -fx-background-color: radial-gradient(center 25% 25%, radius 50%,
                         reflect, dodgerblue, darkblue 75%, dodgerblue);
}

#fancybutton
{
   -fx-font-weight: bold; -fx-font-family: Dialog; -fx-font-size: 36pt;
   -fx-text-fill: linear-gradient(to left,
                         darkviolet 15%,yellow 45%,red 75%,firebrick 85%);
   -fx-background-color: linear-gradient(darkblue 10%, #ABCDEF 65%,
                                         dodgerblue 90%);
   -fx-effect: dropshadow(three-pass-box, black, 5, 0.2, 2, 3);
}
```

Um die Bedienelemente mit den obigen CSS-Definitionen in Verbindung zu bringen, verwendet man Selektoren. Für eine Menge von Bedienelementen eines Typs kann man allgemeine Selektoren etwa `.label,.button` nutzen. Damit haben wir den Stil für den Button `plainButton` vorgegeben. Zudem kann man für spezifische Gruppen einen gemeinsamen Selektor verwenden, wie wir dies für den `loginButton` mit seinem Stil `customloginbutton` getan haben. Die Verknüpfung mit dem Bedienelement geschieht über einen Aufruf von

`getStyleClass().add(String)`. Zudem nutzen wir die Möglichkeit, die Bedienelemente individuell mit einem Stil zu versehen. Dazu wird der ID-Selektor für das gewünschte Bedienelement durch `setID(String)` gesetzt.

Basierend auf den vorangegangenen Informationen ändern wir die `start(Stage)` -Methode folgendermaßen ab:

```
@Override
public void start(final Stage primaryStage) throws Exception
{
    final Button plainButton = new Button("Plain Red Text Button");
    final Button loginButton = new Button("Login Button");
    final Button fancyButton = new Button("Fancy Login");

    // Verknüpfung mit CSS über Style Class bzw. ID
    loginButton.getStyleClass().add("customloginbutton");
    fancyButton.setId("fancybutton");

    final FlowPane flowPane = new FlowPane();
    flowPane.setHgap(7);
    flowPane.setVgap(7);
    flowPane.setPadding(new Insets(7,7,7,7));
    flowPane.getChildren().addAll(plainButton, loginButton, fancyButton);

    primaryStage.setScene(new Scene(flowPane, 420, 160));
    // Verknüpfung von Scene und CSS
    primaryStage.getScene().getStylesheets().add(this.getClass().
                            getResource("buttons.css").toExternalForm());

    primaryStage.setTitle(this.getClass().getSimpleName());
    primaryStage.show();
}
```

Für das Anwenden des CSS müssen wir nichts weiter tun, als es mit der `Scene` zu verknüpfen. Dazu rufen wir `getStylesheets().add(String)` auf und dort die Methode `toExternalForm()`, die eine Stringrepräsentation für eine URL liefert. Diese URL haben wir mithilfe von `getResource(String)` ermittelt und diese referenziert das gewünschte CSS. Führen Sie das Programm ExternalCssExample aus, so erhalten Sie eine Darstellung ähnlich zu Abbildung 14-13.

Abbildung 14-13 *Darstellung mit externer CSS-Datei*

Dynamische Gestaltung

Eingangs erwähnte ich, dass man das Styling auch dynamisch modifizieren kann. Statt aber gleich auf ein vollständig neues Look-and-Feel umzuschalten, das durch eine andere CSS-Datei festgelegt ist, wollen wir uns eine Variante anschauen, die Dynamik rein deklarativ ermöglicht.

Das wird dadurch realisiert, dass man im CSS für verschiedene Zustände eines Bedienelements unterschiedliche CSS-Angaben aufführt. Man kann ein Design für das Darüberfahren (`:hover`), das Drücken (`:pressed`), mit Fokus (`:focused`) usw. angeben. Wir ergänzen das CSS wie folgt:

```
.button:hover
{
    -fx-background-color: linear-gradient(blue, skyblue, white, skyblue,
blue);
}

.button:focused
{
    -fx-background-color: linear-gradient(skyblue, dodgerblue, darkblue);
    -fx-text-fill: white;
    -fx-border-color: firebrick, red, orange, yellow;
    -fx-border-insets: 5, 10, 15, 20;
    -fx-border-width: 3;
}

.button:pressed
{
    -fx-background-color: lightblue;
    -fx-background-radius: 10 10 10 10;
    -fx-text-fill: black;
}
```

Im Sourcecode müssen wir nichts ändern[2], sondern wir sehen die Änderungen und dynamischen Anpassungen, wenn wir das Programm DYNAMICCSSEXAMPLE starten ähnlich zu Abbildung 14-14.

Abbildung 14-14 *Dynamische Anpassungen mit CSS*

Die Grafik deutet die Dynamik lediglich an. Um das Verhalten zu erleben, selektieren und fokussieren Sie die Buttons und schauen, wie sich die Darstellung ändert.

Fazit

Dieser Abschnitt hat einen Einstieg in die vielfältigen Möglichkeiten zur Gestaltung von JavaFX-GUI mithilfe von CSS gegeben. Dabei konnte ich nur an der Oberfläche kratzen. Es gibt viel mehr zu entdecken.

Bei Interesse an den weiteren Möglichkeiten und verfügbaren CSS-Tags sollten Sie einen Blick auf die Dokumentation werfen. Diese finden Sie auf folgender Oracle-Seite: `http://docs.oracle.com/javafx/2/api/javafx/scene/doc-files/cssref.html`.

14.3.2 Effekte

Grafische Benutzeroberflächen, die mit Swing erstellt sind, wirken meistens ein wenig altbacken und fade. Visuelle Effekte kommen dort eher spärlich zum Einsatz. Das liegt einfach daran, dass Effekte aufwendig selbst implementiert werden müssen. JavaFX macht einem das Leben hier wirklich sehr leicht: Es ist auf einfache Weise möglich, die Bedienoberfläche durch grafische Effekte wie Reflexionen, Schatten, Weichzeichner usw. optisch aufzuwerten und dadurch interessant zu gestalten. Praktischerweise bietet JavaFX bereits eine Vielzahl an vordefinierten Effekten zur Auswahl, wodurch sich ohne viel eigenen Implementierungsaufwand recht ansehnliche Ergebnisse erzielen lassen. Zudem können viele Effekte parametriert werden (z. B. lässt sich die Richtung und Intensität eines Schattenwurfs beeinflussen). Außerdem lassen sich Effekte miteinander kombinieren und darüber hinaus zur Laufzeit dynamisch anpassen bzw. verändern.

Das folgende Listing zeigt diverse Effekte, die durch einen Druck auf einen Button aktiviert werden. Die Funktionalität realisieren wir in einem `EventHandler<ActionEvent>`, der mit einem Button verbunden wird:

```
@Override
public void start(final Stage stage) throws Exception
{
    final Node labelNode = new Label("Hello Effects World!");

    final Image image = new Image("example.png", true);
    final Node imageView = new ImageView(image);

    final Button buttonNode = new Button("Activate effects");

    // EventHandler mit Button verbinden
    addEffectHandler(labelNode, imageView, buttonNode);

    final HBox hbox = new  HBox(7);
    hbox.setPadding(new Insets(7,7,7,7));

    final VBox vbox = new  VBox(20);
    vbox.getChildren().addAll(labelNode, buttonNode);
    hbox.getChildren().addAll(vbox, imageView);

    stage.setScene(new Scene(hbox, 400, 300));
    stage.setTitle(EffectExample.class.getSimpleName());
    stage.show();
```

```
}

private void addEffectHandler(final Node labelNode, final Node imageView,
                              final Button buttonNode)
{
    buttonNode.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(final ActionEvent event)
        {
           // Schatteneffekt
           final Effect effect = new DropShadow(2, 7, 7, Color.BLACK);
           labelNode.setEffect(effect);

           // Kombination von Effekten: Schatten, Weichzeichner und Reflexion
           final Effect dropShadoweffect = new DropShadow(5, 3, 5,
Color.BLACK);
           final GaussianBlur gaussianEffect = new GaussianBlur(3);
           gaussianEffect.setInput(dropShadoweffect);
           final Reflection reflectionEffect = new Reflection();
           reflectionEffect.setInput(gaussianEffect);

           // Dynamisch zuweisen
           buttonNode.setEffect(reflectionEffect);
           imageView.setEffect(reflectionEffect);
        }
    });
}
```

Im Listing sehen wir im `EventHandler<ActionEvent>`, dass zunächst ein Schattenwurf für ein `Label` angegeben ist. Danach werden drei Effekte kombiniert und auf den `Button` selbst sowie auf ein Bild vom Typ `javafx.scene.image.Image` mit der korrespondierenden `Node` vom Typ `javafx.scene.image.ImageView` angewendet: ein Schatten, ein Unschärfefilter sowie eine Reflexion. Die drei genutzten Effekte sind vom Typ `DropShadow`, `GaussianBlur` und `Reflection`, besitzen den Basistyp `Effect` und entstammen dem Package `javafx.scene.effect`.

Führen Sie das Programm JAVAFXEFFECTSEXAMPLE aus, so erhalten Sie beim Drücken des Buttons eine Ausgabe ähnlich zu Abbildung 14-15.

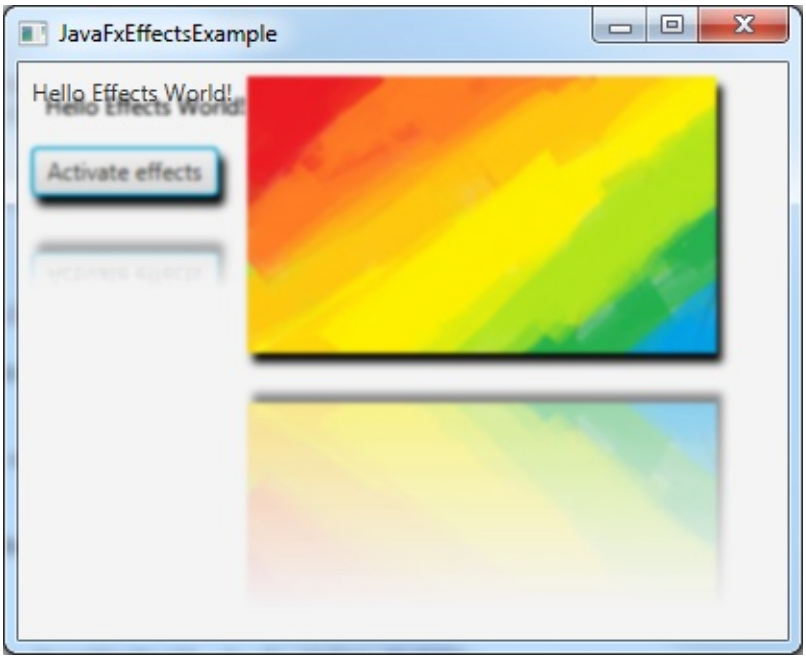


Abbildung 14-15 *Effekte mit JavaFX*

Dieses Beispiel zeigt, wie einfach es ist, Effekte zu verwenden, zu kombinieren und sogar dynamisch einzusetzen. Den gezeigten Unschärfeeffekt könnte man beispielsweise dann nutzen, wenn bei einem Druck auf einen Button eine längere Berechnung ausgeführt wird. Dieser Effekt macht optisch intuitiv klar, dass das betroffene Bedienelement derzeit deaktiviert ist.[3]

Wenn Sie schon einmal versucht haben, ähnliche Effekte, wie die eben gezeigten, mit Java 2D zu zaubern, dann wissen Sie, dass dies einiges an Mühe kostet. Insbesondere gilt dies, wenn Effekte für eine Gruppe von mehreren Bedienelementen wirksam werden sollen. In JavaFX ist das alles kein Problem, denn hier werden Effekte auf konzeptioneller Ebene und nicht auf Pixelebene angewendet. Dadurch wirken Effekte pro Knoten vom Typ `Node`. Ist dieser im Speziellen ein Container, so bedeutet dies, dass sich der Effekt auf alle darin enthaltenen `Nodes` auswirkt. Dort enthaltene einzelne `Nodes` können darüber hinaus aber natürlich mit weiteren, spezifischen Effekten versehen werden. Aber es kommt noch besser. Werfen wir nun einen Blick auf Animationen.

14.3.3 Animationen

Durch den gezielten Einsatz von Effekten kann man eine moderne, ansprechende Bedienoberfläche optisch gestalten. Dies kann jedoch durch die Verwendung von Animationen, wie Ein- und Ausblenden, noch gesteigert werden. Sehr wirksam für eine visuelle Rückmeldung ist auch eine Vergrößerung oder ein Aufglühen, wenn der Benutzer ein Element selektiert oder mit der Maus darüberfährt.

Nachfolgend schauen wir uns eine einfache Transition an, die ein `Label` mehrmals in Folge vergrößert und wieder verkleinert. Um das Ganze aufzupeppen, fügen wir noch eine Rotation um 270 Grad hinzu. Den beiden Animationen werden durch die Angabe verschiedener `Duration`-Instanzen unterschiedliche Ablaufdauern

zugewiesen. Um die Gesamtausführungsdauer der mit 1500 ms schneller ablaufenden Rotation mit der 3 Sekunden dauernden Größenänderung in Einklang zu bringen, wählen wir entsprechende Werte bei den jeweiligen Aufrufen von `setCycleCount(int)`. Durch einen Aufruf von `setAutoReverse(boolean)` wird festgelegt, dass die Animationen zunächst vorwärts und danach automatisch in umgekehrter Richtung abgespielt werden.

Im Listing ist die Kombination zweier Animationen realisiert und durch fett geschriebene Kommentare verdeutlicht:

```
@Override
public void start(final Stage stage)
{
    final Button buttonNode = new Button("Start Animations");
    final VBox vbox = new VBox(10.0);
    vbox.getChildren().addAll(buttonNode);

    addAnimation(buttonNode, vbox);

    stage.setScene(new Scene(vbox, 300, 250));
    stage.setTitle(AnimationExample.class.getSimpleName());
    stage.show();
}
private static void addAnimation(final Node buttonNode, final VBox vbox)
{
    buttonNode.setOnAction(new EventHandler<ActionEvent>()
    {
        @Override
        public void handle(final ActionEvent event)
        {
            final Node labelNode = new Label("Hello Java FX World!");
            vbox.getChildren().addAll(labelNode);

            // Schatteneffekt
            final Effect effect = new DropShadow(2, 5, 5, Color.BLACK);
            labelNode.setEffect(effect);

            // Größenänderung
            final ScaleTransition scaleTransition = new ScaleTransition(
                                                    Duration.millis(3000),
                                                    labelNode);
            scaleTransition.setByX(4);
            scaleTransition.setByY(6);
            scaleTransition.setCycleCount(10);
            scaleTransition.setAutoReverse(true);

            // Rotation
            final RotateTransition rotateTransition = new RotateTransition(
                                                    Duration.millis(1500),
                                                    labelNode);
            rotateTransition.setByAngle(270f);
            rotateTransition.setCycleCount(20);
            rotateTransition.setAutoReverse(true);

            // Kombination der Rotation und Größenänderung
            final ParallelTransition parallelTransition =
                                    new ParallelTransition(labelNode,
                                                           scaleTransition,
                                                           rotateTransition);
```

```
                parallelTransition.play();
            }
        });
    }
```

Im `EventHandler<ActionEvent>` wird ein `Label` erzeugt und diesem ein Schlagschatten hinzugefügt und danach der Text in x-Richtung 4-fach und in y-Richtung 6-fach vergrößert sowie um 270 Grad in die eine Richtung und dann automatisch zurück gedreht. Einen Einblick geben die folgenden Screenshots in Abbildung 14-16. Zur Verdeutlichung sollten Sie das Programm ANIMATIONEXAMPLE starten.

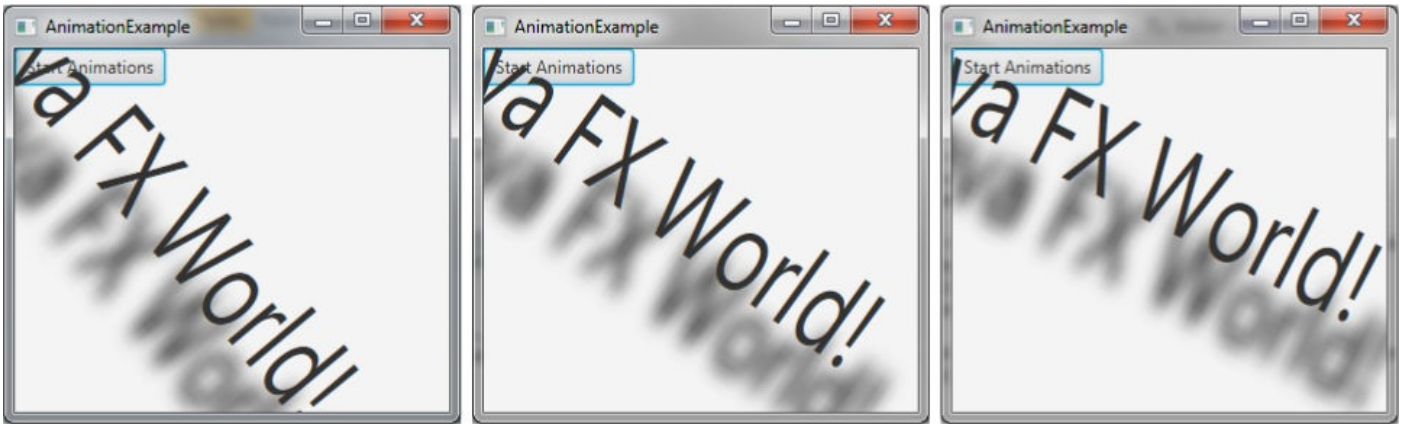


Abbildung 14-16 *Animationen mit JavaFX*

14.4 Neuerungen in JavaFX 8

Mittlerweile haben wir uns ein recht gutes Grundlagenwissen zu JavaFX erarbeitet. Neben dem Einsatz von Bedienelementen mit Action Handling können wir die Benutzeroberfläche auch mit Effekten und Animationen interessant gestalten. Mit diesem Know-how wollen wir uns nun anschauen, welche Neuerungen JavaFX in Version 8 erfahren hat. Es bietet u. a. folgende Erweiterungen und Änderungen:

- Unterstützung von Lambdas als `EventHandler<T extends Event>` (implizit durch die Sprache)
- Texteffekte
- Zwei neue Controls, nämlich `DatePicker` zur Datumsauswahl und `TreeTableView` als Kombination von Tabellen- und Baumdarstellung
- 3D-Support
- Verbesserungen der Performance
- Optische Auffrischung durch ein neues Look-and-Feel namens Modena

Nachfolgend gehen einzelne Abschnitte auf die zuvor aufgelisteten Neuerungen ein, wobei die Verbesserungen der Performance sowie das neue Look-and-Feel namens Modena nicht explizit thematisiert werden. Letzteres wird lediglich im Rahmen der Beschreibung der neuen Bedienelemente gezeigt.

14.4.1 Unterstützung von Lambdas als `EventHandler`

Wie schon bei der Vorstellung der Lambdas kurz gesehen, kann man statt anonymer innerer Klasse nun Lambdas einsetzen, um `EventHandler<T extends Event>` zu realisieren. Das ist möglich, weil das Interface `EventHandler<T extends Event>` ein Functional Interface ist, also genau eine abstrakte, zu realisierende Methode deklariert. Wir beschränken uns auf ein kurzes Beispiel, da die nachfolgenden Beispiele noch Gebrauch von Lambdas als `EventHandler<T extends Event>` machen werden.

```
// Old Style
btn.setOnAction(new EventHandler<ActionEvent>()
{
    @Override
    public void handle(final ActionEvent event)
   {
        System.out.println("Hello World!");
    }
});

// New Style
btn.setOnAction( event -> System.out.println("Hello World!") );
```

14.4.2 Texteffekte

Texte ließen sich bis JavaFX 2.X im Gegensatz zu anderen `Nodes` nicht im gleichen Umfang per CSS stylen. Mit JavaFX 8 ist dies nun möglich. Schauen wir uns ein Beispiel an, das einfarbige Füllungen, solche mit linearem Gradienten sowie unterschiedliche Strichstärken und auch eine Rotation auf verschiedene Texte anwendet.

Abbildung 14-17 *RichTextExample*

Das Beispiel lässt sich als Programm RICHTEXTEXAMPLE starten und basiert

vollständig auf FXML und CSS. Das zugehörige FXML sieht wie folgt aus:

```
<Scene width="600" height="250" fill="white"
xmlns:fx="http://javafx.com/fxml">
    <stylesheets>
        <URL value="@richtext.css" />
    </stylesheets>
    <VBox>
        <TextFlow styleClass="paragraph">
            <Text styleClass="text1">Hello </Text>
            <Text text=" " />
            <Text styleClass="text2, big">Bold</Text>
            <Text text=" " />
            <Text styleClass="text3, dropshadow">Effect</Text>
        </TextFlow>

        <TextFlow styleClass="paragraph">
            <Text styleClass="underlinefancy">
                fancy underline
                <effect>
                    <DropShadow color="BLACK" offsetX="3.0" offsetY="3.0" />
                </effect>
            </Text>
        </TextFlow>

        <TextFlow styleClass="paragraph">
            <Text styleClass="rotatedThai">Thai: ??????????????????</Text>
        </TextFlow>
    </VBox>
</Scene>
```

Im Listing sieht man im Tag `stylesheets`, wie man aus einer FXML-Datei direkt auf das gewünschte CSS verweist. Die korrespondierende CSS-Datei `richtext.css` zeige ich aus Platzgründen nur verkürzt:

```
.paragraph {
    -fx-font-family: Dialog;
    -fx-font-size: 60.0px;
    -fx-vgap: 30.0 px;
}

.text1 {
    -fx-stroke: darkblue;
    -fx-stroke-width: 2.5;
    -fx-fill: darkgoldenrod;
}

.text2 {
    -fx-font-weight: bold;
    -fx-fill: linear-gradient(from 0.0% 0.0% to 100.0% 100.0%, repeat,
                              orange 0.0%, red 50.0%);
    -fx-stroke: black;
    -fx-stroke-width: 2.0;
}

…

.rotatedThai {
    -fx-stroke: black;
    -fx-stroke-width: 2.0;
    -fx-rotate: -5.0;
    -fx-fill: linear-gradient(from 0.0% 0.0% to 100.0% 100.0%, yellow 0.0%,
```

```
                                red 50.0%, blue 75.0%, green 100.0%);
    -fx-font-weight: bold;
}
…
```

14.4.3 Neue Controls

Zwar bot JavaFX auch schon vor Version 8 eine Menge an Bedienelementen, aber es fehlten solche zur Datumsauswahl und zur Darstellung von baumartigen Inhalten in Tabellen, die in anderen grafischen Bibliotheken existieren.[4] Mit JavaFX 8 gibt es nun im Package `javafx.scene.control` die zwei wichtigen und lang ersehnten Bedienelemente `DatePicker` und `TreeTableView`, die nachfolgend vorgestellt werden.

DatePicker

An einem Beispiel wollen wir uns kurz das Bedienelement `DatePicker` anschauen. Dabei verwenden wir die zuvor besprochenen Sprachfeatures Lambdas und das neue Date And Time API. Letzteres nutzen wir dafür, die `DatePicker`-Komponente mit dem aktuellen Datum zu initialisieren sowie das gewählte Datum aus dem Bedienelement wieder auszulesen. Ein `DatePicker` bietet dazu einen Konstruktor mit einem Parameter vom Typ `LocalDate`. Zum Auslesen dient die Methode `getValue()`, die ein `LocalDate`-Objekt liefert:

```
public class DatePickerExample extends Application
{
    @Override
    public void start(final Stage primaryStage)
    {
        // DatePicker mit Date And Time API-Funktionalität initialisieren
        final LocalDate localDate = LocalDate.now();
        final DatePicker datepicker = new DatePicker(localDate);

        datepicker.setOnAction(event ->
        {
            // Gewähltes Datum ermitteln
            final LocalDate selectedDate = datepicker.getValue();
            System.out.println("Selected date: " + selectedDate);
        });

        final FlowPane root = new FlowPane();
        root.getChildren().add(datepicker);

        final Scene scene = new Scene(root, 300, 250);
        primaryStage.setTitle("DatePickerExample");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    // …
}
```

Führen wir das Programm DATEPICKEREXAMPLE aus, so bekommen wir eine

Datumsauswahl ähnlich wie in Abbildung 14-18 präsentiert.

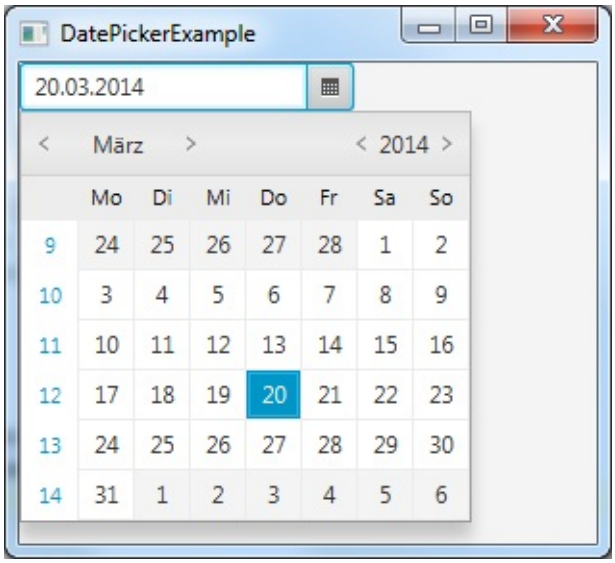


Abbildung 14-18 *Beispiel für den* `DatePicker`

`TreeTableView`

Während der `DatePicker` ein recht einfaches, aber sehr praktisches Bedienelement ist, ist der `TreeTableView` wohl eines der komplexesten Bedienelemente in JavaFX. Um das Bedienelement und vor allem auch das dahinter liegende Datenmodell zu verstehen, schauen wir uns die einzelnen Teile genauer an. Neben den Spalten einer Tabelle stellt ein `TreeTableView` Informationen hierarchisch dar. Nachfolgend wollen wir eine Liste von Personen, die in eine Gruppenhierarchie eingeordnet sind, mit einem `TreeTableView` visualisieren.

Basisdatenmodell und die Klasse `Person`

Die Personen modellieren wir in Form der Klasse `Person`. Diese besitzt die drei Attribute `name`, `age` und `size` und ist wie folgt implementiert:

```
public class Person
{
    private final String name;
    private final Integer age;
    private final Integer size;

    public SimplePerson(final String name, final Integer age,
                        final Integer size)
    {
        this.name = name;
        this.age = age;
        this.size = siz);
    }

    public String getName()
    {
        return name;
    }
```

```
    public Integer getAge()
    {
        return age;
    }
    // …
}
```

Hierarchisches Datenmodell

Das (vereinfachte) Datenmodell besteht aus einer Menge von `Person`-Instanzen. Verschiedene Personen bilden jeweils eine Gruppe. Die Gruppen werden in einer Hauptgruppe `root` namens »All Persons« gebündelt. Die hierarchischen Daten werden über Instanzen vom Typ `TreeItem<T>` modelliert, die auch für den einfacheren `TreeView` zum Einsatz kommen. Sie legen die Baumstruktur fest, die aus Instanzen vom Typ `T` besteht. In diesem Fall sind es `TreeItem<Person>`.

Im folgenden Listing sieht man die Erstellung des Baums aus Objekten vom Typ `TreeItem<T>` durch die Methode `createTreeData()`, die für die Gruppenelemente auf eine Hilfsmethode zurückgreift:

```
private TreeItem<Person> createTreeData()
{
    // Vereinfachtes Datenmodell
    final Person micha = new Person("Micha", 43, 184);
    final Person tom = new Person("Tom", 22, 177);
    final Person lili = new Person("Lili", 34, 170);

    final Person tim = new Person("Tim", 43, 181);
    final Person jens = new Person("Jens", 47, 175);
    final Person andy = new Person("Andy", 31, 178);
    // Hauptgruppe erzeugen

    final TreeItem<Person> root = createGroupTreeItem("All Persons");

    // Gruppe 1 und Subelemente erzeugen
    final TreeItem<Person> group1 = createGroupTreeItem("Group 1");
    final TreeItem<Person> childNode1_1 = new TreeItem<>(micha);
    final TreeItem<Person> childNode1_2 = new TreeItem<>(tom);
    final TreeItem<Person> childNode1_3 = new TreeItem<>(lili);

    // Gruppe 2 und Subelemente erzeugen
    final TreeItem<Person> group2 = createGroupTreeItem("Group 2");
    final TreeItem<Person> childNode2_1 = new TreeItem<>(tim);
    final TreeItem<Person> childNode2_2 = new TreeItem<>(jens);
    final TreeItem<Person> childNode2_3 = new TreeItem<>(andy);

    // Hierarchie und Gruppenzugehörigkeit festlegen
    group1.getChildren().setAll(childNode1_1, childNode1_2, childNode1_3);
    group2.getChildren().setAll(childNode2_1, childNode2_2, childNode2_3);
    root.getChildren().setAll(group1, group2);

    // Aufklappzustand festlegen
    root.setExpanded(true);
    group1.setExpanded(true);

    return root;
}
```

```
private TreeItem<Person> createGroupTreeItem(final String groupName)
{
    return new TreeItem<>(new Person(groupName, null, -1));
}
```

Bereits während der Modellierung fällt auf, dass wir für die Gruppenelemente keine sinnvollen Werte für die Attribute `age` und `size` angeben können Das könnte man durch den Wert `null` ausdrücken. Als Abhilfe kann man sich folgenden Tricks bedienen: Man nutzt einen für das Alter oder die Größe ungültigen Wert, nämlich die zuvor schon eingesetzte `-1`, und sorgt dann in der Darstellung dafür, dass dieser Wert entsprechend behandelt wird. Später erfahren Sie mehr zu dieser Variante.

Erwähnenswert ist noch, dass die Eigenschaft des Aufklappzustands durch `setExpanded(boolean)` gesetzt wird.

Data Binding – Daten und View verknüpfen

Nachdem wir die hierarchische Struktur vorgegeben haben, müssen wir für die Darstellung das sogenannte Data Binding festlegen. Damit ist gemeint, wie die darzustellenden Werte aus den vorliegenden Daten ermittelt (oder im besten Fall bei Änderungen sogar abgeglichen) werden. In diesem Beispiel bestimmen die Attribute der Klasse `Person` die Werte in den Spalten. Die Verbindung zu den Spalten einer `TreeTableView` wird durch Instanzen vom Typ `TreeTableColumn<T,V>` beschrieben, wobei `T` den Typ der Elemente aus den `TreeItem<T>`s repräsentiert. Der Typ `V` entspricht dem Typ des in der Spalte darzustellenden Attributs. Zum Auslesen von Werten nutzt man Instanzen der Klasse `TreeItemPropertyValueFactory`, der man den Namen des Attributs als Text übergibt. Der Wert wird dann per Reflection ausgelesen.

```
private List<TreeTableColumn<Person,?≫ createColumns()
{
   return Arrays.asList(createColumn(“Name”, “name”, 125),
                        createColumn(“Age”, “age”, 50),
                        createColumn(“Size in cm”, “size”, 100));
}

private <V> TreeTableColumn<Person, V> createColumn(final String columnTitle,
                                                    final String
attributeName,
                                                    final int prefWidth)
{
   final TreeTableColumn<Person, V> column = new TreeTableColumn<>
(columnTitle);
   column.setPrefWidth(prefWidth);
   column.setCellValueFactory(
                    new TreeItemPropertyValueFactory<Person, V>
(attributeName));
   return column;
}
```

Kombination zu einem Beispiel

Die beiden zuvor erstellten Methoden kombinieren wir nun zu einem Beispiel für den `TreeTableView` wie folgt:

```
public void start(final Stage stage)
{
    final TreeItem<Person> root = createTreeData();
    final TreeTableView<Person> treeTableView = new TreeTableView<Person>
(root);
    treeTableView.getColumns().addAll(createColumns());

    final VBox vbox = new VBox();
    vbox.getChildren().addAll(treeTableView);

    stage.setScene(new Scene(vbox, 300, 250));
    stage.setTitle("TreeTableViewExample");
    stage.show();
}
```

Starten wir das Programm TREETABLEVIEWEXAMPLE, so sehen wir eine Darstellung ähnlich wie in Abbildung 14-19.

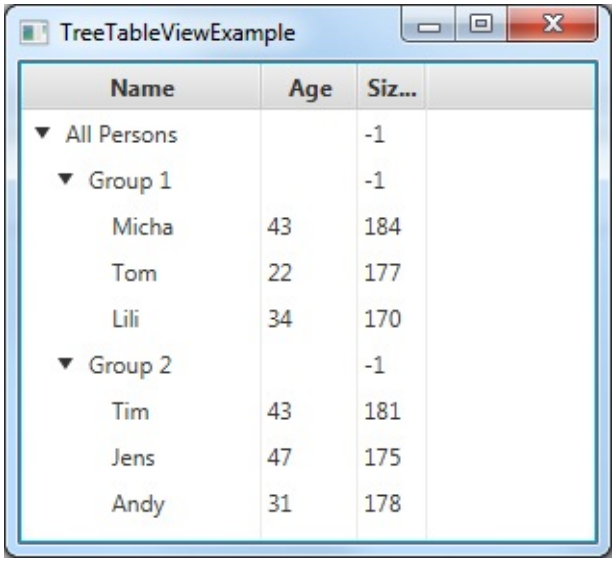


Abbildung 14-19 *`TreeTableView`-Beispiel*

Mit dem Erreichten können wir schon recht zufrieden sein. Es wird bereits einiges an Standardfunktionalität geboten: So wird die Spaltenbreite automatisch dem vom Inhalt benötigten Platz angepasst. Ein Doppelklick auf die Trennlinie zwischen Spalten führt zu einer Größenanpassung und insbesondere lassen sich die Werte einer Spalte durch einen Klick auf den Spalten-Header sortieren.

Hinweis: Sortierbarkeit von Spalten

Die einzelnen Spalten eines `TreeTableView` lassen sich sortieren. Die Sortierung erfolgt automatisch, wenn die Werte das Interface `Comparable<T>` erfüllen und man auf die Spaltenüberschrift klickt. Die Gruppierung bleibt bei einer Sortierung erhalten, es wird also jeweils innerhalb der Gruppen sortiert. Es werden zudem auch mehrere Sortierkriterien unterstützt. Dazu muss man als Benutzer lediglich die Shift-Taste gedrückt halten, wenn man auf den Spalten-Header klickt.

Darstellung anpassen

Bei genauerer Betrachtung stört uns möglicherweise für die Gruppeneinträge der Wert -1 in der Spalte der Größe. Dieser Wert ist durch die gewählte Modellierung bedingt. Zur speziellen Behandlung bei der Darstellung registriert man einen eigenen Renderer, der in JavaFX aus einer Kombination eines Callbacks sowie einer speziellen `Cell<T>`, in diesem Fall einer `TreeTableCell<S,T>`, besteht. Hier wird gezeigt, welche Aktionen man durchführen kann und was dabei ansonsten noch zu beachten ist (siehe fett geschriebene Kommentare). Dazu wird der Aufruf der nachfolgend dargestellten Methode `registerRenderer()` in der Methode `createColumns()` ergänzt und die Methode auch ansonsten minimal angepasst:

```
private List<TreeTableColumn<Person, ?≫ createColumns()
{
    final TreeTableColumn<Person, Integer> sizeColumn = createColumn(“Size”,
                                                                “size”, 70);
    registerRenderer(sizeColumn, ” cm”);

    return Arrays.asList(createColumn(“Name”, “name”, 125),
                         createColumn(“Age”, “age”, 50),
                         sizeColumn);
}

private void registerRenderer(final TreeTableColumn<Person, Integer>
sizeColumn,
                              final String unitPostfix)
{
    sizeColumn.setCellFactory(new Callback<TreeTableColumn<Person, Integer>,
                                           TreeTableCell<Person, Integer≫()

    {
        @Override
        public TreeTableCell<Person, Integer> call(
                            final TreeTableColumn<Person, Integer> p)
        {
            return new TreeTableCell<Person, Integer>()
            {
                @Override
                protected void updateItem(final Integer item,
                                          final boolean empty)


                {
                    super.updateItem(item, empty);

                    if (!empty)
                    {
                        if (item.intValue() == -1)
                        {
                            // Spezialbehandlung für Kein-Wert-Indikator -1
                            setText(null);
                        }
                        else
                        {
                            // Textuelle Ergänzung
                            setText(item + unitPostfix);
                        }
                    }
```

```
                    else
                    {
                        // Ganz wichtig, sonst werden Texte dargestellt,
                        // obwohl Parent zugeklappt ist!
                        setText(null);
                    }
                }
            };
        }
    });
}
```

Diese Korrektur führt dazu, dass keine Werte für den Wert -1 dargestellt und dass die Größenangaben mit dem Zusatz "`cm`" versehen werden, wodurch diese Angabe der Einheit nicht in der Spaltenüberschrift aufgeführt wird.

Zum Ausprobieren können Sie das Programm TREETABLEVIEWEXAMPLE2 starten. Sie erhalten eine Darstellung ähnlich wie in Abbildung 14-20.

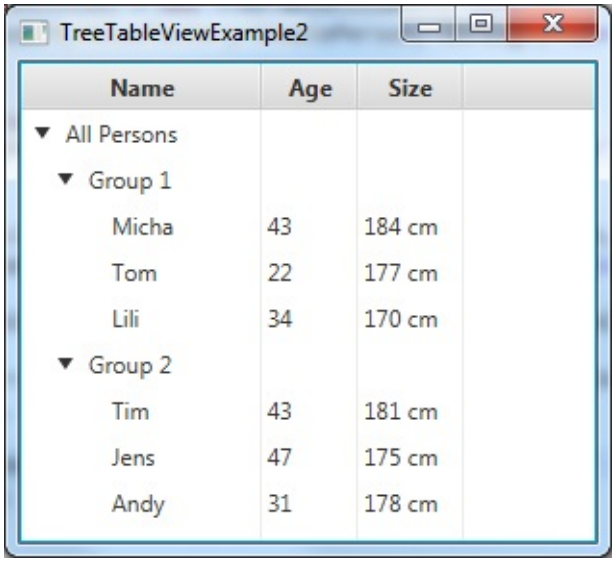


Abbildung 14-20 *Zweites Beispiel zum* `TreeTableView`

14.4.4 JavaFX 3D

Mit JavaFX lassen sich recht ansprechende Anwendungen entwickeln. Teilweise besteht für besondere Visualisierungen aber Bedarf nach einer Darstellung in 3D. Ein Beispiel ist die Simulation eines Containerterminals. Bei Interesse kann man es unter `http://www.youtube.com/embed/AS26gZrYNy8?rel=0` als YouTube-Video anschauen.

Mit JavaFX 8 wurde der Support für die Darstellung von dreidimensionalen Figuren integriert. Zuvor waren lediglich Pseudo-3D-Darstellungen mithilfe perspektivischer Transformationen möglich. Befassen wir uns nun aber mit ein wenig Basiswissen, das wir zur Programmierung eines einfachen 3D-Beispiels benötigen.

Materialien und primitive 3D-Objekte

Objekte in einer Simulation einer dreidimensionalen Welt können durch die Anwendung verschiedener Oberflächen, Materialien und Texturen realitätsnäher gestaltet werden. Als Basis dienen Instanzen von `javafx.scene.paint.PhongMaterial`. Der etwas ungewöhnliche Name geht auf Herrn Phong zurück, der das sogenannte Phong Shading erfunden hat. Ebenso gibt es ein Phong-Beleuchtungsmodell. Beides sorgt dafür, dass 3D-Darstellungen realitätsnah wirken, indem man Polygonflächen mit Farbschattierungen bzw. -abstufungen einfärbt. Dabei bestimmen verschiedene Farben das Reflexionsverhalten.[5] Die Farbbestandteile lassen sich durch Methoden für Instanzen der Klasse `PhongMaterial` setzen. Für Details verweise ich auf die Oracle-Dokumentation.[6]

Zur Erstellung einfacher 3D-Szenarien existieren mit den Klassen `Box`, `Cylinder` und `Sphere` drei vordefinierte Figuren: Diese sind von der abstrakten Klasse `javafx.scene.shape.Shape3D` abgeleitet, die wiederum den Basistyp `Node` besitzt.

Den Figuren können Materialen zugeordnet werden. Zudem lässt sich die Position im dreidimensionalen Raum festlegen und beliebig manipulieren. Beispielsweise ist neben einer Verschiebung entlang der x-, y- oder z-Achse auch eine Rotation möglich.

```
@Override
public void start(Stage primaryStage)
{
    // Zwei Materialien definieren
    final PhongMaterial redMaterial = new PhongMaterial();
    redMaterial.setSpecularColor(Color.FIREBRICK);
    redMaterial.setDiffuseColor(Color.RED);

    final PhongMaterial blueMaterial = new PhongMaterial();
    blueMaterial.setSpecularColor(Color.DODGERBLUE);
    blueMaterial.setDiffuseColor(Color.BLUE);

    // Box als Figur definieren
    final Box redBox = new Box(400, 400, 200);
    redBox.setMaterial(redMaterial);
    redBox.setTranslateX(100);
    redBox.setTranslateY(150);
    redBox.setTranslateZ(500);
    redBox.setRotationAxis(Rotate.Y_AXIS);
    redBox.setRotate(30);

    // Cylinder als Figur definieren
    final Cylinder blueCylinder = new Cylinder(200, 100);
    blueCylinder.setMaterial(blueMaterial);
    blueCylinder.setTranslateX(350);
    blueCylinder.setTranslateY(350);
    blueCylinder.setTranslateZ(150);

    // Gruppieren und etwas nach hinten versetzen
    final Group root = new Group(redBox, blueCylinder);
    root.setTranslateZ(100);
    root.setRotationAxis(Rotate.X_AXIS);
    root.setRotate(25);
```

```
        final Scene scene = new Scene(root, 500, 500);

        // PerspectiveCamera zuordnen
        final PerspectiveCamera perspectiveCamera = new PerspectiveCamera();
        scene.setCamera(perspectiveCamera);

        primaryStage.setTitle("Figures3DExample");
        primaryStage.setScene(scene);
        primaryStage.show();
    }
```

Schlussendlich wird die Darstellung durch eine Instanz einer `PerspectiveCamera` bestimmt. Diese kann man beliebig im 3D-Raum positionieren und sie entspricht etwa dem eigenen Blickwinkel – etwas Ähnliches kennt man z. B. von Google Maps, wo man das Beobachter-Figürchen auch an beliebige Orte setzen kann. Wenn wir das Programm FIGURES3DEXAMPLE starten, erhalten wir eine 3D-Darstellung wie links in Abbildung 14-21.

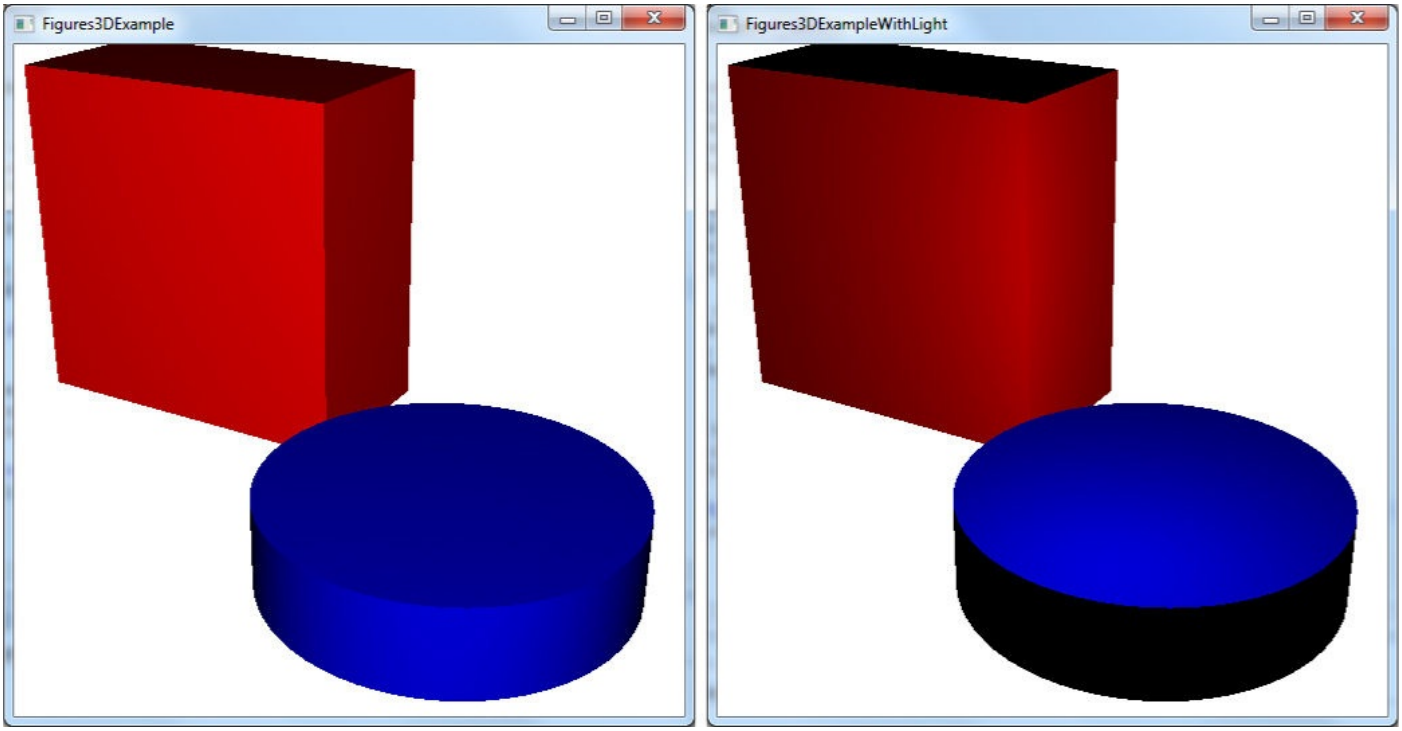


Abbildung 14-21 *3D-Basisfiguren in JavaFX 8 ohne (links) und mit Beleuchtung (rechts)*

Light

In Abbildung 14-21 sieht man, dass eine 3D-Darstellung durch Beleuchtung an Authentizität gewinnt. Lichtquellen sind vom Typ `javafx.scene.LightBase` und können einer `Scene` zugeordnet werden.[7] Es existieren zwei Formen von Lichtquellen:

- `AmbientLight` — Darunter versteht man eine Lichtquelle, die alle Objekte gleichartig ausleuchtet. Ein `javafx.scene.AmbientLight` ist eine Art indirektes Licht, etwa vergleichbar mit Tageslicht.
- `PointLight` – Modelliert eine Lichtquelle mit einer spezifischen Position.

Somit besitzt diese Lichtquelle eine gewisse Entfernung zu anderen Objekten. In der realen Welt sind `javafx.scene.PointLights` etwa Glühbirnen oder Kerzen.

Um den Beleuchtungseffekt auszuprobieren, ergänzen wir einfach folgende vier Zeilen und fügen die Lichtquelle der Gruppe hinzu, wobei wir deren Koordinaten im Raum über verschiedene `setTranslate()`-Methoden festlegen:

```
final PointLight pointLight = new PointLight(Color.ANTIQUEWHITE);
pointLight.setTranslateX(300);
pointLight.setTranslateY(100);
pointLight.setTranslateZ(0);

// Figuren und Lichtquelle bilden eine Gruppe
final Group root = new Group(red, blue, pointLight);
```

Das zugehörige Progamm FIGURES3DEXAMPLEWITHLIGHT produziert eine Ausgabe, wie sie rechts in Abbildung 14-21 gezeigt ist.

14.5 Fazit

Mit den Beispielen zu 3D-Darstellungen endet unser Überblick über JavaFX und dessen Neuerungen in Version 8. Sie sollten mittlerweile einen guten Fundus haben, um mit eigenen Experimenten zu beginnen. Dabei hilft das gelungene und recht intuitive API. Damit kommt man schnell zu ersten Erfolgen, die sich dann zu größeren Beispielen ausbauen lassen.

Abschließend möchte ich noch auf einen Punkt hinweisen: Zwar ist JavaFX ein wirklich umfangreiches GUI-Framework, aber es ist im Gegensatz zu den Werbeversprechungen von Oracle lediglich ein Rich-Client-Framework, aber keine Rich-Client-Plattform (RCP). Letztere definiert sich darüber, dass dort diverse allgemeine Basisfunktionalitäten, die für verschiedenste Applikationen in ähnlicher Form erforderlich sind, bereitgestellt werden. Auf diese Weise vermeidet man einiges an Aufwand bei der Applikationsentwicklung durch die Nutzung der Funktionalitäten aus der RCP. Ein interessanter Artikel, der das Thema im Detail behandelt, ist im Java Magazin 1.14 unter dem Titel »Wo ist die Rich-Client-Plattform für JavaFX?« erschienen.

15 Weitere Änderungen in JDK 8

Dieses Kapitel enthält ein Potpourri verschiedenster Änderungen in JDK 8, von denen man einige in der täglichen Arbeit möglicherweise etwas seltener benötigen wird, aber zumindest sollte man einmal davon gehört haben.

Zunächst stelle ich in Abschnitt 15.1 die praktischen Erweiterungen im Interface `Comparator<T>` vor. In Abschnitt 15.2 wird dann die Klasse `Optional<T>` beschrieben, die optionale Werte als Objekte modelliert. Im Anschluss gehe ich in Abschnitt 15.3 auf parallele Verarbeitungen auf Arrays ein. Danach werden in Abschnitt 15.4 verschiedene Neuerungen im Interface `Map<K, V>` besprochen. Mit JDK 8 wurden auch im Bereich von NIO diverse Erweiterungen vorgenommen. In Abschnitt 15.5 zeige ich einige davon aus der Klasse `Files`. Anschließend besprechen wir mögliche Vereinfachungen bei der Ausführung nebenläufiger Aktionen in Abschnitt 15.6. Darüber hinaus schauen wir uns einführend die neue JavaScript-Engine namens Nashorn an und wie man damit kleine JavaScript-Programme ausführen kann (Abschnitt 15.7). Dann beschreibe ich eine Änderung am Speicheraufbau der JVM (Abschnitt 15.8). Zudem erläutere ich in Abschnitt 15.9 kurz die Möglichkeit, Parameternamen per Reflection abzufragen. Schlussendlich gehe ich in den Abschnitten 15.10 und 15.11 noch überblicksartig auf Erweiterungen bei Base64-Codierungen sowie Annotations ein.

15.1 Erweiterungen im Interface `Comparator<T>`

Wenn man für Objekte eine Sortierung vorgeben möchte, so kann man dazu das Interface `Comparator<T>` passend implementieren. Aus der Beschreibung von Lambdas ist bekannt, dass bei der Realisierung eines `Comparator<T>` in Form einer Klasse einiges an Boilerplate-Code entsteht. Außerdem möchte man meistens nicht den Typ `T` selbst, sondern einige seiner Attribute vergleichen. Zur Verdeutlichung gehen wir von einer Liste von Personen aus. Naheliegend sind die Sortierungen nach Vorbzw. Nachnamen. Diese kann man durch einen `Comparator<Person>` durchführen, dessen `compareTo(Person, Person)`-Methode aus den zwei übergebenen `Person`-Objekten den jeweiligen Namensbestandteil extrahiert und dann vergleicht.

Zunächst werfen wir einen Blick auf eine herkömmliche Realisierung, um zu erkennen, wie umständlich und aufwendig das sein kann. Im Anschluss schauen wir auf einige Erweiterungen im Interface `Comparator<T>` aus JDK 8, die das Erstellen von Komparatoren deutlich vereinfachen.

Herkömmliche Realisierung von Komparatoren bis JDK 7

Zum besseren Verständnis der nachfolgenden Erläuterungen wird im folgenden

Listing eine typische Implementierung eines Komperators gezeigt, welche Personen anhand deren Namen sortiert:

```
final Comparator<Person> compareByName = new Comparator<Person>()
{
    @Override
    public int compare( final Person person1, final Person person2)
    {
        // Spezifische Extraktion eines zu vergleichenden Attributs
        final String value1 = person1.getName();
        final String value2 = person2.getName();

        // Vergleich basierend auf Comparable<T>
        return value1.compareTo( value2);
    }
};
```

Das hier verwendete Prinzip ist für Komparatoren im Grunde immer wieder das gleiche, und der konkrete Ablauf variiert lediglich in der Extraktion der Attribute.

Kurzschreibweise in JDK 8

Seit JDK 8 lässt sich die obige Implementierung mithilfe eines Lambdas deutlich kürzer schreiben. Das gilt nachfolgend im Speziellen, weil dort die Extraktion nicht als einzelne Zwischenschritte aufgeführt ist:

```
final Comparator<Person> compareByName = ( person1, person2) ->
{
    return person1.getName().compareTo( person2.getName());
};
```

Erweiterungen in JDK 8

Weil derartige Vergleiche ein recht häufiger Anwendungsfall sind, wurde das Interface `Comparator<T>` mit JDK 8 um einige nützliche Methoden ergänzt. Diese werden nun aufgezählt und danach anhand kleiner Beispiele genauer vorgestellt:

- `comparing()` – Definiert einen Komparator basierend auf der Extraktion zweier Werte, die sich mithilfe des Interface `Comparable<T>` vergleichen lassen.
- `comparingInt()/-Long()/-Double()` – Definiert einen `Comparator<T>`, wobei der `int-/long-/double`-Wert eines speziellen Attributs des Typs `T` zum Vergleich genutzt wird.
- `naturalOrder()`, `reverseOrder()`, `reversed()` – Diese Methoden erzeugen Komparatoren gemäß der natürlichen, der dazu entgegengesetzten sowie einer zu einem bestehenden Komparator inversen Sortierung.
- `thenComparing()`, `thenComparingInt()/-Long()` und `-Double()` – Hiermit wird die Hintereinanderschaltung von Komparatoren ermöglicht.

`comparing()` – auf `Comparable<T>` basierende Komparatoren

Mithilfe der statischen Methode[1] `Comparator.comparing(Function<? super T,? extends U>)` wird ein Komparator erzeugt. Dazu wird als Parameter ein sogenannter Key-Extractor übergeben, der bestimmt, wie das Attribut aus den zu sortierenden Instanzen, im Beispiel den `Person`-Objekten, ermittelt wird. Der Gewinn dabei ist, dass der fixe Ablauf des Vergleichs basierend auf `Comparable<T>` nicht selbst realisiert werden muss, sondern nur der variable Anteil (die Extraktion) einer klassischen Lösung als Parameter übergeben wird. Nachfolgendes Beispiel zeigt, wie man einen `Comparator<Person>`, der den Namensbestandteil extrahiert und vergleicht, durch Aufruf von `comparing()` erzeugen kann:

```
// Varianten mit Comparator.comparing
Comparator<Person> byName1 = Comparator.comparing(person -> person.getName());
Comparator<Person> byName2 = Comparator.comparing(Person::getName);
```

Hinweis: Internationalisierung und Umlaute

Der zuvor dargestellte Vergleich von Namen basiert auf `compareTo(String)` und vergleicht zwei Strings textuell (basierend auf deren Zeichencode), ohne auf Umlaute und regionale Besonderheiten Rücksicht zu nehmen. Als Abhilfe lässt sich jedoch ein sogenannter Collator vom Typ `java.text.Collator` nutzen. Das ist eine Spezialisierung des Typs `Comparator<T>`, die sprachspezifische Eigenheiten beim Vergleich beachtet, z. B. die Umlaute ä, ö, ü korrekt einsortiert. Weitere Details finden Sie in Abschnitt 10.1.8.

`thenComparing()` – Hintereinanderschaltung von Komparatoren

In gewissen Fällen ist eine Sortierung nach nur einem Kriterium nicht immer ausreichend. Für Personen gilt im Speziellen, dass diese durchaus den gleichen Nachnamen besitzen können. Somit wäre ein zweites Sortierkriterium wünschenswert, um bei Menschen mit gleichen Nachnamen eine eindeutige Reihenfolge zu haben. Dazu werden wir mehrere Komparatoren hintereinander ausführen. Als Basisbausteine definieren wir zunächst verschiedene Komparatoren für einzelne Attribute und kombinieren diese dann durch Aufrufe der Methode `thenComparing(Comparator<? super T>)`:

```
// Komparatoren für ein spezielles Attribut
Comparator<Person> byFirstname = Comparator.comparing(Person::getFirstname);
Comparator<Person> byName = Comparator.comparing(Person::getName);
Comparator<Person> byAge = Comparator.comparing(Person::getAge);

// Kombination von Komparatoren
Comparator<Person> byNameAndFirstname = byName.
                                        thenComparing(byFirstname);
Comparator<Person> byNameAndAge = byName.thenComparing(byAge);
```

Im Listing ist gezeigt, wie einfach sich Komparatoren hintereinander ausführen lassen. Für die Alterswerte erfolgt jedoch ständig ein Auto-Boxing aus einem `int` in ein `Integer`-Objekt, da `thenComparing()` einen auf `Comparable<Integer>` basierenden Komparator erzeugt, `getAge()` aber den Rückgabetyp `int` besitzt.

Verarbeitung primitiver Typen

Nicht in allen Anwendungsfällen ist dieses Auto-Boxing gewünscht oder akzeptabel. Insbesondere bei der Verarbeitung sehr großer Datenmengen oder auf kritischen Pfaden kann sich der Auto-Boxing-Automatismus ungünstig auf die Performance auswirken. Als Abhilfe finden sich im Interface `Comparator<T>` spezialisierte Varianten der `comparing()`-Methode für die primitiven Typen `int`, `long` und `double`. Damit kann man einen Komparator für `Person`-Objekte und deren Attribut `age` als `int`-Wert mithilfe von `comparingInt()` folgendermaßen konstruieren:

```
Comparator<Person> byAge = Comparator.comparingInt(Person::getAge);
```

Spezielle Ordnungen

Manchmal besteht das Bedürfnis, die Sortierreihenfolge zu beeinflussen, z. B. um die Reihenfolge der Elemente umzudrehen. Das gilt etwa für das Sortieren von Tabellenspalten. Das Interface `Comparator<T>` bietet seit JDK 8 die Methode `reversed()` an, hier genutzt, um eine absteigende Sortierung nach Namen zu definieren:

```
final Comparator<Person> byName = Comparator.comparing(Person::getName);
final Comparator<Person> byNameDescending = byName.reversed();
```

Zum Teil möchte man eine natürliche Ordnung mit einem Komparator abbilden. Dazu dient die Methode `naturalOrder()`, die für Strings eine alphabetische und für Zahlen eine aufsteigende Sortierung liefert. Die entgegengesetzte Sortierung erhält man mit der Methode `reverseOrder()`. Wenn man diese umkehrt, erhält man wieder die natürliche Ordnung. Im Listing sind die entsprechenden Methodenaufrufe gezeigt und die resultierende Sortierung durch fett geschriebene Kommentare angedeutet:

```
public static void main(final String[] args)
{
    final Integer[] primes = { 1, 7, 3, 13, 11, 5, 17, 19 };

    // aufsteigend
    final Comparator<Integer> naturalOrder = Comparator.naturalOrder();
    // absteigend
    final Comparator<Integer> reverseOrder = Comparator.reverseOrder();
    // aufsteigend
    final Comparator<Integer> naturalOrderAgain = reverseOrder.reversed();

    sortAndPrint("naturalOrder     ", primes, naturalOrder);
    sortAndPrint("reverseOrder     ", primes, reverseOrder);
```

```
    sortAndPrint("naturalOrderAgain", primes, naturalOrderAgain);
}
private static void sortAndPrint(final String name, final Integer[] primes,
                                 final Comparator<Integer> sortOrder)
{
    Arrays.sort(primes, sortOrder);
    System.out.println(name + ": " + Arrays.toString(primes));
}
```

Das obige Programm NATURALORDEREXAMPLE produziert folgende Ausgaben:

```
naturalOrder     : [1, 3, 5, 7, 11, 13, 17, 19]
reverseOrder     : [19, 17, 13, 11, 7, 5, 3, 1]
naturalOrderAgain: [1, 3, 5, 7, 11, 13, 17, 19]
```

Behandlung von `null`-Werten

Ab und zu sieht man sich der Herausforderung gegenüber, dass die zu sortierenden Datensätze potenziell auch `null`-Werte enthalten. Die üblicherweise genutzten Komparatoren lösen ohne spezielle Behandlung dann eine `NullPointerException` aus. Es gibt aber Anwendungsfälle, bei denen `null`-Werte am Anfang oder am Ende einsortiert werden sollen. Dazu gibt es die beiden Methoden `nullsFirst(Comparator<? super T> comparator)` und `nullsLastComparator<? super T> comparator)`, die wir nachfolgend nutzen, um eine mit `null`-Werten durchsetzte Liste mit Namen zu sortieren:

```
public static void main(final String[] args)
{
    final List<String> names = Arrays.asList("A", null, "B", "C", null, "D");

    // Null-sichere Komparatoren zur Dekoration bestehender Komparatoren
    final Comparator<String> naturalOrder = Comparator.naturalOrder();
    final Comparator<String> nullsFirst = Comparator.nullsFirst(naturalOrder);
    final Comparator<String> nullsLast = Comparator.nullsLast(naturalOrder);

    names.sort(nullsFirst);
    System.out.println("nullsFirst: " + names);
    names.sort(nullsLast);
    System.out.println("nullsLast: " + names);
}
```

Führt man das Programm NULLSAFECOMPARATOREXAMPLE aus, kommt es zu folgenden Ausgaben, die die beschriebene Arbeitsweise zeigen:

```
nullsFirst: [null, null, A, B, C, D]
nullsLast: [A, B, C, D, null, null]
```

Oftmals sind in einer Liste komplexere Objekte gespeichert, und die Sortierung basiert auf einem oder mehreren Attributen. Wenn man dafür einen `null`-sicheren Komparator definieren möchte, muss man achtsam sein (vgl. folgenden Praxistipp). Für die Klasse `Person` und ein über die Methode `getFavoriteColor()` ermitteltes optionales Attribut vom Typ `String` müssen die Komparatoren wie folgt zusammengebaut werden:

```
Comparator<Person> byFavoriteColor = Comparator.comparing(
```

```
                                        Person::getFavoriteColor,
                                        Comparator.nullsFirst(String::compareTo));
```

Hier kommt eine spezielle Variante der Methode `comparing()` zum Einsatz, bei der als erster Parameter der bekannte Key-Extractor übergeben wird. Der zweite Parameter ist ein Komparator, allerdings einer, der für den Typ des zu extrahierenden Attributs spezialisiert ist, hier also für den Typ `String`.

Hinweis: Die Methoden `nullsFirst()` und `nullsLast()`

Trotz ihres sprechenden Namens sind die Methoden `nullsFirst()` und `nullsLast()` nicht ganz so leicht in der Anwendung. Intuitiv könnte man versuchen, sie um eine Komparatorimplementierung zu wickeln. Nachfolgend ist dies etwa zur Sortierung der optionalen Lieblingsfarbe, die als String mit `getFavoriteColor()` ermittelt wird und gegebenenfalls den Wert `null` zurückliefert, gezeigt:

```
Function<Person, String> keyExtractor = Person::getFavoriteColor;
Comparator<Person> byFavoriteColor = Comparator.comparing(keyExtractor);
Comparator<Person> wrongNullsFirst = Comparator.nullsFirst(byFavoriteColor);
```

Diese Variante kompiliert zwar, führt aber zur Laufzeit beim Vergleich von `null`-Werten zu `NullPointerExceptions`.

Folgende Variante kompiliert erst gar nicht:

```
Comparator<Person> invalidNullsFirst = Comparator.nullsFirst(keyExtractor);
```

Das liegt daran, dass hier statt eines `Comparator<Person>` ein Key-Extractor übergeben wird. Dies führt zu einer ziemlich unverständlichen Fehlermeldung:

```
method nullsFirst in interface Comparator<T#2> cannot be applied to given
      types;
  required: Comparator<? super T#1>
  found: Person::ge[…]Color
  reason: cannot infer type-variable(s) T#1
    (argument mismatch; invalid method reference
      method getFavoriteColor in class Person cannot be applied to given
           types
        required: no arguments
        found: T#1,T#1
        reason: actual and formal argument lists differ in length)
  where T#1,T#2 are type-variables:
    T#1 extends Object declared in method <T#1>nullsFirst(Comparator<? super
        T#1>)
    T#2 extends Object declared in interface Comparator

invalid method reference
  non-static method getFavoriteColor() cannot be referenced from a static
      context
```

Sofern man nicht sehr tief in der Materie steckt, wird man wohl aus solchen Fehlermeldungen nicht schlau. Man kann sich zu Recht fragen, ob man dies nicht hätte verständlicher ausdrücken können.

15.2 Die Klasse `Optional<T>`

Für einige Berechnungen kann mitunter kein Ergebnis ermittelt werden. Das ist etwa bei einer erfolglosen Suche oder bei der Bestimmung des Minimums, Maximums usw. für eine leere Menge von Eingaben der Fall. Die Modellierung optionaler oder nicht vorhandener Werte geschah bis JDK 8 in Form von `null`-Werten oder idealerweise mithilfe des NULL-OBJEKT-Musters. Mit JDK 8 bietet sich der Einsatz der Klasse `Optional<T>` an. Ein `Optional<T>` ist ein simpler Container für Werte vom Typ `T` oder aber den Wert `null`. Auf diese Weise können optionale Werte klar ausgedrückt und dadurch `NullPointerExceptions` leichter vermieden werden.

15.2.1 Grundlagen zur Klasse `Optional<T>`

Nachfolgend betrachten wir die Berechnung des Maximums und des Minimums für zwei Arrays von `Integer`-Werten. Das zweite Array enthält allerdings keine Elemente. Hierfür lässt sich das Minimum nicht sinnvoll berechnen. Statt der schon angedeuteten Möglichkeiten zur Modellierung von optionalen Werten wollen wir dies eleganter durch den Einsatz der Klasse `Optional<T>` umsetzen. In unserem Beispiel liefern dazu beide Berechnungen ein `Optional<Integer>` zurück, das einen Wrapper um ein `Integer`-Objekt darstellt. Durch Aufruf von `isPresent()` kann man das `Optional<Integer>`-Objekt befragen, ob tatsächlich ein Wert vorhanden ist. Mit `get()` greift man auf diesen zu. Abschließend sehen wir noch, wie man ein `Optional<T>`-Objekt aus einem Wert durch Aufruf der Methode `of()` bzw. `ofNullable()` konstruieren kann. Letztere nutzt man, falls ein Wert `null` sein kann:

```
public static void main(final String[] args)
{
    final Integer[] sampleValues = {1,3,5,7,11,13,17,19};
    final Integer[] noValues = {};

    // Minimum und Maximum berechnen
    final Comparator<Integer> naturalOrder = Comparator.naturalOrder();
    final Optional<Integer> max =
Arrays.stream(sampleValues).max(naturalOrder);
    final Optional<Integer> min = Arrays.stream(noValues).min(naturalOrder);

    // Minimum und Maximum ausgeben
    System.out.println("max:         " + max);
    System.out.println("min:         " + min);

    // Prüfe, ob es einen Wert gibt
    System.out.println("isPresent?:  " + min.isPresent());

    // Zugriff auf den Wert
    final Integer maxValue = max.get();
    System.out.println("maxValue:    " + maxValue);

    // Konstruktionsmethoden
    final Optional<Integer> optionalFromValue = Optional.of(4711);
```

```
    final Optional<Double> optionalFromNull = Optional.ofNullable(null);
    System.out.println("from Value:  " + optionalFromValue);
    System.out.println("from null:   " + optionalFromNull);
}
```

Führen wir das Programm FIRSTOPTIONALEXAMPE aus, so kommt es zu folgender Ausgabe, die verdeutlicht, dass ein leeres Objekt als `Optional.empty` modelliert wird und ansonsten ein `Optional<T>` ein Container für ein Element vom Typ `T` ist, wobei der hier ummantelte `Integer`-Wert in eckigen Klammern angegeben ist:

```
max:         Optional[19]
min:         Optional.empty
isPresent?:  false
maxValue:    19
from Value:  Optional[4711]
from null:   Optional.empty
```

Wenn das schon alles wäre, hätte man außer der expliziten Kennzeichnung, dass ein Aufrufer nun direkt sieht, dass potenziell kein Ergebnis vorhanden ist, nicht allzu viel gewonnen. Der große Mehrwert besteht aber in verschiedenen Methoden, die Verarbeitungsschritte oder alternative Rückgabewerte erlauben.

Behandlung von Alternativen

Anhand eines Beispiels wird die Behandlung von Alternativen illustriert. Dazu werden die Methoden `ifPresent()`, `orElse()`, `orElseGet()` und `orElseThrow()` genutzt. Diese Methoden erlauben eine recht lesbare Programmierung von Alternativen, falls bei einer Berechnung ein Wert nicht vorhanden sein sollte.

Mit der Methode `orElse()` kann man einen alternativen Wert vorgeben, mit `orElseGet()` eine Berechnung durchführen und mit `orElseThrow()` eine Exception auslösen. Als Parameter dienen jeweils Functional Interfaces, die nachfolgend als Lambdas realisiert sind. Zur Demonstration operieren wir wieder auf einer leeren Eingabe und experimentieren etwas mit den Methoden:

```
public static void main(final String[] args)
{
    final Integer[] noValues = {};

    final Optional<Integer> min = Arrays.stream(noValues).
                                          min(Comparator.naturalOrder());

    // Führe Aktion aus, wenn vorhanden
    min.ifPresent(System.out::println);

    // Alternativen Wert liefern, wenn nicht vorhanden
    System.out.println(min.orElse(-1));

    // Berechne Ersatzwert, wenn nicht vorhanden
    final Supplier<Integer> randomGenerator = () -> (int)(100 *
Math.random());
    System.out.println(min.orElseGet(randomGenerator));

    // Löse eine Exception aus, wenn nicht vorhanden
    min.orElseThrow(() -> new NoSuchElementException("there is no minimum"));
```

```
}
```

Startet man das Programm OPTIONALALTERNATIVESEXAMPLE, so kommt es (durch die Zufallszahlenberechnung) in etwa zu folgenden, hier gekürzten Konsolenausgaben:

```
-1
73
Exception in thread "main" java.util.NoSuchElementException: there is no
minimum
```

Man erkennt, dass die in `ifPresent(Consumer<? super T>)` angegebene Aktion nicht ausgeführt wird, da ja auch kein Objekt in der Eingabe `noValues` verfügbar ist.

Berechnungen werden aber oftmals nicht – wie eben gezeigt – auf Wrapper-Klassen durchgeführt, sondern mitunter auf Werten primitiver Typen. Da `Optional<T>` eine generische Klasse ist, kann man sie für primitive Typen nicht nutzen.

Verarbeitung von primitiven Werten

Weil die Verarbeitung primitiver Werte ein recht gebräuchlicher Anwendungsfall ist, wurde das JDK um besondere Implementierungen von `Optional<T>` erweitert, die für die Typen `int`, `long` und `double` verfügbar sind. Das sind die Klassen `OptionalInt`, `OptionalLong` und `OptionalDouble`. Nachfolgendes Beispiel zeigt die eingangs auf einem `Integer[]` durchgeführten Berechnungen zur Ermittlung von Maximum und Minimum nun für ein Array des primitiven Typs `int`. Darüber hinaus wird durch Aufruf von `average()` der Durchschnitt berechnet:

```
public static void main(final String[] args) {

    final int[] sampleValues = {1,3,5,7,11,13,17,19};

    // Berechnungen auf Streams mit primitiven Werten
    final OptionalInt min = Arrays.stream(sampleValues).min();
    final OptionalInt max = Arrays.stream(sampleValues).max();
    final OptionalDouble avg = Arrays.stream(sampleValues).average();

    System.out.println("min:  " + min);
    System.out.println("max:  " + max);
    System.out.println("avg:  " + avg);
}
```

Beim Betrachten des Listings fällt auf, dass für `min()` und `max()` kein Komparator angegeben werden muss. Hier werden die Zahlen gemäß ihrer natürlichen Ordnung sortiert. Zudem sieht man, dass die Durchschnittsberechnung mit `average()` keinen `int`, sondern einen `double` liefert. Aufgrund seiner Optionalität wird er in Form eines `OptionalDouble`-Objekts zurückgegeben.

Führen wir das Programm OPTIONALPRIMITIVESEXAMPLE aus, so erhalten wir folgende Ausgaben:

```
min: OptionalInt[1]
max: OptionalInt[19]
avg: OptionalDouble[9.5]
```

15.2.2 Weiterführendes Beispiel und Diskussion

Bisher haben wir anhand einfacher Beispiele das API der Klasse `Optional<T>` kennengelernt und bereits ein erstes Gespür für die Möglichkeiten zur Verarbeitung optionaler Werte erhalten. Nun betrachten wir das Ganze mit einem stärkeren Praxisbezug. Als Beispiel wollen wir aus einer Liste von Personen diejenige mit einem bestimmten Namen ermitteln und für diese verschiedene Detailinformationen ausgeben.

Umsetzung mit JDK 7 oder früher

Dazu implementieren wir mit JDK-7-Mitteln etwa folgende zwei Methoden:

```
public Person findByName(final String name, final List<Person> persons)
{
    for (final Person currentPerson : persons)
    {
        if (currentPerson.getName().equals(name))
            return currentPerson;
    }
    return null;
}
void printPersonDetails(final Person person)
{
    final String hometown = person.getHometown();
    //…
}
```

Die gewählte Implementierung erfüllt ihre Aufgabe, jedoch erfordert sie beim Aufrufer zusätzliche Prüfungen für den Fall, dass das gesuchte Element nicht in der Liste enthalten ist. Schauen wir es uns Schritt für Schritt an: Lassen Sie uns eine unbekannte Person wie folgt suchen und danach die Detailinformationen ausgeben:

```
final String desiredName = "Unknown";

// findByName() kann null liefern
final Person searchedPerson = findByName(desiredName, persons);

printPersonDetails(searchedPerson); // NullPointerException bei
person.getXYZ()
```

Wenn die Suche den Wert `null` zurückliefert, so wird bei Aufrufen der Methoden in `printPersonDetails(Person)` eine `NullPointerException` ausgelöst. Oftmals findet man dann als vermeintliche Abhilfe eine `null`-Prüfung wie folgt:

```
final String desiredName = "Unknown";

final Person searchedPerson = findByName(desiredName, persons);
if (searchedPerson != null)
{
```

```
    printPersonDetails(searchedPerson);
}
```

Zwar erhalten wir nun keine `NullPointerException` mehr, allerdings auch keinen Hinweis darauf, dass die gesuchte Person nicht gefunden wurde. Auch vom Design her gibt es eine Schwachstelle: Wir können als Methodenbereitsteller die späteren Nutzer nicht dazu bewegen, mögliche `null`-Werte geeignet abzufragen.

Lassen Sie uns kurz weiter überlegen. Im Allgemeinen ist ein Rückgabe von `null` vieldeutig: Heißt es »nicht gefunden« oder »es ist ein Fehler aufgetreten« oder aber etwas ganz anderes? Im obigen Beispiel wird der erste Fall ausgedrückt und wir könnten folgende Spezialbehandlung ergänzen:

```
final String desiredName = "Unknown";

final Person searchedPerson = findByName(desiredName, persons);
if (searchedPerson != null)
{
    printPersonDetails(searchedPerson);
}
else
{
    showWarnMessage("No such person with name '" + desiredName + "'");
}
```

Beim Betrachten der Methode selbst wie auch des aufrufenden Sourcecodes kann man sich berechtigterweise fragen, ob sich das Ganze nicht schöner realisieren lässt. Wir wissen bereits, dass man mit JDK 8 dazu die Klasse `Optional<T>` nutzen kann.

Variante mit JDK 8 und der Klasse `Optional<T>`

In einem ersten Schritt reimplementieren wir die Suchfunktionalität. Dabei nutzen wir `Optional<Person>` als Rückgabe. Dieses Objekt konstruieren wir mithilfe der Methode `of()`, sofern wir ein Ergebnis ermitteln konnten. Ansonsten geben wir den Wert des Aufrufs `Optional.empty()` als Hinweis für kein Resultat zurück:

```
public static Optional<Person> findByName(final String name,
                                          final List<Person> persons)
{
    for (final Person currentPerson : persons)
    {
        if (currentPerson.getName().equals(name))
            return Optional.of(currentPerson);
    }
    return Optional.empty();
}
```

Unser Beispiel bauen wir durch Aufrufe von `isPresent()` und `get()` wie folgt um:

```
final String desiredName = "Unknown";

final Optional<Person> optionalPerson = findByName (desiredName, persons);
if (optionalPerson.isPresent())
{
    printPersonDetails(optionalPerson.get());
```

```
}
else
{
    showWarnMessage("No such person with name '" + desiredName + "'");
}
```

So wirklich erkennt man den Vorteil hier noch nicht – es ist sogar minimal mehr Sourcecode erforderlich. Der entscheidende Punkt wird dann deutlich, wenn man sich den Sourcecode nochmals genauer anschaut und ein wenig überlegt:

- Die Methode `findByName()` drückt durch Rückgabe von `Optional<Person>` klar aus, dass ein optionaler Wert zurückgeliefert wird.
- Die Methode `printPersonDetails(Person)` erwartet ein Objekt vom Typ `Person`. Wenn man im Sourcecode konsequent `Optional<T>` nutzt, ist offensichtlich, dass hier niemals der Wert `null` übergeben werden darf. Auch ist es nicht möglich, versehentlich ein `Optional<T>` zu übergeben, wie man es als Ergebnis einer Suche erhält.

Gerade der letzte Punkt zeigt, dass sich der konsequente Einsatz von `Optional<T>` positiv auswirken kann und unerwartete Übergaben von `null` recht unwahrscheinlich macht. Das bedeutet auch, dass nach der Einführung von `Optional<T>` der Wert `null` nur noch selten verwendet werden sollte.

15.3 Parallele Operationen auf Arrays

Mit JDK 8 wurde die Utility-Klasse `java.util.Arrays` um die überladene Methode `parallelSort()` erweitert. Damit ist es für diverse Typen möglich, Arrays parallel unter Ausnutzung mehrerer Threads zu sortieren. Zudem gibt es Methoden, die Berechnungen basierend auf dem aktuellen Index oder den Vorgängerwerten ausführen.

Die Methode `parallelSort()`

Nachfolgendes Beispiel illustriert zunächst nur einen einfachen Aufruf des parallelen Sortierens eines `int`-Arrays:

```
public static void main(final String[] args)
{
    final int[] numbers = { 1, 9, 2, 8, 7, 3, 5, 6, 4, 10 };

    System.out.println("Initial: " + Arrays.toString(numbers));
    Arrays.parallelSort(numbers);
    System.out.println("Sorted:  " + Arrays.toString(numbers));
}
```

Das Programm PARALLELARRAYSORTEXAMPLE führt zu folgender Ausgabe:

```
Initial: [1, 9, 2, 8, 7, 3, 5, 6, 4, 10]
Sorted:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Wenn man die parallele Sortierung vorschnell auch für kleinere Datenbestände in der

Hoffnung auf einen Performance-Boost einsetzt, wird man enttäuscht. Die zur Parallelverarbeitung benötigte Aufteilung in mehrere Teilberechnungen sowie die Abstimmung der Threads zum Zusammenführen der Teilergebnisse sorgen für einen gewissen Extraaufwand, der (geringfügig) zusätzliche Laufzeit kostet. Somit lohnt sich die Parallelverarbeitung erst bei größeren Datenmengen oder wenn man Sortierungen mehrmals ausführt. Ansonsten kann sich die Parallelisierung sogar negativ auf die Laufzeit auswirken. Nachfolgendes Beispiel zeigt dies für eine sequenzielle und eine parallele Sortierung für verschiedene Größen von Arrays.

```
public static void main(final String[] args)
{
    for (int i=0; i < 3; i++)
    {
        final long[] limits = {10_000, 100_000, 1_000_000, 10_000_000 };
        for (long currentLimit : limits)
        {
            // IntStream und Zufallszahlen-Generierung
            final IntStream iteratingValues = IntStream.iterate(0,
                            x -> { return (int) (100_000 * Math.random()); }
);
            // Beschränkung auf aktuelles Limit
            final IntStream truncated = iteratingValues.limit(currentLimit);

            // Umwandlung in Array
            final int[] array1 = truncated.toArray();
            final int[] array2 = Arrays.copyOf(array1, array1.length);

            // Sortierung sequenziell und parallel ausführen und messen
            final long start1 = System.nanoTime();
            Arrays.sort(array1);
            final long end1 = System.nanoTime();
            Arrays.parallelSort(array2);
            final long end2 = System.nanoTime();

            printResult(currentLimit, start1, end1, end2);
        }
    }
}
private static void printResult(final long currentLimit, final long start1,
                                final long end1, final long end2)
{
    System.out.println("Current limit:          " + currentLimit);
    System.out.println("sequential sort:        " + (end1-start1));
    System.out.println("parallel sort:          " + (end2-end1));
    final double factor = ((double)(end2-end1))/(end1-start1);
    System.out.println("parallel : sequential:  " +
                       NumberFormat.getPercentInstance().format(factor));
    System.out.println();
}
```

Führt man das Programm ParallelArraySortExample2 aus, so erfolgt die erste Sortierung signifikant langsamer, teilweise bei kleinen Arrays um bis zu Faktor 2000! Bei Arrays mit mehr als etwa 100.000 Elementen ist es häufig noch Faktor 1,5 bis 2. Erst nachfolgende Aufrufe der parallelen Sortierungen sind dann etwa um den Faktor 2 bis 4 schneller. Hier macht sich der Effekt bemerkbar, dass die JVM die HotSpots, also die sehr häufig ausgeführten Programmteile, kompiliert und diese danach extrem

schnell ausgeführt werden können. Sehr deutlich sieht man den Übergang bei 100.000 ausgeführten Anweisungen. Für eine Million und mehr ist die parallele Variante ca. 4-mal schneller:

```
Current limit:          10000
parallel : sequential: 2.138%

Current limit:          100000
parallel : sequential: 272%

Current limit:          1000000
parallel : sequential: 23%

Current limit:          10000000
parallel : sequential: 28%
```

Die Methode `parallelSetAll()`

Nicht nur das Sortieren, sondern auch die Berechnung einer gewünschten Funktion für jeden Wert eines Arrays stellt eine sehr gut parallelisierbare Aktion dar. Nachfolgend zeige ich stellvertretend als Berechnung die Erhöhung um den Wert eins und danach eine Multiplikation bzw. Division mit dem Wert zwei, abhängig vom momentanen Wert. Bitte beachten Sie, dass an den Lambda der Index als Parameter übergeben wird und der Zugriff auf die Elemente des Arrays explizit ausprogrammiert werden muss:

```
public static void main(final String[] args)
{
    final int[] numbers = { 1, 3, 7, 15, 31, 63 };
    System.out.println("Initial:     " + Arrays.toString(numbers));

    // Inkrement - Achtung hier wird der Index übergeben, nicht der Wert
    final IntUnaryOperator increment = i -> { return numbers[i] + 1; };

    Arrays.parallelSetAll(numbers, increment);
    System.out.println("Increment:   " + Arrays.toString(numbers));

    // Alle Werte < 10 durch 2 teilen, alle anderen mit 2 multiplizieren
    final IntUnaryOperator specialMapping = i ->
    {
        final int value = numbers[i];
        return value < 10 ? (value / 2) : (value * 2);
    };

    Arrays.parallelSetAll(numbers, specialMapping);
    System.out.println("Converted:   " + Arrays.toString(numbers));
}
```

Das Programm ParallelArraySetAllExample produziert folgende Ausgabe:

```
Initial:   [1, 3, 7, 15, 31, 63]
Increment: [2, 4, 8, 16, 32, 64]
Converted: [1, 2, 4, 32, 64, 128]
```

An diesem Beispiel sind mir zwei Dinge wichtig: Zum einen sollte man unbedingt beachten, dass der Parameter im Lambda dem Index im Array entspricht und nicht

etwa dem im Array gespeicherten Wert. Zum anderen erkennt man eine Besonderheit in den Signaturen der Methode. Für `int`-Werte übergibt man einen `IntUnaryOperator`, was semantisch eine »`IntToIntFunction`« ist. Für Arrays vom Typ `long`, `double` oder einem beliebigen Referenztyp übergibt man dagegen `IntToLongFunction`, `IntToDoubleFunction` bzw. eine `IntFunction`. Letztere verdeutliche ich im nachfolgenden Beispiel. Dort wird eine Liste von Namen nachbearbeitet, indem Leerzeichen per `trim()` abgeschnitten und `null`-Werte auf `"-n/a-"` abgebildet werden:

```
public static void main(final String[] args)
{
    final String[] names = { "Andy", " Trim ", null, " Trim ", "Ralph" };
    System.out.println("Initial:     " + Arrays.toString(names));

    // Spezielle Nachbearbeitung von Strings
    final IntFunction<? super String> trimAndMapNullToNA = i ->
    {
        final String value = names[i];
        return value == null ? "-n/a-" : value.trim();
    };

    Arrays.parallelSetAll(names, trimAndMapNullToNA);
    System.out.println("Converted:   " + Arrays.toString(names));
}
```

Das Programm PARALLELARRAYSETALLEXAMPLE2 zeigt, wie man Berechnungen für alle Elemente eines Arrays parallel ausführen kann und gibt Folgendes aus:

```
Initial:      [Andy,  Trim  , null,  Trim  , Ralph]
Converted:    [Andy, Trim, -n/a-, Trim, Ralph]
```

Die Methode `parallelPrefix()`

Eine sehr spezielle Funktionalität wird durch die Methode `parallelPrefix()` realisiert: Hierbei werden die Elemente des Arrays jeweils mit dem Wert des Vorgängers verknüpft. Das scheint zunächst gar nicht so nützlich zu sein, tatsächlich kann man damit aber mathematische Berechnungen auf einfache Weise realisieren. Beispielsweise kann man die Summe für alle Werte bilden oder aber die Fakultät berechnen, und das Ganze sogar parallel.

Nachfolgendes Listing zeigt dies für ein `int`-Array und die Werte 1 bis 10, für die wir die Summe und die Fakultät berechnen:

```
public static void main(final String[] args)
{
    final int[] numbers1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    System.out.println("Initial: " + Arrays.toString(numbers1));

    //  Berechne die Summe: sum = 1 + 2 + … + n
    final IntBinaryOperator sum = (x, y) -> x + y;
    Arrays.parallelPrefix(numbers1, sum);
    System.out.println("sum:     " + Arrays.toString(numbers1));

    //  Berechne die Fakultät: n! = 1 * 2 * … * n
    final int[] numbers2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
    final IntBinaryOperator fak = (x, y) -> x * y;
    Arrays.parallelPrefix(numbers2, fak);
    System.out.println("fak:      " + Arrays.toString(numbers2));
}
```

Im Listing sieht man, dass zwei gleiche Arrays definiert werden. Das geschieht, damit man jeweils die gleiche Ausgangsbais besitzt, weil die Summierung die Daten ändert und die Fakultät auch wieder für die Wert 1 bis 10 berechnet werden soll. Führt man das Programm PARALLELARRAYPREFIXEXAMPLE aus, so erhält man folgende Ausgabe:

```
Initial: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sum:     [1, 3, 6, 10, 15, 21, 28, 36, 45, 55]
fak:     [1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

15.4 Erweiterungen im Interface `Map<K, V>`

Das Interface `Map<K, V>` wurde in JDK 8 erweitert, beispielsweise in Form der Methoden `getOrDefault(K, V)`, `putIfAbsent(K, V)` usw. Anhand eines Beispiels wollen wir nachvollziehen, wie wir die neuen Methoden im Interface `Map<K, V>` gewinnbringend einsetzen können. Nehmen wir an, wir müssten für eine Liste von Wörtern deren Häufigkeiten bestimmen. Weil uns die dazu benötigten Testdaten in einigen Listings begleiten werden, zeige ich zunächst einmalig die Methode, die diese Werte bereitstellt:

```
private static List<String> createTestData()
{
    final List<String> wordList = Arrays.asList("Dies", "ist", "eine",
"Liste",
                        "Eine", "Liste", "kann", "Worte", "enthalten",
                        "Dies", "ist", "das", "Ende", "der", "Liste");
    return wordList;
}
```

Realisierung mit JDK 7 oder früher

Um die Häufigkeiten von Wörtern in einem Text zu ermitteln, haben wir schon Streams und die `groupingBy()`-Methode genutzt. Hier zeige ich, wie man dies herkömmlich mithilfe einer `Map<K, V>` ausprogrammieren könnte:

```
final List<String> wordList = createTestData();

final Map<String, Integer> wordCounts = new TreeMap<>();
for (final String word : wordList)
{
    // Wortvorkommen hoch zählen bzw. anlegen, wenn zuvor nicht existent
    if (wordCounts.containsKey(word))
    {
        final Integer oldValue = wordCounts.get(word);
        wordCounts.put(word, oldValue + 1);
    }
    else
    {
       wordCounts.put(word, 1);
    }
```

```
}

System.out.println(wordCounts);
```

Wir sehen die Behandlung verschiedener Sonderfälle, beispielsweise wenn kein Wert vorhanden ist sowie das Auslesen des alten und Setzen des neuen Werts. Das wirkt bereits ein wenig unelegant. Insbesondere problematisch sind zwei Dinge: Erstens muss man diese Funktionalität für andere, ähnliche Anwendungsfälle immer wieder erneut ausprogrammieren, im Speziellen auch, wenn lediglich andere Typen für Key oder Value genutzt werden. Zweitens ist eine derartige Verarbeitung kritisch, falls durch andere Threads Änderungen während der Verarbeitung erfolgen. Die Probleme bei Multithreading entstehen dadurch, dass die Abarbeitung der Anweisungen nahezu jederzeit unterbrochen werden kann und andere Threads Veränderungen in der Map bewirken können, sodass im Anschluss ein anderer Zustand existiert als vor der Unterbrechung und z. B. bei der Prüfung. Natürlich kann man durch Synchronisierung für einen kritischen Bereich und die exklusive Ausführung sorgen, dies geschieht jedoch auf Kosten der Möglichkeit zur parallelen Abarbeitung. Eine weitere Alternative wären Locks.

Zusammenfassend lässt sich feststellen, dass man diese Methoden zwar relativ einfach in ihrer Funktionalität nachbauen kann, es jedoch eher schwierig ist, dies Thread-sicher und bei konkurrierenden Zugriffen korrekt hinzubekommen. Umso angenehmer ist es, dass diese Methoden von der für Multithreading und konkurrierende Zugriffe ausgelegten Klasse `ConcurrentHashMap<K, V>` angeboten werden. Nachfolgend wollen wir uns vor allem um Lesbarkeit und Verständlichkeit und weniger um Multithreading kümmern. Dazu lernen wir einige neue Methoden im Interface `Map<K, V>` kennen.

Die Methode `getOrDefault()`

Oftmals wünscht man sich beim Zugriff auf eine Map, dass ein Defaultwert zurückgeliefert werden kann, falls kein Eintrag zu einem gewünschten Schlüssel existiert. Diese Funktionalität wird in JDK 8 durch die Methode `getOrDefault(Object, V)` realisiert. Man vermeidet dadurch ansonsten notwendige Spezialbehandlungen. Eine Methode, die analog arbeitet, könnte man wie folgt realisieren:

```
// Achtung: Nur für Singlethreading korrekt
public Object getOrDefaultSimplified(final Object key,
                                      final V defaultValue)
{
    if (!map.containsKey(key))
       return defaultValue;

    final Object value = map.get(key);
    return value;
}
```

Die Methoden `putIfAbsent()` und `replace()`

Im Interface `Map<K, V>` konnte man bis JDK 8 durch Aufrufe von `put(K, V)` Werte zu einem Schlüssel sowohl in die Map einfügen als auch einen bereits existierenden Wert überschreiben. In JDK 8 werden nun mit `putIfAbsent(K, V)` und `replace(K, V)` zwei neue speziellere Funktionen geboten. Wie bereits am Namen zu vermuten ist, fügt `putIfAbsent(K, V)` nur dann einen Wert ein, wenn zuvor noch keiner existierte. Für `replace(K, V)` gilt es andersherum: Mit der Methode `replace(K, V)` werden lediglich schon vorhandene Einträge ersetzt. Existiert kein Wert, passiert nichts.

Beispiel Wörterzählen

Die drei Methoden `getOrDefault(Object, V)`, `putIfAbsent(K, V)` und `replace(K, V)` kombinieren wir für das Wörterzählen wie folgt:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        // Initialen Wert vorgeben, Achtung 0, weil später Inkrement erfolgt
        wordCounts.putIfAbsent(word, 0);
        // Wert ermitteln, wenn vorhanden
        final Integer value = wordCounts.getOrDefault(word, 0);
        // Wert ersetzen
        wordCounts.replace(word, value + 1);
    }

    System.out.println(wordCounts);
}
```

Zwar ist diese Realisierung kürzer, jedoch möglicherweise nicht intuitiv verständlich. Insbesondere muss beim ersten Hochzählen ein wenig getrickst werden.

Die Methoden `computeIfAbsent()` und `computeIfPresent()`

Manchmal soll nicht nur ein ganz bestimmter Wert mit `putIfAbsent(K, V)` in eine Map eingefügt werden, sondern stattdessen eine Berechnung ausgeführt werden. Das kann man unter anderem dazu nutzen, um eine sogenannte Multi Map zu realisieren, bei der für einen Schlüssel mehrere Werte gespeichert werden können. Existiert noch kein Wert für einen Schlüssel, so muss zunächst eine Collection angelegt werden. Das implementieren wir wie folgt:

```
// Achtung: Nur für Singlethreading korrekt
if (!map.containsKey(key))
{
    map.put(new ArrayList<>();
```

```
}
```

Die obige Realisierung ist nur für Singlethreading korrekt, bei Multithreading könnten mehrere Threads zeitgleich die Prüfung vornehmen und danach unterbrochen werden. Nachfolgendes Hinzufügen von Elementen wird dann möglicherweise durch frisch initialisierte `ArrayList<E>`-Instanzen wieder zunichtegemacht. Die Standardimplementierung als Default-Methode im Interface `Map<K, V>` löst dieses Problem zwar nicht, verbessert aber die Lesbarkeit:

```
map.computeIfAbsent(key, it -> new ArrayList<>());
```

Nutzen wir als konkrete Realisierung eine `ConcurrentHashMap<K, V>`, so läuft die Ausführung der Methode `computeIfAbsent(K, Function<? super K, ? extends V>)` atomar ab und vermeidet Multithreading-Probleme.

Beispiel Wörterzählen

Für das Beispiel des Wörterzählens kann man die Verarbeitung deutlich klarer wie folgt schreiben:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        wordCounts.computeIfPresent(word, (str, val) -> val + 1);
        wordCounts.computeIfAbsent(word, (val) -> 1);
        // Alternativ: wordCounts.putIfAbsent(word, 1);
    }

    System.out.println(wordCounts);
}
```

Zum Erhöhen des Zählers nutzen wir hier im Aufruf von `computeIfPresent(K, BiFunction<? super K, ? super V, ? extends V>)` einen Lambda, der als Eingabe sowohl den Wert des Schlüssels als auch des Werts erhält und Letzteren um eins erhöht zurückgibt. Falls es noch keinen Eintrag für ein Wort gibt, kann man entweder einen Aufruf von `computeIfAbsent(K, Function<? super K, ? extends V>)` oder die Methode `putIfAbsent(K, V)` nutzen.

Die Methode `merge()`

Die abschließend vorgestellte Methode `merge(K, V, BiFunction<? super V, ? super V, ? extends V>)` realisiert eine Funktionalität, ähnlich zu `computeIfAbsent(K, Function<? super K, ? extends V>)`, die einen existierenden Eintrag mit einer übergebenen Funktion verknüpft: Der neue Wert wird aus dem alten Wert und einer binären Operation ermittelt. Für den Fall, dass es den Wert noch nicht gibt, wird der an die Methode `merge(K, V, BiFunction<? super V, ? super V, ? extends V>)` übergebene

Startwert vom Typ `V` in der Map gespeichert. Zur Verdeutlichung möchte ich dies für das Beispiel des Wörterzählens wie folgt nutzen:

```
public static void main(final String[] args)
{
    final List<String> wordList = createTestData();

    final Map<String, Integer> wordCounts = new TreeMap<>();
    for (final String word : wordList)
    {
        wordCounts.merge(word, 1, Integer::sum);
    }

    System.out.println(wordCounts);
}
```

An der schrittweisen Weiterentwicklung des Beispiels erkennen wir sehr schön, dass man sich von der imperativen Programmierung hin zu einem deklarativen Programmierstil bewegt. Die erste Variante hat den Algorithmus mit 15 Zeilen umgesetzt. Zum Schluss benötigt man nur noch fünf Zeilen. Neben der Kürze kommuniziert die obige Lösung vor allem die gewünschte Funktionalität viel besser.

Fazit

Die zuvor vorgestellten Methoden sind für viele Anwendungsfälle praktisch und erleichtern die tägliche Arbeit. Man kann sich dadurch wieder mehr auf das zu lösende Problem als auf die Details der Zugriffe auf die Map konzentrieren. Insgesamt sinkt durch den deklarativen Ansatz und durch die im Framework implementierten Algorithmen die Wahrscheinlichkeit für Flüchtigkeitsfehler – im Gegensatz dazu kann beim imperativen Ansatz ein kleiner Fehler viel schneller zu unerwarteten Resultaten führen.

Hinweis: Streams und Maps

Möglicherweise haben Sie sich schon im Kapitel über Streams gefragt, wie man denn Maps und Streams zusammenbringt. Tatsächlich werden Streams durch Maps nicht unterstützt. Weil aber Massenoperationen auch für Maps sehr praktisch sind, wurde das Interface `Map<K, V>` um einige Bulk Operations erweitert, nämlich um die Methoden `forEach(BiConsumer<? super K, ? super V>)` und `replaceAll(BiFunction<? super K, ? super V, ? extends V>)`. In der Klasse `ConcurrentHashMap<K, V>` finden sich neben der allgemeinen `forEach()`-Methode noch Spezialisierungen nur für Schlüssel (`forEachKey()`) und nur für Werte (`forEachValue()`).

15.5 Erweiterungen im NIO und der Klasse `Files`

Mit JDK 8 gibt es auch im Bereich NIO (New Input Output[2]) einige Neuerungen. Stellvertretend dafür betrachten wir die Utility-Klasse `java.nio.file.Files`.

Diese wurde um verschiedene Hilfsmethoden erweitert, unter anderem um folgende:

- `lines(Path)` – Stellt eine Datei zeilenweise in Form eines `Stream<String>` bereit.
- `readAllLines(Path)` – Liest eine Datei zeilenweise ein und gibt die Zeilen als `List<String>` zurück.
- `list(Path)` – Liefert den Inhalt eines Verzeichnisses als `Stream<Path>`. Das Besondere dabei ist, dass der Inhalt sukzessive bei Bedarf ermittelt wird und nicht direkt von vornherein. Es wird, wie im Abschnitt über Streams beschrieben, immer nur ein Teil der Daten angefordert, wenn eine Terminal Operation ausgeführt wird.
- `write(Path, Iterable<? extends CharSequence>, OpenOption...)` – Schreibt die übergebenen Textzeilen in die durch den `Path`-Parameter referenzierte Datei. Dabei wird der Schreibmodus durch die angegebene `OpenOption` bestimmt, etwa `APPEND` oder `WRITE`. Ersteres fügt an die Datei an, Letzteres schreibt von Anfang an – und überschreibt gegebenenfalls vorhandene Informationen.

Diese Aufzählung kann nur einen unvollständigen Überblick über die Neuerungen aus JDK 8 geben, da diese deutlich umfangreicher sind. Die oben dargestellten Funktionalitäten sind allerdings besonders nützlich, was ich anhand eines Beispiels verdeutlichen möchte. Hierbei erzeugen und befüllen wir eine Textdatei namens `WriteText.txt` im Temp-Verzeichnis des Systems mit ein wenig Inhalt und nutzen dazu die Methode `write()`. Die in die Datei geschriebenen Informationen ermitteln wir in Form eines `Stream<String>` durch Aufruf von `lines()`. Auf dem Ergebnis wenden wir eine Filterung und eine Gruppierung an. Abschließend inspizieren wir mithilfe von `list()` das Temp-Verzeichnis und ermitteln alle Dateien, die mit `.txt` enden. Dabei gilt es zu beachten, dass man nicht die Methode `endsWith(String)` des `Path`-Objekts nutzen kann, da diese auf Pfadbestandteilen arbeitet und nicht auf Namen. Daher muss zuvor eine Umwandlung in einen String erfolgen:

```
public static void main(final String[] args) throws IOException
{
    final String tempDirPath = System.getProperty("java.io.tmpdir");
    final Path destinationFile = Paths.get(tempDirPath + "/WriteText.txt");
    final List<String> content = Arrays.asList("This", "is", "the",
"content");

    // Datei schreiben
    final Path resultFile = Files.write(destinationFile, content,
                                        StandardOpenOption.CREATE,
                                        StandardOpenOption.APPEND);

    // Zeilenweise als Stream<String> einlesen
    final Stream<String> contentAsStream = Files.lines(resultFile);

    // Filtern und Gruppieren
    final Map<Integer, List<String≫ filterdAndGrouped = contentAsStream.
```

```
                                filter( word -> word.length() > 3).
                                collect( Collectors.groupingBy( String::length));
    System.out.println( filterdAndGrouped);

    // Verzeichnis als Stream<Path> einlesen
    final Stream<Path> tmpDirContent = Files.list( Paths.get( tempDirPath));

    // Fallstrick: endsWith arbeitet auf Path-Komponenten, nicht auf
Dateinamen!
    tmpDirContent.filter( path -> path.toString().endsWith( ".txt")).
                                                    forEach( System.out::println);
}
```

Nach der zweiten Ausführung des obigen Programms FILESEXAMPLE erhalten Sie in etwa folgende Ausgaben, weil wir den APPEND-Modus gewählt haben:

```
{4=[This, This], 7=[content, content]}
C:\tmp\WriteText.txt
```

15.6 Erweiterungen im Bereich Concurrency

Im Bereich Concurrency wurden in JDK 8 verschiedene Erweiterungen realisiert. Das betrifft vor allem die bereits besprochenen Möglichkeiten zur Parallelverarbeitung mit Streams und Arrays sowie das Package `java.util.concurrent` und seine Subpackages mit unter anderem folgenden wichtigen Änderungen:

- `CompletableFuture<T>` – Die Klasse `CompletableFuture<T>` erweitert das Interface `Future<T>` um eine Vielzahl an Methoden. Diese interessante Neuerung werden wir im Anschluss an diese Aufzählung genauer betrachten.
- `ConcurrentHashMap<K, V>` – In der Klasse `ConcurrentHashMap<K, V>` wurde eine Vielzahl an Methoden ergänzt. Das sind unter anderem `computeIfAbsent()`, `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`, `merge()`, `reduce()` und `search()`. Einige davon haben wir schon bei der Betrachtung der Neuerungen im Basisinterface `Map<K, V>` besprochen. Hier möchte ich nur kurz nochmals stellvertretend auf die Methode `putIfAbsent()` eingehen: Insbesondere bei Multithreading vermisst man eine Funktionalität, einen Wert für einen Schlüssel in der Map zu speichern, falls dieser dort noch nicht existiert. Obwohl das einfach klingt, ist es eine Mehrschrittoperation. Eine solche kann zu nahezu beliebigen Zeitpunkten unterbrochen werden, wodurch Inkonsistenzen und Berechnungsfehler durch konkurrierende Zugriffe entstehen können. Herkömmlicherweise musste man entweder mit `synchronized` oder Locks arbeiten, um einen kritischen Abschnitt zu realisieren, in dem zuerst ein Lesezugriff und danach gegebenenfalls ein Schreibzugriff erfolgte. Mithilfe der Methode `putIfAbsent()` kann man sich derartige »Verrenkungen« ersparen.
- `StampedLock` – Die Klasse `java.util.concurrent.locks.StampedLock` ist eine spezielle Variante

eines Locks. Diese ist in ihrer Intention ähnlich einem `java.util.concurrent.locks.ReadWriteLock`. Ein `StampedLock` arbeitet zunächst mit optimistischen Sperren und erlaubt damit performantere Verarbeitungen, wenn viel Parallelität in Form gleichzeitiger Lesezugriffe mit seltenen Schreibzugriffen erfolgt. Führt das optimistische Sperren jedoch zu Konflikten, so muss man auf eine konventionelle Sperrung zurückgreifen. Die Hoffnung besteht darin, dass solche Situationen eher selten auftreten und man im Normalfall von den Performance-Vorteilen der optimistischen Verfahren profitieren kann.

Die Klasse `CompletableFuture`

Bei asynchronen Berechnungen steht man häufig vor der Herausforderung, diese miteinander zu synchronisieren, gewisse Abläufe abzugleichen oder Ergebnisse auszutauschen. Wenn man dies über `Runnable` oder `Callable<T>` realisieren möchte, so wird das recht schnell unhandlich. Schon mit JDK 5 wurde das Interface `java.util.concurrent.Future<V>` eingeführt, um Ergebnisse von asynchronen Berechnungen auszudrücken. Damit kann man eine erste Verbesserung erzielen. Das mit JDK 8 eingeführte `java.util.concurrent.CompletableFuture<T>` führt zu einer deutlichen Vereinfachung. Mithilfe der Klasse `CompletableFuture<T>` beschreibt man Berechnungen als eine Folge von Tasks. Diese kann man durch spezielle Methodenaufrufe miteinander verknüpfen:

- `supplyAsync()` – Konstruiert ein neues `CompletableFuture<T>`-Objekt.
- `thenApply()` – Führt eine Aktion in Form einer `Function<T, R>` aus.
- `thenAccept()` – Führt eine abschließende Aktion aus.

Die Aufrufe erfolgen im aktuellen Thread (blockierend). Darüber hinaus existieren asynchrone Varianten, deren Methodenname mit `Aysnc` endet.

Weil hier ein ganz neues Konzept eingeführt wird, beginne ich mit einem stark vereinfachten Beispiel. Normalerweise nutzt man `CompletableFuture<T>` zur Ausführung länger dauernder Aktionen. Diese sind hier durch einfache Berechnungen stilisiert, etwa das initiale Berechnen eines Ergebnisses als simple Rückgabe eines Strings. Danach werden symbolisch für komplexe Berechnungen zunächst eine Wandlung eines Strings in ein `Integer`-Objekt und dann eine einfache Multiplikation ausgeführt. Abschließend ist gezeigt, wie auf das Berechnungsergebnis zugegriffen wird:

```
public static void main(final String[] args) throws InterruptedException,
                                                     ExecutionException
{
    // Schritt 1: Aufwendige Berechnung, hier nur Rückgabe von einem String
    final Supplier<String> longRunningAction = () ->
    {
        System.out.println("Current thread: " + Thread.currentThread());
```

```
            return "101";
        };
        final CompletableFuture<String> step1 =
                              CompletableFuture.supplyAsync(longRunningAction);

        // Schritt 2: Konvertierung, hier nur Abbildung von String auf Integer
        final Function<String, Integer> complexConverter = Integer::parseInt;
        final CompletableFuture<Integer> step2 =
step1.thenApply(complexConverter);

        // Schritt 3: Konvertierung, hier nur Multiplikation mit .75
        final Function<Integer, Double> complexCalculation = value -> .75 * value;
        final CompletableFuture<Double> step3 =
step2.thenApply(complexCalculation);

        // Explizites Auslesen per get() löst die Verarbeitung aus
        System.out.println(step3.get());
}
```

Mit `CompletableFuture<T>`s kann man auf einfache Weise asynchrone Abläufe erzeugen und an definierten Punkten wieder zusammenlaufen lassen. Führt man das obige Programm FIRSTCOMPLETEABLEFUTUREEXAMPLE aus, so wird dies anhand der Protokollierung des aktuellen Threads im `Supplier<String>` und folgender Konsolenausgabe deutlich:

```
Current thread: Thread[ForkJoinPool.commonPool-worker-1,5,main]
75.75
```

Die Komplexität im Vergleich zu einer Lösung mittels Threads ist erheblich geringer. Man muss sich weder um die Erzeugung von Threads noch um deren Interaktion selbst kümmern. Stattdessen erfolgen die Berechnungen automatisch in einem oder mehreren Threads, die aus dem mit JDK 8 eingeführten `commonPool` des Fork-Join-Frameworks stammen.

Dieses einführende Beispiel sollte nur die grundsätzlichen Abläufe verdeutlichen, damit die Grundlagen für komplexere Ausführungen gelegt sind. Nachfolgend soll der Inhalt einer Datei eingelesen und analysiert werden. Weil man beim (professionellen) Programmieren immer auch ein Augenmerk auf Wartbarkeit und Testbarkeit legen sollte, entwickeln wir die Funktionalität als kleine Bausteine. Diese realisieren wir in Form von Methoden mit folgenden Aktionen: Das Einlesen der Datei geschieht zeilenweise. Die Zeilen werden dann als einzelne Worte aufbereitet. Diese Funktionalität ist Schritt 1 und wird durch die Methode `extractWordsFromFile(Path)` und dem Aufruf von `supplyAsync()` auf einem `CompletableFuture<List<String≫` realisiert. Im Anschluss daran sollen in einem zweiten Schritt zwei Filterungen parallel auf den bereitgestellten Daten durchgeführt werden: Zum einen werden zu ignorierende Wörter herausgefiltert, zum anderen werden Wörter mit weniger als vier Buchstaben aus dem Ergebnis entfernt. Beide Verarbeitungen werden durch `thenApplyAsync()` angestoßen und laufen parallel zu einander ab. In Schritt 3 werden nun die beiden parallel berechneten Ergebnisse miteinander per `thenCombine()` verbunden. Dass dies erst nach Abschluss beider Berechnungen geschieht, darum kümmert sich die Logik aus der Klasse `CompletableFuture<T>`.

Das folgende Listing zeigt, dass man durch den Einsatz der Klasse `CompletableFuture<T>` und ihrer Methoden die Abläufe gut nachvollziehbar gestalten kann:

```
public static void main(final String[] args) throws Exception
{
    final Path exampleFile = Paths.get("src/chxx_jdk8/misc/Example.txt");

    // Schritt 1: Möglicherweise längerdauernde Aktion
    final CompletableFuture<List<String≫ contents =
              CompletableFuture.supplyAsync(extractWordsFromFile(exampleFile));

    contents.thenAccept(text -> System.out.println("Initial: " + text));

    // Schritt 2: Filterungen parallel ausführen
    final CompletableFuture<List<String≫ filtered1 =
                           contents.thenApplyAsync(removeIgnorableWords());

    final CompletableFuture<List<String≫ filtered2 =
                           contents.thenApplyAsync(removeShortWords());
    // Schritt 3: Verbinde die Ergebnisse
    final CompletableFuture<List<String≫ result =
                           filtered1.thenCombine(filtered2,
calcIntersection());

    System.out.println("result: " + result.get());

}
```

Im Listing sind die Verarbeitungsschritte und deren Kombination gut erkennbar, auch ohne die einzelnen Berechnungen im Detail zu verstehen. Die zur Realisierung eingesetzten einzelnen Bausteine haben wir in ähnlicher Form bereits alle einmal kennengelernt. Wir nutzen verschiedene `Predicate<T>`-Objekte, mit denen wir Filterbedingungen formulieren, die wir auf Streams anwenden. Außerdem lesen wir Daten aus einer Datei mithilfe von `Files.readAllLines()`. Diese Daten bearbeiten wir mit `flatMap()`, `map()`, `sorted()` usw., um den Inhalt der Datei als eine Menge von Wörtern zurückzuliefern, wobei zuvor Satzzeichen von den Wörtern abgeschnitten werden. Nachfolgendes Listing zeigt die zuvor aufgerufenen Hilfsmethoden:

```
private static Supplier<List<String≫ extractWordsFromFile(final Path
inputFile)
{
    return () ->
    {
        try
        {

            final List<String> lines = Files.readAllLines(inputFile);

            final Stream<String> words = lines.stream().flatMap(line ->
                                              Stream.of(line.split(" ")));

            final Function<String, String> removePunctationMarks =
                                             removePunctationMarks();

            final Stream<String> mapped = words.map(removePunctationMarks);
```

```
            final Stream<String> sorted = mapped.sorted(
                                                    String.CASE_INSENSITIVE_ORDER);

            return sorted.collect(Collectors.toList());
        }
        catch (final Exception e)
        {
            return Collections.emptyList();
        }
    };
}

private static Function<List<String>, List<String≫ removeIgnorableWords()
{
    final List<String> wordsToIgnore = Arrays.asList("this", "This", "text");
    final Predicate<String> isIgnorable = word ->
wordsToIgnore.contains(word);

    return input -> { return input.stream().filter(isIgnorable.negate())
                                           .collect(Collectors.toList()); };
}

private static Function<List<String>, List<String≫ removeShortWords()
{
    final Predicate<String> isShortWord = word -> word.length() <= 3;
    final Predicate<String> notIsShortWord = isShortWord.negate();

    return input -> { return input.stream().filter(notIsShortWord)
                                           .collect(Collectors.toList()); };
}

private static BiFunction<? super List<String>,
                          ? super List<String>,
                          ? extends List<String≫ calcIntersection()
{
    return (list1, list2) -> { list1.retainAll(list2); return list1; };
}

private static Function<String, String> removePunctationMarks()
{
    final Function<String, String> removePunctationMarks = str ->
    {
        if (str.endsWith(".") || str.endsWith(":") || str.endsWith("!"))
        {
            return str.substring(0, str.length()-1);
        }
        return str;
    };
    return removePunctationMarks;
}
```

Dieses Beispiel hat Ihnen einen ersten Eindruck von den Möglichkeiten der Klasse `CompletableFuture<T>` geliefert. Ein Blick in die API-Dokumentation verrät, dass es noch einiges zu entdecken gibt: Hier finden sich etwa 50 Methoden.

15.7 »Nashorn« – die neue JavaScript-Engine

Seit Version 6 enthält das JDK eine JavaScript-Engine. Mit JDK 8 wurde diese

überarbeitet. Insbesondere ist sie deutlich performanter und besitzt eine bessere Kompatibilität zum JavaScript-Standard als die JavaScript-Engine namens Rhino aus JDK 7.

Bevor wir die Ausführung von JavaScript innerhalb eines Java-Programms betrachten, zeige ich zunächst, wie man ganz allgemein auf Scripting Engines zugreift und sich dazu Informationen beschafft.

Verfügbare Scripting Engines auflisten

Die Funktionalität des JDKs kann durch Scripting Engines ergänzt werden. Standardmäßig ist im JDK 8 eine JavaScript-Engine namens Nashorn vorhanden. Darüber hinaus können aber auch weitere Engines verfügbar sein, z. B. für Groovy.

Den Ausgangspunkt bildet die Klasse `javax.script.ScriptEngineManager`, die eine Liste von `javax.script.ScriptEngineFactory`-Instanzen liefert. Von einer solchen kann man verschiedene charakteristische Eigenschaften ermitteln:

```
public static void main(final String args[])
{
   final ScriptEngineManager manager = new ScriptEngineManager();

   for (final ScriptEngineFactory factory : manager.getEngineFactories())
   {
      System.out.println(factory.getEngineName());
      System.out.println(factory.getEngineVersion());
      System.out.println(factory.getLanguageName());
      System.out.println(factory.getLanguageVersion());
      System.out.println(factory.getExtensions());
   }
}
```

Führen wir das Programm ListScriptingEngines aus, so erhalten wir folgende Ausgaben – falls Sie weitere Scripting Engines installiert haben sollten, etwa Groovy, so werden natürlich auch deren Informationen ausgegeben:

```
Oracle Nashorn
1.8.0
ECMAScript
ECMA - 262 Edition 5.1
[js]
```

Einfache JavaScript-Anweisungen ausführen

Das nachfolgende Beispiel verdeutlicht, wie man JavaScript-Anweisungen ausführen kann. Ausgangspunkt ist wiederum die Klasse `ScriptEngineManager`. Von dieser kann man basierend auf einem Namen mithilfe von `getEngineByName(String)` die dazu passende `javax.script.ScriptEngine`-Instanz abfragen. Scriptcode lässt sich damit durch die Methode `eval(String)` ausführen.

Jede nicht triviale Berechnung benötigt Eingaben oder einen Ablaufkontext. Dazu

kann man Werte mithilfe von `javax.script.Bindings` setzen, die dann in JavaScript-Berechnungen referenziert werden:

```
// Nur für dieses Beispiel throws Exception - in realen Programmen behandeln
public static void main(final String[] args) throws Exception
{
   final ScriptEngineManager manager = new ScriptEngineManager();
   final ScriptEngine engine = manager.getEngineByName("js");

   // Kommando println() ist mit JDK 7 noch erlaubt; JDK 8: nur noch print()
   engine.eval("print('Hello! JavaScript executed from a Java program.')");

   // Data Binding
   engine.put("a", 2);
   engine.put("b", 7);

   final Bindings bindings = engine.getBindings(ScriptContext.ENGINE_SCOPE);
   final Object a = bindings.get("a");
   final Object b = bindings.get("b");
   System.out.println("a = " + a);
   System.out.println("b = " + b);

   // Berechnung ausführen
   final Object result = engine.eval("a + b;");
   System.out.println("a + b = " + result);

   // Ergebnis der Berechnung wird einer JavaScript-Variablen zugewiesen
   final String script = "var ergebnis = Math.max(a, b)";
   engine.eval(script);

   // Wert der Variablen von Engine ermitteln
   final Object result2 = engine.get("ergebnis");
   System.out.println("Math.max(a, b) = " + result2);

   // Typen der Variablen ermitteln
   System.out.println("typeof a = " + engine.eval("typeof a"));
   System.out.println("typeof ergebnis = " + engine.eval("typeof ergebnis"));
}
```

Führt man das obige Programm SIMPLEJAVASCRIPTANDBINDINGDEMO aus, so erhält man folgende Ausgaben:

```
a = 2
b = 7
a + b = 9
Math.max(a, b) = 7.0
typeof a = number
typeof ergebnis = number
```

Hinweis: Derzeit keine Konsolenausgabe mit JavaScript und `println`

Ich habe festgestellt, dass interessanterweise mit JDK 7 und dessen JavaScript-Engine namens Rhino Ausgaben per `println` problemlos ausgeführt werden können. Mit JDK 8 ist das in der ersten offiziellen Version nicht möglich. Allerdings kann man das Kommando `print` sowohl in JDK 7 als auch in JDK 8 ohne Probleme ausführen.

Dynamische Berechnungen ausführen

Meiner Meinung nach wird der große Mehrwert der Scripting Engines leider viel zu selten klar herausgestellt. Bei vielen einfachen Beispielen, die man so findet, fragt man sich nach dem Nutzen.

Meines Erachtens kann man die JavaScript-Engine immer dann gewinnbringend einsetzen, wenn man dynamische Berechnungen ausführen möchte, wie z. B. eine vom Benutzer eingegebene Funktionen in einem bestimmten Wertebereich berechnen, um eine Wertetabelle oder einen Funktionsplotter zu implementieren. Dies ist mit Java-Bordmitteln nur extrem schwierig zu realisieren, insbesondere wenn die Funktion frei vom Benutzer in einem Textfeld eingegeben werden kann.

Weil es hier lediglich um das Prinzip der dynamischen Auswertung geht, nutze ich im Folgenden der Einfachheit halber einen String, der die zu berechnende Formel enthält, die ein Benutzer hätte eingeben können.

```
public static void main(final String[] args) throws Exception
{
    final ScriptEngineManager manager = new ScriptEngineManager();
    final ScriptEngine engine = manager.getEngineByName("js");

    final String calculation = "7 * (x * x) + (3 - x) * (x + 3) / 10";
    System.out.println("f(x) = " + calculation);

    for (int x = -10; x <= 10; x++)
    {
        engine.put("x", x);

        final Object calculationResult = engine.eval(calculation);
        System.out.println("x = " + x + "\t / f(x) = " + calculationResult);
    }
}
```

Führt man diese Programmzeilen als DynamicCalculationExample aus, so wird für den Wertebereich für `x` von -10 bis 10 die entsprechende Formel ausgewertet und das Ergebnis per `eval(String)` berechnet und anschließend folgendermaßen ausgegeben (einige Werte sind ausgelassen):

```
x = -10  / f(x) = 690.9
x = -9   / f(x) = 559.8
…
x = -2   / f(x) = 28.5
x = -1   / f(x) = 7.8
x = 0    / f(x) = 0.9
x = 1    / f(x) = 7.8
x = 2    / f(x) = 28.5
…
x = 9    / f(x) = 559.8
x = 10   / f(x) = 690.9
```

15.8 Keine Permanent Generation mehr

Im Speicherbereich namens Permanent Generation (kurz: Perm Gen) finden sich vor

allem die Metadaten zu Klassen, also Informationen zum Typ der Klasse, implementierten Interfaces und bereitgestellten Methoden. Mit JDK 8 wurde die Perm Gen aus den Speicherbereichen der JVM entfernt und auf eine andere Art realisiert: Zuvor war die Perm Gen von der Größe beschränkt, was immer mal wieder zu Problemen geführt hat. Mit JDK 8 werden die Metadaten der Klassen in einem größenveränderlichen Bereich namens Metaspace abgelegt, der zudem durch nativen Speicher und nicht aus dem der JVM bereitgestellt wird. Warum wurde diese Änderung notwendig?

In umfangreichen Applikationen, die viele externe Abhängigkeiten zu anderen Bibliotheken besitzen und Klassen dynamisch erzeugen, kann es bis einschließlich JDK 7 immer mal wieder vorkommen, dass während der Programmausführung ein `OutOfMemoryError: PermGen Space` auftritt. Ursache ist, dass die Perm Gen mitunter zu klein dimensioniert ist und vollläuft. Das passiert insbesondere dann, wenn keine explizite (Vor-)Einstellung der Perm Gen durch Aufrufparameter der JVM erfolgt. Zwar lässt sich so die initiale (d. h. auch maximale) Größe der Perm Gen zum Programmstart festlegen, allerdings eben nur auf einen fixen Wert, der möglicherweise die Dynamik des Programmablaufs nicht genügend berücksichtigt: Wenn diverse Klassen dynamisch erzeugt oder nachgeladen werden, besteht weiterhin die Gefahr des `OutOfMemoryError`s, selbst wenn zu Programmstart die Perm Gen scheinbar ausreichend groß gewählt schien. Als potenzielle Abhilfe sieht man mitunter eine extrem groß dimensionierte Perm Gen, was wiederum zu Speicherplatzverschwendung führt. Mit der Änderung in der Speicherverwaltung der JVM trägt man diesem Umstand Rechnung und insbesondere verhindert man, dass die früher teilweise als Abhilfe genutzte Überdimensionierung der Perm Gen nicht mehr notwendig ist und damit auch eine Speicherverschwendung vermieden wird.

Sollte man versehentlich trotzdem noch die JVM mit dem entsprechenden Parameter zur Größenanpassungen der Perm Gen, etwa `-XX: PermSize=512m`, starten, so erhält man folgende Warnung, die besagt, dass die Angabe mit JDK 8 ignoriert wird: »`Java HotSpot(TM) 64-Bit Server VM warning: ignoring option PermSize=512m; support was removed in 8.0.`«

15.9 Erweiterungen im Bereich Reflection

Mit Reflection kann man Programme inspizieren und Informationen zu Klassen, Methoden, Parametern usw. ermitteln. Bis JDK 8 war es per Reflection jedoch nicht möglich, die Namen der Parameter einer Methode (oder eines Konstruktors) zu ermitteln. Diese wurden jeweils nur als `arg0`, `arg1` usw. repräsentiert. Für verschiedene Einsatzzwecke kann der Zugriff auf die Namen der Parameter aber durchaus wünschenswert sein. Im Bereich der Webservices existieren dafür spezielle Annotations etwa `@QueryParam` sowie `@PathParam` und in JavaFX 8 wurde dazu die Annotation `@NamedArg` eingeführt. Diese Annotations können per Reflection abgefragt werden.

Die Abfrage von Parameternamen geht mit Java 8 generell etwas einfacher und erfordert keine Annotations und insbesondere keine Angabe bzw. Wiederholung der Namen von Parametern. Das folgende Listing zeigt die Klasse `ReflectionParameterNamesExample`, die eine Selbstinspektion durchführt. Zur Ermittlung und Ausgabe der Informationen zu den Parametern wird auf die mit JDK 8 eingeführte Klasse `java.lang.reflect.Parameter` und auf dort definierte Methoden zurückgegriffen:

```
public class ReflectionParameterNamesExample
{
    public static void main(final String[] args)
    {
        inspectClass(ReflectionParameterNamesExample.class);
    }

    public static void inspectClass(final Class<?> clazz)
    {
        System.out.println("Untersuchte Klasse: " + clazz.getCanonicalName());

        // Zugriff und Ausgabe aller öffentlichen Methoden
        final Method methods[] = clazz.getDeclaredMethods();
        System.out.println("Methoden: ");
        for (final Method method : methods)
        {
            // Neu in JDK 8
            final Parameter[] paramters = method.getParameters();

            final String asString = Modifier.toString(method.getModifiers()) +
                                    // zeigt auch Generics
                                    " " + method.getReturnType().getTypeName()
+
                                    " " + method.getName() +
                                    buildParameterString(paramters);

            System.out.println(methodAsString);
        }
    }


    public static String buildParameterString(final Parameter[] parameters)
    {
        final StringBuilder sb = new StringBuilder("(");

        for (final Parameter param : parameters)
        {
            sb.append(Modifier.toString(param.getModifiers()) + " ");
            sb.append(param.getParameterizedType().getTypeName() + " ");
            sb.append(param.getName());
        }

        return sb.append(")").toString();
    }
}
```

Startet man das Programm REFLECTIONPARAMETERNAMESEXAMPLE, so gibt es die korrekten Namen der Parameter aus:

```
Untersuchte Klasse: jdk8.misc.ReflectionExample
Methoden:
```

```
public static void main(final java.lang.String[] args)
public static java.lang.String buildParameterString(final
                                java.lang.reflect.Parameter[] parameters)
public static void inspectClass(final java.lang.Class<?> clazz)
```

Allerdings funktioniert dies nur, sofern der Sourcecode mit einem speziellen Compiler-Flag übersetzt wurde (siehe Abbildung 15-1 zur Einstellung in Eclipse). Ansonsten würden die Parameternamen einfach als `arg0`, `arg1` usw. ausgegeben.

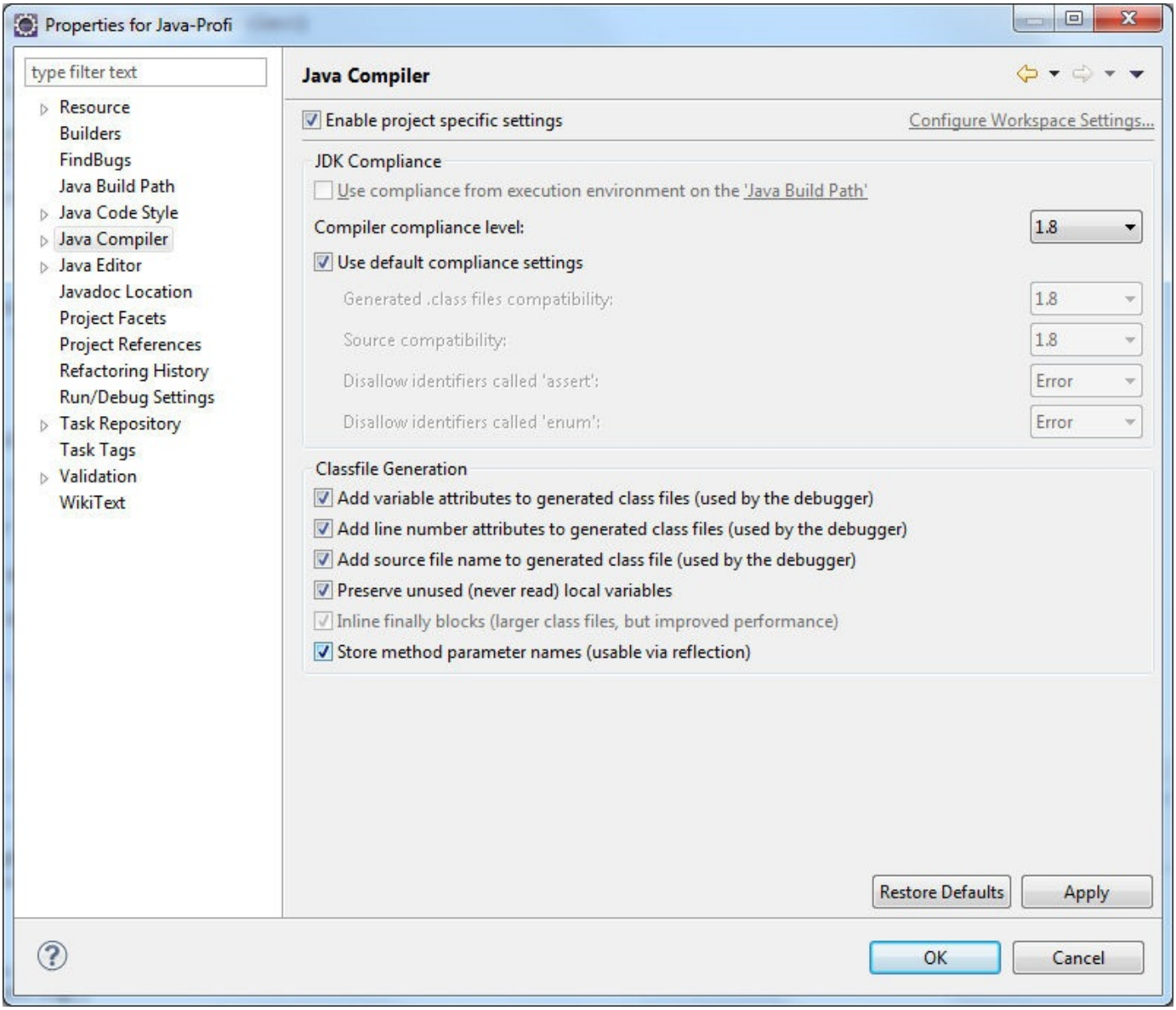


Abbildung 15-1 *ParameterCompilerOption*

15.10 Base64-Codierungen

Lange Zeit war es nur durch Nutzung externer Bibliotheken oder inoffizieller Klassen des JDKs (`BASE64Encoder`/`BASE64Decoder` aus dem Package `sun.misc`) möglich, Base64-Codierungen zu verarbeiten. Diese Funktionalität findet sich endlich auch offiziell im JDK in der Klasse `java.util.Base64`.

Die Base64-Codierung wird etwa für E-Mail-Anhänge im MIME-Format (Multipurpose Internet Mail Extensions) verwendet. Die Art der Codierung wurde

entwickelt, weil mit dem SMTP (Simple Mail Transfer Protocol) nur Zeichen mit 7 Bit übertragen werden konnten. Das ist aber weniger, als die Codierung von ASCII-Zeichen benötigt. Daher wurde eine Codierung erdacht, die auf das oberste Bit verzichtet. Dazu werden jeweils drei Byte der Eingabe (=24 Bit) in vier 6-Bit-Blöcke aufgeteilt. Jeder dieser Blöcke kann eine Zahl zwischen 0 und 63 darstellen, was zu dem Namen Base64 geführt hat. Der Vorteil dieser Codierung ist, dass die resultierende Zeichenfolge nur aus wenigen, codepage-unabhängigen ASCII-Zeichen besteht. Es werden lediglich die Zeichen `A-Z`, `a-z`, `0-9`, `+` und `/` verwendet sowie das Zeichen = als Endemarkierung.

Mit der Base64-Codierung wird dadurch der problemlose Transport von beliebigen Binärdaten möglich. Das kann man dazu nutzen, um Binärdaten, etwa Bilddaten, in Form eines Strings zu verarbeiten. Vereinfachend nutzen wir nachfolgend die Bytes eines Strings zur Umwandlung:

```
public static void main(final String[] args)
{
    final byte[] bytes = "This is the Base64 Test".getBytes();

    final String encoded = Base64.getEncoder().encodeToString(bytes);
    System.out.println("Base64 encoded = " + encoded);

    final byte[] decoded = Base64.getDecoder().decode(encoded);
    System.out.println("Base64 decoded = " + new String(decoded));
}
```

Bei Ausführung des Programms BASE64EXAMPLE wird der String »`This is the Base64 Test`« in Base64 und zurück konvertiert:

```
Base64 encoded = VGhpcyBpcyB0aGUgQmFzZTY0IFRlc3Q=
Base64 decoded = This is the Base64 Test
```

Wie man sieht, hat man bei einer möglichen Übertragung von einem Rechner zu einem anderen ganz nebenbei den Vorteil, dass die Base64-Codierung eine nicht lesbare Darstellung erstellt. Dies bietet zumindest einen kleinen Schutz vor unbeabsichtigten Blicken. Allerdings entsteht durch die Reduktion auf die ausgewählten Zeichen ein Overhead: Das zu übertragende Datenvolumen nimmt um rund 30 % zu.

15.11 Änderungen bei Annotations

Während bis JDK 7 Annotations nur an ganz bestimmten Stellen im Sourcecode erlaubt waren, wurde dies mit JDK 8 deutlich ausgeweitet. Dadurch wird es leichter, Constraints (Randbedingungen) an beliebige Elemente im Sourcecode als Annotations anhängen zu können. Beispielsweise kann man sich für Codeprüfungen etwa folgende Annotations vorstellen:

- `@NonNull`, `@Nullable` – Attribut oder Parameter darf nicht `null` sein bzw. kann den Wert `null` enthalten.
- `@Immutable` – Das Attribut oder die Klasse ist unveränderlich.

- `@ReadOnly` – Es sollen lediglich Lesezugriffe durch andere Objekte erlaubt sein.

Leider werden derartige Annotations im JDK 8 nicht bereitgestellt und auch nicht ausgewertet. Diese selbst zu definieren ist recht leicht folgendermaßen möglich:

```
@Target( value = { ElementType. TYPE_USE})
@Retention( value = RetentionPolicy. SOURCE)
@interface NonNull
{
}

@Target( value = { ElementType. TYPE_USE})
@Retention( value = RetentionPolicy. SOURCE)
@interface Nullable
{
}

// …
```

Wollte man etwa für eine Liste von Strings festlegen, dass diese keine `null`-Elemente enthält, so könnte man Folgendes schreiben:

```
List<@NonNull String> containsNoNullElements = …
```

Diese Annotation führt jedoch nicht dazu, dass diese Bedindung auch tatsächlich eingehalten wird. Dazu bedarf es Tools, wie etwa dem Checker-Framework, das frei zum Download unter `http: //types. cs. washington. edu/checker-framework/` bereitsteht und ähnliche Annotations wie die obigen mitbringt und auch deren Einhaltung prüft.

Meiner Meinung nach wäre es sehr wünschenswert gewesen, gewisse Prüf-Annotations bereits im JDK zu definieren, ähnlich zu denen aus JavaEE, und diese auch durch den Compiler auszuwerten. Weil dies leider nicht der Fall ist, werde ich das Thema Annotations hier nicht weiter vertiefen.

Teil IV

Fallstricke und Lösungen im Praxisalltag

16 Bad Smells

Nachdem wir uns bereits mit einigen fortgeschrittenen Java-Techniken sowie kleineren API-Problemen beschäftigt haben, werfen wir nun einen Blick auf immer wiederkehrende Fallstricke und Probleme, die einem Entwickler regelmäßig begegnen. Man spricht in diesem Zusammenhang auch von »Bad Smells«. Darunter versteht man Abschnitte des Sourcecodes, die im übertragenen Sinne einen schlechten Geruch verbreiten – an denen potenziell etwas »faul« ist. Dieser eingängige Begriff wurde von Kent Beck und Martin Fowler im Buch »Refactoring« [27] zur Beschreibung derartiger Programmabschnitte eingeführt.

In diesem Kapitel werden Sie erfahren, welche Tücken und Fehler sich leicht in den Sourcecode einschleichen. Wenn wir diese Bad Smells erkennen, lokalisieren und verstehen können, sind wir auch in der Lage, Gegenmaßnahmen zu ergreifen. In den folgenden Abschnitten präsentiere ich dazu einen Katalog von Bad Smells, der sich in die Abschnitte 16.1 »Programmdesign«, 16.2 »Klassendesign« und 16.3 »Fehlerbehandlung und Exception Handling« gliedert und diverse Sourcecode-Auszüge unterschiedlicher Qualität zeigt. Wir analysieren jeweils die darin enthaltenen Probleme und schulen damit Ihr Auge. Das Erkennen eines Problems ist gut, allerdings sollten wir auch in der Lage sein, es zu beheben. Daher werden zu jedem Bad Smell Tipps zur Vermeidung gegeben und kleinere Abhilfemaßnahmen sofort vorgestellt. Allgemeingültige und größere Umbaumaßnahmen stelle ich separat in Kapitel 17 »Refactorings« vor. Den Abschluss dieses Kapitels bildet in Abschnitt 16.4 »Häufige Fallstricke« eine Liste mit Beispielen, die nicht die Schwere eines Bad Smells haben, aber dennoch hier erwähnt werden sollen, um diese in eigenen Programmen zu vermeiden.

Bad Smells am Beispiel

Bekanntermaßen beschreiben Bad Smells Sourcecode-Abschnitte, an denen potenziell etwas nicht in Ordnung ist. Diese Programmteile vergrößern die Gefahr für Fehlfunktionen oder enthalten bereits Fehler – manchmal sind solche Sourcecode-Stellen aber auch akzeptabel. Diesen Sachverhalt sollte man immer zunächst genau prüfen und gegebenenfalls durch einen Kommentar beschreiben.

Mit ein wenig Übung springen beim Analysieren von Sourcecode potenziell gefährliche Stellen schnell ins Auge. Außerdem ist es in der Regel so, dass Bad Smells gehäuft auftreten. Zum besseren Verständnis betrachten wir ein Beispiel mit der folgenden Methode `setDataState(int, OperationState)`:

```
public void setDataState(final int department, final OperationState state)
{
    if (this.dataState == null)
    {
        this.dataState = new HashMap<Integer, OperationState>();
    }
```

```
    this.dataState.put(new Integer(department), state);

    if (log.isInfoEnabled())
    {
        String msg = "device data state for department " + department + ": ";
        switch (state)
        {
            case RUNNING:
                msg += "running";
                break;
            case GOING_DOWN:
                msg += "going down";
                break;
            case DOWN:
                msg += "down";
                break;
            case GOING_UP:
                msg += "going up";
                break;
            default:
                msg += "unknown";
        }
        log.info(msg);
    }
}
```

Diese Methode scheint zunächst durchaus in Ordnung. Allerdings offenbaren sich dem geübten Auge hier einige Schwachstellen:

1. **Fehlende Plausibilitätsprüfungen der Parameter** – Durch unerwartete Parameterwerte lauern Probleme: Nur ein kleiner Teil des `int`-Wertebereichs des Parameters `department` entspricht gültigen Werten. Außerdem ist für den Parameter `state` der Wert `null` ungültig und sollte zurückgewiesen werden.
2. **Unpassender Einsatz von Lazy Initialization** – Die Gefahr für mögliche Multithreading-Probleme wird durch den unsynchronisierten Einsatz von Lazy Initialization[1] für das Attribut `dataState` erhöht. Tatsächlich traten diese Probleme beim Einsatz des obigen Sourcecode-Abschnitts in der Praxis auf und führten zu Inkonsistenzen in den gespeicherten Werten.
3. **Unausgewogenheit beim Logging** – Der Logging-Code dominiert die Funktionalität: Nur die ersten Programmzeilen tragen zur Anwendungsfunktionalität bei. Die restlichen Zeilen bereiten Log-Informationen auf und sollten herausfaktoriert werden, um die Struktur klar erkennbar zu machen.

16.1 Programmdesign

Nach diesem einführenden Beispiel stelle ich nun einen Katalog von Bad Smells vor und beginne dabei mit möglichen Problemen im Programmdesign.

16.1.1 Bad Smell: Verwenden von Magic Numbers

Oftmals kommen an beliebigen Stellen im Sourcecode verschiedene Zahlenwerte vor, die explizit als Zahlenliteral – als sogenannte ***Magic Number*** – angegeben werden. Zum Teil existieren funktionale und semantische Abhängigkeiten zwischen diesen Werten, die jedoch nicht im Sourcecode, sondern bestenfalls durch Kommentare ausgedrückt werden können.

Wir betrachten hier eine Klasse `CommandExecutor`, die im Konstruktor einen `int`-Wert erhält, der steuert, wie neue Kommandos verwaltet und hinzugefügt werden. Für diesen Zweck gibt es mehrere Konstanten, unter anderem die Konstante `ADD_AS_LAST` mit dem Wert `2`. Im folgenden Beispiel werden zwei Instanzen der Klasse `CommandExecutor` erzeugt, einmal unter Verwendung der Konstantendefinition und einmal anhand des korrespondierenden Zahlenwerts:

```
final CommandExecutor executor1 = new CommandExecutor( ADD_AS_LAST);
// Magic Number 2
final CommandExecutor executor2 = new CommandExecutor( 2);
```

Warum ist das ein Bad Smell?

Bereits anhand dieses einfachen Beispiels sieht man, dass die Lesbarkeit unter dem Einsatz von Magic Numbers leidet. Dass liegt daran, dass ein reiner Zahlenwert wenig semantische Aussagekraft besitzt. Erschwerend kommt hinzu, dass Änderungen an den Werten – etwa wenn obige Konstante z. B. später den Wert 7 erhält – überall im nutzenden Sourcecode nachgezogen werden müssen. Das ist aufwendig und fehleranfällig, denn alle betroffenen Vorkommen einer Magic Number lediglich anhand ihres Werts aufzuspüren, ist problematisch: Logischerweise repräsentiert nicht jedes Vorkommen der Zahl `2` die Konstante `ADD_AS_LAST`. Wird bei einer solchen Korrektur eine Stelle übersehen, so kann dies zu unerwarteten Resultaten und schwer zu findenden Fehlern führen: Wird irgendwann einmal der Wert der Konstante auf `4711` verändert und eine andere Konstante erhält den Wert `2`, so besagt diese `2` nun beispielsweise das Einfügen an erster Stelle und verändert den Programmablauf damit komplett. Wird dagegen eine andere Programmstelle versehentlich geändert, so kommt es dort zu Fehlern im Programmverhalten, weil der neue Wert (die `4711`) dort keinen Sinn ergibt und möglicherweise sogar außerhalb des zulässigen Wertebereichs liegt.

Tipps und Refactorings

Mit sprechenden Konstantennamen kann man leicht ausdrücken, was bewirkt werden soll, hier das Einfügen an letzter Position. Zudem stellen nachträgliche Änderungen der Werte kein Problem im nutzenden Sourcecode dar. Ein weiterer Vorteil der Verwendung von Konstanten ist, dass diese im Nachhinein, wie in diesem Beispiel sicher sinnvoll, leicht in eine `enum`-Aufzählung umgewandelt werden können. Dies kann ohne größere Änderungen am bestehenden Sourcecode durchgeführt werden.

Wurden jedoch Magic Numbers verwendet, so muss später jedes Vorkommen einzeln überprüft und gegebenenfalls angepasst werden.

16.1.2 Bad Smell: Konstanten in Interfaces definieren

Java erlaubt es, Konstanten in einem Interface zu definieren. Dabei kann man zwei Varianten unterscheiden. In der ersten wird ein Interface ausschließlich zur Definition von Konstanten verwendet, hier durch das folgende Interface `FigureConstants` gezeigt:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureConstants
{
    // Randbreite und -höhe
    int BORDER_WIDTH = 5;
    int BORDER_HEIGHT = 5;

    // Abstand der Punkte im Raster in X- und Y-Richtung
    int GRID_SIZE_X = 20;
    int GRID_SIZE_Y = 20;
}
```

In der zweiten Variante – hier am Beispiel des Interface `FigureIF` – werden sowohl Konstanten als auch Methoden definiert:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureIF
{
    // Rückgabewerte für hitTest
    int BORDER_HIT = 0;
    int TITLE_HIT = 1;
    int SIZE_BOX_HIT = 2;

    int hitTest(final int x, final int y);
    void draw(final Graphics g);
}
```

Warum ist das ein Bad Smell?

Die Definition von Konstanten in Interfaces ist in vielerlei Hinsicht ungünstig. Zunächst ist durch die JLS jede dort definierte Konstante implizit `public static final` und jede Methode `public abstract`. Daher kann die Sichtbarkeit nicht eingeschränkt werden.

Zudem ist es möglich, ein reines Konstanten-Interface durch eine Klasse zu implementieren. Dadurch werden die Konstantennamen in den Namensraum der jeweiligen Klasse aufgenommen, und somit geht der Verweis auf den tatsächlichen Definitionsort der Konstante verloren. Für Konstanten als Bestandteile eines Methoden deklarierenden Interface besteht das Problem gleichermaßen.

In der folgenden Klasse `BadFigure`, die zu Demonstrationszwecken das Interface `FigureConstants` implementiert, ist dies in der Methode `isInside(int, int)` mit der Verwendung der Konstanten `BORDER_WIDTH` und `BORDER_HEIGHT` gezeigt, wobei zum einen eine Referenzierung über `FigureConstants` und zum anderen über `BadFigure` erfolgt:

```
// ACHTUNG: Interfaces (möglichst) so nicht verwenden
public class BadFigure implements FigureConstants
{
    final Rectangle boundingRect;

    BadFigure(final Rectangle boundingRect)
    {
        this.boundingRect = boundingRect;
    }

    public boolean isInside(final int x, final int y)
    {
        final Rectangle innerRect = new Rectangle(boundingRect);

        // Definitionsort der Konstanten BORDER_WIDTH und BORDER_HEIGHT
        // ist bei Referenzierung über BadFigure unklar
        innerRect.grow(-FigureConstants.BORDER_WIDTH,
                       -BadFigure.BORDER_HEIGHT);

        return innerRect.contains(x, y);
    }
}
```

Die beschriebene und im Listing gezeigte »Namensraumverschmutzung« scheint zunächst ein etwas hergeholtes Argument zu sein. Doch in der Praxis erschwert eine solche Vorgehensweise mögliche Erweiterungen, Refactorings und die Nachvollziehbarkeit. Schauen wir uns dazu die Problematik einmal genauer an: Nehmen wir an, eine Klasse `BaseFigure` implementiert versehentlich ein Konstanten-Interface `Constants` mit der Konstante `OK`, anstatt die dort definierten Werte zu referenzieren. Eine Erweiterung der Klassenhierarchie um eine Subklasse `ProcessingFigure` führt zu Kompilierfehlern, wenn diese Subklasse ein Interface `ProcessingResults` implementiert, in dem ebenfalls eine Konstante `OK` definiert ist: Die Konstante `OK` ist dann mehrdeutig. Dies deutet Abbildung 16-1 an.

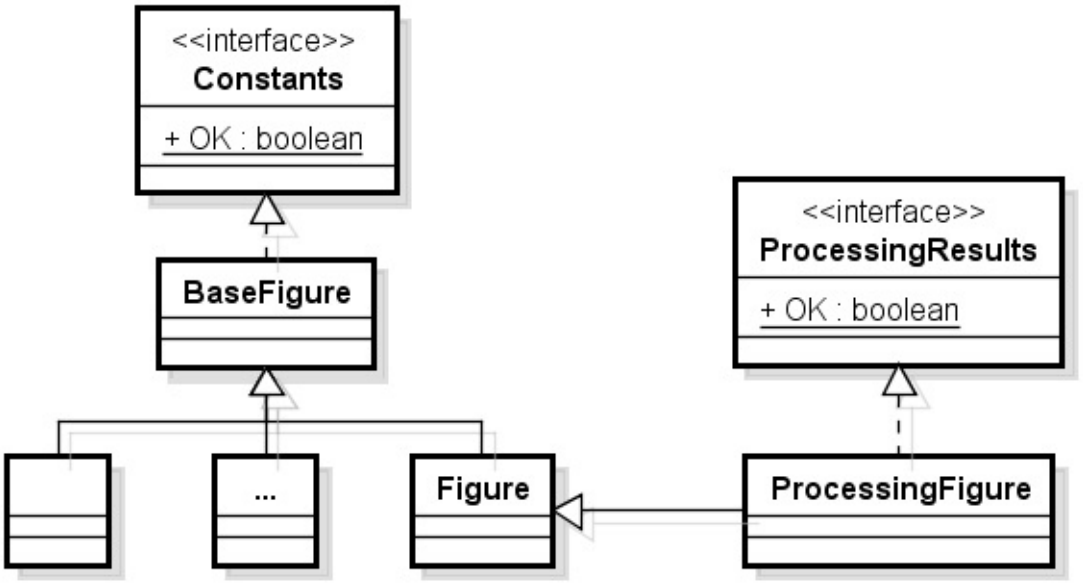


Abbildung 16-1 *Doppeldeutigkeit der Konstante* `OK`

Tipps und Refactorings

Benutzen Sie Interfaces nur dazu, wozu sie aus OO-Sicht dienen sollen, nämlich, um eine Schnittstelle mit angebotenen Methoden zu definieren.

Reine Konstanten-Interfaces können in der Regel in finale Konstantensammlungsklassen mit privaten Konstruktoren oder `enum`-Aufzählungen umgewandelt werden. Wie man dabei vorgeht, beschreibt das Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM in Abschnitt 17.4.12 als Schritt-für-Schritt-Anleitung.

Wurden solche Interfaces allerdings bereits (versehentlich) implementiert, dann muss mühselig jedes Vorkommen einer Konstante mit einer Referenz auf die neue Konstantenklasse versehen werden. Dabei kann es zu Problemen kommen, wenn man nicht alle Klassen im Zugriff hat oder verändern kann, die das Interface implementiert haben. Daher ist es sinnvoll, alle bekannten Nutzer vor einer solchen Maßnahme über den Umbau zu informieren. Ist die Nutzerbasis groß (z. B. für das Java-API), so ist ein solches Vorgehen nicht möglich und eine solche Designsünde lässt sich nicht mehr korrigieren, sondern nur als veraltet markieren. Dazu nutzt man das Javadoc-Tag `@deprecated` oder die Annotation `@Deprecated`. ***An diesem Beispiel erkennt man, wie wichtig es ist, gleich von Anfang an mit großer Sorgfalt zu arbeiten***. Dies gilt insbesondere bei der Implementierung aller nach außen sichtbaren Klassen und Methoden.

Info: Ähnliche Probleme durch statischen Import (`import static`)

Statische Import ermöglichen es, Attribute oder Methoden anderer Klassen ohne Angabe des qualifizierenden Klassennamens zu nutzen. Es werden dadurch weitere Namen in den Namensraum der eigenen Klasse aufgenommen.

Beim Import statischer Attribute kommt es bei gleichnamigen, eigenen Klassenattributen zu Namenskonflikten und Kompilierproblemen, wie dies z. B. beim Implementieren verschiedener Interfaces mit gleichnamigen Konstanten der Fall ist.

Bei Mehrdeutigkeiten in Bezug auf Methoden treten keine Fehler beim Kompilieren auf, stattdessen überdeckt eine Objektmethode aus der eigenen Klasse eine statisch importierte Methode. Dadurch kommt es gegebenenfalls allerdings zu folgendem Problem: ***Wird bei einer Erweiterung eine Objektmethode eingeführt, so wird das Verhalten unerwartet verändert, da als Folge nicht mehr die statisch importierte Methode aufgerufen wird, sondern die neue eingeführte Methode der eigenen Klasse.*** Dies gilt übrigens auch dann, wenn die aufrufende Methode selbst statisch ist!

16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert

Müssen einige Werte als Konstanten definiert werden, wie etwa die Einfügestrategien für Kommandos aus dem Beispiel der Klasse `CommandExecutor`, sieht man häufiger Definitionen zusammengehörender Konstanten als `int`-Werte:

```
/** altes Kommando durch neues ersetzen */
public static final int REPLACE_OLD_OR_ADD_AS_LAST = 0;

/** Kommando als erstes Kommando neu erzeugen */
public static final int ADD_AS_FIRST               = 1;

/** Kommando als letztes Kommando neu erzeugen */
public static final int ADD_AS_LAST                = 2;
```

Warum ist das ein Bad Smell?

Werden Konstanten als `int` definiert, so sind als Werte grundsätzlich alle aus dem Wertebereich des Typs erlaubt. Dadurch können auch unsinnige Werte Anwendung finden, wie dies folgendes Beispiel zeigt:

```
public static void main(final String[] args)
{
    // Was mag die 4711 bedeuten?
    final CommandExecutor executor = new CommandExecutor(4711);
    // …
}
```

Wünschenswert ist es, sicherzustellen, dass die übergebenen Werte im Bereich der definierten Konstanten liegen. Das erfordert eine manuelle Bereichsprüfung und wird dadurch schnell unübersichtlich. Dieser Effekt verstärkt sich, wenn die Prüfungen an verschiedenen Stellen durchgeführt werden müssen, die Werte keinen zusammenhängenden Bereich abbilden oder sich Werte nachträglich ändern bzw.

neue Konstanten hinzukommen. Betrachten wir dies konkret für den Konstruktor der Klasse `CommandExecutor`:

```
public CommandExecutor(final int strategy)
{
    if (strategy < REPLACE_OLD_OR_ADD_AS_LAST || strategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'strategy' is invalid: "
+
                        "value='" + strategy + "' not in range [" +
                        REPLACE_OLD_OR_ADD_AS_LAST + " – " + ADD_AS_LAST + "]");
    }
    this.registrationStrategy = strategy;
}
```

Prinzipiell gut an der Parameterprüfung ist, dass sie überhaupt vorhanden ist und der Benutzer bei einem Fehler im Aufruf detailliert sowohl über die Wertebelegung als auch über gültige Parameterwerte informiert wird.

Allerdings stehen diesen Vorteilen einige Nachteile gegenüber. Die gesamte Wertebereichsprüfung und Fehlerbehandlung ist aus folgenden Gründen ziemlich fragil:

1. Es werden Annahmen über die konkreten Werte der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` und `ADD_AS_LAST` gemacht, insbesondere, dass der Wert von `ADD_AS_LAST` den größeren der beiden Werte repräsentiert. Wird der Wert der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` bzw. `ADD_AS_LAST` derart verändert, dass diese Annahme nicht mehr gilt, scheitert die Wertebereichsprüfung.
2. Der Vergleich setzt zudem voraus, dass durch die Konstanten ein zusammenhängender Wertebereich abgedeckt wird. Diese Forderung erschwert unter Umständen eine sinnvolle Vergabe der Parameterwerte.
3. Wird im Nachhinein eine zusätzliche Kommandoart hinzugefügt, erfordert dies nicht nur die Definition einer neuen Konstanten, sondern es müssen überall im verwendenden Sourcecode Anpassungen in den Wertebereichsprüfungen erfolgen.

Tipps und Refactorings

Tatsächlich möchte man Aufrufern lediglich erlauben, symbolische Werte, d. h. Konstantennamen, zu verwenden. Das lässt sich sehr elegant und einfach durch das Sprachmittel `enum` erreichen, mit dem man Konstanten zu einem neuen Typ zusammenfassen kann:

```
enum RegistrationStrategyType
{
    REMOVE_OLD_AND_ADD_AS_LAST(0, "altes Kommando durch neues ersetzen"),
    ADD_AS_FIRST(1, "Kommando als erstes Kommando neu erzeugen"),
    ADD_AS_LAST(2, "Kommando als letztes Kommando neu erzeugen");

    final int    value;
    final String description;
```

```
    RegistrationStrategyType(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }
}
```

Die durch Einsatz eines `enum` gewonnene Typsicherheit löst sowohl die Probleme der Bereichsprüfung – diese ist nun überflüssig, da sie implizit durch den Compiler basierend auf der Typangabe sichergestellt wird – als auch die der nicht zusammenhängenden Wertebereiche oder sich ändernder Konstantenwerte. Eine Schritt-für-Schritt-Anleitung liefert das Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM in Abschnitt 17.4.12.

16.1.4 Bad Smell: Programmcode im Logging-Code

Programmcode und Logging-Code sollte man immer funktional trennen. Dieser Bad Smell betrachtet, was passieren kann, wenn man dies nicht tut.

In folgendem Beispiel wird im Log-Level `debug` die Methode `resetLineCounter()` aufgerufen, für andere Log-Level jedoch nicht:

```
if (log.isDebugEnabled())
{
    log.debug("…");
    resetLineCounter();
}
```

Warum ist das ein Bad Smell?

Erfolgt das Logging, z. B. während der Entwicklung, zunächst immer im Debug-Level, scheint die Software fehlerfrei zu funktionieren. Wird im späteren Verlauf eines Projekts der Log-Level allerdings erhöht, so erzeugt man dadurch mit hoher Wahrscheinlichkeit einen Softwaredefekt. Das Rücksetzen der Variablen durch die Methode `resetLineCounter()` wird nun nicht mehr durchgeführt. Noch schlimmer ist es, wenn ein Kunde die Funktion bereits abgenommen hat und selbstständig später die Log-Konfiguration ändert.

Tipps und Refactorings

Ist die Ausführung von Programmcode abhängig vom Log-Level, so kann dies zu extrem schwer zu findenden Fehlern führen! Daher ist Programmcode innerhalb von Log-Ausgaben zu vermeiden. Der Aufruf von `get()`-Methoden ist meistens akzeptabel. Allerdings muss man selbst hierbei achtsam sein, denn man sieht leider immer wieder `get()`-Methoden, die Seiteneffekte verursachen.

Warnung: Ähnliche Probleme mit Assertions

Nach der Lektüre von Abschnitt 4.7.5 wissen wir, dass Assertions standardmäßig deaktiviert sind und Package-weise (de)aktiviert werden können. Daher darf man sich niemals darauf verlassen, dass Anweisungen innerhalb eines `asserts` abgearbeitet werden. Folgende Methode ist daher fragil und löscht je nach Einstellung Daten oder eben auch nicht:

```
protected boolean deleteValues(final String key)
{
    if (exists(key))
    {
        assert delete(key);
        return true;
    }
    return false;
}
```

16.1.5 Bad Smell: Dominanter Logging-Code

Wie zuvor erkannt, sollte man Programmcode und Logging-Code funktional voneinander trennen. Teilweise ist aber die eigentliche Programmfunktionalität kaum mehr zu erkennen, da der Sourcecode extrem mit Logging-Code durchsetzt ist.

Folgendes Beispiel stammt aus einem Fahrgastinformationssystem zur Bereitstellung von Busabfahrten. Das Listing zeigt die Erzeugung von Informationen zu neuen Abfahrten mithilfe der Methode `DepartureFactory.createNewDepartures()` und deren Verarbeitung durch `processNewDepatures(List<Departure>)`:

```
log.info("Start creating new departures …");
final List<Departure> newDepartures = DepartureFactory.createNewDepartures();
log.info("Finished creation of departures");

if (log.isDebugEnabled())
{
    log.debug("newly created depatures:");
    final Iterator<Departure> it = newDepartures.iterator();
    while (it.hasNext())
    {
        final Departure departure = it.next();
        log.debug("new Departure: " + "time = '" + departure.getTime() + "'"
                  + "line = '" + departure.getLine() + "'"
                  + "destination = '" + departure.getDestination());
    }
}

log.info("process new departures");
processNewDepartures(newDepartures);
```

Warum ist das ein Bad Smell?

Obwohl beide Aufgaben schon in eigene Methoden ausgelagert sind, erschließt sich in diesem Ausschnitt deren Zusammenhang und auch deren Aufruf nicht sofort. Das

liegt insbesondere daran, dass der Logging-Code den Applikationscode dominiert und stark durchsetzt. Tatsächlich finden sich zwei Zeilen Nutzcode und über 10 Zeilen Logging-Code, in dem zum einen einfache Abläufe protokolliert werden und zum anderen eine längere (und bereits leicht unübersichtliche) Aufbereitung von Log-Ausgaben erfolgt.

Verschärft wird das Ganze dadurch, das Sprünge zwischen verschiedenen Abstraktionsebenen vorliegen, wodurch wir beim Betrachten des Sourcecodes immer wieder gedanklich zwischen verschiedenen Abstraktionsgraden hin und her wechseln müssen. Darauf gehe ich in Abschnitt 19.2 ein, wo wir etwas zur Wahrnehmungspsychologie erfahren und wie sich das Layout des Sourcecodes auf dessen Verständlichkeit auswirkt.

Tipps und Refactorings

Wenn man den Logging-Code in eine Hilfsmethode `logAllDepartureInfos(List<Departure>)` auslagert, so wird der Sourcecode-Abschnitt bereits um einiges klarer. Dadurch kann man vermutlich auch auf die protokollierenden Informationsmeldungen verzichten – oder diese in die beiden Nutzcodemethoden verlagern. Als Letztes kann man noch Logging und Informationsverarbeitung vertauschen. Damit wird das Verarbeiten und das Ermitteln der Informationen noch stärker verknüpft und der Sourcecode kommuniziert ohne den feingranularen Logging-Code sehr gut, was er tut:

```
final List<Departure> newDepartures = DepartureFactory.createNewDepartures();
processNewDepartures(newDepartures);

logAllDepartureInfos(newDepartures);
```

16.1.6 Bad Smell: Unvollständige Betrachtung aller Alternativen

Teilweise sieht man Sourcecode-Abschnitte, die verschiedene Alternativen über `if` oder `case` unterscheiden, jedoch einige mögliche Fälle nicht betrachten.

Dieses Beispiel stammt aus einer Applikation, die aus mehreren Teilkomponenten besteht. Jede dieser Komponenten verwaltet einen eigenen Systemzustand. Dieser kann abgefragt werden und dient als Eingabe für die Berechnung eines Gesamtzustands. Die Einzelzustände werden durch den Aufruf einer Methode `calcState(ComponentId)` als Aufzählungskonstanten ermittelt und in der Variablen `stateMap` gespeichert. Wird von `calcState(ComponentId)` unerwartet der Wert `null` zurückgeliefert, so findet allerdings keine Speicherung statt:

```
private Map<ComponentId, ComponentState> stateMap = new HashMap<>();

public void updateState(final ComponentId componentId)
```

```
{
    final ComponentState state = calcState(componentId);

    if (state != null)
    {
        stateMap.put(componentId, state);
    }
}
```

Warum ist das ein Bad Smell?

Betrachtet man obiges Listing, so ist zunächst kein Fehler zu erkennen. Bei einem fehlenden `else`-Zweig sollte man sich aber immer fragen, was in einem solchen Fall passiert bzw. ob ein solcher Fall eintreten kann.

Tatsächlich kam es bei diesem Sourcecode in ganz seltenen und ungünstigen Fehlersituationen dazu, dass die Methode `calcState(ComponentId)` nicht einen der definierten Aufzählungswerte, sondern einen `null`-Wert lieferte. Ursache war, dass ein Spezialfall übersehen worden war und dort fälschlicherweise eine Rückgabe von `null` erfolgte. Da dieser Wert weder protokolliert wurde noch eine Aktualisierung des Zustands stattfand, wurde folglich der zuvor in der Map gespeicherte Systemzustand beibehalten. In ungünstigen Fällen wurde irrtümlich also weiterhin von einem funktionierenden System ausgegangen, obwohl ein Fehler vorlag. Das Nachvollziehen dieser ganz speziellen, seltenen Fehlersituation war zeitaufwendig, da das gesamte System keinen Hinweis auf eine Fehlfunktion lieferte. ***Der Einsatz von Aufzählungen schützt zwar vor ungültigen Werten, jedoch müssen dann von Klienten gegebenenfalls Referenzen auf `null` geprüft werden.*** Ähnliche Probleme behandeln BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.6 sowie BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN in Abschnitt 16.3.8.

Tipps und Refactorings

Selbst wenn man der Meinung ist, einen Fall gänzlich ausschließen zu können, sollte man immer einen `else`-Zweig einführen. Dort kann für diese unerwartete Situation eine Warnmeldung in eine Log-Datei erfolgen oder bevorzugt eine `IllegalStateException` ausgelöst werden. Durch Letzteres hat man die Gewissheit, mögliche Programmier- oder Denkfehler aufzudecken. Ein großer Vorteil dieses offensiven Umgangs mit Fehlersituationen ist, dass dadurch häufig Fehler noch relativ früh im Entwicklungsprozess erkannt werden können. Selbst wenn ein solcher Fehler erst während der Wartungsphase auftritt, kann eine Exception auf veränderte Umgebungsbedingungen hindeuten und eine Fehlerkorrektur erleichtern. Folgende Modifikation hilft, mögliche Fehler bei der Ermittlung des Systemstatus aufzudecken. Insbesondere wird nun die Fehlersituation sofort deutlich und der Normalfall erfordert keine Sonderbehandlung mehr. Folgender Sourcecode ist also verständlicher:

```
public void updateStateImproved(final ComponentId componentId)
```

```
{
    final ComponentState state = calcState(componentId);

    if (state == null)
        throw new IllegalStateException("variable 'state' must not be null!");

    stateMap.put(componentId, state);
}
```

16.1.7 Bad Smell: Unvollständige Änderungen nach Copy-Paste

Bei Erweiterungen wird teilweise Funktionalität ähnlich zu bereits implementierten Programmteilen benötigt. Aus Bequemlichkeit wird dann mitunter Copy-Paste genutzt, um die erforderlichen Teile zu übernehmen und nur an einigen Stellen zu modifizieren.

Betrachten wir dies anhand der Methode `createWatchdog(byte[])`:

```
public Watchdog createWatchdog(final byte[] data) throws
      UnknownProtocolException
{
    final Encoding encoding = EncodingFactory.getEncodingFor(Watchdog.class);
    if (encoding == null)
        throw new UnknownProtocolException(Watchdog.class);

    return new Watchdog(data, encoding);
}
```

Diese Methode dient als Ausgangsbasis und Kopiervorlage, um eine ziemlich ähnliche Methode `createContent(byte[])` neu zu erzeugen. Die Kopie sieht nach ein paar Modifikationen wie folgt aus:

```
public Content createContent(final byte[] data) throws
UnknownProtocolException
{
    final Encoding encoding = EncodingFactory.getEncodingFor(Content.class);
    if (encoding == null)
        throw new UnknownProtocolException(Watchdog.class);
    //                                      ********
    return new Content(data, encoding);
}
```

In beiden Fällen wird zunächst ein passendes Encoding durch eine `EncodingFactory` bereitgestellt. Ist kein solches registriert, wird von der Methode `getEncodingFor(Class<T>)` der Rückgabewert `null` geliefert. Die Applikation wirft daraufhin eine `UnknownProtocolException`. Existiert ein passendes Encoding, so wird das gewünschte Objekt aus den übergebenen Daten und dem Encoding erzeugt.

Es fällt auf, dass in den kopierten Zeilen der Fehlerbehandlung beim Auslösen der `UnknownProtocolException` die Anpassung der Typangabe von `Watchdog.class` auf `Content.class` vergessen wurde.

Warum ist das ein Bad Smell?

Der Copy-Paste-Ansatz bietet scheinbar einen schnellen Weg, gewünschte Funktionalität übernehmen und modifizieren zu können. Als Folge steigt jedoch der Sourcecode-Umfang, und es entstehen diverse ähnliche Blöcke mit lediglich einigen (kleineren) Veränderungen. Solange die funktionale Korrektheit des Programms gegeben ist, scheint die Duplikation zunächst nur ein kosmetisches Problem zu sein. Allerdings wird der Überblick erschwert. Dadurch ist solcher Sourcecode schwieriger erweiter- und wartbar. Zudem schleichen sich bei Änderungen sehr leicht Fehler ein. Durch die Duplikation müssen Änderungen an einer Stelle des ursprünglichen Sourcecodes möglicherweise an allen bereits kopierten Stellen nachgezogen werden. Im hektischen Alltag wird schnell eine davon übersehen, wodurch jedoch unvollständige Anpassungen entstehen, die Inkonsistenzen auslösen. Diese verringern wiederum die Verständlichkeit. Es beginnt ein Wartungsalbtraum.

In diesem Beispiel wird im Fehlerfall zwar lediglich eine irreführende Fehlermeldung generiert. Eine solche Fehlinformation erschwert jedoch eine spätere Fehlersuche. Mit etwas Fantasie kann man sich ausmalen, welche Folgen und Verarbeitungsfehler an kritischeren Stellen drohen, wenn nicht nur eine Warnmeldung aus einer Exception erzeugt wird.

Tipps und Refactorings

Beim Verwenden von Copy-Paste sollte man immer sehr große Sorgfalt walten lassen und sich den kopierten Sourcecode und alle nachfolgenden Anpassungen genau anschauen. Häufig lässt sich Copy-Paste vermeiden, indem man Convenience-Methoden einführt, den betroffenen Sourcecode-Ausschnitt dorthin verlagert und die neue Methode an der Original- und Zielstelle aufruft, anstatt die betreffenden Stellen zu kopieren. In diesem Beispiel würde es sich anbieten, eine Methode zur Prüfung auf ein gültiges Encoding in die Klasse `EncodingFactory` aufzunehmen und im Fehlerfall hier eine `UnknownProtocolException` zu werfen, statt `null` zurückzugeben. Derartige Problemfälle behandelt BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL in Abschnitt 16.3.5.

Soll bestehender Sourcecode auf Duplikation oder unerwartete Abhängigkeiten geprüft werden, so können letztere manchmal bereits durch einen Blick auf die `import`-Anweisungen entdeckt werden. Deutlich genauere Ergebnisse liefern die Detektoren für Copy-Paste-Duplikation, die in einige Tools integriert sind (vgl. Abschnitt 19.4).

16.1.8 Bad Smell: Casts auf unbekannte Subtypen

Casts werden in Programmen verwendet, wenn eine Typumwandlung gewünscht wird, die vom Compiler nicht garantiert werden kann. In wenigen Fällen ist ein Einsatz unumgänglich, beispielsweise wenn man ein Zahlenliteral vom Typ `int` an

eine Methode mit `short`-Parametern übergeben möchte. In der Regel weist der Einsatz eines Casts allerdings auf eine potenzielle Fehlerquelle hin, und man sollte einen genaueren Blick auf die entsprechende Sourcecode-Stelle werfen.

Betrachten wir die Problematik am Beispiel einer Methode `getPersons()`, die in einem Interface `IDataAccess` wie folgt definiert ist:

```
interface IDataAccess
{
    List<Person> getPersons();
}
```

Der folgende Sourcecode-Ausschnitt nutzt eine Realisierung dieses Interface zum Aufruf von `getPersons()`. Vor der Zuweisung an die Variable `persons` findet ein Cast auf eine `LinkedList<Person>` statt:

```
final LinkedList<Person> persons = (LinkedList<Person>)
dbAccessor.getPersons();
final Iterator<Person> it = persons.iterator();
while (it.hasNext())
{
    processValue(it.next());
}
```

Warum ist das ein Bad Smell?

In diesem Beispiel erfolgt eine Umwandlung der zurückgelieferten Referenz auf den Typ `LinkedList<Person>`, obwohl im Interface `IDataAccess` lediglich die Rückgabe einer `List<Person>` garantiert wird. Sofern die momentane Implementierung des Interface `IDataAccess` für `getPersons()` eine `LinkedList<Person>` verwendet und zurückliefert, kommt es zu keinem Laufzeitfehler. Der aufrufende Sourcecode verlässt sich hier allerdings auf ein Implementierungsdetail. Würde die Implementierung, etwa aufgrund von Performance-Messungen, auf den Einsatz einer `ArrayList<Person>` umgestellt, so wäre dies immer noch Interface-konform. Es käme jedoch im Beispiel in der aufrufenden Methode zu einer `ClassCastException`.

Casts sind schon an sich kein schöner Programmierstil, da sie Typumwandlungen vornehmen, deren Fehlerfreiheit vom Compiler nicht garantiert werden kann. Problematisch werden sie jedoch, wenn ein sogenannter ***Down Cast*** von einem Basistyp auf einen Subtyp erfolgt. Dies entspricht in der Ableitungshierarchie dem »Weg nach unten«, der allerdings nicht zugesichert ist und häufig zu `ClassCastExceptions` führt, beispielsweise wenn man folgende Methode `downcastToSub(Object)` mit einem Objekt vom Typ `Person` aufruft:

```
public Sub downcastToSub(final Object obj)
{
    final Sub sub = (Sub) obj;          // Down Cast: Object -> Sub
    // …
```

Erinnern wir uns an die OO-Grundlagen: Es gilt die Beziehung Subtyp »is-a« Basistyp, aber nicht umgekehrt. Nur für den Fall, dass an die obige Methode ein

Objekt mindestens vom Typ `Sub` übergeben wird, ist der Cast erfolgreich. Ansonsten kommt es zu einer Inkompatibilität von Typen, wodurch eine `ClassCastException` ausgelöst wird. Einen solchen Cast sollte man daher nur nach expliziter Typprüfung durchführen:

```
public Sub downcastToSub(final Object obj)
{
    if (obj instanceof Sub)
    {
        // Cast ist nun sicher
        final Sub sub = (Sub) obj;      // Down Cast: Object -> Sub
        // …
    }
    else
    {
        // sollte nicht auftreten, z. B. IllegalStateException werfen
    }
    // …
```

Der Cast ist nun zwar sicher, aber es stellt sich die Frage, wie auf nicht erwartete Typen reagiert werden soll. Eine ähnliche Situation wird als BAD SMELL: UNVOLLSTÄNDIGE BETRACHTUNG ALLER ALTERNATIVEN in Abschnitt 16.1.6 behandelt.

Tipps und Refactorings

Anhand des eingangs gezeigten Beispiels sieht man einerseits, dass der Cast in diesem Fall vollkommen überflüssig ist, und andererseits, dass es durch Programmieren gegen das Interface `List<T>` gar nicht erst zu dem Problem gekommen wäre. Als Merksatz gilt: ***Vermeide Casts auf konkrete Klassen und programmiere stattdessen gegen Interfaces***. Sofern kein gemeinsames Interface vorhanden ist, sollte man ein solches einführen. Casts kann man vermeiden, wenn man Referenzen auf das gemeinsame Interface zum polymorphen Aufruf von Methoden nutzt. Dieses Vorgehen entspricht eher der objektorientierten Denkweise.

16.1.9 Bad Smell: Pre-/Post-Increment in komplexeren Statements

Die Verwendung von Pre-/Post-Increments bzw. -Decrements erlaubt häufig eine kompaktere Schreibweise des Sourcecodes. Nutzen wir diese jedoch in Kombination mit Abfragen oder in Array-Zugriffen, so kann dies schnell unübersichtlich werden.

Betrachten wir diesen Bad Smell konkret am Beispiel einer Variablen `pos`, die innerhalb einer Schleife in einem Array-Zugriff inkrementiert wird. Das kann mit Pre-Increment folgendermaßen realisiert werden:

```
while (pos < message.length)
{
    if (message[++pos] != EOL_BYTE_VALUE)
    {
```

Mit Post-Increment schreibt man Folgendes:

```
while ( pos < message.length)
{
    if ( message[ pos++] != EOL_BYTE_VALUE)
    {
```

Warum ist das ein Bad Smell?

Beides sieht auf den ersten Blick sehr ähnlich aus. Es gibt jedoch einen subtilen Unterschied: Im Fall des Post-Increments wird zunächst der Wert aus dem `message`-Array ausgelesen und *anschließend* der Wert von `pos` erhöht. Im Fall des Pre-Increments wird der Wert von `pos` jedoch *vor* dem Auslesen erhöht. Dadurch greift man auf einen anderen Wert im Array zu und vergleicht andere Werte als beim Post-Increment. Der entscheidende Punkt ist, dass die unterschiedliche Bedeutung nicht sofort ins Auge springt. Das Ganze hat folgende Konsequenzen:

1. **Die Position des Array-Zugriffs ist nicht intuitiv ersichtlich:** Möglicherweise werden falsche (unbeabsichtigte) Werte miteinander verglichen, sodass der Vergleich ein unerwartetes Ergebnis liefert. Zusätzlich wird häufig, wie in diesem Fall, der Index durch das Pre-Increment am Ende der Schleifendurchläufe ungültig: Für den Fall, dass `pos` den Wert `message.length-1` erreicht, kommt es beim Aufruf des Zugriffs `message[ ++pos]` zu einer `ArrayIndexOutOfBoundsException`. Selbst wenn eine solche Exception nicht auftritt, gilt das nächste Argument.
2. **Es wird Vergleichslogik mit Schleifenlogik vermischt:** Wird eine Schleifenvariable mitten in der Schleife geändert, wird es schwieriger, deren Veränderungen nachzuvollziehen. Dies gilt vor allem, wenn an verschiedenen Stellen in der Schleife nochmals auf die Schleifenvariable zugegriffen wird. Noch komplizierter wird es, wenn mehrere solcher Anweisungen hintereinander, geschachtelt oder sogar bedingt ausgeführt werden, wie dies im folgenden Beispiel angedeutet ist:

```
while ( pos < message.length)
{
    // erhöht in jedem Fall
    if ( message[ pos++] == START_TAG_VALUE)
    {
        // erhöht nur, wenn START_TAG_VALUE gefunden wurde
        if ( message[ pos++] != EOL_BYTE_VALUE)
        {
```

Welchen Wert die Schleifenvariable dann in jedem Schritt hat, ist mühseliger nachzuvollziehen als bei einer Zuweisung und Veränderung in separaten Schritten. Zudem vergrößert sich die Gefahr nicht abgesicherter Array-Zugriffe und daraus resultierender `ArrayIndexOutOfBoundsExceptions`.

Tipps und Refactorings

Man sollte Vergleichs- und Zuweisungslogik möglichst immer von Variablenänderungen trennen. Das Durchlaufen einer Schleife und das Verändern der Schleifenvariablen sollte in der Schleifenbedingung oder aber als separate Zeile am Ende der Schleife erfolgen.

Hinweis: Extrembeispiel

Betrachten wir folgendes Extrembeispiel, um die Idee dieses Bad Smells zu verdeutlichen. Es basiert auf dem Puzzle »Tricky Assignment« aus dem Buch »Java Puzzlers« von Joshua Bloch und Neil Gafter [10].

```
int tricky = 0;
for (int i=0; i < 10; i++)
    tricky += tricky++;
```

Raten Sie mal, welchen Wert die Variable `tricky` als Ergebnis dieser Schleife hat: 0, 10, 45 oder einen ganz anderen Wert? Die richtige Antwort ist: 0. Merkwürdig! Wie kommt das? Schreiben wir die Zeile einmal so, dass wir einige Zwischenschritte erkennen:

```
// tricky += tricky++;
=> tricky = tricky + tricky++;
```

Auch diese Anweisung können wir noch weiter aufschlüsseln und erkennen dann die Ursache: Die Variable `tricky` wird zwar inkrementiert, jedoch wird zuvor der alte Wert von `tricky` in einer Hilfsvariablen `temp` zwischengespeichert und anschließend wieder an `tricky` zugewiesen. Die Erhöhung geht dadurch verloren:

```
// tricky = tricky + tricky++;
1) temp = tricky + tricky;
2) tricky++;
3) tricky = temp;
```

16.1.10 Bad Smell: Keine Klammern um Blöcke

Im Rahmen dieses Bad Smells wird beschrieben, warum es sinnvoll ist, Blöcke normalerweise durch Klammern zu kennzeichnen, und welche Probleme auftreten können, wenn man dies nicht tut.

Das folgende Beispiel stammt aus einer Implementierung einer Tabelle, die sortierbare Spalten anbietet. Der Sourcecode sieht vereinfacht so aus:

```
if (COLUMN_PATH.equals(strColumnName))
    // do not sort path

if (COLUMN_CREATION_DATE.equals(strColumnName))
    Collections.sort(documents, new CreationDateComparator(ascending));

if (COLUMN_MODIFICATION_DATE.equals(strColumnName))
    Collections.sort(documents, new ModificationDateComparator(ascending));

…
```

Obwohl für alle Spalten (außer dem Pfad) passende `Comparator`-Objekte hinterlegt

sind, zeigen erste Tests, dass die Inhalte gewisser Spalten nicht sortiert werden können. Mit ein wenig Testaufwand findet man heraus, dass dieser Fehler tatsächlich nur den Inhalt der Spalte `COLUMN_CREATION_DATE` betrifft.

Wie ist das zu erklären? Für alle Leser, die bereits sensibilisiert sind, ist der Fall sofort klar. Für alle anderen vereinfache ich den Sourcecode. Wenn die Variable `strColumnName` den Wert `COLUMN_CREATION_DATE` besitzt, steht dort:

```
if (false)
    // do not sort path

if (true)
    System.out.println("sort");
```

Führt man den Sourcecode aus, so gibt er nichts aus! Das liegt daran, dass ein einzeiliger Kommentar laut JLS keine Anweisung ist. Aufgrund dessen wird der Sourcecode folgendermaßen ausgewertet:

```
if (false)
{
    // do not sort path

    if (true)
        System.out.println("sort");
}
```

Derart formatiert und mit eingefügten Hilfsklammern ist klar, dass nichts ausgegeben wird.

Warum ist das ein Bad Smell?

Fehlende Klammern um Blöcke erschweren es, den Programmablauf basierend auf dem Sourcecode-Layout nachzuvollziehen. Die Auswertung von Bedingungen geschieht dann zum Teil nicht so, wie man es aufgrund der Formatierung erwarten würde. Im Besonderen gilt dies, wenn mehrere `if`-Anweisungen aufeinanderfolgen.

Tipps und Refactorings

Man sollte bevorzugt Klammern um Blöcke verwenden, um solche Probleme im Voraus zu verhindern. Ausgenommen davon sind einzeilige Shortcut-Returns oder `throw`-Anweisungen in Zustandsprüfungen am Methodenanfang. Werden diese jedoch um weitere Anweisungen oder ein `else` erweitert oder möchte man sich bei Erweiterungen der Anweisungen im `if`-Block niemals Gedanken machen müssen, ob man etwas am Ablaufverhalten ändert, empfiehlt es sich, einfach immer Klammern zu verwenden.

Achtung: Probleme durch '; ' an unerwarteten Stellen

Die irrtümliche Angabe eines '; ' an falscher Position kann ähnliche Probleme wie fehlende Klammern auslösen. Glücklicherweise kann eine IDE dies automatisch prüfen und eine Warnmeldung ausgeben.

```
public static void wrongLoop()
{
    int i = 0;
    for (; i < 10; i++);
    {
        System.out.println("i= " + i);
    }
}
```

Die hier vorgestellte Schleife wird nicht so durchlaufen, wie man es zunächst intuitiv erwartet. Hier wird lediglich `i=10` ausgegeben! Betrachtet man den Sourcecode genau, dann wird klar, dass jeweils zehn Mal die leere Anweisung ausgeführt wird: Es wird lediglich der Schleifenrumpf bestehend aus der Leeranweisung ';' wiederholt. Abschließend erfolgt die einmalige Ausführung des eigentlich gewünschten Schleifenrumpfs. Abhilfe kann man leicht dadurch schaffen, indem die Schleifenvariable innerhalb der `for`-Anweisung definiert wird:

```
for (int i= 0; i < 10; i++);
```

Die Variablendefinition ist dann für den folgenden Block mit der Ausgabe der Variablen `i` über `System.out.println()` nicht mehr sichtbar. Es kommt zu einem Kompilierfehler. Ein solcher Flüchtigkeitsfehler wird somit schneller ersichtlich. ***Eine Variable sollte also nach Möglichkeit immer im engsten Sichtbarkeitsbereich definiert werden***. Im Detail bespricht dies BAD SMELL: VARIABLENDEKLARATION NICHT IM KLEINSTMÖGLICHEN SICHTBARKEITSBEREICH in Abschnitt 16.1.14.

16.1.11 Bad Smell: Mehrere aufeinanderfolgende Parameter gleichen Typs

Werden Methoden überladen, so unterscheiden sich diese lediglich in ihrer Parameterliste. Unübersichtlich wird dies, wenn in den Signaturen mehrere Übergabeparameter des gleichen Typs aufeinanderfolgen.

Betrachten wir folgende Signaturen einer Methode `clearRect()`:

```
public void clearRect(int x, int y, int width, int height)
public void clearRect(int x1, int y1, int x2, int y2, long colorRGB)
public void clearRect(int red, int green, int blue,
                      int x, int y, int width, int height)
```

Warum ist das ein Bad Smell?

Bei mehreren aufeinanderfolgenden Parametern gleichen Typs in der Parameterliste besteht die Gefahr der Verwechselung von Positionen und Bedeutungen. Bereits die Anzahl der Parameter macht das Ganze unübersichtlich und erhöht die Wahrscheinlichkeit für Fehler. Je mehr gleichartige Parameter verwendet werden, desto stärker macht sich das Problem bemerkbar.

Im obigen Beispiel variiert bei der zweiten Variante der überladenen Methode die

Bedeutung der Parameter drei und vier gegenüber der ersten Methode. Dies ist problematisch, wenn man beispielsweise beim Löschen eine Farbe angeben möchte und intuitiv den ursprünglichen Aufruf lediglich um einen Parameter zur Farbinformation ergänzt. Man erhält so ein ganz anderes Applikationsverhalten als beim ursprünglichen Aufruf. Dies ist dadurch bedingt, dass die zweite Methode als Parameter drei und vier ein zweites Koordinatenpaar statt einer Angabe von Breite und Höhe erwartet.

Für Referenzparameter führt das Verwechseln von Positionen nicht nur zu geänderten Eingabewerten, sondern eventuell auch zu Exceptions: Wird an unerwarteter Stelle ein `null`-Wert übergeben, kommt es als Folge potenziell zu einer `NullPointerException` statt zu einem unterschiedlichen Eingabewert.

In diesem Beispiel ist eine weitere Inkonsequenz vorhanden: Die Methode mit fünf Parametern erwartet die Farbinformation in einem anderen Format als die Methode mit sieben Parametern, die einzelne RGB-Werte verwendet. Außerdem stimmen die Positionen nicht überein.

Tipps und Refactorings

Um die zuvor diskutierten Probleme zu lösen, ist es zum einen sinnvoll, ***die Reihenfolge von Parametern immer so einheitlich wie möglich zu halten und einer programmweiten Systematik folgen zu lassen***. Zum anderen erhält man durch das Zusammenfassen von Parametern zu Gruppen – sogenannten Parameter Value Objects (vgl. Abschnitt 3.4.5) – verständlichere Signaturen, die eine Verwechselung von Positionen unwahrscheinlich machen. Folgendes Listing zeigt eine mögliche Realisierung mit den Parameter Value Objects `Point`, `Dimension` und `Color`:

```
public void clearRect( Point point, Dimension size)
public void clearRect( Point point1, Point point2, long colorRGB)
public void clearRect( Color clearColor, Point point, Dimension size)
```

16.1.12 Bad Smell: Grundloser Einsatz von Reflection

Mithilfe von Reflection können Metainformationen über Klassen ermittelt werden. Man kann dadurch Sourcecode entwickeln, der zur Laufzeit auf Klassen zugreifen kann, die bei der Kompilierung der eigenen Klassen noch nicht zur Verfügung stehen und deren Name aber textuell vorliegt, z. B. durch Angabe in einer Konfigurationsdatei. Statt einer Referenzierung über `import` erfolgt der Zugriff über eine textuelle Repräsentation des vollqualifizierten Klassennamens.

Schauen wir uns ein einfaches Beispiel an, das per Reflection mit `newInstance()` ein neues Objekt vom Typ `AnyClass` erzeugt und dieses einer Referenzvariablen vom selben Typ zuweist. Diese Zuweisung an einen konkreten Typ erfordert aber, dass diese Klasse bereits zur Kompilierzeit bekannt ist. Das setzt

wiederum eine `import`-Anweisung voraus. Reflection wird anschließend nicht mehr verwendet, sondern es erfolgt ein »normaler« Aufruf der Methode `output(String)`:

```
import util.AnyClass;

// …

try
{
    final Class<?> c = Class.forName("util.AnyClass");
    final AnyClass newObject = (AnyClass) c.newInstance();
    newObject.output("1,2,3");        // Direkter, normaler Methodenaufruf
}
catch (final ClassNotFoundException e)
{
    handleReflectionException(e);
}
catch (final InstantiationException e)
{
    handleReflectionException(e);
}
catch (final IllegalAccessException e)
{
    handleReflectionException(e);
}
```

Warum ist das ein Bad Smell?

Die Verwendung von Reflection macht ein Programm in der Regel schlechter lesbar und dadurch schwieriger wartbar. Außerdem sind diverse Exceptions abzufangen. Insgesamt handelt es sich dabei um ein fragiles Gebilde, wenn Änderungen an Klassen, der Package-Struktur oder an Methodensignaturen zu erwarten sind. Der Einsatz von Reflection erschwert damit alle Basis-Refactorings (u. a. MOVE TO PACKAGE, RENAME), die detailliert im Buch »Refactoring« von Martin Fowler [27] beschrieben sind.

Der Einsatz von Reflection sollte daher immer gut begründet sein. Für die obige Konstruktion fällt es allerdings schwer, eine Begründung zu finden, da es sich bei der angesprochenen Klasse offenbar um eine dem Compiler bekannte Klasse handelt, die problemlos wie folgt hätte erzeugt und genutzt werden können:

```
final AnyClass newObject = new AnyClass();
newObject.output("1,2,3");
```

Tipps und Refactorings

Reflection birgt einige potenzielle Fehlerquellen und sollte daher nur eingesetzt werden, wenn kein konventioneller Zugriff möglich ist, etwa bei optionalen Komponenten. Hat man jedoch zur Kompilierzeit Zugriff, sollte man auf Reflection verzichten.

16.1.13 Bad Smell: `System.exit()` mitten im Programm

Ein Programmabbruch durch einen Aufruf von `System.exit(int)` mitten im Programmablauf birgt einige Gefahren und sollte daher immer genau hinterfragt werden.

Exemplarisch betrachten wir dies an der Methode `startConnection(int)`, die einen Verbindungsaufbau abhängig von einem durch den `int`-Parameter `transmissionMode` gewählten Übertragungsmodus startet. Wird ein unbekannter Modus angegeben, so landet man im `default`-Zweig der `case`-Anweisung, wo das Programmende mit `System.exit(int)` erzwungen wird:

```
private void startConnection(final int transmissionmode)
{
    switch (transmissionmode)
    {
        case TRANSMISSION_MODE_RADIO:
            startRadioConnection();
            break;
        case TRANSMISSION_MODE_NETWORK:
            startNetworkConnection();
            break;
        default:
            // FALSCHER ÜBERTRAGUNGSMODUS
            log.fatal("Unknown transmission mode: " + transmissionmode);
            System.exit(-1);
    }
}
```

Vor dem Programmende wird eine Information über den falsch gewählten Übertragungsmodus in die Log-Ausgabe geschrieben. Dadurch hat man zumindest im Nachhinein die Chance, das Problem einzugrenzen.

Warum ist das ein Bad Smell?

Der Ausstieg mit `System.exit(int)` ist problematisch, da ein Programm dadurch sofort terminiert wird, ohne Aufräumarbeiten durchführen zu können, etwa offene Dateien, Sockets, Datenbankverbindungen oder sonstige Ressourcen zu schließen bzw. freizugeben.[2] Im Extremfall stehen dann bei einem möglichen Neustart eines anderen Programms bzw. einer anderen JVM die zuvor belegten Ressourcen nicht oder noch nicht wieder zur Verfügung und verhindern so die Programmausführung. Das gilt insbesondere für Programme, die eigentlich immer laufen sollten (24/7) und die bei Programmabstürzen oder beim Programmende automatisch (z. B. durch Skripte) neu gestartet werden. Wenn dann ein Netzwerkfehler für andauernde Neustarts sorgt, sind die zuvor beschriebenen Probleme leicht nachvollziehbar.

Tipps und Refactorings

Zunächst ist es fehlerträchtig, dass Aufrufer durch die unbedachte Wahl des Parametertyps `int` überhaupt in der Lage sind, einen falschen Wert an diese Methode zu übergeben. BAD SMELL: ZUSAMMENGEHÖRENDE KONSTANTEN NICHT ALS TYP DEFINIERT diskutiert das Ganze in Abschnitt 16.1.3 genauer. Der Einsatz eines Aufzählungstyps stellt eine Lösung dar. Eine Schritt-für-Schritt-Anleitung liefert das in Abschnitt 17.4.12 beschriebene Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM. Neben dieser offensichtlichen Korrektur sind zwei weitere Verbesserungen denkbar:

1. Das Einführen eines booleschen Rückgabewerts, der über den (Miss-)Erfolg beim Verbindungsaufbau informiert.
2. Das Auslösen einer Exception, um eine geordnete Fehlerbehandlung im aufrufenden Programmteil zu ermöglichen. Hier könnte man eine `IllegalArgumentException` nutzen oder eine selbstdefinierte `UnsupportedTransmissionModeException`.

Tipp: Aufräumarbeiten und Shut-down-Hooks

Über einen sogenannten ***Shut-down-Hook*** können belegte Systemressourcen sauber wieder freigegeben werden – dies gilt selbst dann, wenn ein Aufruf von `System.exit(int)` mitten im Programmablauf erfolgt. Ein Shut-down-Hook ist eine spezielle Implementierung eines Threads. Beim Aufruf von `System.exit(int)` wird dieser Thread vor dem Ende der JVM ausgeführt. Gleiches gilt beim Beenden eines Programms durch eine nicht gefangene Exception. Wird die JVM jedoch anders beendet (etwa manuell über »Prozess beenden«), so ist nicht garantiert, dass ein Shut-down-Hook ausgeführt wird.

Idealerweise definiert man sich in denjenigen eigenen Klasse, die Ressourcen freigeben sollen, eine Methode `onShutdown()`, die die Aufräumarbeiten implementiert. Des Weiteren initialisiert man möglichst beim Programmstart oder kurz nach der Erzeugung eines Objekts einen Shut-down-Hook, sodass alle folgenden Aufrufe an `System.exit(int)` eine geordnete Terminierung erlauben.

Ein entsprechendes Programmfragment könnte so aussehen:

```
Runtime.getRuntime().addShutdownHook(new Thread()
{
    public void run()
    {
        onShutdown();
    }
});
```

16.1.14 Bad Smell: Variablendeklaration nicht im kleinstmöglichen Sichtbarkeitsbereich

Manchmal sieht man die Deklaration oder Definition diverser Variablen am Methodenanfang. In C++- bzw. C ist dies eine beliebte Technik, um alle verwendeten Variablen an zentraler Stelle sichtbar zu machen.

Ein Beispiel dafür ist folgende etwas umständlich arbeitende Methode `getImageForName(String)`, die zu einem übergebenen Namen einer Bilddatei ein passendes Bild vom Typ `Image` ermittelt und bei der Suche die Groß- und Kleinbuchstaben außer Acht lässt. Dazu wird eine Liste `imageNames` nach einem passenden Namen durchsucht und gegebenenfalls auf eine zweite Liste `images` mit Bilddaten zugegriffen. Dort sind die Daten jedoch um einen Wert `DEFAULT_IMAGE_OFFSET` versetzt gespeichert. Hier werden vier Variablen am Methodenanfang definiert:

```
private static Image getImageForName(final String imageNameToSearch)
{
    boolean found = false;
    String imageName = null;
    Image image = null;
    int index = 0;

    for (int i = 0; i < imageNames.size() && found == false; i++)
    {
        imageName = imageNames.get(i);
        if (imageName.equalsIgnoreCase(imageNameToSearch))
        {
            found = true;
            index = i + DEFAULT_IMAGE_OFFSET;
            image = images.get(index);
        }
    }

    return (found ? image : NO_IMAGE);
}
```

Alle vier Variablen werden erst im Schleifenrumpf mit aussagekräftigen Werten belegt, wobei drei Variablen sogar nur dann besetzt werden, wenn der gesuchte Name in der übergebenen Liste `imageNames` vorhanden ist.

Warum ist das ein Bad Smell?

Die Definition von Variablen zu Methodenbeginn ist nicht mehr zeitgemäß. Die Begründung ist relativ einfach: Eine solche Initialisierung macht nicht klar, wo eine Variable verwendet wird, und zudem vergrößert sich die Gefahr, dass eine Variable für mehrere, unabhängige Zwecke eingesetzt wird. Außerdem ist meistens noch keine sinnvolle Vorbelegung möglich. Des Weiteren kann eine solche Variable nicht als `final` deklariert werden, selbst wenn sie es bei der Deklaration am passenden Ort sein könnte. Ein ganz entscheidender Nachteil jedoch ist, dass diese Art der Variablendeklarationen Refactorings erschwert, die gewisse Funktionalität aus einem Block in eine separate Methode verlagern.

Tipps und Refactorings

Variablen sollten so lokal wie möglich deklariert werden, also in dem kleinstmöglichen Sichtbarkeitsbereich. Dies hilft, schnell zu erkennen, welche Variablen an welcher Stelle im Sourcecode eingesetzt oder verändert werden. Wir können anschließend sämtliche Variablen, die sich nicht ändern, `final` definieren. Dadurch verhindert man den Missbrauch oder ein versehentliches Benutzen der Variablen für andere Zwecke. Durch die verringerten Abhängigkeiten werden Refactorings und das Herauslösen von Methoden erleichtert.

Nehmen wir an dem zuvor gezeigten Beispiel einige Änderungen vor: Wir verschieben die Variablendeklarationen in den kleinstmöglichen Sichtbarkeitsbereich und vereinfachen den Sourcecode weiter. Man erkennt beispielsweise, dass die Variable `index` immer den Wert `i + DEFAULT_IMAGE_OFFSET` besitzt. Auch die Variable `found` ist künstlich und steuert den Schleifenabbruch. In dieser kurzen Methode profitiert man vom Einsatz eines Shortcut-Returns, der die Variable `found` überflüssig macht. Durch den Shortcut-Return kann auch auf die Variable `image` verzichtet werden. Es bleibt lediglich die Variable `imageName` übrig. Sie wird direkt zu ihrer Verwendung verschoben, wodurch die Methode kürzer, übersichtlicher und lesbarer wird:

```
private static Image getImageForNameImproved(final String imageNameToSearch)
{
    for (int i = 0; i < imageNames.size(); i++)
    {
        final String imageName = imageNames.get(i);
        if (imageName.equalsIgnoreCase(imageNameToSearch))
        {
            return images.get(i + DEFAULT_IMAGE_OFFSET);
        }
    }

    return NO_IMAGE;
}
```

16.2 Klassendesign

16.2.1 Bad Smell: Unnötigerweise veränderliche Attribute

In der täglichen Arbeit kommt es immer wieder zu Änderungen in Klassen, um diese an neue Anforderungen anzupassen. Häufig werden dazu neue Attribute eingeführt. Zum Teil werden dann ohne weitere Bedenken korrespondierende `get()`- und `set()`-Methoden zum Zugriff angeboten.

Betrachten wir exemplarisch eine Klasse `Telegram`, bei der dies der Fall ist. Die Erzeugung eines `Telegram`-Objekts erfolgt per Defaultkonstruktor mit anschließender Initialisierung durch den Aufruf einiger `set()`-Methoden, die das Objekt dann vollständig »gebrauchsfertig« machen:

```
// => Konstruktion mit (eventuell unsinnigen) Defaultwerten
final Telegram telegram = new Telegram();
// => Initialisierung
telegram.setTransmissionMode(TransmissionMode.TCP);
telegram.setReceiver(hostAndPort);
telegram.setPayload(data);
// => hier erst korrektes Objekt !!!
```

Warum ist das ein Bad Smell?

Der Sourcecode sieht auf den ersten Blick nicht besonders fehleranfällig aus. Problematisch ist jedoch, dass zunächst durch den Konstruktoraufruf ein Objekt erzeugt wird, das diverse Attribute lediglich mit Defaultwerten versorgt. Wird nur der Konstruktor aufgerufen und erfolgen anschließend keine weiteren Initialisierungen, so verbleibt das Objekt in einem möglicherweise ungültigen Zustand (vgl. Abschnitt 3.1.5). Auf einer solchen Objektreferenz könnten bereits Methoden aufgerufen werden – ein korrektes Verhalten ist jedoch höchst unwahrscheinlich. Die anschließende mehrstufige Initialisierung über den Aufruf einzelner `set()`-Methoden ist zudem problematisch, weil dadurch weder sichergestellt werden kann, dass alle zur Objektkonstruktion benötigten Eingabewerte tatsächlich über entsprechende `set()`-Methoden gesetzt wurden, noch, dass eine gegebenenfalls erforderliche Reihenfolge der Initialisierung eingehalten wird.

Eine standardmäßige Definition von `set()`-Methoden verhindert, dass Attribute `final` deklariert werden können, obwohl diese eigentlich unveränderlich sind. Häufig sieht man zusätzlich bei der Deklaration eine Zuweisung eines unsinnigen, weil beliebigen Werts, wie dies folgendes Beispiel demonstriert:

```
public class Telegram
{
    private String           receiver         = "undefined";
    private TransmissionMode transmissionMode = TransmissionMode.UNDEFINED;
    private byte[]           payload          = null;

    public Telegram()
    {
    }

    public void setReceiver(final String receiver)
    {
        this.receiver = receiver;
    }

    public void setTransmissionModed(final TransmissionMode transmissionMode)
    {
        this.transmissionMode = transmissionMode;
    }

    public void setPayload(final byte[] payload)
    {
        this.payload = payload;
    }
}
```

Solche nichtssagenden (unsinnigen) Werte sind nur bis zum Aufruf einer

entsprechenden `set()`-Methode »gültig«. Erfolgt allerdings keine nachträgliche Initialisierung eines »pseudoinitialisierten« Attributs mit einem sinnvollen Wert, so liegt eine unvollständige Initialisierung vor, die jedoch nur schwer zu erkennen ist.

Tipps und Refactorings

Durch Codegeneratoren oder aus Unachtsamkeit werden manchmal öffentliche `set()`-Methoden für Attribute angeboten, die eigentlich unveränderlich sein könnten. Man sollte vermeiden, *standardmäßig* eine `set()`-Methode anzubieten, sondern diese nur bei tatsächlichem Bedarf einführen.

Durch die Abarbeitung von Konstruktoren sollte ein Objekt in einen gebrauchsfertigen Grundzustand versetzt werden. Konstruktoren müssen daher alle dafür notwendigen Eingabeparameter entgegennehmen. Wenn noch kein solcher Konstruktor angeboten wird, sollte man einen solchen einführen und die Klasse gemäß dem Refactoring MINIMIERE VERÄNDERLICHE ATTRIBUTE aus Abschnitt 17.4.2 anpassen. Dadurch werden falsch oder nur teilweise initialisierte Objekte und damit ungültige Objektzustände vermieden. Die `Telegram`-Klasse kann so umgewandelt werden, dass lediglich die Nutzlast (Attribut `payload`) variabel und über eine `setPayload(byte[])`-Methode änderbar ist:

```
public class Telegram
{
    private final String           receiver;
    private final TransmissionMode transmissionMode;
    private byte[]                 payload = null;

    public Telegram(final String receiver,
                    final TransmissionMode transmissionMode)
    {
        this.receiver = receiver;
        this.transmissionMode = transmissionMode;
    }

    public void setPayload(final byte[] payload)
    {
        this.payload = payload;
    }
}
```

16.2.2 Bad Smell: Herausgabe von `this` im Konstruktor

Eine Technik, die man häufiger sieht und die dabei harmloser wirkt, als sie in Wirklichkeit ist, ist die Übergabe der `this`-Referenz aus eigenen Konstruktoren an andere Objekte. Dies wird als »***Escaping Reference***« bezeichnet.

Im folgenden Beispiel betrachten wir den Konstruktionsprozess der Klasse `MainService`, die Referenzen auf die Klassen `CallBack` und `String` als Attribute

hält:

```
public class MainService
{
    private final CallBack caller;
    private final String   info;

    MainService()
    {
        caller = new CallBack(this);
        info = "Initialized " + MainService.class.getSimpleName();
    }

    public String getInfo()
    {
        return info;
    }
}
```

Im Konstruktor der Klasse `MainService` werden jeweils Objekte der Klassen `CallBack` und `String` erzeugt, wobei der Konstruktor von `CallBack` erstere wir folgt definiert ist:

```
CallBack(final MainService mainService)
{
    this.mainService = mainService;

    // Zugriff auf Methode der teilinitialisierten(!) Klasse MainService
    final byte[] infoAsBytes = mainService.getInfo().getBytes();
}
```

Warum ist das ein Bad Smell?

Bei der Herausgabe der `this`-Referenz verhält sich auf den ersten Blick korrekt erscheinender Sourcecode merkwürdig. Es treten Probleme durch falsch oder nicht initialisierte Attribute auf.

Konkretisieren wir dies an obigem Beispiel: Der Aufruf des Konstruktors der Klasse `CallBack` erhält als Eingabe die `this`-Referenz des gerade in der Erzeugung befindlichen `MainService`-Objekts. Zu diesem Zeitpunkt ist die Konstruktion der Klasse `MainService` allerdings noch nicht vollständig abgeschlossen, es steht noch die Initialisierung des Attributs `info` aus. ***Während der Konstruktion der Klasse `MainService` ist es der Klasse `CallBack` allerdings bereits möglich, auf alle öffentlichen Daten und Methoden der Klasse `MainService` zuzugreifen.*** Im Speziellen gilt dies auch für das Attribut `info`. Wenn die Klasse `CallBack` die Methode `getInfo()` der Klasse `MainService` aufruft, so kommt es durch die nicht initialisierte Referenz `info` zu einer `NullPointerException` beim Zugriff per `getBytes()`.

Tipps und Refactorings

Durch die Herausgabe der `this`-Referenz können andere Objekte ein unvollständig initialisiertes Objekt sehen und auf dessen Attribute zugreifen. ***»Escaping References«***

sind daher zu vermeiden. Es bietet sich der Einsatz des Musters ERZEUGUNGSMETHODE aus Abschnitt 18.1.1 an. Man kann dann ein Objekt zunächst ohne aggregierte Objekte erzeugen. Anschließend werden die Konstruktoren der aggregierten Objekte aufgerufen, die die Referenz benötigen. Die eigene Klasse muss allerdings um entsprechende `set()` - bzw. `init()` -Methoden erweitert werden, um die ansonsten im Konstruktor erzeugten Objekte zu initialisieren.

Achtung: »Escaping References« und »Immutable Objects«

»Immutable Objects« sollen zur ihrer Lebenszeit nur einen definierten Zustand besitzen, der zum Konstruktionszeitpunkt bestimmt wird. Reicht ein solches »Immutable Object« allerdings seine `this`-Referenz während seiner Erzeugung nach außen, kann es dadurch mehrere sichtbare Zustände einnehmen. »Escaping References« führen den Einsatz von »Immutable Objects« ad absurdum: Trotz der Realisierung als unveränderliche Klasse kann man Zwischenschritte sehen.

16.2.3 Bad Smell: Aufruf abstrakter Methoden im Konstruktor

Manchmal wünscht man sich, in Basisklassen einen gewissen Ablauf bei der Initialisierung vorgeben zu können, etwa den Aufruf einer speziellen `init()`-Methode aus dem Konstruktor. Die konkrete Realisierung der Initialisierung ist in der Basisklasse jedoch noch unbekannt und soll von Subklassen realisiert werden. Um dieses Ziel zu erreichen, erfolgt dann der Aufruf einer abstrakten Methode `init()` aus dem Konstruktor der Basisklasse.

Wieso dieses zunächst sinnvoll erscheinende Vorgehen problematisch sein kann, betrachten wir an folgendem Beispiel: Die Basisklasse `AbstractBase` definiert eine Variable `baseValue` und deklariert eine abstrakte Methode `init()`, die in deren Konstruktor aufgerufen wird. Die abgeleitete Klasse `Derived` implementiert diese `init()`-Methode und gibt dort die Werte der `Integer`-Variablen `baseValue` sowie `value` aus.

```
public final class AbstractMethodInCtorExample
{
    abstract static class AbstractBase
    {
        protected final Integer baseValue = 42;

        AbstractBase()
        {
            // Aufruf der Initialisierung
            init();
        }

        abstract void init();
    }

    static class Derived extends AbstractBase
```

```
    {
        private final Integer value = 13;

        Derived()
        {
        }

        void init()
        {
            // Zugriff auf Attribut der Basisklasse
            System.out.println("baseValue = " + baseValue);

            // Zugriff auf Attribut dieser Klasse
            System.out.println("value = " + value);
        }
    }

    public static void main(String[] args)
    {
        // Konstruktion zur Demonstration der Probleme
        new Derived();
    }
}
```

Listing 16.1 *Ausführbar als* **'ABSTRACTMETHODINCTOREXAMPLE'**

Intuitiv erwartet man die Ausgabe der Zahlen `42` und `13`. Schauen wir mal, ob Intuition und Wirklichkeit zusammenpassen, und führen das Programm ABSTRACTMETHODINCTOREXAMPLE aus. Wir erhalten folgende Ausgabe:

```
baseValue = 42
value = null
```

Wie ist das zu erklären? Folgen wir dem Programmfluss, so ist der Ablauf wie folgt: In der Methode `main()` wird ein Objekt der Klasse `Derived` konstruiert: Der Aufruf des Defaultkonstruktors ruft wiederum den Konstruktor der Basisklasse auf (Details zum Ablauf sind in folgendem Hinweis beschrieben). Die dortige Ausführung der `init()`-Methode führt aufgrund der Polymorphie und des dynamischen Bindens zum Aufruf der `init()`-Methode der Klasse `Derived`. Da die Variable `value` zu diesem Zeitpunkt aber noch nicht zugewiesen wurde – der Konstruktor von `Derived` ist noch nicht vollständig abgearbeitet –, wird der Wert `null` ausgegeben. Erst nach Beendigung des Aufrufs der Methode `init()` wird auch die Abarbeitung des Basisklassenkonstruktors abgeschlossen. Danach wird erst das Attribut `value` als Teilschritt der Konstruktion und Initialisierung der Klasse `Derived` auf den Wert `13` gesetzt.

Hinweis: Ablauf der Objektkonstruktion

Machen wir uns einmal klar, welche Schritte beim Konstruktionsprozess eines Objekts ablaufen:

1. Bevor ein Objekt einer Klasse per Konstruktor erzeugt wird, wird die Klassenbeschreibung von einem sogenannten `ClassLoader` geladen, wenn die Klasse bisher unbenutzt war.

2. Der Konstruktionsprozess wird begonnen. Dazu wird zunächst der Speicherplatz für die Attribute bereitgestellt und diese mit ihrem Defaultwert initialisiert. Dies geschieht ausgehend von der Klasse `Object` für alle Basisklassen bis zur eigentlich zu konstruierenden Klasse.
3. Für alle Klassen erfolgen in umgekehrter Reihenfolge (von `Object` bis zur eigenen Klasse) folgende Schritte:
   (a) Alle Instanzvariablen, die in ihrer Deklaration einen initialen Wert angegeben haben, werden nun zugewiesen.
   (b) Anschließend werden die Instanz-Initializer ausgeführt. Diese kann man beispielsweise für aufwendigere Berechnungen der initialen Werte einsetzen. Beim Erzeugen eines Objekts werden die Initializer entsprechend ihrer Reihenfolge im Sourcecode ausgeführt. Nach Ausführung des letzten Initializer werden die Anweisungen im Konstruktor ausgeführt.
   (c) Der Konstruktor wird abgearbeitet und es erfolgen die Zuweisungen und Methodenaufrufe, so wie sie im Konstruktor stehen.

Bei einer Ableitungshierarchie wird das Objekt sozusagen Stück für Stück aus seinen Basisklassenbestandteilen zusammengesetzt, und zwar beginnend bei der obersten Basisklasse `Object`.

Warum ist das ein Bad Smell?

Der auf den ersten Blick korrekte Sourcecode verhält sich unerwartet und es treten merkwürdige Effekte durch uninitialisierte Variablen auf. Daher ist der Aufruf abstrakter Methoden im Konstruktor ein Problem. ***Noch schwerer nachvollziehbar ist dies, wenn der Konstruktor eine konkrete `init()`-Methode verwendet und darin abstrakte Methoden aufgerufen werden.***

Tipps und Refactorings

Man sollte abstrakte Methoden niemals aus dem Konstruktor aufrufen. Man umgeht Probleme, indem man eine `init()`-Methode einführt und diese nach der Objektkonstruktion aufruft. Es bietet sich der Einsatz der Muster ERZEUGUNGSMETHODE oder FABRIKMETHODE (vgl. Abschnitt 18.1.1 und 18.1.2) an, um diesen Ablauf sicherzustellen.

Durch die Refactorings ÜBERPRÜFE EINGABEPARAMETER (Abschnitt 17.5.2) und FÜHRE EINE ZUSTANDSPRÜFUNG EIN (Abschnitt 17.5.1) erreicht man zum einen die Prüfung von Eingabeparametern sowie zum anderen im Idealfall die Sicherstellung eines gültigen Objektzustands. Wird im Falle einer fehlerhaften Initialisierung eine Exception geworfen, so erhält man Hinweise auf die Ursache eines zuvor möglicherweise unerklärlichen Fehlers.

Variante

Dieses Beispiel zeigt ein mögliches Initialisierungsproblem im Zusammenhang mit Threads. Betrachten wir folgendes Konstrukt, das eine Utility-Klasse `ThreadAutoStart` definiert, die einen Thread erzeugt und diesen automatisch startet:

```
public class ThreadAutoStart implements Runnable
{
    public ThreadAutoStart()
    {
        new Thread(this).start();
    }
}

public class ErroneousDataAccessThread extends ThreadAutoStart
{
    private final DataService service;

    public ErroneousDataAccessThread(final DataService service)
    {
        // Aufwendige Initialisierung
        // …
        this.service = service;
    }

    @Override
    public void run()
    {
        // Möglicherweise hier schon Zugriff, weil Initialisierung
        // im Konstruktor noch nicht abgeschlossen => NullPointerException
        final SomeData data = service.retrievData(…);
        // …
    }
}
```

Werden Klassen analog zu `ErroneousDataAccessThread` von der Basisklasse `ThreadAutoStart` abgeleitet, würde bereits während einer Objektkonstruktion ein Thread gestartet und dessen `run()`-Methode ausgeführt. Allerdings wären zu dem Zeitpunkt die Subklassenbestandteile noch nicht vollständig konstruiert. Dadurch sind ähnliche Probleme wie die zuvor geschilderten möglich.

Versteckte Initialisierungsprobleme

Im folgenden Beispiel findet sich ein ähnlicher Fehler, der durch eine zusätzliche Indirektion allerdings schwieriger zu erkennen ist. Die Basisklasse `CommunicationBase` definiert eine abstrakte Methode `createComponents()`, die von ihren Subklassen überschrieben werden muss, um Kommunikationskomponenten zu erzeugen:

```
public abstract class CommunicationBase
{
    CommunicationBase()
    {
        createComponents();
```

```
    }

    abstract protected void createComponents();
}
```

Eine Klasse `RadioCommunication` realisiert eine Funkkommunikation und erzeugt in der Methode `createComponents()` dafür jeweils einen Sender und einen Empfänger. Um eine unidirektionale Kommunikation zu erreichen, also exklusiv nur zu empfangen oder zu senden, kann man auf die Idee kommen, ein gemeinsames Synchronisationsobjekt `sharedSyncObject` folgendermaßen zu definieren und zu nutzen:

```
public final class RadioCommunication extends CommunicationBase
{
    // Definition des Synchronisationsobjekts
    private final Object  sharedSyncObject = new Object();
    private RadioSender   dataSender       = null;
    private RadioReceiver dataReader       = null;

    protected final void createComponents()
    {
        // Übergabe des Synchronisationsobjekts
        dataSender = new RadioSender( sharedSyncObject);
        // …
```

Die `RadioSender`-Klasse erhält das gemeinsame Synchronisationsobjekt im Konstruktor und nutzt es, um ein gleichzeitiges Senden und Empfangen zu unterbinden, in der Methode `send( byte[])` wie folgt:

```
public final class RadioSender
{
    private final Object sharedSyncObject;

    public RadioSender( final Object sharedSyncObject)
    {
        this. sharedSyncObject = sharedSyncObject;
    }

    public void send( final byte[] msg)
    {
        synchronized ( sharedSyncObject)
        {
            sendBytes();
        }
    }
    //…
```

Zur Laufzeit beobachten wir allerdings eine `NullPointerException` in der Zeile mit der Anweisung `synchronized ( sharedSyncObject)`. Zunächst scheint der Fehler unerklärlich: Das `sharedSyncObject` wird während der Objekterzeugung initialisiert und ist zudem `final`. Wie kann es dann `null` sein?

Eine Prüfung des Eingabeparameters im Konstruktor der Klasse `RadioSender` hätte einen Hinweis darauf gegeben, dass statt der erwarteten Referenz auf das Synchronisationsobjekt tatsächlich der Wert `null` übergeben wurde. Erinnern wir uns an die Diskussion bezüglich der Initialisierung: Das `RadioCommunication`-Objekt ist noch nicht vollständig konstruiert, wenn der Aufruf von

`createComponents()` aus dem Konstruktor der Basisklasse `CommunicationBase` erfolgt. Demnach ist auch das Attribut `sharedSyncObject` noch nicht korrekt initialisiert, sondern nur mit dem Defaultwert `null`.

16.2.4 Bad Smell: Referenzierung von Subklassen in Basisklassen

Entsprechend dem OO-Gedanken der Generalisierung definieren Basisklassen gemeinsame Funktionalität potenzieller Subklassen. In diesen Spezialisierungen sollten lediglich Unterschiede zur Basisklasse beschrieben werden. Teilweise sieht man aber Klassen, die ihre Spezialisierungen kennen.

Schauen wir uns dies in Abbildung 16-2 anhand der Basisklasse `DataManager` an, die drei konkrete Subklassen `DataManagerA` bis `DataManagerC` besitzt. In der `main()`-Methode der Klasse `DataManager` wird abhängig von einem Konfigurationsparameter `managerType` die jeweils zu erzeugende Subklasse gewählt.

Warum ist das ein Bad Smell?

Basisklassen sollten ihre Spezialisierungen nicht kennen. Andernfalls werden häufig Anpassungen in der Basisklasse nötig, wenn neue Subklassen definiert werden oder dort Erweiterungen stattfinden. Dies widerspricht dem OO-Gedanken der Basis- und Subklassen und dem Herausfaktorieren gemeinsamer Funktionalität in Basisklassen.

Eine Referenzierung von Subklassen in Basisklassen ist ähnlich unelegant wie ein Down Cast, wird aber leider auch in einigen Klassen des JDKs verwendet: Die Klasse `NumberFormat` kennt ihre Spezialisierung `DecimalFormat` und erzeugt sogar Instanzen davon.

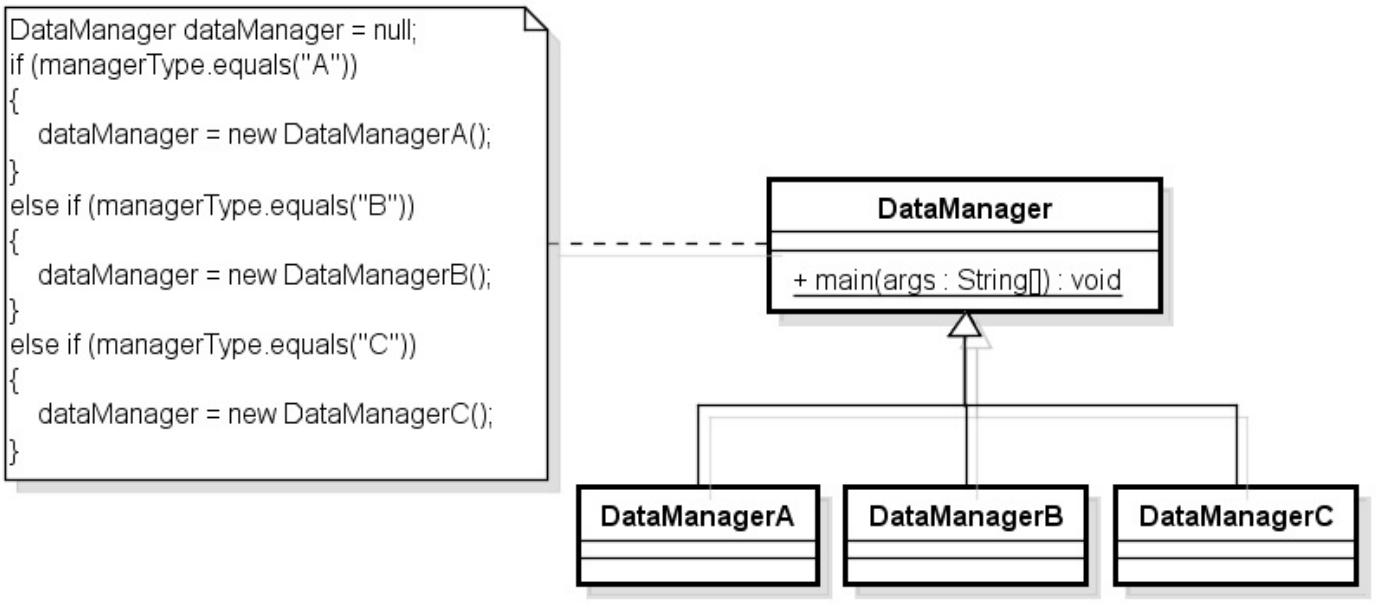


Abbildung 16-2 *Referenzierung von Subklassen in der Basisklasse*

Tipps und Refactorings

Durch den Einsatz des FABRIKMETHODE-Musters (vgl. Abschnitt 18.1.2) kann man das Problem der Referenzierung von Subklassen lösen. Die Klasse `DataManager` wird in drei Bestandteile zerlegt. Sie war ursprünglich für zu viele Dinge verantwortlich, nämlich a) das Starten des Programms, b) die Auswahl der konkreten Subklasse und c) das Agieren als Basisklasse. Nur die letztere Funktionalität verbleibt im neuen Design bei ihr. Zum Applikationsstart wird die `main()` -Methode in eine neu erzeugte Klasse `DataManagerStarter` verlagert. Die Auswahl und Erzeugung von Spezialisierungen erledigt die Klasse `DataManagerFactory`. Eine weitere Verbesserung bestünde darin, statt eines Strings einen `enum` als Konfigurationsparameter zu verwenden. Abbildung 16-3 zeigt die Verbesserungen.

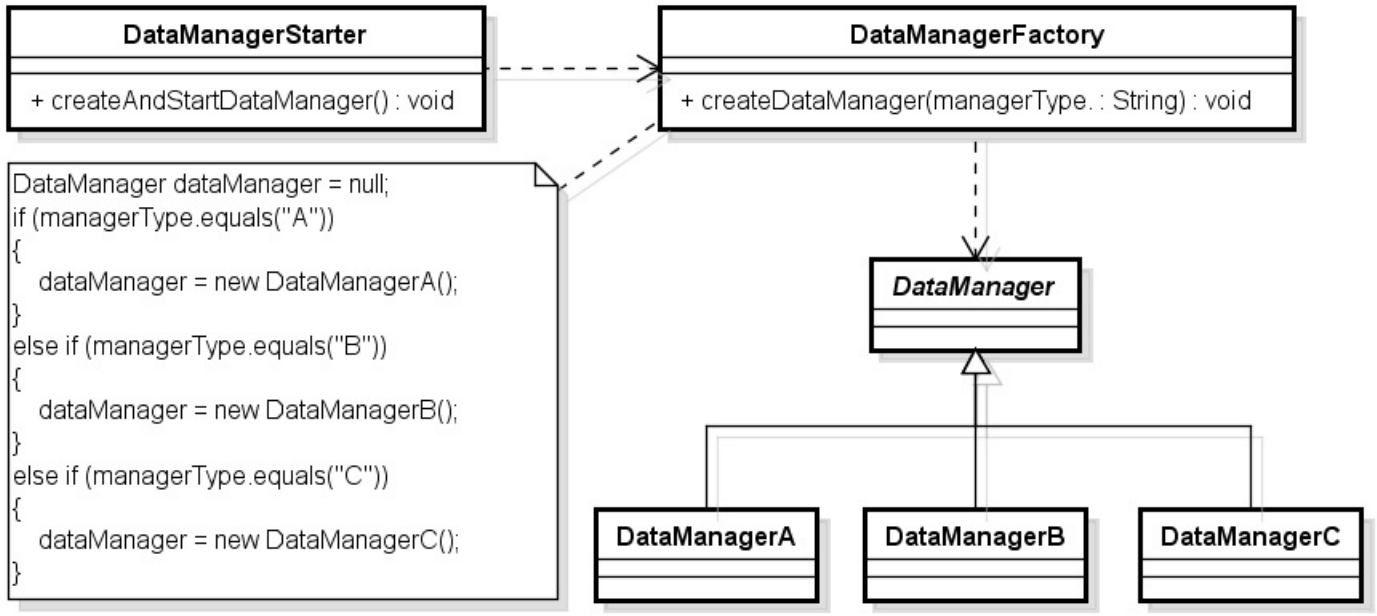


Abbildung 16-3 *Einführen einer Starter-Klasse und einer Fabrikmethode*

16.2.5 Bad Smell: Mix abstrakter und konkreter Basisklassen

In komplexen Vererbungshierarchien werden Basisklassen zum Teil nicht abstrakt definiert, obwohl sich dieses empfehlen würde, weil man davon ausgehen sollte, dass die Basisklassen noch nicht alle Funktionalitäten enthalten. Eine Instanziierung dieser Klassen ist demnach möglich, aber wenig sinnvoll.

Betrachten wir ein Beispiel einer derartigen Ableitungshierarchie, dargestellt als UML-Diagramm in Abbildung 16-4.

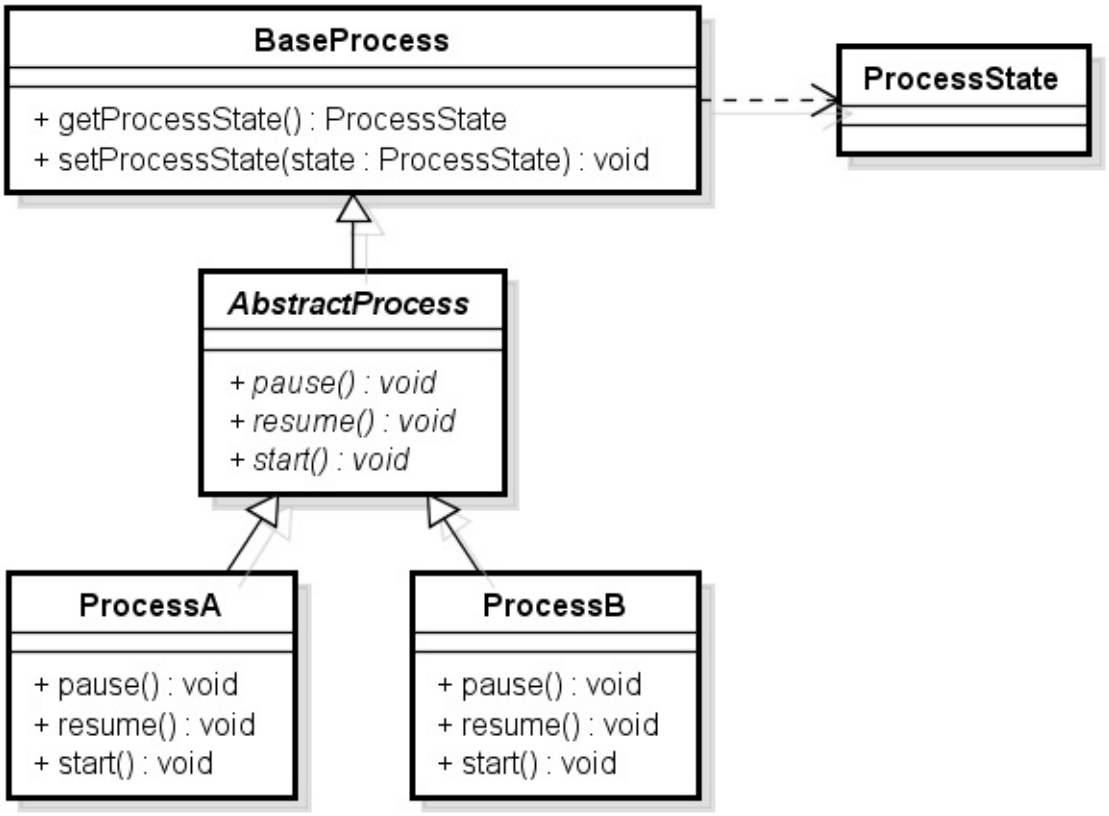


Abbildung 16-4 *Mix abstrakter und konkreter Basisklassen*

Die konkrete Basisklasse `BaseProcess` definiert die Basisfunktionalität für einen Prozessschritt und erlaubt mit den Methoden `getProcessState()` und `setProcessState(ProcessState)` eine Verarbeitung von Zustandsinformationen. Die davon abgeleitete Klasse `AbstractProcess` ist abstrakt und deklariert Methoden zum Starten, Anhalten und Fortsetzen eines Prozessschritts. Diese Methoden sind wiederum abstrakt, weil die Realisierungen der auszuführenden Aktionen in dieser abstrakten Basisklasse noch unbekannt sind und in den Subklassen definiert werden müssen. Nur diese »wissen«, welche Zustandsinformationen beim Anhalten zu speichern sind, um ein Fortsetzen der Verarbeitung sicherzustellen. Die konkreten Subklassen `ProcessA` und `ProcessB` definieren die Methoden `start()`, `pause()` und `resume()` entsprechend ihren Anforderungen.

Warum ist das ein Bad Smell?

Bei einer Klasse, die nicht abstrakt definiert ist, sollte man als Nutzer davon ausgehen können, dass diese eigenständig zu instanziieren und zu benutzen ist. Basisklassen nicht abstrakt zu machen, birgt also die Gefahr, dass diese (versehentlich) instanziiert werden und zum Einsatz kommen. In der Regel wollen wir dies jedoch nicht erlauben, da derartige Basisklassen noch kein funktional vollständiges Objekt darstellen. Eine Verwendung ist daher eigentlich nicht vorgesehen und das Objektverhalten unbestimmt. Noch verwirrender wird das Ganze, wenn in einer Vererbungshierarchie die Basisklassen teilweise abstrakt sind, teilweise nicht. Welche Schlussfolgerungen soll man daraus ziehen? Eine Basisklasse ist für sich allein

einsatzfähig, aber eine davon abgeleitete Subklasse nicht? Das macht wenig Sinn.

Tipps und Refactorings

In diesem Beispiel bieten sich folgende zwei Lösungsmöglichkeiten an:

1. Die vermeintliche Basisklasse `BaseProcess` definiert nur Hilfsmethoden zur Statusverwaltung eines Prozesses. Dies stellt in der Regel einen Designfehler dar, weil Vererbung lediglich zur Übernahme aus der Basisklasse benötigter Funktionalität eingesetzt wird. Für die *konkrete* Klasse `BaseProcess` und ihre *abstrakte* Subklasse `AbstractProcess` liegt keine »is-a«-Beziehung vor. Daher nutzt man besser eine »has-a«-Beziehung, die durch eine Assoziation ausgedrückt werden kann. Die Klasse `BaseProcess` wird dann als Hilfsklasse verwendet. Durch Umbenennen in `ProcessStateUtils` wird deren Utility-Charakter verdeutlicht. Eine mögliche Realisierung ist in Abbildung 16-5 dargestellt.

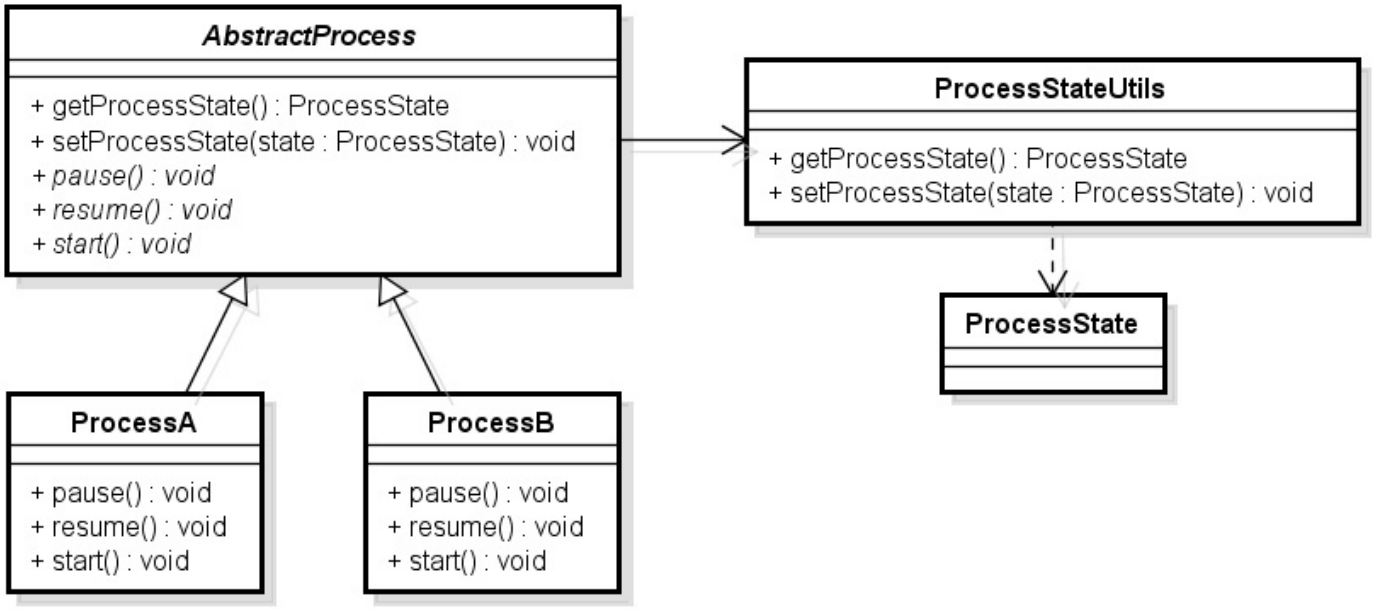


Abbildung 16-5 *Abhilfe durch Einführen einer Utility-Klasse*

2. Nehmen wir an, es läge tatsächlich eine »is-a«-Beziehung vor und die Funktionalität wäre eher »künstlich« oder willkürlich auf die Basisklassen verteilt. In diesem Fall sollten die beiden Basisklassen zu einer gemeinsamen verschmolzen werden. Für dieses Beispiel kann die gesamte Funktionalität aus der dann überflüssigen Basisklasse `BaseProcess` in die abstrakte Klasse `AbstractProcess` integriert werden. Eine sinnvolle Erweiterung besteht darin, ein Interface `IProcess` zu erzeugen. Dieses definiert die Methoden aus der Klasse `BaseProcess` sowie die abstrakten Methoden `pause()`, `resume()` und `start()`. Durch den Einsatz wird eine bessere Kapselung und damit auch eine losere Kopplung erreicht. Eine mögliche Umsetzung zeigt Abbildung 16-6.

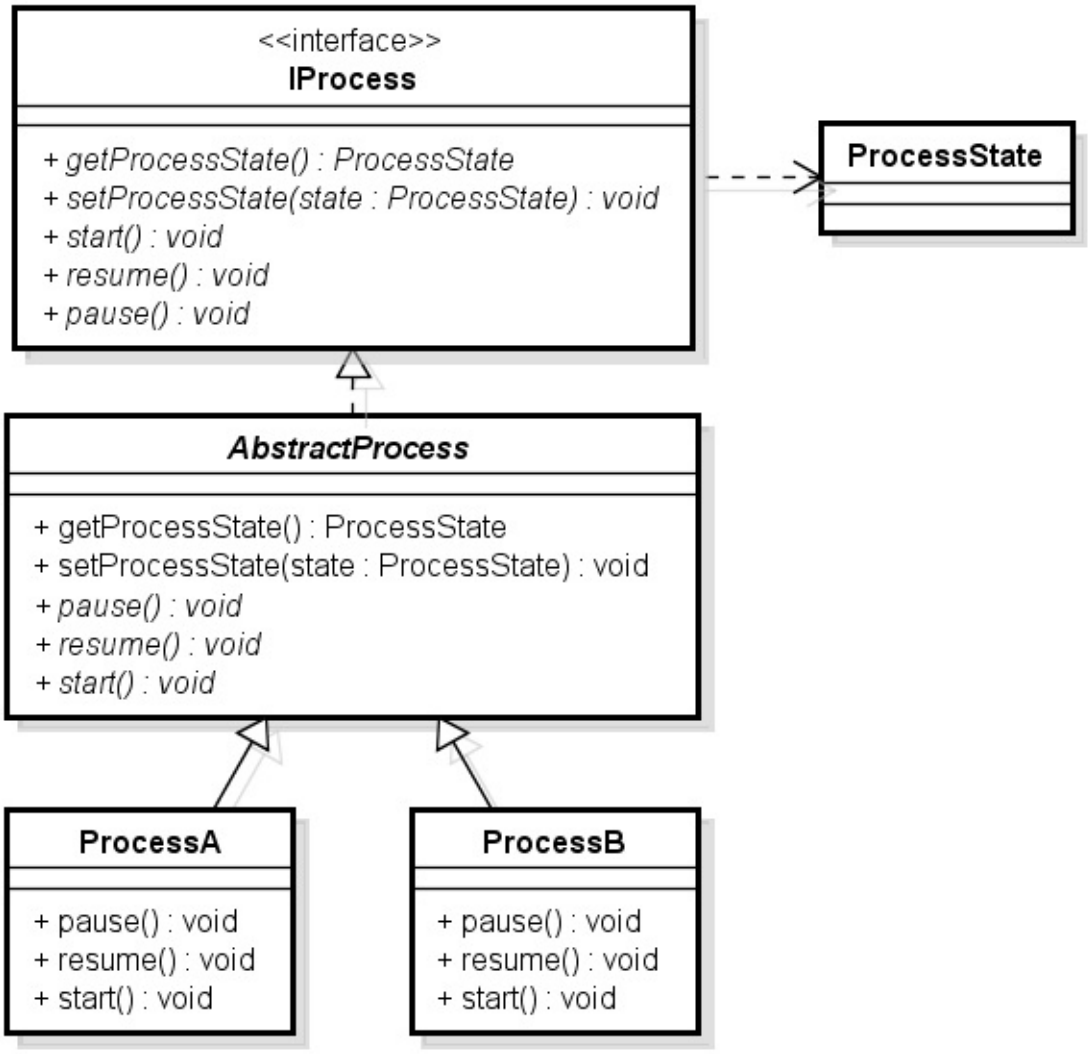


Abbildung 16-6 *Abhilfe durch Interface und Zusammenfassen von Funktionalität*

16.2.6 Bad Smell: Öffentlicher Defaultkonstruktor lediglich zum Zugriff auf Hilfsmethoden

Man sieht häufig Klassen, in denen ein öffentlicher Defaultkonstruktor existiert. Selten wird ein solcher aber tatsächlich benötigt. Zum Teil ist dessen Existenz lediglich dadurch begründet, dass ein Zugriff auf Konstanten oder Hilfsmethoden erfolgen soll, die fälschlicherweise nicht statisch definiert sind.

Betrachten wir dies am Beispiel der folgenden Utility-Klasse `FigureDrawUtils`. Hier werden eine Hilfsmethode `drawBorder(Graphics, Color, BaseFigure)` sowie eine Konstante `BORDER_COLOR` definiert – beide sind nicht statisch:

```
public final class FigureDrawUtils
{
    public final Color BORDER_COLOR = Color.RED;

    public FigureDrawUtils()
    {
    }
```

```
    public void drawBorder(final Graphics graphics, final Color borderColor,
                           final BaseFigure figure)
    {
        graphics.setColor(borderColor);
        graphics.drawRect(figure.getX(), figure.getY(),
                          figure.getWidth(), figure.getHeight());
    }
}
```

Möchte man die Konstante `BORDER_COLOR` oder die gezeigte Hilfsmethode verwenden, um beispielsweise einen Selektionsrahmen zu zeichnen, so benötigt man ein (beliebiges) `FigureDrawUtils`-Objekt. Im Extremfall würde für jeden Zugriff eine Objektkonstruktion erfolgen:

```
new FigureDrawUtils().drawBorder(graphics,
                                 new FigureDrawUtils().BORDER_COLOR,
                                 circleFigure);
```

Warum ist das ein Bad Smell?

Zum Zugriff auf nicht statische Konstanten oder Utility-Methoden muss man entweder bereits ein Objekt der Klasse referenzieren oder dafür sogar ein neues »Dummy«-Objekt erzeugen. Letzteres ist problematisch, da eventuell in einem Nutzungskontext kein Objekt erzeugt werden kann, beispielsweise, weil zu übergebende Werte für Konstruktorparameter nicht sinnvoll gewählt werden können. Als vermeintliche »Lösung« wird aus diesem Grund dann zum Teil ein Defaultkonstruktor bereitgestellt, ohne die daraus resultierenden Konsequenzen zu bedenken: Ein später hinzugefügter Defaultkonstruktor birgt die Gefahr, Objekte ohne sinnvollen Initialisierungszustand erzeugen zu können. Mögliche Auswirkungen davon wurden bereits in Abschnitt 16.2.1 als BAD SMELL: UNNÖTIGERWEISE VERÄNDERLICHE ATTRIBUTE angesprochen.

Ein solches Vorgehen provoziert einerseits Inkonsistenzen, andererseits leiden die Verständlichkeit sowie die Lesbarkeit. Inkonsistenzen entstehen dadurch, dass der Zugriff auf Methoden und Attribute möglich ist, obwohl derart erzeugte Objekte häufig keinen gültigen Zustand besitzen. Die Lesbarkeit wird eingeschränkt, weil Objektkonstruktionen notwendig werden, um auf diese nicht statischen Utility-Methoden zuzugreifen. Ein solcher Konstruktor führt demnach zu hässlichem und unverständlichem Sourcecode.

Tipps und Refactorings

Existiert ein öffentlicher Defaultkonstruktor ohne Initialisierungen nur zum »bequemeren« Zugriff auf Hilfsmethoden oder Konstanten, so sind alle Utility-Methoden und Konstanten statisch zu deklarieren, um einen Zugriff ohne vorherige Objektkonstruktion zu ermöglichen. Anschließend kann dieser Defaultkonstruktor entfernt oder als `private` deklariert werden. ***Methoden, die keinen Objektzustand abfragen oder modifizieren, sollten statisch sein und gegebenenfalls in eine separate***

Utility-Klasse ausgelagert werden. Das dazu notwendige Vorgehen wird als Refactoring WANDLE IN UTILITY-KLASSE MIT STATISCHEN HILFSMETHODEN UM in Abschnitt 17.4.14 beschrieben.

16.3 Fehlerbehandlung und Exception Handling

16.3.1 Bad Smell: Unbehandelte Exception

Unbehandelte Exceptions können verschiedenste Probleme verursachen. Gerade wenn man mit einer Entwicklungsumgebung arbeitet, findet man leere `catch`-Blöcke, die durch automatische Sourcecode-Vervollständigungen entstehen. Dort finden sich meistens ein `TODO`-Kommentar und ein `printStacktrace()`. Noch schlimmer ist das »Verschlucken« der Exception wie im folgenden Beispiel:

```
catch (final RemoteException e)
{
    System.out.println("to make Volker happy :-)");
}
```

Ob man mit einer solchen Ausgabe, wie mit dem `System.out.println()` angedeutet, wirklich jemanden glücklich macht, kann man bezweifeln. Ich konnte dem Auffinden dieser Ausgabe im Produktionscode jedenfalls nicht wirklich Freude abgewinnen, denn ich durfte eine sinnvolle Fehlerbehandlung implementieren.

Warum ist das ein Bad Smell?

Bleiben Exceptions unbehandelt, so erfährt der Programmanwender eventuell niemals von möglichen Problemen. Eine `RemoteException`, wie im obigen Beispiel, deutet aber auf ein Netzwerkproblem hin. Derartige Probleme sollten dem Benutzer kommuniziert oder zumindest für eine spätere Analyse in eine Log-Datei geschrieben werden.

Tipps und Refactorings

Man sollte Exceptions innerhalb der Methode, in der die Exception auftreten kann, behandeln, sofern dort eine sinnvolle Fehlerbehandlung möglich ist. Erfolgt keine Behandlung, so sollte die Exception weiter propagiert werden. Dadurch können höhere Applikationsschichten auf eine Fehlersituation reagieren. In der Regel ist dies dort zudem angemessener möglich als beispielsweise inmitten einer Berechnung.

Anhand von `IOExceptions` und `RemoteExceptions` kann man sich dies gut verdeutlichen: Bei einem Zugriffsproblem kann eine ausführende Methode oft nicht adäquat reagieren. Ein Aufrufer kann aber eine Wiederholung veranlassen oder eine neue Verbindung herstellen.

Ist zunächst offen, ob auf eine Fehlersituation angemessen reagiert werden kann

oder eben nicht, so sollte man zumindest eine Log-Ausgabe durchführen. Sinnvoll ist es, einen `@TODO`-Kommentar als eine Erinnerung an die noch ausstehende Fehlerbehandlung im Sourcecode zu hinterlassen.

Eine Sache sollte man noch bedenken: Im Umfeld von Applikationsservern können während der Verarbeitung ausgelöste `Exceptions` zu Rollbacks von Transaktionen führen und somit ungewollt den Programmablauf beeinflussen.

Achtung: Gefahr des Verschleierns

Eine Ausgabe von Exceptions in eine Log-Datei kann hilfreich sein. Als einzige Reaktion ist dies meistens jedoch für eine sinnvolle Fehlerbehandlung nicht ausreichend, da die Exception lautlos »versickert«. Allerdings ist ein solches Logging für eine nachträgliche Fehlersuche und -analyse hilfreich. Folgende Methode `storeTask(ITask)` liefert ein Beispiel für das »Versickern« einer `RemoteException`:

```
private void storeTask(final ITask task)
{
    try
    {
        taskService.storeTask(task);
    }
    catch (final RemoteException e)
    {
        log.error("Unable to store task in db!  task = " + task, e);
    }
}
```

Im gezeigten Exception Handling fehlt eine weitere Behandlung der auftretenden `RemoteException`, die auf ein Verbindungsproblem hinweist. Nehmen wir an, in einer höheren Applikationsschicht existiere genau für solche Verbindungsprobleme eine eingebaute Logik zum Neuverbinden. Diese wird als Folge der fehlenden Propagation jedoch nicht mehr aufgerufen. Daher sollte in diesem Fall die Exception an die aufrufende Methode weitergeleitet werden.

16.3.2 Bad Smell: Unpassender Exception-Typ

Beim Auftreten von Fehlern sollten diese möglichst aussagekräftig propagiert werden. Wenn dazu eine Exception geworfen wird, bedeutet dies, dass besondere Aufmerksamkeit bei der Wahl des Exception-Typs erforderlich ist. Ein unpassend gewählter Exception-Typ erschwert eine Analyse des Programms im Fehlerfall.

Betrachten wir dazu als Beispiel folgende Methode `fill(String)`, in der auf unterschiedliche Fehlersituationen jeweils auf dieselbe Art und Weise reagiert wird: Eine fehlende Initialisierung des Attributs `startFolder` sowie die Belegung des Attributs `ascending` mit `false` lösen jeweils eine `NullPointerException` aus:

```
public void fill(final String path)
{
```

```
    if (this.startFolder != null)
    {
        if (this.ascending)
            fillFolderContent(path);
        else
            throw new NullPointerException("Sort order not implemented!");
    }
    else
    {
        throw new NullPointerException("No startfolder!");
    }
}
```

Warum ist das ein Bad Smell?

Die hier für die Fehlerbehandlung gewählten `NullPointerExceptions` werden mit Informationstexten erzeugt und geben Hinweise auf eine mögliche Fehlerursache. Damit sind auftretende Fehler zwar anhand des Stacktrace relativ gut nachvollziehbar, allerdings ist der verwendete Exception-Typ irreführend. Es ist schlecht, durch eine `NullPointerException` eine unvollständige Implementierung der unterstützten Sortierreihenfolgen auszudrücken. Zum Teil werden in der Praxis leider Exceptions ohne Hinweistext geworfen. Das erschwert eine mögliche Fehlersuche zusätzlich. Dieser Effekt verstärkt sich, wenn – wie hier – mehrfach der gleiche Exception-Typ Anwendung findet. Nachfolgende Änderungen anderer Entwickler können dazu führen, dass die Zeilennummern aus einem Stacktrace nicht mehr mit denen des aktuellen Sourcecodes übereinstimmen. Welche Situation zum Fehler geführt hat, ist dann nur noch schwierig zu ermitteln.

Tipps und Refactorings

Zunächst führen wir eine Parameterprüfung ein, wie dies im Refactoring ÜBERPRÜFE EINGABEPARAMETER in Abschnitt 17.5.2 beschrieben wird. Ein nicht initialisiertes Attribut `startFolder` löst nun eine `IllegalStateException` anstelle einer `NullPointerException` aus. Dadurch wird nicht der offensichtliche Fehler (Wert ist `null`), sondern der zugrunde liegende semantische Fehler (fehlerhafte Initialisierung des Objektzustands) deutlicher ausgedrückt. Noch kritischer ist das Auslösen einer `NullPointerException` für den Fall einer fehlenden Implementierung, hier der nicht implementierten, absteigenden Sortierung. Dafür ist eine `UnsupportedOperationException` viel besser geeignet. Befolgen wir diese Hinweise, so erhalten wir nach einigen Änderungen folgenden Sourcecode, der zunächst mögliche Fehlersituationen und benötigte Vorbedingungen prüft und gegebenenfalls bei Verstößen Exceptions auslöst. Aufgrund der einzeiligen `throw`-Anweisungen wird dabei auf eine Klammerung verzichtet. Sind alle Vorbedingungen erfüllt, kommt es zur Ausführung der Methodenaktion `fillFolderContent()`:

```
public void fill(final String path)
{
   // Parameter-Check
   Objects.requireNonNull("parameter 'path' must not be null!");
```

```
    // Initialization-Check
    if (startFolder == null)
        throw new IllegalStateException("attribute 'startFolder' must not be "
+
                        "null! Call initialize() before any other method
call!");

    // Implementation-Check
    if (!ascending)
        throw new UnsupportedOperationException("descending sort order " +
                                                "not implemented!");
    fillFolderContent(path);
}
```

Beachten Sie dabei die Unterschiede in den Hinweistexten der Exceptions: Diese enthalten nun zusätzliche Informationen zu Methodenparametern und Attributen, um mögliche Fehlerursachen klarer nachvollziehbar zu machen.

16.3.3 Bad Smell: Exceptions zur Steuerung des Kontrollflusses

Dieser Abschnitt erklärt, warum Exceptions nicht zur Steuerung des Kontrollflusses eingesetzt werden sollten.

Betrachten wir dazu folgende Methode `sum(int[])`, die die Summe eines übergebenen `int[]` berechnen soll. In einer Endlosschleife wird der aktuelle Wert an Position `i` auf die Variable `sum` addiert:

```
private int sum(final int[] values)
{
    int i = 0;
    int sum = 0;
    try
    {
        while (true)
        {
            sum += values[i];
            i++;
        }
    }
    catch (final Exception ex)
    {
        // Ende des Arrays erreicht, Summenberechnung stoppen
    }
    return sum;
}
```

Warum ist das ein Bad Smell?

Der indizierte Zugriff löst irgendwann – je nach Größe des übergebenen Arrays – beim indizierten Zugriff eine `ArrayIndexOutOfBoundsException` aus. Dadurch endet die Schleife. Diese ungewöhnliche Art des Schleifendurchlaufs ist

einige Zeilen länger und durch das Exception Handling schlechter lesbar als eine »normale« `for`- oder `while`-Schleife. Auch die eigentliche Programmlogik lässt sich schlechter nachvollziehen. Darüber hinaus birgt diese »Exception«-Schleife das Problem, dass alle möglichen Arten von Exceptions im `catch`-Block gefangen werden. Dadurch werden möglicherweise Fehlersituationen »verschluckt«. Weitere Probleme solcher `catch`-Blöcke werden als BAD SMELL: FANGEN DER ALLGEMEINSTEN EXCEPTION in Abschnitt 16.3.4 sowie als BAD SMELL: UNBEHANDELTE EXCEPTION in Abschnitt 16.3.1 diskutiert.

Tipps und Refactorings

Die obige Variante des Schleifendurchlaufs ohne Prüfung auf Array-Grenzen war in früheren JVMs schneller als eine `for`- oder `while`-Schleife mit einer Prüfung der Array-Grenzen – allerdings natürlich bei Weitem schlechter lesbar. Die mit JDK 5 eingeführte for-each-Schleife erlaubt es, weiter zu abstrahieren, und hilft, Fehler bei der Prüfung auf Array-Grenzen und beim indizierten Zugriff zu vermeiden.

Exceptions sollten nur zur Behandlung von außergewöhnlichen Fehlersituationen eingesetzt werden. Eine Steuerung des Kontrollflusses erfolgt vorzugsweise durch konventionelle Java-Sprachmittel.

16.3.4 Bad Smell: Fangen der allgemeinsten Exception

Leider sieht man viel zu häufig das Fangen der unspezifischen Exception als Fehlerbehandlung, wie dies bereits im vorherigen Beispiel der `sum(int[])`-Methode der Fall war:

```
catch (final Exception ex)
{
    // beliebiger Sourcecode …
}
```

Warum ist das ein Bad Smell?

Ein solcher `catch`-Block verfehlt nahezu immer das Ziel, angemessen auf die jeweilige Fehlersituation reagieren zu können. Es findet eine Vermischung der Behandlung erwarteter und beliebiger anderer, unerwarteter Exceptions statt. Viele Fehlersituationen werden demnach durch dieses unspezifische Exception Handling verdeckt. Dies lässt sich sehr schön anhand der `sum(int[])`-Methode aus dem vorherigen Bad Smell nachvollziehen. Die (versehentliche) Übergabe einer `null`-Referenz für die Variable `values` löst dort eine `NullPointerException` aus. Dieser Fehler bleibt vor dem Aufrufer verborgen, weil er vermeintlich in `catch (Exception ex)` behandelt wird, allerdings erfolgt keine Summierung und es kommt zur Rückgabe von 0 als Ergebnis. Der auslösende, schwerwiegende

Applikationsfehler, hier das nicht initialisierte `values`-Array, fällt so allerdings nicht direkt auf: Es ist lediglich ein Fehler im Algorithmus, jedoch kein Programmabsturz beobachtbar.

Tipps und Refactorings

Eine mögliche Verbesserung stellt das Fangen eines besser geeigneten Exception-Typs dar, für die `sum(int[])`-Methode etwa eine `ArrayIndexOutOfBoundsException`. Aber selbst das funktioniert nur für den Fall, dass die Methode `sum(int[])` keine weitere Methode aufruft, die eine solche Exception auslösen kann. Ansonsten würde die Summierung einfach an einer beliebigen Stelle abgebrochen und falsche Ergebnisse liefern.

Ausnahmen

Es gibt ganz spezielle Fälle, in denen ein `catch (Exception ex)` tatsächlich etwas mehr Klarheit in den Sourcecode bringen kann. Dies ist immer dann möglich, wenn auf verschiedenste Exceptions exakt gleich reagiert wird. Wir betrachten dazu folgendes Beispiel, das auf drei verschiedene Exceptions (`Remote-`, `Finder-` und `RemoveException`) jeweils eine identische Log-Ausgabe produziert:

```
public void removeFromHistory(final int taskId)
{
    try
    {
        taskService.remove(Integer.valueOf(taskId));
    }
    catch (final RemoteException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
    catch (final FinderException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
    catch (final RemoveException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
}
```

Diese Sourcecode-Duplikation ist zu vermeiden. Denken wir kurz nach und schreiben die Methode dann um. Wir müssen dabei beachten, unerwartete Exceptions nicht wie die drei explizit abzufangenden und erwarteten Exceptions zu behandeln. Diese Aufgabe können wir mit einigen `instanceof`-Prüfungen wie folgt lösen:

```
public void removeFromHistory(final int taskId)
{
    try
    {
        taskService.remove(Integer.valueOf(taskId));
    }
    catch (final Exception ex)
```

```
    {
        if (ex instanceof RemoteException ||
            ex instanceof FinderException ||
            ex instanceof RemoveException)
        {
            log.error("Unable to remove task with id = " + taskId, ex);
            return;
        }
        throw new RuntimeException(ex); // Ummantelung mit RuntimeException
    }
```

Hiermit vermeidet man Sourcecode-Duplikation und erhöht die Lesbarkeit sowie Übersichtlichkeit – das Exception Handling erfordert aber einen Trick: Die ursprüngliche Methode definiert keine Checked Exception in der Signatur. Um dazu kompatibel zu sein, muss eine auftretende Exception mit einer `RuntimeException` ummantelt werden, die nicht in der Signatur angegeben werden muss (vgl. Abschnitt 4.7.2).

Vereinfachungen mit JDK 7

Mit JDK 7 wird das Exception Handling erleichtert (vgl. Abschnitt 4.7.4). Wir nutzen das sogenannte »Multi Catch«, mit dem mehrere Exceptions gefangen werden können, zur Vereinfachung des Exception Handlings in der Methode `removeFromHistory(int)`:

```
public void removeFromHistory(final int taskId)
{
    try
    {
        taskService.remove(Integer.valueOf(taskId));
    }
    // Multi Catch ab JDK 7
    catch (final RemoteException | FinderException | RemoveException ex)
    {
        log.error("Unable to remove task with id = " + taskId, ex);
    }
}
```

16.3.5 Bad Smell: Rückgabe von `null` statt Exception im Fehlerfall

Dieser Bad Smell schildert, warum schwerwiegende Fehlersituationen besser durch das Auslösen einer Exception als durch die Rückgabe von `null` (oder eines Fehlerwerts) ausgedrückt werden sollten. Die Rückgabe eines Werts erlaubt es Aufrufern, diesen zu ignorieren. Eine Checked Exception besitzt den Vorteil, nicht so einfach wie ein Rückgabewert ignoriert werden zu können: Eine Checked Exception muss entweder durch einen `catch`-Block behandelt oder mit `throw` weiter propagiert werden. Für Unchecked Exceptions ist beides optional auch möglich.

Betrachten wir folgende Methode `getServerAndPort()`, die aus einer

Property-Datei über die Methode `getPropertyValue( String)` zwei Werte liest. Sind beide vorhanden, wird aus diesen ein Servername mit Portnummer zusammengebaut. Fehlt eine der Informationen, so wird der Wert `null` zurückgeliefert. Dies ist jedoch weder im Methodenkommentar beschrieben, noch stellt der Wert `null` einen sinnvollen oder gültigen Rückgabewert dar:

```
/**
  * @return der Name des Servers in der Form 'host:port'.
  */
public String getServerAndPort()
{
    final String host = getPropertyValue(KEY_HOST);
    final String port = getPropertyValue(KEY_PORT);

    if (host != null && port != null)
    {
        return host + ":" + port;
    }

    return null;
}
```

Warum ist das ein Bad Smell?

Bei Referenzparametern ist die unbedachte Rückgabe des Werts `null` problematisch, wenn Aufrufer mit dem Rückgabewert ohne Prüfung weiterarbeiten. Dies beschreibt detailliert BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.6.

Verlässt sich aufrufender Sourcecode auf eine gültige Rückgabe und führt keine Prüfung auf `null` durch, so kommt es zu einer unbehandelten `NullPointerException`. Im obigen Beispiel bleibt bei korrekter Konfiguration der beschriebene Softwaredefekt unentdeckt – vielleicht sogar während der gesamten Entwicklungs- und Testphase. Wird die Konfiguration jedoch im Nachhinein (versehentlich) dahingehend verändert, dass einer der beiden Werte fehlt, löst dies unerwartet eine `NullPointerException` aus. Der Zusammenhang mit einem geänderten oder fehlenden Property-Wert ist jedoch nicht ersichtlich.

Tipps und Refactorings

Das Auslösen einer Exception ermöglicht die Ausgabe eines Stacktrace, wodurch sich Fehler in der Regel gut nachvollziehen lassen. Mithilfe einer `IllegalStateException` wird ein möglicher Konfigurationsfehler frühzeitig erkennbar und semantisch besser beschrieben. Eventuell kann ein Problem dann umgehend durch Einfügen eines fehlenden Werts in eine Property-Datei behoben werden. Dazu ist es hilfreich, im Text der Exception zusätzlich den Pfad und den Namen der verwendeten Property-Datei anzugeben. Das erleichtert eine gegebenenfalls notwendige Fehlerbehandlung ungemein. Eine Realisierung könnte wie folgt aussehen:

```
public String getServerAndPort()
{
```

```
    final String host = getPropertyValue(KEY_HOST);
    final String port = getPropertyValue(KEY_PORT);

    if (host != null && port != null)
    {
        return host + ":" + port;
    }

    throw new IllegalStateException("missing property entries for server!" +
                  " no parameters '" + KEY_HOST + "' and/or '" + KEY_PORT +
                  "' in file '" + getPropertyFilePath() + "'.");
}
```

An diesem Beispiel erkennt man sehr schön, dass eine offensive Fehlerbehandlung eine spätere Wartung ungemein erleichtern kann. Exceptions sind zwar im Programmablauf ärgerlich, zur Fehlersuche sind sie meistens aber sehr hilfreich. Wird ein Fehler durch `null`-Prüfungen oder leere `catch`-Blöcke verschluckt, so macht er sich möglicherweise nur durch ein merkwürdiges Programmverhalten bemerkbar. Er lässt sich dann nur mit Mühe zurückverfolgen und durch intensive Sourcecode-Analyse finden und beheben.

16.3.6 Bad Smell: Unbedachte Rückgabe von `null`

Dieser Bad Smell beschreibt, warum die unbedachte Rückgabe von `null` Probleme bereiten kann. Nur in wenigen Fällen sollte man überhaupt mit dem Wert `null` als Rückgabe arbeiten. Dies gilt etwa für Suchfunktionen, um ausdrücken zu können, dass kein Treffer gefunden wurde. Für »normale« Verarbeitungsroutinen ist die Rückgabe von `null` meistens nicht sinnvoll, weil Aufrufer mit diesem Wert nicht rechnen. Im vorherigen BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL habe ich dargestellt, dass Fehlersituationen häufig besser durch Exceptions ausgedrückt werden können.

Betrachten wir eine Umwandlung zweier Eingabewerte in der folgenden `main()`-Methode. Dort erfolgt ein Aufruf der Methode `toHashSeparatedString(String)`, die nach jedem Zeichen im übergebenen String ein '#' einfügt:

```
public static void main(final String[] args)
{
    final String value1 = toHashSeparatedString("012");
    final String value2 = toHashSeparatedString("");

    System.out.println("Encoded='" + value1 + "' / length=" +
value1.length());
    System.out.println("Encoded='" + value2 + "' / length=" +
value2.length());
}

public static String toHashSeparatedString(final String message)
{
    if (message.length() == 0)
        return null;
```

```
    final StringBuffer sb = new StringBuffer("#");
    for (int i = 0; i < message.length(); i++)
    {
        sb.append(message.charAt(i)).append('#');
    }
    return sb.toString();
}
```

Listing 16.2 *Ausführbar als* **'DONTRETURNNULLEXAMPLE'**

Warum ist das ein Bad Smell?

Rückgaben von `null` müssen im aufrufenden Sourcecode speziell behandelt werden. Geschieht dies, wie in der obigen `main()`-Methode, nicht, so kommt es unerwartet zu einer `NullPointerException`: In diesem Beispiel also bei der Ausgabe der Länge des transformierten, zweiten Eingabewerts.

Aus der Methodensignatur und aufgrund der fehlenden Dokumentation ist aber nicht ersichtlich, dass der Wert `null` überhaupt als Rückgabewert möglich ist. Zudem wird hier mit der Rückgabe des Werts `null` ein erlaubter Randfall anders als gewöhnliche Eingabewerte ausgewertet. Ein derartiges Problem beschreibt BAD SMELL: SONDERBEHANDLUNG VON RANDFÄLLEN in Abschnitt 16.3.7. Eine Nachricht der Länge 0 sollte auf keinen Fall im nachfolgenden Sourcecode eine `NullPointerException` auslösen. Diese Realisierung provoziert unnötigerweise Exceptions in aufrufenden Methoden.

Tipps und Refactorings

Die Rückgabe eines leeren Strings für leere Eingabedaten ist viel eleganter und die einfachste Form des NULL-OBJEKT-Musters (vgl. Abschnitt 18.3.2). Mögliche Aufrufer müssten dadurch keine Abfragen auf `null` vornehmen und könnten direkt mit den Rückgabewerten arbeiten. Das sorgt für mehr Lesbarkeit.

Ausbreitung dieses Bad Smells

Problematisch ist vor allem die Rückgabe von `null`-Werten für Referenzen auf Arrays, Listen oder sonstige Containerklassen, da dies häufig zu einer Ausbreitung dieses Bad Smells führt. Betrachten wir dazu folgende vereinfachte Methode `getPersonsByPrefix(String)`, die als Rückgabe ein `Person[]` besitzt. In dieser Methode wird durch Aufruf der Hilfsmethode `findObjectsByPrefix(Class<?>, String)` der Utility-Klasse `DBAccess` ein `Object`-Array ermittelt. Ist dieses `null` wird hier auch der Wert `null` zurückgeliefert. Ansonsten erfolgt eine Umwandlung des Ergebnisses in ein `Person`-Array:

```
public static List<Person> getPersonsByPrefix(final String prefix)
{
   final Person[] result = (Person[])
DBAccess.findObjectsByPrefix(Person.class,
```

```
                                                                    prefix);

    if (result != null)
    {
        return Arrays.asList(result);
    }

    // Problematische Rückgabe von null
    return null;
}
```

Eine leere Treffermenge drückt man für Arrays und Containerklassen besser durch NULL-OBJEKTE aus (vgl. Abschnitt 18.3.2). Tut man dies nicht, so pflanzen sich `null`-Prüfungen immer weiter fort. In diesem Beispiel sieht man das an der Methode `findObjectsByPrefix(Class<?>, String)`. Würde diese ein leeres Array statt `null` zurückliefern, könnte man sich die Spezialbehandlung in der aufrufenden Methode `getPersonsByPrefix(String)` sparen. Wäre dem so, könnte man kurz und elegant Folgendes schreiben:

```
public Person[] getPersonsByPrefix(final String prefix)
{
    final Person[] result = (Person[])
DBAccess.findObjectsByPrefix(Person.class,
                                                                    prefix);

    return Arrays.asList(result);
}
```

Mögliche Schwierigkeiten

Wenn man externe Bibliotheken einsetzt, ist es manchmal schwierig zu verhindern, dass sich dieser Bad Smells ausbreitet. Das gilt insbesondere, wenn Methoden `null`-Werte für Collections zurückliefern. Oftmals hat man das Problem, dass man die Implementierung von Hilfsmethoden aus Bibliotheken nicht ändern kann. In solchen Fällen sollte man sich eine eigene Hilfsmethode schreiben, die die Bibliotheksmethode aufruft und um eine Fehlerbehebung erweitert. Dadurch wird die ansonsten notwendige Spezialbehandlung vor der eigenen Applikation versteckt und der nutzende Sourcecode vereinfacht. Außerdem kann man bei eigenen Realisierungen von Methoden darauf achten, leere Collections statt `null` als Rückgabe zu verwenden. Falls wir die Implementierung von `DBAccess` nicht im Zugriff haben und ändern können, so können wir doch etwas tun. Durch Befolgen es eben genannten Vorgehens können wir die Methode `getPersonsByPrefix()` folgendermaßen korrigieren, um eine Fortpflanzung von `null`-Abfragen zu verhindern:

```
public Person[] getPersonsByPrefix(final String prefix)
{
    // …

    return new Person[0];
}
```

16.3.7 Bad Smell: Sonderbehandlung von Randfällen

Zum Teil findet man Sonderbehandlungen für erlaubte, aber seltene oder ungewöhnliche Eingabewerte. Gleiches gilt für Rückgabewerte. Gerade deren geeignete Wahl kann das Erkennen von Fehlersituationen erleichtern.

Betrachten wir dies beispielhaft an der folgenden Methode `toHexString()`, die eine Stringrepräsentation in hexadezimaler Form aus einem übergebenen `byte`-Array `message` erzeugt. Sowohl bei der Übergabe von `null` als auch für eine leere Eingabe wird `null` zurückgegeben:

```
public static String toHexString(final byte[] message)
{

    if (message == null)
        return null;
    if (message.length == 0)
        return null;

    final StringBuffer sb = new StringBuffer("| ");
    for (int i = 0; i < message.length; i++)
    {
        // Wandle byte in einen Hex-String
        final String hexStr = "0" + Integer.toHexString((int) message[i]);
        sb.append(hexStr.substring(hexStr.length() - 2)).append('| ');
    }
    return sb.toString();
}
```

Warum ist das ein Bad Smell?

In diesem Beispiel stellt eine Eingabe von `null` eigentlich einen Fehler dar, was anhand des Rückgabewerts nicht deutlich wird. Die Eingabe eines leeren Arrays ist dagegen ein seltener, aber erlaubter Randfall.

Die Gleichbehandlung von Fehlern und Randfällen durch Rückgabe gleicher Werte führt zu Inkonsistenzen. Man stellt eine normale Situation – hier die Eingabe eines Arrays der Länge 0 – wie eine Fehlersituation dar. Betrachten wir die verschiedenen Fälle für den Eingabewert `message` genauer, um zu verstehen, warum Sonderbehandlungen von Randfällen und daraus resultierende, inkonsistente Rückgabewerte einen Bad Smell darstellen:

1. Wird ein Array der Länge 0 übergeben, so kann dafür eine sinnvolle Stringrepräsentation erzeugt werden, etwa "`[]`". Dieser Randfall sollte demnach genauso bearbeitet werden wie ein Array mit Eingabedaten. Von diesem Vorgehen sollte nur abgewichen werden, wenn dies explizit in der Methodendokumentation erwähnt wird. In diesem Beispiel führt ein leeres Array aber zur Rückgabe von `null` und ist damit anhand des Rückgabewerts nicht mehr von einem Fehler zu unterscheiden.
2. Im Fall einer ungültigen Eingabe (`null`) kann die Methode keine geeignete

Stringrepräsentation erzeugen. Die Rückgabe von `null` drückt dies nur unzureichend aus. Wenn dies explizit dokumentiert ist, stellt eine derartige Rückgabe nicht unbedingt einen Fehler dar. Es besteht aber immer die Gefahr, dass es zu einer Verschleierung eines Fehlers kommt.

Tipps und Refactorings

Die Spezialbehandlung für das leere Eingabe-Array ist überflüssig und kann komplett entfallen. Auf die Eingabe von `null` sollte mit einer `IllegalArgumentException` reagiert werden, wie dies bereits in Abschnitt 16.3.5 als BAD SMELL: RÜCKGABE VON NULL STATT EXCEPTION IM FEHLERFALL besprochen wurde. Der Aspekt der Parameterprüfung zur Sicherstellung eines gültigen Objektzustands ist in Abschnitt 16.3.8 Thema von BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN. Weshalb die Rückgabe von `null` problematisch ist und man die dadurch notwendigen `null`-Prüfungen nicht auf Klienten übertragen sollte, wurde bereits als BAD SMELL: UNBEDACHTE RÜCKGABE VON NULL in Abschnitt 16.3.6 vorgestellt.

16.3.8 Bad Smell: Keine Gültigkeitsprüfung von Eingabeparametern

Eine fehlende Prüfung von Eingabeparametern kann schnell zu inkonsistenten Objektzuständen führen. Meistens lassen sich Inkonsistenzen durch ungültige Parameterwerte bereits beim Aufruf von `set()`-Methoden oder im Konstruktor feststellen.

Betrachten wir dazu den folgenden Konstruktor der Klasse `FramedDisplayMsg`. Die Parameter `receiverNo` und `message` werden ungeprüft an entsprechende Attribute zugewiesen:

```
private final int    receiverNo;
private final byte[] message;

public FramedDisplayMsg(final int receiverNo, final byte[] message)
{
    this.receiverNo = receiverNo;
    this.message = message;
}
```

Weiterhin ist unter anderem folgende Zugriffsmethode definiert:

```
public byte[] getMessageWithoutFraming() throws IOException
{
    if (MsgSupport.checkMessage(this.message) != MsgSupport.CHECK_MSG_OK)
        throw new IOException("parameter 'message' has an invalid format " +
                              Arrays.toString(message));

    return ByteArrayUtils.extractByteArray(message, FRAME_OFFSET,
                                           message.length - FRAME_OFFSET);
```

```
}
```

Warum ist das ein Bad Smell?

Bei fehlerhaften Eingabewerten verbleibt ein neu erzeugtes `FramedDisplayMsg`-Objekt in einem nicht korrekt initialisierten Zustand, ohne dies zu kommunizieren. Auf ungültige Eingaben sollte allerdings so früh wie möglich reagiert werden, um inkonsistente Objektzustände zu vermeiden. Dies geschieht hier nicht und kann diverse unerklärliche Folgefehler auslösen.

Erst bei einem Aufruf der `getMessageWithoutFraming()`-Methode wird auf Inkonsistenzen geprüft. Werden diese erkannt, wird eine `IOException` ausgelöst, um einen Fehler zu signalisieren. Allerdings ist das eine irreführende Wahl des Exception-Typs – hier, um auf falsche Attributwerte zu reagieren. Das wurde bereits als BAD SMELL: UNPASSENDER EXCEPTION-TYP in Abschnitt 16.3.2 vorgestellt.

Diese Reaktion ist jedoch problematisch, da die Konsistenzprüfung zu spät erfolgt. Zwischen der Konstruktion und einem Aufruf der obigen Methode mit Konsistenzprüfung könnten Klienten bereits mit dem Objekt arbeiten. In diesem Beispiel sind `null`-Werte für die Attribute nicht erlaubt, aber durch die fehlende Parameterprüfung nicht ausgeschlossen. Das Auslesen eines Attributs kann unerwartet den Wert `null` liefern und damit `NullPointerExceptions` bei Aufrufern auslösen.

Das Schlimme in beiden Fällen ist, dass es sich hierbei um eine Fehlerverschleierung handelt, für die wir in unserer Arbeitsgruppe den Spitznamen »Delayed Exception« verwenden. Er besagt, dass Probleme erst beim Zugriff und nicht bereits beim Initialisieren auftreten. ***Das Ungerechte daran ist, dass nicht der Verursacher der Inkonsistenzen mit einer Exception bestraft wird, sondern irgendein späterer Nutzer der Klasse. Ist dies z. B. das GUI, so wird der Fehler schnell einem falschen Verursacher zugeordnet.***

Tipps und Refactorings

Das Prüfen aller Parameter öffentlicher Methoden und von Konstruktoren hilft, Initialisierungsprobleme so früh wie möglich aufzuspüren und falsch initialisierte Objekte (invalide Objektzustände) zu vermeiden. Ein entsprechendes Vorgehen ist in Abschnitt 17.5.2 als Refactoring ÜBERPRÜFE EINGABEPARAMETER detailliert beschrieben. Dort wird auch diskutiert, ob man alle Parameter oder nur eine Auswahl überprüfen sollte.

In diesem Fall kopieren wir die Konsistenzprüfung aus der `getMessageWithoutFraming()`-Methode und entfernen diese dort:

```
public byte[] getMessageWithoutFraming() throws IOException
{
    return ByteArrayUtils.extractByteArray(message, FRAME_OFFSET,
                                           message.length - FRAME_OFFSET);
}
```

Anschließend fügen wir diese Konsistenzprüfung in den Konstruktor ein und ergänzen noch eine Prüfung das Parameters `message` auf `null`. Außerdem wählen wir eine passendere Exception, die im Falle der fehlgeschlagenen Überprüfung ausgelöst wird:

```
public CorrectedFramedDisplayMsg(final int receiverNo, final byte[] message)
{
    if (MsgSupport.checkMessage(message) != MsgSupport.CHECK_MSG_OK)
        throw new IllegalArgumentException("parameter 'message' has an " +
                                "invalid format" + Arrays.toString(message));

    // Parameterprüfung mit Utility-Klasse Objects
    this.message = Objects.requireNonNull(message, "parameter 'message' must "
+
                                                  "not be null");
    this.receiverNo = receiverNo;
}
```

16.3.9 Bad Smell: Fehlerhafte Fehlerbehandlung

Zum Teil findet man Fehlerbehandlungsroutinen, die selbst wiederum Fehler enthalten oder die Fehlerbehandlung nur unvollständig oder nicht korrekt durchführen.

Das folgende Beispiel der Methode `synchronizeTime(long)` zeigt eine Abfrage des Attributs `currentSender` auf den Wert `null`. Im Falle eines `null`-Werts erfolgt eine Fehlermeldung in Form einer Log-Ausgabe.

```
public void synchronizeTime(final long newTime)
{
    if (currentSender == null)
    {
        log.error("Called synchronizeTime for a null current sender.");
    }
    currentSender.synchronizeTime(new Date(newTime));
}
```

Warum ist das ein Bad Smell?

Eine Fehlerbehandlung, die selbst Fehler auslöst, die vom eigentlichen Fehler ablenken, ist nicht wirklich sinnvoll. Im Beispiel hilft die Ausgabe des fehlerhaften Zustands nur bedingt weiter, da mit dem nächsten Statement `currentSender.synchronizeTime()` eine `NullPointerException` beim Dereferenzieren der `null`-Referenz geworfen wird. Wie man leicht sieht, fehlt in diesem Fall ein Methodenabbruch. Der fehlerhafte Objektzustand (die unvollständige Initialisierung) sollte per `IllegalStateException` an den Aufrufer kommuniziert werden. Die ansonsten auftretende `NullPointerException` hat nicht die gleiche Aussagekraft, da diese den ungültigen Objektzustand semantisch nicht so klar ausdrückt.

Außerdem könnte das Attribut `currentSender` privat sein, um ein

Implementierungsdetail zu verbergen, und wäre damit nach außen nicht sichtbar. Ein Aufrufer würde sich dann fragen: Was ist "currentSender" im Stacktrace der Exception?

Tipps und Refactorings

Eine Zustandsprüfung sollte sinnvollerweise in eine eigene Methode ausgelagert werden. Das beschreibt das Refactoring FÜHRE EINE ZUSTANDSPRÜFUNG EIN in Abschnitt 17.5.1. Wenden wir dies an, so entsteht die folgende Methode `checkInitialization()`:

```
private void checkInitialization()
{
    if (currentSender == null)
    {
        log.error("Illegal state: attribute 'currentSender' is
uninitialized");
        throw new IllegalStateException("Illegal state: attribute " +
                                        "'currentSender' is uninitialized");
    }
}
```

Durch den Einsatz einer solchen Prüfmethode wird der Sourcecode besser lesbar. Bei Bedarf kann diese Statusabfrage auch in anderen Methoden eingesetzt werden, ohne die Prüfungen durch Copy-Paste im Sourcecode zu duplizieren.

```
public void synchronizeTime(final long newTime)
{
    checkInitialization();

    currentSender.synchronizeTime(new Date(newTime));
}
```

Spezialfälle

In diesem einfachen Beispiel war der Fehler noch recht schnell zu finden, im folgenden Beispiel der Methode `send(Object)` und dem Einsatz von `instanceof` und `getClass()` ist er jedoch besser verborgen:

```
public void send(final Object telegram)
{
    if (!(telegram instanceof RadioTelegram))
    {
        log.error("Invalid telegram class " + telegram.getClass().getName());
        return;
    }
    // …
}
```

Der Aufruf von `instanceof` ist zwar laut JLS sicher bezüglich `null`. Das bedeutet: Ein Test `null instanceof XYZ` liefert immer `false`. Vorsicht ist jedoch geboten, wenn man die Aussage negiert. Denn eine Negation besagt nicht, dass die Variable ungleich `null` ist! Wird an die obige Methode eine `null`-Referenz als Parameter übergeben, so löst der Aufruf von `getClass()` eine

`NullPointerException` aus.

Tipp: Test der Fehlerbehandlung

Achten Sie darauf, auch die Fehlerbehandlung zu testen. Dazu sollten bewusst Testfälle definiert werden, in denen beispielsweise Referenzvariablen mit `null` initialisiert sind. Auch andere invalide Einträge sollten getestet werden, etwa ungültige Wertebereiche: Fehler finden sich häufig nicht im normalen Wertebereich, sondern bei Extremwerten. Es sollte dazu ein ***Extremwerttest*** durchgeführt werden. Der normale Programmablauf wird durch ***Äquivalenzklassentests*** abgesichert.

16.3.10 Bad Smell: I/O ohne `finally`

Bei der Arbeit mit Stream-Objekten werden zusätzlich zu den Java-Objekten auch Betriebssystemressourcen angefordert. Wird ein entsprechendes Java-Objekt zerstört, so werden nicht in jedem Fall als Folge davon auch die belegten Systemressourcen freigegeben. Dazu ist eine spezielle `close()`-Methode vom Entwickler selbst aufzurufen.

Betrachten wir hier einen typischen Sourcecode-Abschnitt, der Properties aus einer Datei lesen soll. Im Regelfall erfolgt nach einem Zugriff auf die Datei ein Aufruf von `close()`. Im `catch`-Block wurde dieser Aufruf vergessen. Die im folgenden Listing gezeigte Methode `readProperties(String)` macht dies deutlich:

```
private boolean readProperties(final String fileName)
{
    try
    {
        final FileInputStream inputStream = new FileInputStream(fileName);
        loadProperties(inputStream);
        inputStream.close();
    }
    catch (final IOException ex)
    {
        log.warn("can't read file '" + fileName + "'", ex);
        return false;
    }
    return true;
}
```

Warum ist das ein Bad Smell?

Problematisch an dieser Methode ist das Verhalten im Falle eines Fehlers beim Zugriff auf die Property-Datei. Tritt dabei eine `IOException` auf, so werden Ressourcen nicht garantiert wieder freigegeben, da kein direkter Aufruf von `close()` erfolgt.[3]

Tipps und Refactorings

Wie bereits in Abschnitt 4.7 diskutiert, sollte man Aufräumarbeiten immer in einem `finally`-Block durchführen. Tritt dann eine `IOException` auf, so kann das zuvor geöffnete `InputStream`-Objekt in jedem Fall umgehend wieder geschlossen und somit auch die allozierten Betriebssystemressourcen wieder freigegeben werden. Wir korrigieren dies, und es entsteht folgende verbesserte Methode `readProperties(String)`:

```
private boolean readProperties(final String fileName)
{
    InputStream inputStream = null;
    try
    {
        inputStream = new FileInputStream(fileName);
        loadProperties(inputStream);
    }
    catch (final IOException ex)
    {
        log.warn("can't read file '" + fileName + "'", ex);
        return false;
    }
    finally
    {
        // Sicheres Schließen
        IOUtils.closeQuietly(inputStream);
    }
    return true;
}
```

Wenn man JDK 7 nutzt, so sollte man besser auf das damit neu eingeführte Sprachfeature ARM (Automatic Resource Management) (vgl. Abschnitt 4.7.4) zurückgreifen. Dadurch lässt sich der Sourcecode wie folgt vereinfachen:

```
private boolean readProperties(final String fileName)
{
    try (final InputStream inputStream = new FileInputStream(fileName))
    {
        loadProperties(inputStream);
    }
    catch (final IOException ex)
    {
        log.warn("can't read file '" + fileName + "'", ex);
        return false;
    }
    return true;
}
```

16.3.11 Bad Smell: Resource Leaks durch Exceptions im Konstruktor

Zum Teil werden bereits im Konstruktor Zugriffe auf Streams durchgeführt. Dabei werden Betriebssystemressourcen alloziert. Allerdings können dabei Exceptions ausgelöst werden. Werden diese im Konstruktor behandelt, so ist selten ein sinnvoller

Objektzustand zu erreichen. Betrachten wir das an einem Beispiel.

Der folgende Sourcecode löst `IOExceptions` (genauer die Spezialisierung `FileNotFoundException`) im Konstruktor aus, wenn eine übergebene Datei zum Einlesen nicht gefunden oder aber keine Datei zum Schreiben erzeugt werden kann. In beiden Fällen werden möglicherweise bereits Dateiressourcen belegt, aber eventuell nicht (sofort) freigegeben.

```
public class ZipConverter
{
    private final InputStream  inStream;
    private final OutputStream outStream;

    public ZipConverter(final String inputFileName, final String
outputFileName)
            throws FileNotFoundException
    {
        // beide Konstruktoren können eine FileNotFoundException auslösen
        inStream = new FileInputStream(inputFileName);
        outStream = new FileOutputStream(outputFileName);
    }

    protected final InputStream getInputStream()
    {
        return inStream;
    }

    protected final OutputStream getOutputStream()
    {
        return outStream;
    }
    // …
```

Warum ist das ein Bad Smell?

Durch I/O ausgelöste Exceptions im Konstruktor führen dazu, dass Objekte nicht vollständig konstruiert werden können und damit der Programmablauf gestört wird. Eine sinnvolle Fehlerbehandlung fällt schwer. Bereits belegte Ressourcen können zum Teil nicht korrekt freigegeben werden, da nach dem Abbruch der Objektkonstruktion durch Auslösen der Exception kein Zugriff auf die Ressourcenattribute möglich ist.

Tipps und Refactorings

Betrachten wir Möglichkeiten, die genannten Probleme zu lösen, und beginnen mit der Idee, im Konstruktor die erwarteten Exceptions abzufangen. Die Konstruktion der Streams wird dazu jeweils mit einem eigenen `try-catch`-Block wie folgt umschlossen:

```
private static class ZipConverter
{
    // Indikator, ob bereits initialisiert
    private InputStream inStream = null;
    private OutputStream outStream = null;
```

```
    public ZipConverter(final String inputFileName, final String
outputFileName)
    {
        try
        {
            inStream = new FileInputStream(inputFileName);
        }

        catch (final FileNotFoundException fnfe)
        {
            // Achtung: „silent fail", d. h., Exception wird nicht propagiert
            IOUtils.closeQuietly(inStream);
        }

        try
        {
            outStream = new FileOutputStream(outputFileName);
        }
        catch (final FileNotFoundException fnfe)
        {
            // Achtung: „silent fail", d. h., Exception wird nicht propagiert
            IOUtils.closeQuietly(outStream);
        }
```

Um die Fehlerbehandlung lesbar zu gestalten, nutzen wir Funktionalität aus der Java-Bibliothek Apache Commons IO. In der Klasse `org.apache.commons.io.IOUtils` sind verschiedene überladene Varianten der Methode `closeQuietly()` definiert.

Der Einsatz einer derartigen Fehlerbehandlung im Konstruktor führt beim Auftreten von I/O-Fehlern allerdings nur dazu, dass die eigentliche Objektkonstruktion vermeintlich erfolgreich verläuft. Das neu erzeugte Objekt ist jedoch anschließend in einem unbrauchbaren Zustand. Im Extremfall sind beide Streams geschlossen und jede Aktion darauf löst Probleme durch `IOExceptions` aus. Man könnte solche Initialisierungsprobleme dadurch dokumentieren, dass die korrespondierenden Stream-Attribute auf `null` gesetzt werden. Allerdings erfordert dies wiederum Fehler- und Spezialbehandlungen im aufrufenden Sourcecode: Dort muss nun vor jedem Zugriff zunächst eine `null`-Prüfung der Stream-Attribute erfolgen.

Dadurch verlagert man das Problem jedoch nur: Nun sind die Aufrufer und nicht das Objekt selbst verantwortlich, Aufräumarbeiten durchzuführen. Dazu benötigen andere Klassen jedoch Zugriff auf die privaten, internen Stream-Attribute, d. h., die beiden `get()`-Methoden müssen als `public` definiert werden. Außerdem müssen Klienten nun wissen, dass ein `null`-Wert auf ein nicht erzeugtes Stream-Objekt hindeutet. Eine derartige Realisierung ist wenig objektorientiert. Es ist problematisch, diverse Annahmen nach außen zu publizieren und andere Objekte die eigenen Aufräumarbeiten ausführen zu lassen. Wie lösen wir das Problem?

Wir verwenden das Entwurfsmuster ERZEUGUNGSMETHODE (vgl. Abschnitt 18.1.1). Nur für den Fall, dass sich ein Objekt erzeugen lässt, ohne dass es dabei zu einer `IOException` kommt, geben wir eine gültige Objektreferenz nach außen.

Ansonsten schließen wir die Streams und propagieren die Exception. Um eine saubere Fehlerbehandlung zu ermöglichen, müssen wir die Aktionen aus dem Konstruktor in eine `init()`-Methode verlagern. Somit steht in jedem Fall eine gültige Objektreferenz zur Verfügung, die eine Abfrage der Stream-Attribute erlaubt. Folgendes Listing zeigt die entsprechende Erzeugungsmethode `createConverter()` sowie die anderen notwendigen Anpassungen in der Klasse:

```
private static class ZipConverter
{
    // Indikator, ob bereits initialisiert
    private InputStream inStream = null;
    private OutputStream outStream = null;

    public static final ZipConverter createConverter(final String
inputFileName,
            final String outputFileName) throws FileNotFoundException
    {
        // Definition, damit wir Streams schließen können
        ZipConverter zipConverter = null;
        try
        {
            zipConverter = new ZipConverter();
            zipConverter.init(inputFileName, outputFileName);
            return zipConverter;
        }
        catch (final FileNotFoundException ex)
        {
            // Die Variable zipConverter ist hier ungleich null
            IOUtils.closeQuietly(zipConverter.getInputStream());
            IOUtils.closeQuietly(zipConverter.getOutputStream());
            throw ex;
        }
    }

    private ZipConverter()
    {
    }

    private void init(final String inputFileName, final String outputFileName)
          throws FileNotFoundException
    {
        inStream = new FileInputStream(inputFileName);
        outStream = new FileOutputStream(outputFileName);
    }
    // …
```

Es stellt sich zuletzt noch die Frage, was passiert eigentlich, wenn das `ZipConverter`-Objekt nicht mehr referenziert wird? Wer gibt die Systemressourcen dann wieder frei? Als erste Idee kommt uns die `finalize()`-Methode in den Sinn (vgl. Abschnitt 8.5.6). Die Methode wird vor der Zerstörung eines Objekts durch den Garbage Collector aufgerufen. ***Die Ausführung der `finalize`-Methode wird jedoch nicht garantiert. Ein sicherer Aufräumvorgang ist auf diese Weise nicht möglich.***

Es bietet sich daher folgendes Vorgehen an: Man führt eine Methode `release()` ein, die benötigte Aufräumarbeiten erledigt, hier die geöffneten Streams

schließt. Diese Methode sollte dann im Fehlerfall aus der Erzeugungsmethode sowie zusätzlich aus der `finalize()`-Methode aufgerufen werden. Zudem können Klienten diese Methode nach Abschluss ihrer Arbeiten aufrufen. Dadurch werden Aufräumarbeiten im Normalfall vom Programm selbst vorgenommen. Nur in unerwarteten Situationen steht als letztes Sicherheitsnetz eine Freigabe über `finalize()` bereit. Diese Variante wird auch von Joshua Bloch in »Effective Java« vorgeschlagen.

Jetzt könnte man sich natürlich fragen, worin der Unterschied dieser Lösung zu dem zuvor kritisierten Zugriff über `get()`-Methoden auf die Streams liegt. Die Antwort ist bereits in der Frage enthalten: Bei der ursprünglichen Lösung wird ein Zugriff auf die Streams und die Kenntnis der Implementierungsdetails des Objekts benötigt. Eine `release()`-Methode ist dagegen objektorientiert und versteckt diese Details. Folgendes Listing zeigt eine mögliche Umsetzung:

```
private static class ZipConverter
{
    // Indikator, ob bereits initialisiert
    private InputStream inStream = null;
    private OutputStream outStream = null;

    public static final ZipConverter createConverter(final String
inputFileName,
            final String outputFileName) throws FileNotFoundException
    {
        // Definition, damit wir release() aufrufen können
        ZipConverter zipConverter = null;
        try
        {
            zipConverter = new ZipConverter();
            zipConverter.init(inputFileName, outputFileName);
            return zipConverter;
        }
        catch (final FileNotFoundException ex)
        {
            // Die Variable zipConverter ist hier ungleich null
            zipConverter.release();
            throw ex;
        }
    }

    protected void finalize() throws Throwable
    {
        release();
        super.finalize();
    }

    public void release()
    {
        IOUtils.closeQuietly(getInputStream());
        IOUtils.closeQuietly(getOutputStream());
    }
    // …
```

Fazit

Für Klassen, die Systemressourcen als Attribute halten und bei deren Konstruktionsprozess Exceptions ausgelöst werden können, wird eine Erzeugungsmethode zur sicheren Objektkonstruktion benötigt. Weiterhin ist es empfehlenswert, eine `release()`-Methode zur kontrollierten Freigabe von Systemressourcen bereitzustellen.

Um im Fehlerfall verbleibende Aufräumarbeiten durchzuführen und Resource Leaks zu verhindern, dient als allerletzte Instanz eine `finalize()`-Methode. Wichtig ist hierbei, auch die Implementierung der Basisklasse aufzurufen, damit die Aufräumarbeiten für alle »Objekt-Teile«, d. h. alle Klassenbestandteile aller Basisklassen, durchgeführt werden können.

16.4 Häufige Fallstricke

Dieser Abschnitt beschreibt einige weitere Programmierprobleme, die nicht die Schwere eines Bad Smells aus dem zuvor vorgestellten Katalog haben, die man aber trotzdem kennen sollte, um sie vermeiden zu können.

Fallstrick: Nutzung statischer Attribute statt Membervariablen

Manchmal sieht man den Einsatz von statischen Attributen, die zur Speicherung von Eigenschaften eines bestimmten Objekts verwendet werden (sollen). Statische Attribute sind aber keinem Objekt konkret zugeordnet und sollten daher nur in Ausnahmefällen von Objektmethoden verändert werden, z. B. zur Referenzzählung im Konstruktor.

Fallstrick: Änderung statischer Attribute im Konstruktor / in Methoden

Werden statische Attribute im Konstruktor oder durch Objektmethoden verändert, so liegt meistens ein Problem vor. Wie bereits angedeutet, gibt es Situationen und Anwendungsfälle, in denen dies Sinn macht. Häufig geschieht die Veränderung aber lediglich aus Unachtsamkeit. Eine Veränderung bewirkt aber das Umsetzen eines Werts eines statischen Attributs. Diese Wertänderung ist für alle Objekte dieser Klasse sichtbar.

Fallstrick: Missachtung der Initialisierungsreihenfolge statischer Attribute

Es ist wichtig, die Initialisierungsreihenfolge statischer Attribute zu kennen und zu beachten. Statische Attribute werden in der Reihenfolge ihrer Deklaration – d. h. entsprechend ihrem Auftreten im Sourcecode – initialisiert. Dies ist immer zu

beachten, um Probleme durch eine falsche Ausführungsreihenfolge zu verhindern. Das gilt insbesondere beim Aufruf statischer Methoden, die auf diese statischen Attribute zugreifen. Wir betrachten dies anhand eines Beispiels. Hier wird die `Logger`-Referenz `log` fälschlicherweise nicht als erstes statisches Attribut definiert:

```
public final class LogInitExceptionExample
{
  private static final long test_fail = init();
  private static final Logger log =
Logger.getLogger("LogInitExceptionExample");

  private static long init()
  {
    log.info("init()");
    return 4712;
  }

  public static void main(final String[] args)
  {
    System.out.println("LogInitExceptionExample");
  }
}
```

Listing 16.3 *Ausführbar als* **'LOGINITEXCEPTIONEXAMPLE'**

Führen wir das obige Programm aus, so kommt es zu einem Initialisierungsproblem in Form einer zunächst unerklärlichen `NullPointerException`: Das statische Attribut `log` wurde noch nicht zugewiesen und besitzt noch den Defaultwert `null`, wenn in der Methode `init()` darauf zugegriffen wird. Bei der Definition statischer Attribute mit primitiven Typen kommt es nicht zu Exceptions, sondern stattdessen sind einige Werte je nach Typ einfach unerwartet mit dem Wert 0 bzw. `false` initialisiert.

Die Missachtung der Initialisierungsreihenfolge statischer Attribute kann somit zu merkwürdigen und schwierig auffindbaren Fehlern führen.

Fallstrick: Statische Methoden über Objektreferenzen aufrufen

An den beiden statischen Methoden `yield()` und `sleep(long)` der Klasse `Thread` kann man sehr schön verdeutlichen, warum es irreführend ist, wenn man statische Methoden so ausführt, als ob sie Objektmethoden wären. Nehmen wir an, der `main`-Thread wäre aktiv und ein Thread `threadX` lauffähig. Es werden folgende Aufrufe ausgeführt:

```
threadX.sleep(2000);
threadX.yield();
```

Auf den ersten Blick meint man, der Thread `threadX` würde zunächst zwei Sekunden pausieren und danach kurz die Kontrolle abgeben. Tatsächlich wirken sich beide Methoden jedoch auf den zum Zeitpunkt des Methodenaufrufs aktiven Thread aus. Dies ist zunächst der laufende `main`-Thread. Auf welchen der beiden Threads sich der Aufruf von `yield()` auswirkt, ist zufällig und nicht vorhersagbar.

Fallstrick: Utility-Klasse mit öffentlichem Konstruktor

Eine Utility-Klasse mit öffentlichem Konstruktor suggeriert, dass es sich dabei um eine normale Klasse handeln würde, von der Instanzen erzeugt werden können. Dies sollte bei einer Utility-Klasse jedoch nicht der Fall sein. Stattdessen sollten hier nur einige (statische) Hilfsmethoden bereitgestellt werden. Der Konstruktor sollte daher immer als `private` definiert werden. Weiterhin sollte eine Ableitung vermieden werden, indem die Klassendefinition mit `final` durchgeführt wird.

```
public final class MyUtilityClass
{
    private MyUtilityClass()
    {
        // Verhindert Konstruktoraufrufe durch andere Klassen
    }

    public static int someUtilityMethod(final int value1, final int value2)
    {
        // …
    }
}
```

Fallstrick: Einsatz von Vererbung und statischen Methoden

Wird Vererbung bei Utility-Klassen eingesetzt, die nur statische Methoden bereitstellen, so ist kein Polymorphismus möglich, sondern es erfolgt nur ein Überdecken von Methoden. Dies deutet Abbildung 16-7 für eine `print()`-Methode an, die sowohl in der Basisklasse `Base` als auch in der Subklasse `Sub` definiert ist. Bei einem Aufruf dieser Methode erfolgt *kein* dynamisches Binden, sondern es wird die Methode basierend auf dem Kompiliertyp ausgewählt. Insbesondere ist das problematisch, wenn die Utility-Klassen Konstruktoren anbieten. Folgende Zeilen verdeutlichen das und zeigen Probleme durch den vorherigen Fallstrick nochmals auf.

```
final Base base = new Base();    // => Base::print()
final Base baseSub = new Sub(); // => Base::print()
final Sub sub = new Sub();       // => Sub::print()
```

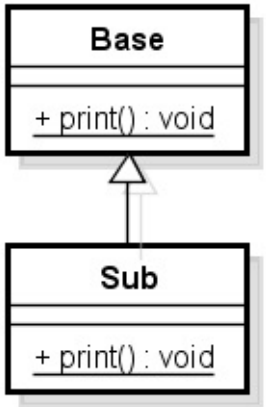


Abbildung 16-7 *Vererbung, aber nur statische Methoden*

Fallstrick: Chaotische Konstruktor-/Methoden-Aufruffolgen

Werden Methoden oder Konstruktoren mit unterschiedlicher Anzahl an Parametern definiert und rufen sich diese gegenseitig auf, so führt dies schnell in ein Chaos. Dies gilt insbesondere dann, wenn die Methoden untereinander nicht hierarchisch geordnet sind und die Aufrufe unstrukturiert erfolgen. Die Grafik auf der linken Seite in Abbildung 16-8 verdeutlicht dies. Jede Methode ist hier als Klammerpaar dargestellt. Die Parameter sind durch Punkte symbolisiert. In der Praxis ist eine derartige chaotische Aufrufhierarchie leider nicht ungewöhnlich. Dadurch wird jedoch eine korrekte Initialisierung ungewiss. Die rechte Seite der Grafik zeigt eine streng hierarchische Aufrufreihenfolge. Der Verlauf ist leicht nachzuvollziehen, und die Wahrscheinlichkeit für Fehler beim Aufruf von Methoden mit mehreren Parametern sinkt.

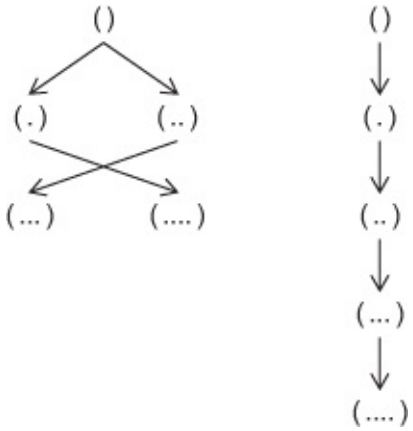


Abbildung 16-8 *Konstruktordelegation*

Fallstrick: Missverständliches API durch Überladen

Wenn man nicht genau hinschaut, sieht es im folgenden Beispiel zunächst so aus, als ob ein Überschreiben der Methode `wait()` der Klasse `Object` stattfindet. Tatsächlich ist dies nicht möglich, da die Methode dort `final` definiert ist. Es erfolgt hier vielmehr ein Überladen:

```
public void wait(final int millis)
{
    try
    {
        Thread.sleep(millis);
    }
    catch (final InterruptedException ex)
    {
        Thread.currentThread().interrupt();
    }
}
```

Eine solche Definition suggeriert, dass eine ähnliche oder erweiterte Funktionalität wie bei der originalen Methode geboten werden soll. Dies ist hier jedoch nur begrenzt der Fall! Im Gegensatz zur Methode `wait()` der Klasse `Object`, die zur Steuerung von Threads und im Speziellen zur Verarbeitung von Benachrichtigungen

über `notify()` bzw. `notifyAll()` dient, wartet diese Methode `wait(int)` einfach nur eine angegebene Zeitdauer. Dabei behält der gerade aktive Thread alle akquirierten Locks. Im Gegensatz dazu wartet die Methode `Object.wait(long)` auch eine gewisse Zeitdauer auf das Eintreffen einer Nachricht, allerdings wird hierbei der Lock auf das Objekt wieder freigegeben (vgl. Abschnitt 7.2.2).

Fallstrick: Mehrfachverkettung der ».«-Notation

Die Hintereinanderschaltung verschiedener Methodenaufrufe mit der ».«-Notation birgt die Gefahr nicht initialisierter Referenzen. Methodenaufrufe lösen eine `NullPointerException` aus, wenn eine der Referenzen `null` ist. Außerdem widerspricht die mehrfache ».«-Notation dem in Abschnitt 3.5.2 vorgestellten Law Of Demeter und wird schnell unübersichtlich. Das folgende Beispiel zeigt eindrucksvoll, dass schon wenige Verkettungen problematisch sind, wobei wir den Fokus auf die Lesbarkeit und Verständlichkeit legen, da im Beispiel die Referenzvariablen korrekt initialisiert sind:

```
public static long cslcMinutesToGo(final Departure departure)
{
    return departure.getDepartureTime().longValue()  -
           Calendar.getInstance().getTimeInMillis() / 1000 / 60 /*[min]*/;
}
```

Abhilfe schafft hier das Eclipse-Refactoring »EXTRACT LOCAL VARIABLE« aus dem Menü REFACTOR mit dem Tastaturkürzel ALT+SHIFT+L. Damit kann man Ausdrücke in lokale Variablen ausgliedern und den ursprünglichen Ausdruck vereinfachen, wie dies im folgenden Listing gezeigt ist.

```
public static long cslcMinutesToGo(final Departure departure)
{
    final long depatureTimeInMillis =
departure.getDepartureTime().longValue();
    final long nowInMillis = Calendar.getInstance().getTimeInMillis();

    return (depatureTimeInMillis - nowInMillis) / 1000 / 60 /*[min]*/;
}
```

Durch die deutlich übersichtlichere und verständliche Umsetzung entdeckt man auch den im obigen Ausdruck versteckten Berechnungsfehler: Dort wurde nur der zweite Wert in Minuten umgewandelt, der erste Wert war noch in Millisekunden. Zur Korrektur muss lediglich die Klammerung um die Subtraktion erfolgen.

Soll der Sourceocde die Umwandlung von Millisekunden in Minuten noch klarer kommunizieren, kann man dazu die Aufzählung `TimeUnit` wie folgt nutzen:

```
public static long cslcMinutesToGo(final Departure departure)
{
    final long depatureTimeInMillis =
departure.getDepartureTime().longValue();
    final long nowInMillis = Calendar.getInstance().getTimeInMillis();

    return TimeUnit.MILLISECONDS.toMinutes(depatureTimeInMillis -
nowInMillis);
```

```
}
```

Fallstrick: Unnötige Methoden und Komplexität

Leider sieht man immer wieder Sourcecode, der keinen Mehrwert hat, sondern eher der Übersicht sowie der Klarheit schadet. Unglaublich, aber wahr, ist das folgende Beispiel der Methode `getString(Object)`:

```
public static final String getString(final Object argument)
{
    return argument.toString();
}
```

Diese Methode ist vollkommen überflüssig, da sie lediglich zusätzliche Komplexität einführt. Ein direkter Aufruf von `toString()` würde den Sourcecode klarer machen und das intuitive Verständnis erleichtern. Als Faustregel gilt: ***Eine Methode sollte nur dann erstellt werden, wenn sie neue Funktionalität hinzufügt oder für eine bessere Struktur oder Lesbarkeit des bestehenden Sourcecodes sorgt***.

Abgesehen davon existiert ein noch viel schwerwiegenderes Problem: Jeder Aufrufer besitzt nun eine Abhängigkeit von der Utility-Klasse, in der die Methode `getString(Object)` definiert ist. Verweist diese Utility-Klasse auf weitere Klassen, so bekommt man schnell eine Menge ungewünschter und unerwarteter Abhängigkeiten, wie dies Abbildung 16-9 andeutet.

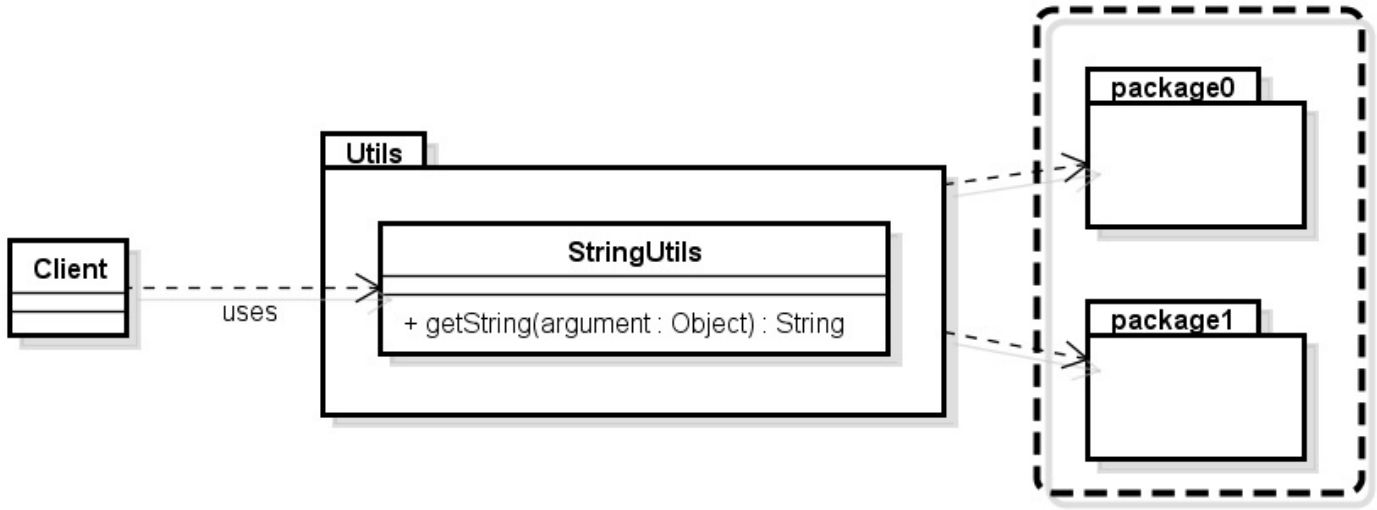


Abbildung 16-9 *Unerwartete Abhängigkeiten durch* `getString(Object)`

Fallstrick: Objektvergleich durch Einsatz von `toString()` und `equals()`

Die Methode `equals(Object)` dient bekanntermaßen dazu, einen semantischen Vergleich von Objekten zu ermöglichen (vgl. Abschnitt 4.1.2). Manchmal wird für ein Objekt aber die `equals(Object)`-Methode nicht überschrieben. Um dennoch inhaltliche Vergleiche ausführen zu können, sieht man teilweise zunächst eine Umwandlung in einen String durch einen Aufruf von `toString()`. Die so

erzeugten Stringrepräsentationen werden dann mit deren `equals(Object)`-Methode verglichen, wie dies im folgenden Beispiel geschieht: Dort wird von einem Objekt `databaseJob` über den Aufruf von `toString()` eine textuelle Repräsentation erzeugt und diese mit einer Variablen `jobId` verglichen:

```
if (databaseJob.toString().equals(jobId))
```

Dieser Vergleich verlässt sich auf das Implementierungsdetail der `toString()`-Methode dieses Objekts, dass in diesem Beispiel nur den Wert der `jobId` des Objekts zurückgeliefert wird. Eine derartige Annahme ist gewagt und der resultierende Vergleich ist sehr fragil. Wird die Ausgabe der `toString()`-Methode im Nachhinein verändert, z. B. um weitere Ausgaben von Attributwerten ergänzt oder einfach besser lesbar gestaltet, so schlägt der gezeigte Vergleich immer fehl. Ein solcher Fehler ist schwer zu finden, da er sich nur indirekt durch Merkwürdigkeiten im Programmablauf bemerkbar macht.

Liefert die selbst definierte Methode `toString()` gar den Wert `null` zurück, so ergibt sich direkt der nächste Fallstrick.

Fallstrick: Rückgabe von `null` in `toString()`-Methoden

Eine `toString()`-Methode soll eine textuelle Repräsentation eines Objekts erzeugen. Sie sollte allerdings niemals den Wert `null` zurückgeben, denn dadurch provoziert sie `NullPointerException`s im aufrufenden Sourcecode.

Fallstrick: Zugriff ohne Bereichsprüfung

Zum Teil sieht man Zugriffe auf Arrays, Strings, Listen, Iteratoren und Enumerations ohne vorherige Prüfung, ob dieser Zugriff überhaupt möglich und erlaubt ist (Index im Bereich, Daten vorhanden usw.). Ich möchte hier nicht zum übervorsichtigen Prüfen vor jedem Zugriff aufrufen, sondern lediglich motivieren, die Validität von Zugriffen abzusichern. Häufig kann dies an zentraler Stelle einmal geschehen. Nach einer Prüfung müssen aufgerufene Methoden diese nicht wiederholen, sofern sich die Zusammensetzung der Daten nicht ändert. Ansonsten kann auch eine erneute Prüfung notwendig sein. Im Einzelnen sollten zumindest einmalig folgende Tests durchgeführt werden:

- Bei Array-Zugriffen sollte die Indexposition mit der Array-Größe verglichen werden. Gleiches gilt für Listen und einen indizierten Zugriff mit `get(int)`.
- Indizierte Zugriffe per `charAt(int)` in Strings sollten die Stringlänge prüfen.
- Beim Einsatz eines Iterator- oder Enumeration-Zugriffs ist es erforderlich, durch Aufruf der Methode `hasNext()` bzw. `hasMoreElements()` festzustellen, ob weitere Elemente existieren.

Fallstrick: `default` mitten in den `case`-Anweisungen

versteckt und/oder unerwartetes Fallthrough

Ein weiterer Fallstrick ist die Angabe der `default`-Aktion zwischen beliebigen Alternativen einer `switch`-Anweisung. Ein intuitives Verständnis des Ablaufs wird dadurch erschwert. Es ist guter Programmierstil, die Standardaktion, die nicht durch `case` abgedeckt wird, am Ende der `case`-Alternativen aufzuführen.

```
switch (value)
{
   case 0:
   case 2:
   case 4:
       System.out.println(value + " accepted even value [0 - 4]");
       // Fehlendes break => Fallthrough

   default:
       System.out.println(value + " performing default action");
       break;

   case 1:
   case 3:
       System.out.println(value + " is odd");
}
```

Das Listing zeigt aber eine weitere Unschönheit: Standardmäßig erfolgt bei Angaben in `switch` ein Fallthrough, d. h., für die Werte 0, 2 und 4 wird nicht nur ihr `case`-Zweig ausgeführt, sondern auch noch der nachfolgende Defaultzweig, weil kein `break` aufgeführt ist.

Fallstrick: Berechnungen in `case`-Anweisungen

Fallunterscheidungen mit `case` sollte man möglichst lesbar und verständlich halten. Obwohl es möglich ist, Berechnungen innerhalb von `case` auszuführen, sollte man dies vermeiden, da darunter die Nachvollziehbarkeit und Wartbarkeit leiden. Folgendes Listing zeigt dies eindrucksvoll:

```
public static void main(String[] args)
{
    final int byteValue = 7;

    int value = (int) (10* Math.random());
    switch (value)
    {
        case 7:
        case 19 - 11:            // 8
        case 17 - 8:             // 9
        case 2 * byteValue - 4:  // 10
            System.out.println(value + " accepted [7 - 10]");
            break;
        default:
            System.out.println(value + " is not in range 7 - 10");
    }
}
```

Listing 16.4 *Ausführbar als* **'BADCASEEXAMPLE'**

Fallstrick: Einsatz komplizierter boolescher Bedingungen

Der Einsatz boolescher Bedingungen dient der Auswertung verschiedener Zustände. Nicht immer muss eine solche Bedingung aber so kompliziert geschrieben werden, wie dies häufig ohne viel Nachdenken getan wird. Betrachten wir folgendes Beispiel, um einen Wechsel im Betriebszustand festzustellen. Dazu ist das boolesche Attribut `lastWorkingState` zur Speicherung des alten Zustands und die boolesche Variable `newWorkingState` für den aktuellen Zustand vorgesehen. Die Zustandswechsel werden jeweils durch explizite Vergleiche festgestellt:

```
if ((lastWorkingState == false && newWorkingState == true) ||
    (lastWorkingState == true  && newWorkingState == false))
```

Tatsächlich lässt sich dies mühelos wie folgt klarer umschreiben:

```
if (lastWorkingState != newWorkingState)
```

Fallstrick: Einsatz doppelter Verneinung

Der Einsatz von booleschen Variablen kann zur Klarheit von Programmen enorm beitragen. Werden jedoch Bedingungen umständlich oder als Verneinung formuliert, so stört dies die Lesbarkeit. Ein Beispiel, das von mir exakt so im Sourcecode eines realen Projekts gefunden wurdet, ist die Variable `dontCleanUpDatabase`:

```
boolean dontCleanUpDatabase = false;
```

Für den Wert von `false` wird die Datenbank aufgeräumt. Für den Wert `true` geschieht dies nicht und es bleiben Datensätze erhalten. Diese doppelte Verneinung erschwert das Verständnis und ist nicht intuitiv. Je nach Einsatzzweck wären folgende Variablennamen besser verständlich gewesen:

```
boolean cleanUpDatabase = true;
boolean keepDatabaseEntries = false;
```

Fallstrick: Intensive Nutzung von Sprungmarken, `break` und `continue`

Das intensive Nutzen von Sprungmarken, `break` und `continue` erinnert an die GOTO-Programmierung vergangener Tage. In den allermeisten Fällen lassen sich Abbruchbedingungen klarer durch boolesche Variablen formulieren.

Fallstrick: Kommentierte Klammern

Manchmal werden schließende Klammern kommentiert, um einen Hinweis auf die Anweisung der zugehörigen öffnenden Klammer zu geben. Das zeigt eigentlich nur, dass die Struktur des Programms unübersichtlich ist und die Blöcke zu groß gewählt sind. Betrachten wir ein Beispiel, das von der absoluten Länge her in Ordnung ist, aber durch seine kommentierten Klammern bereits unübersichtlich wird:

```
public void writeXML(final String filename)
{
    try
    {
        final Element root = toDOMElement();
        final Document doc = new Document(root);
        try
        {
            final XMLOutputter xmlout = createXMLOutputter();
            xmlout.output(doc, new FileOutputStream(filename));
        } // try
        catch (final IOException e)
        {
            log.error("Can't write " + filename, e);
        } // catch
    } // try
    catch (final NullPointerException npe)
    {
        log.error("Can't create " + filename, npe);
    } // catch
} // writeXML
```

Durch die massive, unsinnige Kommentierung der Klammern kann man dem Programmfluss in diesem Beispiel eher schlecht folgen. Tatsächlich sieht man erst auf den zweiten Blick, dass hier keine Fehlerbehandlung von `IOExceptions` erfolgt. Das ist Thema von BAD SMELL: I/O OHNE `FINALLY` in Abschnitt 16.3.10.

Fallstrick: Missverständliche Methodennamen

Bedenken Sie beim Entwurf der Schnittstellen Ihrer Klassen, dass diese möglichst überraschungsfrei sind und somit dem »Principle of Least Astonishment« (POLA) – auch »Principle of Least Surprise« (POLS) genannt – folgen sollten. Schauen wir auf ein Gegenbeispiel:

```
private Client client = null;

public void addSubscriber(final Client client)
{
    Objects.requireNonNull(client, "subscriber must not be null");

    this.client = client;
}
```

Aufgrund des Methodennamens würde man als Aufrufer vermuten, dass man sich zu einer Liste von Beobachtern hinzufügt. Stattdessen ersetzt man den gerade registrierten Subscriber (wozu hier auch noch das etwas missverständlich benannte Attribut `client` genutzt wird). Würde man die Methode `setSubscriber()` oder `replaceSubscriber()` nennen, so würde besser nach außen kommuniziert, was tatsächlich passiert.

Bitte verstehen Sie mich nicht falsch: Es ist nicht generell wünschenswert, alle Implementierungsdetails nach außen zu tragen, sondern hier geht es darum, High-Level-Verhalten zu kommunizieren, das möglicherweise Einfluss auf andere Klassen oder Subsysteme besitzt. Höchstwahrscheinlich sollte die obige Methode zudem wie

folgt korrigiert werden, um eine Liste von Subscribern verwalten zu können:

```
private final List<Client> subscribers = new CopyOnWriteArrayList<>();

public void addSubscribers(final Client client)
{
    Objects.requireNonNull(client, "subscriber must not be null");
    this.subscribers.add(client);
}

public void removeSubscribers(final Client client)
{
    Objects.requireNonNull(client, "subscriber must not be null");
    this.subscribers.remove(client);
}
```

Durch Einsatz der Klasse `CopyOnWriteArrayList<E>` kann man auf eine eigene Realisierung von Synchronisation verzichten und vermeidet zudem Probleme in Bezug auf Multithreading bei gleichzeitigen Regristrierungswünschen mehrerer Subscriber.

16.5 Weiterführende Literatur

Zwei empfehlenswerte Bücher, die sich sowohl mit Bad Smells, aber vor allem auch mit dem Schreiben von verständlichem Sourcecode beschäftigen, sind:

- »**Code Craft: The Practice of Writing Excellent Code** « von Pete Goodlife [33]

  Pete Goodlife hat ein fantastisches Buch zu gutem Softwareentwurf geschrieben. Es werden viele Tipps gegeben, um sauberen, robusten Sourcecode zu schreiben. Ich empfehle es ausdrücklich. Ein Must-have!

- »**Clean Code: A Handbook of Agile Software Craftsmanship**« von Robert C. Martin [62]

  Ein weiteres Buch von Robert C. Martin, in dem man verschiedenste hilfreiche Informationen zu sauberem Sourcecode und gutem Programmierstil findet. Es ist vielleicht manchmal ein wenig dogmatisch.

Bad Smells und Merkwürdigkeiten im Java-API

Neben den vorgestellten Bad Smells ist es sinnvoll, einige Probleme in den Java-APIs zu kennen. Diverse Beispiele finden Sie unter anderem in folgenden Büchern:

- »**Java Puzzlers**« von Joshua Bloch und Neil Gafter [10]

  Joshua Bloch und Neil Gafter sind vielen Leuten bekannt. In diesem Buch veröffentlichen sie verschiedene Rätsel rund um die Sprache Java. Einige davon sind allerdings wenig praxisrelevant, andere dafür umso spannender.

- »**Java Pitfalls: Time-Saving Solutions and Workarounds to Improve**

Programs« von Michael C. Daconta, Eric Monk, J. Paul Keller und Keith Bohnenberger [19]

In diesem Buch werden verschiedene Fallstricke aufgezeigt. Dabei wird ein breites Spektrum von Design bis hin zu diversen API-Unschönheiten im JDK behandelt. Es ist eine Sammlung von Lösungen für Probleme, die immer mal wieder beim täglichen Entwickeln auftauchen.

17 Refactorings

Bei der Entwicklung und Pflege von Software ist das Gesetz der Entropie (der zunehmenden Unordnung) aus der Physik zu beobachten. Man kennt Ähnliches aber auch aus dem täglichen Leben. In einer Wohnung nimmt die Unordnung ständig zu, wenn man nicht von Zeit zu Zeit aufräumt. Auf Software übertragen gilt, dass sich im Laufe der Zeit durch Änderungen die Struktur und Lesbarkeit derart verschlechtern kann, dass sich Fehlerbehebungen oder Erweiterungen immer schwieriger in den bestehenden Sourcecode integrieren lassen. Das Thema Wartbarkeit ist beim professionellen Programmieren aber sehr wichtig, denn oftmals hat man eine große Sourcecode-Basis zu pflegen und zu erweitern. Nur selten kommt man in den Genuss, ein System vollständig neu entwerfen zu dürfen. Und auch in diesem Fall wandelt sich die Situation schnell, wenn man nicht fortlaufend Qualitätssicherung betreibt und kontinuierlich Überarbeitungen durchführt. Wurde dies versäumt, so hilft – wie im realen Leben – nur eine gründliche Aufräumaktion oder gar ein Umzug. Letzteres entspräche auf die Software übertragen einer kompletten Neuimplementierung. Eine solche ist aufgrund des hohen Risikos, zu scheitern, jedoch meistens keine Alternative. Es verbleibt die Aufräumaktion, die Umbaumaßnahmen im Sourcecode, sogenannten ***Refactorings***, entspricht. Diese sollen problembehafteten Sourcecode derart verändern, dass dieser besser verständlich, lesbar und auch leichter wartbar wird. Martin Fowler definiert den Begriff Refactoring folgendermaßen: »Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify without changing the observable behavior of that software component« [27].

Dieses Kapitel stellt verschiedene in der Praxis erprobte Überarbeitungen von Sourcecode vor. In Abschnitt 17.1 betrachten wir ein einführendes Beispiel. Beim Überarbeiten des Sourcecodes ist ein Standardvorgehen hilfreich, das ich in Abschnitt 17.2 beschreibe. Dieses sorgt zunächst mit einigen Vorbereitungsmaßnahmen und der Beachtung von Coding Conventions (vgl. Kapitel 19) dafür, dass die weitere Bearbeitung und gewünschte Transformation leichter fallen. Je nach erkannter Schwachstelle kann man gemäß den Schritt-für-Schritt-Anleitungen aus Abschnitt 17.4 vorgehen. ***Nicht bei allen davon handelt es sich im strengen Sinne um Refactorings, da diese mitunter das nach außen sichtbare Verhalten leicht ändern und somit die obige Forderung nicht strikt einhalten***. Allerdings gibt es auch Meinungen, die die Aussage auf relevantes Verhalten lockern. Unabhängig von der genauen Auslegung spreche ich der Einfachheit halber immer von Refactorings. Natürlich müssen derartige Änderungen mit Vorsicht und Sorgfalt ausgeführt werden, um dadurch keine neuen Fehler einzuführen oder das Programmverhalten grundlegend zu ändern. Dazu ist es sinnvoll, Veränderungen in kleinen, überschaubaren Schritten durchzuführen und für eine Absicherung durch Unit Tests zu sorgen (vgl. Kapitel 20).

17.1 Refactorings am Beispiel

Anhand eines von mir so in der Praxis gefundenen Beispiels möchte ich darstellen, welche Möglichkeiten zur Vereinfachung sich mit Refactorings ergeben können. Folgende statische Methode `isNumber(String)` der Klasse `NumberUtilsV1` prüft auf naive Weise, ob ein übergebener String eine ganze Zahl darstellt. Dazu wird vor allem die Methode `Character.isDigit(char)` verwendet:

```
public static boolean isNumber(final String value)
{
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!(Character.isDigit(value.charAt(i))))
            {
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    return true;
}
```

In der Methode finden zwei `if`-Abfragen statt. Zunächst wird das erste Zeichen geprüft. Nur wenn dieses eine Ziffer ist, wird anschließend in einer `for`-Schleife beginnend ab Index 1 der `isDigit(char)`-Vergleich wiederholt, bis entweder das Ende des Strings erreicht oder das betrachtete Zeichen keine Ziffer ist. Durch etwas Sourcecode-Analyse erkennen wir, dass die Methode `isNumber(String)` folgende Probleme enthält:

1. **Unerwartete Exception bei leerer Eingabe** – Wenn die Eingabe nicht mindestens ein Zeichen enthält, wird eine `java.lang.StringIndexOutOfBoundsException` ausgelöst. Die Ursache ist der indizierte Zugriff per `charAt(0)`, ohne zuvor die Länge des Parameters `value` zu überprüfen. Ein solches Verhalten ist zu vermeiden. Dies gilt im Speziellen, wenn die Werte aus einer Benutzereingabe stammen.
2. **Fehleranfällig für `null`** – Bei Übergabe eines `null`-Werts löst die Methode erst bei der Verarbeitung und dem Aufruf von `charAt(0)` eine `NullPointerException` aus. Öffentliche Methoden sollten gemäß Design by Contract (vgl. Abschnitt 3.1.5) ihre Vorbedingungen sicherstellen und dazu vor der eigentlichen Verarbeitung die Eingabeparameter auf Gültigkeit prüfen. Damit wird eine Störung der Abarbeitung durch fehlerhafte Übergabewerte vermieden.
3. **Zu kompliziert** – Die initiale `if`-Bedingung und das `if` innerhalb der `for`-Schleife sorgen für eine weitere Schachtelungsebene und sind konträr zueinander formuliert. Bei flüchtigem Hinsehen könnte man die initiale Prüfung übersehen und sich dann fragen, wieso die Schleife mit 1 und nicht bei 0 startet.

4. **Missverständlicher Name bzw. unklares Verhalten** – Der Methodenname `isNumber(String)` suggeriert die Möglichkeit der Verarbeitung beliebiger Zahlen. Momentan werden aber weder Zahlen mit Vorzeichen noch solche mit Nachkommastellen unterstützt.
5. **Viele `return`-Anweisungen** – Für eine derart kurze Methode existieren mit drei `return`-Anweisungen schon recht viele Ausgänge.

Um die Funktionalität zu prüfen und mögliche Fehler aufzudecken, entwickeln wir einige elementare Unit Tests. Dabei können uns die eben ermittelten obigen Kritikpunkte helfen, mögliche Testfälle zu identifizieren und diese Schwachstellen für die Zukunft auszuschließen. Das ist nützlich, weil wir auch einige kleinere Änderungen und Erweiterungen realisieren wollen. Damit beginnen wir jedoch erst, nachdem wir ein Sicherheitsnetz aus diversen Testfällen erstellt haben. Wir folgen hier der in Abschnitt 20.2.1 beschriebenen testgetriebenen Entwicklung.

Problem 1: Unerwartete Exception bei leerer Eingabe

Bevor wir uns dem eigentlichen Problem bei leeren Eingaben zuwenden, erstellen wir einige Funktionstests: Wir prüfen die Verarbeitung einer gültigen Eingabe, etwa `"12345"`, und eines fehlerhaften Werts, etwa `"ABC"`. Im ersten Fall erwarten wir `true` und im zweiten `false` als Ergebnis. Dies lässt sich mithilfe der Methoden `assertTrue(boolean)` und `assertFalse(boolean)` wie folgt ausdrücken:

```
@Test
public void testValidNumberInput()
{
    assertTrue(NumberUtilsV1.isNumber("12345"));
}

@Test
public void testInvalidInput()
{
    assertFalse(NumberUtilsV1.isNumber("ABC"));
}
```

Randfälle prüfen: Eingabe der Länge 0 und 1

Weil die obigen zwei Tests bestanden werden, prüfen wir nun auch zwei Randfälle, nämlich eine leere Eingabe und eine Eingabe mit nur einer Ziffer. Unsere Voranalyse hat schon aufgedeckt, dass es dabei Probleme gibt. Wir schreiben nun passende Unit Tests, die das bestätigen. In der Zukunft soll die Methode für eine leere Eingabe keine `StringIndexOutOfBoundsException` auslösen, sondern den Wert `false` für die Aussage »keine Zahl« liefern. Eine einzelne Ziffer gilt als Zahl. Das Ganze lässt sich wie folgt absichern:

```
@Test
public void testNumberInputLength0()
{
    assertFalse(NumberUtilsV1.isNumber(""));
```

```
}

@Test
public void testNumberInputLength1()
{
    assertTrue( NumberUtilsV1.isNumber( "1" ));
}
```

Wie erwartet, schlägt der Test `testNumberInputLength0()` mit leerer Eingabe fehl.

Korrektur

Bevor wir weitere Tests ergänzen, korrigieren wir die Funktionalität. Wir fügen folgende Sicherheitsprüfung am Anfang der Methode hinzu:

```
if ( value.isEmpty())
{
    return false;
}
```

Damit werden dann die Unit Tests wieder bestanden. Allerdings ist unser Nutzcode durch die Abfrage und das weitere `return` etwas komplizierter geworden, aber wir leben erstmal damit. Darum kümmern wir uns, sobald wir das Sicherheitsnetz aus Unit Tests ausgebaut haben. Dadurch können dann Änderungen an der inneren Struktur mit mehr Vertrauen in die Korrektheit ausgeführt werden.

Problem 2: Fehleranfällig für `null`

In der initialen Analyse haben wir erkannt, dass die Eingabe von `null` unbehandelt ist und eine `NullPointerException` auslöst – im Originalcode allerdings nicht durch eine Parameterprüfung, die auch eine `IllegalArgumentException` nutzen könnte, sondern erst beim Zugriff `value.charAt(0)` durch die JVM beim Dereferenzieren. Wünschenswert ist eine explizite Behandlung und ein Hinweistext, der mitteilt, welcher Parameter `null` war. Der Test ist etwas komplizierter zu formulieren:

```
@Test
public void testNullInput()
{
    try
    {
        NumberUtilsV1.isNumber( null);
        fail(); // es wird als Reaktion eine Exception erwartet
    }
    catch ( final Exception ex)
    {
        // NullPointerException wird als Reaktion erwartet
        assertTrue( ex instanceof NullPointerException);
        // Teste die Existenz eines Textes => keine Standardexception
        assertFalse( StringUtils.isEmpty( ex.getMessage()));
    }
}
```

Bei der Abarbeitung erwarten wir, dass eine Exception auftritt. Normalerweise könnte man dazu die Notation `@Test(expected=NullPointerException.class)` nutzen. Das geht hier nicht, da wir zudem prüfen wollen, ob ein Hinweistext in der Exception hinterlegt ist. Das erwarten wir, wenn Exceptions aus dem eigenen Programm heraus ausgelöst werden. Nur wenn dort passende Hinweise und Kontextinformationen bereitgestellt werden, ist eine sinnvolle Fehlerbehandlung möglich. Im Gegensatz dazu sind die Texte leer, wenn Exceptions durch die JVM automatisch beim Dereferenzieren einer `null`-Referenz ausgelöst werden.

Wir verwenden die Methode `fail()` für den Fall, dass fälschlicherweise keine Exception geworfen und `isNumber(null)` normal terminieren würde. Beim Auftreten einer Exception prüfen wir auf die erwartete `NullPointerException`, sodass der Unit Test als fehlgeschlagen interpretiert wird, falls eine andere Exception auftritt. Abschließend testet die Methode `isEmpty(String)` der Klasse `StringUtils` aus Apache Commons Lang[1], ob ein Hinweistext hinterlegt ist. Und, wie erwartet, schlägt auch dieser Test zunächst fehl.

Korrektur

Wir haben in Kapitel 6 die Bibliothek Google Guava[2] und deren Utility-Klasse `Preconditions` zum Sicherstellen von Vorbedingungen kennengelernt. Wir fügen eine Sicherheitsprüfung am Anfang der Methode hinzu:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    if (Character.isDigit(value.charAt(0)))
    {
        for (int i = 1, n = value.length(); i < n; i++)
        {
            if (!(Character.isDigit(value.charAt(i))))
            {
                return false;
            }
        }
    }
    else
    {
        return false;
    }
    return true;
}
```

Damit werden dann die Unit Tests wieder bestanden. Allerdings ist unser Nutzcode durch die Abfrage nochmals etwas komplizierter geworden. Da wir mittlerweile ein recht gutes Sicherheitsnetz erstellt haben, ist es nun an der Zeit, den Nutzcode –

sofern möglich – ein wenig aufzuräumen.

Problem 3: Zu kompliziert

Wenn wir uns die Methode und insbesondere die Schleife und dortigen `if`-Abfragen anschauen, ist das schon einigermaßen kompliziert. Bei genauem Hinsehen fällt auf, dass man Vereinfachungen erreichen kann: Die abgefragten Bedingungen sind semantisch gleich, aber negiert. Somit lassen sich die beiden `if`-Abfragen zusammenfassen, indem die `if`-Startbedingung in die `if`-Abfrage der `for`-Schleife integriert wird. Als Folge kann die Schleife bei 0 gestartet werden.

Auch die Formulierung der `for`-Schleife ist uns zu kompliziert. Zur Vereinfachung entfernen wir die Zuweisung `n = value.length()` aus dem Initialisierungsteil der `for`-Schleife, die zur Optimierung des Vergleichs `i < n` diente.[3] Dies schreiben wir kürzer als `i < value.length()`. Das erhöht die Lesbarkeit und Verständlichkeit. Insgesamt ergibt sich folgende Korrektur:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    for (int i = 0; i < value.length(); i++)
    {
        if (!(Character.isDigit(value.charAt(i))))
        {
            return false;
        }
    }
    return true;
}
```

Wir führen die Unit Tests aus und diese bestätigen uns, dass alle bisher akzeptierten Zahlen weiterhin gültig sind.

Problem 4: Missverständlicher Name bzw. unklares Verhalten

Nach Rücksprache mit dem Kunden oder Requirements Engineer wird deutlich, dass die Methode `isNumber(String)` neben positiven selbstverständlich auch negative Ganzzahlen verarbeiten können sollte. Eventuell wird später sogar eine Erweiterung auf Gleitkommazahlen gewünscht.

Wie bisher gehen wir testgetrieben vor, d. h., wir erstellen zuerst den Testfall. Dieser sollte fehlschlagen, da die Funktionalität noch nicht existiert. Daraufhin korrigieren wir die Funktionalität. Gerade bei Sourcecode, den man nicht so gut kennt und in dem kleinere Erweiterungen oder Verbesserungen zu realisieren sind, ist dieses Vorgehen hilfreich. Nicht so empfehlenswert finde ich dieses Vorgehen, wenn

man ganz genau weiß, was man realisieren möchte, und das Design und die Implementierung schon im Kopf hat. Dann können die Sprünge zwischen Test und Codierung stören. Genauer gehe ich darauf nochmals in Kapitel 20 ein.

Nach diesem kleinen gedanklichen Ausflug kommen wir wieder zu der Erweiterung zurück. Wir wollen die Verarbeitung von Vorzeichen prüfen und nutzen die Werte "+4711" und "-4711" als Eingaben, die in beiden Fällen das Ergebnis `true` liefern sollten, was wir folgendermaßen prüfen:

```
@Test
public void testNumberPositive_PlusSignShouldBeAccepted()
{
    assertTrue("plus sign should be accepted", NumberUtils.isNumber("+4711"));
}

@Test
public void testNumberNegative_MinusSignShouldBeAccepted()
{
    assertTrue("minus sign should be accepted",
NumberUtils.isNumber("-4711"));
}
```

Weil wir die Implementierung mittlerweile ziemlich gut kennen, wissen wir, dass keine Vorzeichen unterstützt werden. Wie erwartet, werden demnach beide Tests auch nicht bestanden. Aber durch das mittlerweile erworbene Know-how fällt die Korrektur der Methode nicht schwer:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    if (value.isEmpty())
    {
        return false;
    }
    // Verarbeite Vorzeichen
    String number = value;
    if (value.startsWith("-") || value.startsWith("+"))
    {
        number = value.substring(1, value.length());
    }
    // Weitere Prüfung auf Zahl wie zuvor
    for (int i = 0; i < number.length(); i++)
    {
        if (!(Character.isDigit(number.charAt(i))))
        {
            return false;
        }
    }
    return true;
}
```

Zwar werden nun alle Tests bestanden, die Komplexität des Applikationscodes hat aber schon wieder zugenommen. Spätestens jetzt sollten wir uns fragen, ob es nicht eine adäquate Funktionalität im JDK oder externen Bibliotheken gibt. Dadurch kann man sich in der Regel einige Arbeit sparen. Immerhin haben wir nun einen guten Satz an Unit Tests. Damit können wir dann auch gleich das letzte Problem der eingangs

aufgestellten Mängelliste angehen. Schauen wir einmal, was möglich ist.

Problem 5: Viele `return`-Anweisungen

Die Methode `isNumber(String)` sollte auf Ganzzahlen prüfen. Statt diese Funktionalität, wie initial geschehen, selbst zu programmieren, bietet sich der Einsatz der Java-Bibliotheken zum Parsen von Zahlen an. Da unsere Tests bereits recht ausgereift sind, müssen wir hier nichts ergänzen. In der Methode `isNumber(String)` verwenden wir nun die Methode `Integer.parseInt(String)` (vgl. Abschnitt 4.2.2). Durch deren Einsatz können im Gegensatz zu der eigenen Realisierung sowohl positive als auch negative Zahlen verarbeitet werden.

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    try
    {
        Integer.parseInt(value); // Rückgabe ignorieren, nur prüfen
        return true;
    }
    catch (final NumberFormatException ex)
    {
        return false;
    }
}
```

Wir nutzen wieder unsere mittlerweile auf 7 Testfälle angewachsene Testklasse, um die korrekte Funktionalität zu prüfen. Alle Unit Tests werden bestanden. Ziehen wir ein Fazit und schauen auf mögliche Erweiterungen.

Was haben wir erreicht?

Die Intention und Realisierung der Methode ist nun deutlich klarer, da nicht mehr auf Basis einzelner Zeichen geprüft wird. Vielmehr erfolgt auf logischer Ebene eine Umwandlung in eine Zahl. Die Lesbarkeit hat dadurch enorm zugenommen. Wir kommentieren zudem die bewusst fehlende Auswertung des Rückgabewerts von `Integer.parseInt(String)`, um möglicherweise aufkommende Fragen anderer Entwickler sofort zu klären.

Einschränkungen und mögliche Erweiterungen

Die obige Realisierung ist deutlich klarer und besser verständlich als die Originalmethode sowie alle zuvor gezeigten Zwischenschritte.

Allerdings gibt es doch noch eine Kleinigkeit zu bedenken bzw. zu bemängeln: Durch die Prüfung mit `Integer.parseInt(String)` wird der Wertebereich auf `Integer.MIN_VALUE` bis `Integer.MAX_VALUE` begrenzt, also in etwa auf ±2

Milliarden. Für größere Wertebereiche können wir auf die Wrapper-Klasse `Long` zurückgreifen. Um Gleitkommazahlen zu unterstützen, können wir die Klassen `Float` bzw. `Double` verwenden. All dies führt jedoch zu Veränderungen im nach außen sichtbaren Verhalten der Methode, weil dadurch weitere Eingabewerte erlaubt sind. Wir akzeptieren hier die Erweiterung des Wertebereichs, müssen dabei aber die Anmerkungen im folgenden Hinweis »Vorsicht selbst bei minimalen Modifikationen am Verhalten« bezüglich der Änderung von Verhalten bedenken.

Hinweis: Vorsicht selbst bei minimalen Modifikationen am Verhalten

In einigen Schritten wurde durch die initiale Prüfung sowie den Einsatz von Bibliotheksfunktionen zum Parsing minimal etwas am nach außen veröffentlichten Verhalten verändert. So etwas übersieht man leicht. Allerdings können dadurch Probleme durch Inkompatibilitäten verursacht werden. Betrachten wir dies im Detail.

Auswirkungen der Korrekturen für Problem 1 Die ursprüngliche Methode hat bei Übergabe leerer Eingaben eine `StringIndexOutOfBoundsException` ausgelöst. Die neue Realisierung greift gar nicht erst indiziert zu, wenn der übergebene Text leer ist, sondern es wird direkt `false` zurückgegeben.

Auswirkungen der Korrekturen für Problem 4 und 5 Durch die Akzeptanz von Vorzeichen sowie den Einsatz von Bibliotheksfunktionen beim Parsing werden nun je nach gewählter Realisierung auch Vorzeichen und Nachkommastellen unterstützt. Die ursprüngliche Methode hat lediglich Ganzzahlen ohne Vorzeichen, dafür aber beliebiger Länge, als Zahl erkannt. Dies war höchstwahrscheinlich nicht als Feature gedacht, sondern war vermutlich eher ein »Unfall«.

Kompatibilität und Einsatz von Bibliotheksfunktionen

Die Kompatibilität zum ursprünglichen Verhalten kann entscheidend sein, wenn man nicht alle Aufrufer im Zugriff hat und somit nicht weiß, ob vielleicht einer von diesen die Information »keine Zahl« durch Abfangen einer `StringIndexOutOfBoundsException` ermittelt. Andere Aufrufer könnten beliebig lange Ziffernfolgen als Zahlen auswerten wollen.

Weil Ersteres meiner Ansicht nach ein Designfehler ist, werde ich hier keine Lösung angeben. Die Auswertung beliebig langer Ziffernfolgen ist ein denkbarer Anwendungsfall. Hätte die Aufgabe der ursprünglichen Methode `isNumber(String)` tatsächlich darin bestanden, nur eine derartige Prüfung durchzuführen, kann man dies elegant und kompatibel zu allen Unit Tests mit der Klasse `java.math.BigInteger` wie folgt realisieren:

```
public static boolean isNumber(final String value)
{
    Preconditions.checkNotNull(value, "parameter 'value' must not be null");

    try
```

```
        {
            new BigInteger( value); // nur prüfen
            return true;
        }
        catch ( final NumberFormatException ex)
        {
            return false;
        }
    }
```

17.2 Das Standardvorgehen

Das einleitende Beispiel hat Ihnen einen ersten Eindruck von Refactoring-Schritten vermittelt. Um reproduzierbare Ergebnisse zu erreichen und mehr Sicherheit zu haben, halten wir uns an eine Art Checkliste zur Überarbeitung von Sourcecode. Diese beschreibt das bereits angesprochene Standardvorgehen, das folgende Schritte umfasst:

1. **Testen** – Führe vorhandene Unit Tests aus. Im Speziellen sollte dies kontinuierlich nach jedem der folgenden Schritte geschehen.
2. **Coding Conventions anwenden** – Beachte die später in Abschnitt 19.3 vorgestellten Coding Conventions:
   - Formatiere den Sourcecode
   - Mache, falls möglich, die Übergabeparameter `final`
   - Definiere, falls möglich, lokale Variablen `final`
   - Sorge für verständliche Konstanten-, Variablen- und Methodennamen
3. **In Einzelbestandteile zerlegen** – Zerlege, wenn notwendig und sinnvoll, eine komplexere Programmstelle zunächst mithilfe (des mehrmaligem Einsatzes) des Basis-Refactorings EXTRACT METHOD, das in Eclipse im Menü REFACTOR –> EXTRACT METHOD oder über das Tastaturkürzel ALT+SHIFT+M erreichbar ist, in handhabbare und sinnvolle kleinere Bestandteile.
4. **Unit Tests erstellen** – Nutze, sofern die Anforderungen von der Fachseite oder dem Kunden oder aus einem Requirements-Dokument bekannt sind, diese, um daraus Testfälle zu gestalten. Ansonsten schreibe für zuvor extrahierte Methoden entsprechende Unit-Test-Methoden:
   - Für normale Eingaben
   - Für ungültige Eingaben
   - Für Rand- oder Spezialfälle
5. **Aufräumen** – Räume den Sourcecode auf:
   - Entferne unbenutzte Variablen und unbenutzte Methoden
   - Entferne alte, unnötige oder (mittlerweile) falsche Kommentare

6. **Vereinfachen** – Reduziere die Komplexität:
   - Entferne duplizierten Sourcecode, erzeuge gegebenenfalls Hilfsmethoden
   - Vereinfache Bedingungen
   - Füge bei Bedarf erklärende Kommentare ein – oftmals sollte man zunächst sprechende Bezeichner für Attribute und Methoden in Betracht ziehen.
7. **Konkrete Refactorings durchführen** – Verbessere den Sourcecode durch den Einsatz eines für den erkannten Schwachpunkt passenden Refactorings aus dem Refactoring-Katalog aus Abschnitt 17.4.

Die einzelnen Schritte müssen nicht sklavisch exakt in dieser Reihenfolge abgearbeitet werden. Es ist durchaus möglich, die Reihenfolge zu variieren und einige Schritte auszulassen oder auch mehrfach auszuführen. Um ein Gespür für die Vorgehensweise beim Einsatz in der Praxis zu bekommen, wollen wir die obigen Schritte an einem weiteren kurzen Beispiel nachvollziehen.

Beachten Sie bitte, dass ich sowohl im folgenden Beispiel als auch bei der Vorstellung der Refactorings auf eine detaillierte Darstellung der eigentlich notwendigen Unit Tests verzichten werde, um so den Fokus auf die eigentliche Transformation zu legen. In der Praxis entspricht die Vorgehensweise jedoch dem zuvor gezeigten einleitenden Beispiel, in dem Unit Tests zur Absicherung erstellt und eingesetzt wurden.

Beispiel

Als Ausgangsbasis dient folgender überarbeitungswürdiger `catch`-Block, den ich tatsächlich – abgesehen von Details – so in Produktionscode vorgefunden habe:

```
catch (final Fault aF)
{
    final String stSrc = aF.getSource();
    final String stErr = aF.getFaultCode();
    if (stErr != null && stErr.equalsIgnoreCase("FileNotFound"))
    {
        stErrorMsg = aF.getFaultString() + ": " + stSrc;
    }
    else
    {
        stErrorMsg = aF.getFaultString() + ": " + stSrc;
    }
}
```

Wir führen folgende Schritte (Nummerierung laut Standardvorgehen) durch:

Schritt 2: Coding Conventions anwenden

Befolgt man Namenskonventionen führt das zu mehr Lesbarkeit. Es fällt auf, dass in diesem Sourcecode-Abschnitt einige Präfixe in den Variablennamen verwendet werden. Wie später in Abschnitt 19.3.1 genauer beschrieben, stellen Präfixe in

Variablennamen eher ein Relikt aus alten Tagen dar, weil damals die IDEs noch nicht in der Lage waren, Informationen aus dem Sourcecode on the fly zu extrahieren und etwa Attribute farblich anders darzustellen. Heutzutage sollte man Präfixe selten verwenden oder besser ganz vermeiden – ***beim Überarbeiten eines bestehenden Programmteils ist es aber ratsam, nicht alles umzukrempeln und dem vorhandenen Stil einigermaßen zu folgen***, sofern dies nicht (allzu sehr) gegen die eigenen vorgegebenen Konventionen verstößt. Im Beispiel gilt, dass ein Präfix nutzlos ist, wenn kein sinnvoller Variablenname folgt wie hier für die Variable `aF`. Diese wird daher in `fault` umbenannt. Solche Namensänderungen sind für lokale Variablen immer möglich. Wir benennen die Variable `stErr` in `strErr` um. Aus `stSrc` wird `strSource`. Beide Male wird das merkwürdige Präfix `st` dabei zu `str` für Stringvariablen. Das Beibehalten der Kürzel ist ein Zugeständnis, nicht allzu viel am Stil zu ändern. Zudem wird `strErr` durch Umbenennung in `strFaultCode` besser lesbar und verständlich. Die Namensähnlichkeit mit dem privaten Attribut `stErrorMsg` und die daraus resultierende Verwechselungsgefahr existiert nach dieser Namensänderung nicht mehr. Das Attribut wird außerdem durch den Einsatz der `this`-Notation explizit als solches gekennzeichnet und mit dem Präfix `str` zu `strErrorMsg`. Es lässt sich somit visuell gut von den lokalen Variablen abgrenzen.[4]

```
catch (final Fault fault)
{
    final String strSource = fault.getSource();
    final String strFaultCode = fault.getFaultCode();
    if (strFaultCode != null && strFaultCode.equalsIgnoreCase("FileNotFound"))
    {
        this.strErrorMsg = fault.getFaultString() + ": " + strSource;
    }
    else
    {
        this.strErrorMsg = fault.getFaultString() + ": " + strSource;
    }
}
```

Nicht in jedem Fall ist eine Korrektur von Namen derart möglich. Private Attribute können problemlos umbenannt werden – sofern darauf nicht per Reflection zugegriffen wird. ***Im nachfolgenden Text dieses Kapitels gehe ich auf den Spezialfall Reflection und Refactorings kaum mehr explizit ein.*** Die Änderung der Namen öffentlicher Attribute kann Änderungen in einsetzenden Klassen erfordern und ist daher möglicherweise problematisch. Eine Verbesserung der Kapselung erreicht man mit dem Refactoring REDUZIERE DIE SICHTBARKEIT VON ATTRIBUTEN (vgl. Abschnitt 17.4.1), das auf dem Basis-Refactoring ENCAPSULATE FIELD basiert. Als Folge kann bei Bedarf das Design geändert werden. Statt eines Attributs kann man eine Delegation nutzen oder den Wert dynamisch berechnen – das bietet sich etwa für die Eigenschaft Alter einer Person an, das sich aus dem aktuellen Datum und dem Geburtstag ergibt.

Schritt 6: Vereinfachen

In diesem einfachen Beispiel findet man eine exakte Duplikation der Anweisungen im

`if`- und `else`-Anweisungsblock. Zwar ist das realer Anwendungscode, aber meistens ist eine derart extreme Duplikation selten und eher als Wiederholung von Teilabschnitten zu beobachten. In diesem Beispiel kann eine enorme Vereinfachung erzielt werden, weil die Auswertung der Bedingung überflüssig ist. Der Anweisungsblock muss dadurch lediglich einmal notiert werden:

```
catch (final Fault fault)
{
    final String strSource = fault.getSource();
    final String strFaultCode = fault.getFaultCode();

    this.strErrorMsg = fault.getFaultString() + ": " + strSource;
}
```

Schritt 5: Aufräumen

Als Folge dieser Vereinfachungen wird ersichtlich, dass die Variable `strFaultCode` nun unbenutzt und damit überflüssig ist. Damit kann sie entfallen. Auch die Variable `strSource` ermittelt nur Daten aus dem übergebenen `Fault`-Objekt. Wir schreiben einfacher Folgendes:

```
catch (final Fault fault)
{
    this.strErrorMsg = fault.getFaultString() + ": " + fault.getSource();
}
```

Fazit

Das Resultat ist beeindruckend: Aus zehn Zeilen Sourcecode ist eine Zeile geworden. Ähnlich wie sich Bad Smells ausbreiten, wenn man unachtsam ist, gibt es glücklicherweise auch einen gegenteiligen Effekt: ***Je mehr man für Klarheit und Struktur sorgt, desto leichter fallen weitere Verbesserungsmaßnahmen.***

Jedoch haben wir ein Problem nicht entfernen können: Nach wie vor gibt es eine Zuweisung an das Attribut `strErrorMsg` im `catch`-Block. Ein solcher Seiteneffekt ist zu vermeiden, da es an unerwarteter Stelle im Programm zu Zustandsänderungen kommt. ***Durch Refactorings wollen wir normalerweise das nach außen sichtbare Programmverhalten nicht verändern, sondern lediglich die innere Struktur verbessern***. An dieser Stelle können wir im Sourcecode einen Kommentar mit einem Hinweis auf diesen Seiteneffekt einfügen und das Ganze später nochmal prüfen bzw. überarbeiten.

17.3 Kombination von Basis-Refactorings

Bevor wir uns einen Katalog einiger komplexerer Refactorings – und zum Teil genau genommen sogar leicht verhaltensverändernder Transformationen des Sourcecodes – anschauen, wollen wir zunächst an einem Beispiel verschiedene durch die IDE unterstützte Basis-Refactorings, wie sie Martin Fowler in seinem Buch »Refactoring:

Improving the Design of Existing Code« [27] beschreibt, betrachten. Die Basis-Refactorings zeichnen sich dadurch aus, dass sie oftmals keine Modifikation am sichtbaren Verhalten vornehmen. Die hohe Kunst ist es, die Schritte so klein und sicher zu gestalten, dass es dabei möglichst selten zu Kompilierfehlern oder anderweitigen Problemen kommt. Damit es klappt, muss man sich der Refactoring-Automatiken aus der IDE bedienen. Die Wahrscheinlichkeit für Komplikationen steigt, wenn man eher freihändig refaktorisiert.

17.3.1 Refactoring-Beispiel: Ausgangslage und Ziel

Als Ausgangsbasis dient eine Utility-Klasse `TimeStampUtils` mit einer Methode `createTimeStampString()`. Schauen wir zunächst auf einen Aufruf eines Nutzers:

```
final String timeStamp = TimeStampUtils.createTimeStampString(currentPeriod,
                                                               frequency);
```

Weil der Aufruf unproblematisch scheint, betrachten wir nun die statische öffentliche Methode an sich, um mögliche Schwachpunkte zu erkennen:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final DateTime start = currentPeriod.getDateTime();
    final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
    final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Ein erster Blick zeigt eine vermeintlich einfache Realisierung, die Jahresangaben gefolgt von Monat oder Quartal ausgeben soll.[5] Das Ganze ist recht kurz, aber vielleicht durch die Abfragen mit dem `?`-Operator ein wenig unübersichtlich. Problematischer ist jedoch, dass die Methode ungewünschte Abhängigkeiten auf die zwei Klassen `ExtTimePeriod` und `ComplexFrequency` besitzt, die aus einem externen Package (`external`) stammen. Ein genauerer Blick offenbart zusätzlich folgende Probleme:

- Die Methode scheint für beliebige Frequenzen des Typs `ComplexFrequency` ausgelegt zu sein, Tatsächlich ist sie es aber nicht, denn durch einen versteckten Logikfehler wird alles außer der Frequenz monatlich auf Quartale abgebildet. Dadurch können nur Monats- oder Quartalswerte korrekt verarbeitet werden.
- Es ist unklar, welches der gewünschte Rückgabewert ist. Gerade im Bereich von Datumsarithmetik findet man 0- oder 1-basierte Werte: Startet `getMonthOfYear()` also mit 0 oder 1? Und wieso erfolgt eine Subtraktion von 1?

Für die nachfolgenden Refactorings steht zunächst die Auflösung der Abhängigkeiten

im Fokus. Auf die beiden anderen Details der Verarbeitung gehe ich später ein.

Definition des Ziels

Die Methode `createTimeStampString(ExtTimePeriod, ComplexFrequency)` soll nun mithilfe von Basis-Refactorings so umgestaltet werden, dass nur Abhängigkeiten auf Standards wie Joda-Time[6] oder besser noch JDK-Klassen bestehen und der Sourcecode verständlicher wird. Bevor wir mit den Umbauarbeiten beginnen, erstellen wir ein UML-Klassendiagramm von der Ausgangslage und insbesondere auch einem möglichen Zieldesign. Das beides ist in Abbildung 17-1 dargestellt. Das gezeigte Ziel ist nicht ganz starr, sondern eher ein Anhaltspunkt, da man beim Entwickeln gewöhnlich immer noch kleinere Änderungen vornimmt. Das ist auch der Grund, warum wir hier keine Typparameter in den Signaturen angeben.

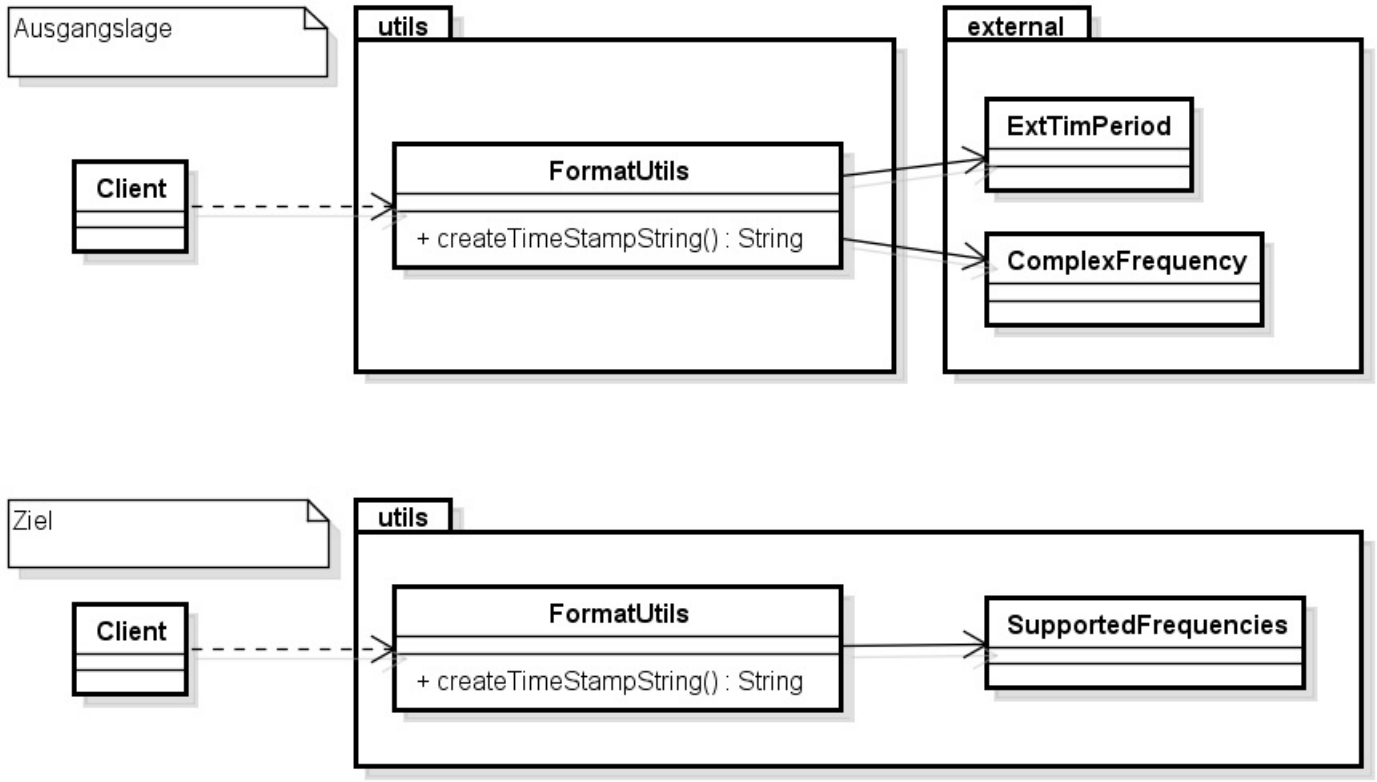


Abbildung 17-1 *Refactoring der Methode* `createTimeStampString()`

Wir wollen nun folgende Schritte ausführen, um die angemerkten Probleme zu beseitigen und das dargestellte Ziel zu erreichen:

- **Auflösen der Abhängigkeiten** – In einem ersten Schritt wollen wir die Abhängigkeiten zum Package `external` auflösen, indem wir einen `enum` namens `SupportedFrequencies` als Ersatz für `ComplexFrequency` einführen und anstelle der Klasse `ExtTimePeriod` die Klasse `DateTime` aus Joda-Time nutzen.
- **Vereinfachungen** – Einige der Berechnungen in der Methode sind etwas komplex und nicht gut zu lesen. Wir werden ein paar Vereinfachungen vornehmen.

- **Verlagern von Funktionalität** – Abschließend schauen wir, wie wir durch eine kleine Änderung von Zuständigkeiten für mehr Klarheit im Design sorgen.

17.3.2 Auflösen der Abhängigkeiten

Um den Sourcecode klarer und besser verständlich zu gestalten, werden wir folgende Refactorings, deren Tastaturkürzel in Eclipse in Klammern notiert sind, nutzen:[7]

- EXTRACT LOCAL VARIABLE (ALT+SHIFT+L)
- EXTRACT METHOD (ALT+SHIFT+M)
- INLINE (ALT+SHIFT+I)
- CHANGE METHOD SIGNATURE (ALT+SHIFT+C)

Schritt 1: Hilfsvariable einführen (EXTRACT LOCAL VARIABLE)

Zum leichteren Verständnis wird die zu bearbeitende Methode nochmals gezeigt:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final DateTime start = currentPeriod.getDateTime();
    final int divisor = frequency == ComplexFrequency.P1M ? 1 : 3;
    final String addition = frequency == ComplexFrequency.P1M ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Als Erstes selektieren wir den Ausdruck `ComplexFrequency.P1M` und nutzen das Refactoring EXTRACT LOCAL VARIABLE (ALT+SHIFT+L), um die lokale Variable `isMonthly` zu extrahieren, wodurch sich der Sourcecode vereinfachen lässt:

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final DateTime start = currentPeriod.getDateTime();
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Schritt 2: Abhängigkeit zur Frequenz entfernen (EXTRACT METHOD)

Als Nächstes wollen wir die Abhängigkeit zur Klasse `ComplexFrequency` auflösen. Wir erzeugen dazu eine separate Methode `createTimeStampString()`, jedoch

mit anderer Signatur. Damit wir diese aus der Originalmethode extrahieren können, ordnen wir die Zeilen um und nutzen dazu die Tastaturkürzel ALT+UP/DOWN. Damit verschieben wir die boolesche Variable `start` direkt zu der ersten Verwendung, also vor die Definition von `value`. Die Variable `isMonthly` schieben wir ganz nach oben an den Methodenanfang:[8]

```
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final DateTime start = currentPeriod.getDateTime();
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Danach selektieren wir alle Zeilen nach der Definition von `isMonthly` und setzen das Refactoring EXTRACT METHOD (ALT+SHIFT+M)[9] ein. Damit entsteht eine gleichnamige Methode mit der Sichtbarkeit `public`, die als Parametertyp `boolean` statt `ComplexFrequency` besitzt.

```
@Deprecated
public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final ComplexFrequency frequency)
{
    final boolean isMonthly = frequency == ComplexFrequency.P1M;
    return createTimeStampString(currentPeriod, isMonthly);
}

public static String createTimeStampString(final ExtTimePeriod currentPeriod,
                                            final boolean isMonthly)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final DateTime start = currentPeriod.getDateTime();
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Die unerwünschte Abhängigkeit zur Klasse `ComplexFrequency` wurde damit aufgelöst – allerdings durch einen booleschen Parameter, was meistens kein gutes Design ist. Später komme ich darauf zurück und wir beheben auch diese Schwachstelle.

Indem wir die ursprüngliche Methode als `@Deprecated` markieren, signalisieren wir, dass zukünftige Nutzer stattdessen die neu erstellte Methode verwenden sollten. Für die bisherigen Aufrufer hat sich nichts geändert.

Schritt 3: Inlining des Methodenaufrufs (INLINE)

Da wir die Abhängigkeit zur Klasse `ComplexFrequency` in der Utility-Klasse

eliminieren wollen, sollte überall die neu erstellte statt der alten Methode eingesetzt werden.

Die Aufrufstellen sind ähnlich zu folgender:

```
final String timeStamp = createTimeStampString(currentPeriod, frequency);
```

Die Aufrufstellen könnten wir zwar von Hand korrigieren, aber es ist sinnvoller und weniger fehleranfällig, dazu die in die IDE integrierten Refactorings zu nutzen.[10] Dazu markieren wir den Namen der ursprünglichen Methode und nutzen dann das Refactoring INLINE (ALT+SHIFT+I). Dieses transformiert alle Aufrufstellen folgendermaßen:

```
final boolean isMonthly = frequency == ComplexFrequency.P1M;
final String timeStamp = createTimeStampString(currentPeriod, isMonthly);
```

Optional kann die nicht mehr benötigte Methode automatisch gelöscht werden, wodurch nur noch die zuvor extrahierte, neue Methode in der Utility-Klasse verbleibt. Die bisher durchgeführten Änderungen lassen erahnen, dass sich für Umgestaltungen die Nutzung von Refactoring-Automatiken anbietet, um die Wahrscheinlichkeit für Fehler zu reduzieren und für konsistente Änderungen zu sorgen.

Schritt 3 (optional): Inlining der Hilfsvariablen (INLINE)

Die Aufrufstellen sehen nun nach Schritt 3 – unter anderem durch den booleschen Übergabeparameter – etwas ungelenk aus, was wir später noch mit einer Designänderung adressieren werden. Zunächst könnte man in einem weiteren Schritt statt der lokalen Variablen den Ausdruck direkt als Methodenparameter angeben. Dabei hilft wiederum das Refactoring INLINE (ALT+SHIFT+I), diesmal für die Variablendeklaration. Zum Ausführen ist die Variable `isMonthly` zu selektieren:

```
final boolean isMonthly = frequency == ComplexFrequency.P1M;
final String timeStamp = createTimeStampString(currentPeriod, isMonthly);
```

Durch das Refactoring INLINE wird der Aufruf folgendermaßen abgewandelt:

```
final String timeStamp = createTimeStampString(currentPeriod,
                                                frequency == MyFrequency.P1M);
```

Jedoch reduziert dieser Schritt mitunter die Verständlichkeit und Lesbarkeit.

Schritt 4: Abhängigkeit zur Klasse `ExtTimePeriod` entfernen

Kommen wir wieder zu der eigentlichen Methode `createTimeStampString()` zurück. Unser Ziel besteht darin, die Abhängigkeiten aufzulösen, nun die auf die Klasse `ExtTimePeriod`. Dabei hilft uns ein scharfer Blick auf die Methode und das Refactoring EXTRACT METHOD: Abgesehen von der ersten Zeile der Methode selektieren wir den Rest und extrahieren eine gleichnamige Methode. Die Utility-

Klasse sieht wie folgt aus, nachdem wir die alte Methode noch als `@Deprecated` markiert haben:

```
@Deprecated
public static String createTimeStampString(final MyTimePeriod currentPeriod,
                                            final boolean isMonthly)
{
    final DateTime start = currentPeriod.getDateTime();
    return createTimeStampString(isMonthly, start);
}
public static String createTimeStampString(final boolean isMonthly,
                                            final DateTime start)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Wie schon zuvor, hat das Refactoring keine Auswirkungen auf bisherige Nutzer, außer, dass diese nun durch das Hinzufügen von `@Deprecated` auf unsere Änderungen in der Utility-Klasse aufmerksam gemacht werden.

Schritt 5: Für konsistente Parameterreihenfolge sorgen

Beim Extrahieren der Methode fällt uns auf, dass durch die Automatik die Parameterreihenfolge vertauscht wurde und `isMonthly` nun der erste Parameter ist. Das wollen wir korrigieren. Dazu nutzen wir das Refactoring CHANGE METHOD SIGNATURE (ALT+SHIFT+C) und vertauschen die beiden Parameter, womit wir wieder eine konsistente Reihenfolge erzielen.

Schritte 6: Inlining des Methodenaufrufs

Wir führen weitere Aufräumarbeiten aus und selektieren die alte Methode und nutzen das Refactoring INLINE. Dadurch verdichten wir die Utility-Klasse:

```
public static String createTimeStampString(final DateTime start,
                                            final boolean isMonthly)
{
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Auch die Aufrufstelle wird automatisch durch die IDE angepasst:

```
final DateTime start = currentPeriod.getDateTime();
final String timeStamp = createTimeStampString(start,
                                                frequency ==
EasyFrequency.P1M);
```

Schritte 6 (optional): Inlining der Hilfsvariablen

Wenn gewünscht, kann man nochmals das Refactoring INLINE ausführen, um die Hilfsvariable zu entfernen und direkt in den Methodenaufruf zu integrieren:

```
final String timeStamp = createTimeStampString(currentPeriod.getDateTime(),
                                                frequency ==
EasyFrequency.P1M);
```

Zwischenfazit

Die erstellte Methode besitzt keine Abhängigkeiten auf externe Klassen mehr oder zumindest nur auf Klassen, die Standardbibliotheken wie Joda-Time entstammen. Damit lässt sich das Ganze viel einfacher mit Unit Tests überprüfen.

Tests

Bisher haben wir keine Unit Tests ausgeführt. Das lag zum einen daran, dass es keine gab, und zum anderen daran, dass wir bisher Refactorings nur so genutzt haben, dass das nach außen sichtbare Verhalten sich nicht ändert. Bitte beachten Sie, dass dies selbst für die Basis-Refactorings nicht immer gilt.[11] Allerdings sind diese deutlich sicherer, als Änderungen von Hand auszuführen. In jedem Fall empfiehlt sich bei Umstrukturierungen die Ausführung von Unit Tests. Weil es noch keine gibt, erstellen wir hier exemplarisch zwei einfache Testfälle. In der Praxis sollten Utility-Klassen umfangreicher getestet werden, als es der Platz hier erlaubt:

```
@Test
public void testCreateTimeStampString_Monthly()
{
    final boolean MONTHLY = true;
    assertEquals("2000-2", createTimeStampString(
                           new DateTime(2000, 2, 7, 0, 0), MONTHLY));
    assertEquals("2000-7", createTimeStampString(
                           new DateTime(2000, 7, 14, 0, 0), MONTHLY));
}

@Test
public void testCreateTimeStampString_Quarterly()
{
    final boolean QUARTERLY = false;
    assertEquals("2000-Q1", createTimeStampString(
                            new DateTime(2000, 2, 7, 0, 0), QUARTERLY));
    assertEquals("2000-Q3", createTimeStampString(
                            new DateTime(2000, 7, 14, 0, 0), QUARTERLY));
}
```

Wir führen die Tests aus und sie zeigen – wie erwartet – Grün. Normalerweise würden wir noch ein paar mehr Testfälle ergänzen. In diesem Kontext sollen uns aber diese zwei reichen, um mögliche Probleme aufzuzeigen.

Unzulänglichkeit: Boolescher Parameter

Durch die Refactoring-Schritte haben wir zwar die Abhängigkeiten zum Package `external` gelöst, jedoch – wie schon zuvor erwähnt – auch eine Unschönheit in unsere öffentliche Schnittstelle eingefügt: einen booleschen Parameter. Was ist daran störend? Aufrufer müssen dadurch immer genau wissen, was die Werte `true` bzw. `false` ausdrücken sollen. Dies ist jedoch nur durch Betrachten der Methode, nicht aber der Aufrufstelle allein möglich.

Die Verwendung der booleschen Konstanten `MONTHLY` und `QUARTERLY` in den Tests macht deutlich, dass sich ein weiteres Refactoring anbietet, um den booleschen Parameter in der öffentlichen Schnittstelle zu eliminieren. In unserem Beispiel hatten wir die Klassen im Zugriff und durften dort auch ändern. Das ist jedoch nicht immer der Fall, sodass man mitunter mit der Signatur leben muss. Wie kann man trotzdem für besser verständlichen Sourcecode sorgen? Schauen wir auf verschiedene Abhilfen.

Abhilfen ohne Änderungen der Signatur

Wie wir es beim Erstellen der Tests kennengelernt haben, kann man zwei Konstanten mit sprechenden Namen definieren:

```
public static final boolean MONTHLY = true;
public static final boolean QUARTERLY = false;
```

Oftmals besser lesbar ist es, eine Hilfsmethode wie folgt zu implementieren:

```
private static boolean isMonthly(final ComplexFrequency frequency)
{
    return frequency == ComplexFrequency.P1M;
}
```

Bei der zweiten Variante verbleibt allerdings die Abhängigkeit von Aufrufern an das Package `external`, was aber möglicherweise akzeptabel ist.

Es gibt eine weitere Möglichkeit, die so elegant in der Nutzung und offensichtlich ist, dass man sie leicht übersieht: Man definiert zwei Methoden mit sprechendem Namen, die jeweils den erwarteten Wert zurückgeben:[12]

```
private static boolean monthly()
{
    return true;
}

private static boolean quarterly()
{
    return false;
}
```

Die gezeigten Varianten lösen das eigentliche Problem nicht, lindern jedoch ein wenig die »API-Schmerzen«. Das ist für die Fälle praktisch, in denen man die Schnittstelle der Klasse nicht ändern kann, die Aufrufe aber klarer gestalten möchte. Der Aufruf für monatlich würde wie folgt aussehen:

```
final String timeStamp = createTimeStampString(currentPeriod.getDateTime(),
                                                monthly());
```

Abhilfen mit Änderungen der Signatur

Wenn man auf die Klassen Zugriff hat und die Methode ändern kann, bietet sich die Definition eines `enums` an:

```
public enum SupportedFrequencies
{
    MONTHLY, QUARTERLY;
}
```

Damit können wir die Signatur der Methode wie folgt abändern und eine Hilfsvariable `isMonthly` einführen:

```
static String createTimeStampString(final DateTime start,
                                     final SupportedFrequencies frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Diese Lösung ist für Aufrufer klarer, was ein sehr wichtiger Punkt ist, da damit auch die Benutzbarkeit verbessert wird. Allerdings fängt das Ganze intern an, unübersichtlich zu werden. Es wird höchste Zeit, nach ein paar Vereinfachungen Ausschau zu halten.

17.3.3 Vereinfachungen

In diesem Abschnitt sehen wir uns zwei Arten von Vereinfachungen an. Zunächst entzerren wir die Anweisungen und die etwas komplexere Logik. Daraus ergeben sich weitere Möglichkeiten, die Formel zur Berechnung an sich zu vereinfachen.

Vereinfachung der Anweisungen

Die Komplexität innerhalb der Methode entsteht vor allem dadurch, dass hier die zwei Fälle »Monatlich« und »Quartalsweise« ineinander verwoben behandelt werden:[13]

```
public static String createTimeStampString(final DateTime start,
                                            final SupportedFrequencies
frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor = isMonthly ? 1 : 3;
    final String addition = isMonthly ? "" : "Q";
    final int value = ((start.getMonthOfYear() - 1) / divisor + 1);

    return start.getYear() + "-" + addition + value;
}
```

Teilen wir das Ganze doch einfach so auf, dass wir abhängig von `isMonthly` zwei Wertebelegungen erhalten. Versuchen wir schrittweise dorthin zu kommen.

Als Vorbereitung führen wir das Refactoring INLINE für die Variable `value` aus, um den Ausdruck zur String- und Wertekonkatenation zusammenzuführen.

Den ternären Operator (`?`-Operator) kann man in eine `if-else`-Anweisung umwandeln. Dazu nutzen wir das Tastaturkürzel CTRL+1 für QUICK FIX und wählen dort REPLACE CONDITIONAL WITH 'IF-ELSE'. Das machen wir für beide `?`-Operatoren. Damit ergibt sich folgende Variante der ursprünglichen Methode, wobei die entstehenden `if-else`-Anweisungen allerdings (noch) keine Blöcke sind:

```
public static String createTimeStampString(final DateTime start,
                                            final SupportedFrequencies
frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    if (isMonthly)
        divisor = 1;
    else
        divisor = 3;
    final String addition;
    if (isMonthly)
        addition = "";
    else
        addition = "Q";

    return start.getYear() + "-" + addition + ((start.getMonthOfYear() - 1)
          / divisor + 1);
}
```

Das Ergebnis sieht ein wenig chaotisch aus und scheint in die falsche Richtung zu gehen, da viel mehr Zeilen entstanden sind. Lassen Sie sich nicht entmutigen. Die Tests zeigen, dass sich das Verhalten der Methode nicht geändert hat. Wir sind wohl auf dem richtigen Weg. Insbesondere sind die einzelnen Abfragen viel weniger komplex.

Jetzt wollen wir die jeweiligen Zeilen für die beiden Bedingungen zusammen gruppieren. Voraussetzung dazu ist aber, dass wir die `if-else`-Anweisungen in Blöcke umwandeln. Dazu selektieren wir das `if` und nutzen wiederum das Tastaturkürzel CTRL+1, was uns nun die Option CHANGE 'IF-ELSE' STATEMENTS TO BLOCKS anbietet, die wir wählen. Danach ordnen wir die Zeilen um, indem wir die Zeilen aus den jeweiligen Bedingungen gruppieren. Als Folge entsteht im unteren Teil ein leeres `if-else`-Gebilde, was wir entfernen. Es ergibt sich folgende Methode:

```
public static String createTimeStampString(final DateTime start,
                                            final SupportedFrequencies
frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    final String addition;
```

```
    if (isMonthly)
    {
        divisor = 1;
        addition = "";
    }
    else
    {
        divisor = 3;
        addition = "Q";
    }

    return start.getYear() + "-" + addition + ((start.getMonthOfYear() - 1)
            / divisor + 1);
}
```

Der Sourcecode ist erneut länger geworden, aber zumindest sind die logischen Einheiten gruppiert. Bevor wir vereinfachen können, wird es noch etwas unübersichtlicher und wir benötigen auch etwas Handarbeit, um die inverse Variante des Basis-Refactorings CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENT[14] durchzuführen. Normalerweise will man damit duplizierte Elemente in `if`-Zweigen zu einer Anweisung am Ende zusammenfügen. Hier machen wir das Gegenteil und duplizieren die `return`-Anweisung, um sie dann in jedem `if-else`-Zweig bereitzustellen:

```
public static String createTimeStampString(final DateTime start,
                                            final SupportedFrequencies
frequency)
{
    final boolean isMonthly = frequency == SupportedFrequencies.MONTHLY;
    final int divisor;
    final String addition;
    if (isMonthly)
    {
        divisor = 1;
        addition = "";
        return start.getYear() + "-" + addition + ((start.getMonthOfYear() -
1)
            / divisor + 1);
    }
    else
    {
        divisor = 3;
        addition = "Q";
        return start.getYear() + "-" + addition + ((start.getMonthOfYear() -
1)
            / divisor + 1);
    }
}
```

Wir machen weiter. Die Variablen `divisor` und `addition` sind eigentlich überflüssig, da sie jeweils nur einfache Konstanten enthalten. Wir können nun die Werte für beide Variable direkt im Sourcecode ersetzen. Je nach verwendeter IDE müssen wir etwas Handarbeit leisten, um die in den Zweigen jeweils konstanten Werte in die `return`-Anweisungszeile zu integrieren sowie die überflüssigen Variablendeklarationen zu entfernen:

```
public static String createTimeStampString(final DateTime start,
```

```
                                                    final SupportedFrequencies
frequency)
{
    if (frequency == SupportedFrequencies.MONTHLY)
    {
        return start.getYear() + "-" + ((start.getMonthOfYear() - 1) / 1 + 1);
    }
    else
    {
        return start.getYear() + "-Q" + ((start.getMonthOfYear() - 1) / 3 +
1);
    }
}
```

Hinweis: Viele Zwischenschritte

Diese vielen kleinen Schritte für diesen einfachen Sourcecode-Abschnitt sehen möglicherweise übertrieben aus. Allerdings bietet das den Vorteil, das man sich in jedem Einzelschritt auf das Wesentliche konzentrieren kann.

Wenn die Programmabschnitte komplexer werden, dann profitiert man am meisten von kleinen Schritten, die im Falle eines Irrwegs bei Bedarf auch leicht zurückgenommen werden können. Flüchtigkeitsfehler lassen sich dadurch eher vermeiden als bei »Freihand«-Refactorings.

Vereinfachung der Berechnung in mehreren Schritten

Wenn man sich die Ausdrücke anschaut, sollte zumindest im ersten Fall eine Vereinfachung möglich sein. Damit wir nicht abgelenkt werden, schauen wir hier wirklich nur auf den Ausdruck der Monatsberechnung an sich, wobei die äußere Klammerung wegen der Stringkonkatenation nicht weiter betrachtet wird:

```
(start.getMonthOfYear() - 1) / 1 + 1
```

Die Division / 1 ist nutzlos

Eine Division durch 1 ändert das Ergebnis nicht, macht aber den Ausdruck komplizierter und den Sourcecode schlechter lesbar. Also entfernen wir diese Division und erhalten folgende Vereinfachung:

```
(start.getMonthOfYear() - 1) + 1
```

Weitere Schritte 1

Aufgrund der Vereinfachung ergeben sich neue Möglichkeiten. In der Praxis sieht man es immer mal wieder, dass Ausdrücke eher zu viel geklammert sind. In diesem Fall ist die äußere Klammerung um die Subtraktion überflüssig und wird entfernt:

```
start.getMonthOfYear() - 1 + 1
```

Weitere Schritte 2

In einem letzten Schritt kann die Berechnung `- 1 + 1` entfallen, weil sie den Wert `0` ergibt und hier somit nutzlos ist. Damit verbleibt für die Berechnung der Monate nur noch der Aufruf `start.getMonthOfYear()` und wir können die Methode wie folgt vereinfachen:

```
public static String createTimeStampString(final DateTime start,
                                            final SupportedFrequencies
frequency)
{
    if (frequency == SupportedFrequencies.MONTHLY)
    {
        return start.getYear() + "-" + start.getMonthOfYear();
    }
    else
    {
        return start.getYear() + "-Q" + ((start.getMonthOfYear() - 1) / 3 +
1);
    }
}
```

Komplexität der Berechnung für Quartale

Die Berechnung des Quartals ist komplexer, was sich kaum vermeiden lässt. In der hier genutzten Klasse `DateTime` aus der Bibliothek Joda-Time beginnt die Zählung der Tage und Monate jeweils bei 1, wie es der menschlichen Denkweise entspricht. Die gezeigten Berechnungen für das Quartal bilden die Werte von 1–12 durch die Subtraktion von 1 auf 0–11 ab, wodurch die Division durch 3 einen Wertebereich von 0–3 liefert. Die Addition von 1 ergibt dann den Wertebereich 1–4. Allerdings liest sich das `/ 3 + 1` potenziell falsch. Um die Berechnung klarer zu gestalten, kann man die Addition von `1` nach vorne ziehen:

```
return start.getYear() + "-Q" + (1 + (start.getMonthOfYear() - 1) / 3);
```

Fazit

Wenn man bedenkt, mit welch kompliziertem Ausdruck wir ins Rennen gestartet sind, ist das Erreichte beeindruckend. Bei einem Blick auf die ursprüngliche Berechnung wäre weder eine Aussage möglich gewesen, was das Resultat denn nun genau ist, noch, ob man die Berechnung vereinfachen kann. Durch unseren letzten Schritt ist keine Vereinfachung für die Monatsberechnung mehr nötig und das Ergebnis offensichtlich. Nur die Quartalsberechnung ist eben etwas komplizierter.

17.3.4 Verlagern von Funktionalität

Wenn wir uns die zuvor verbesserte Methode `createTimeStampString(DateTime, SupportedFrequencies)` genauer ansehen, erkennen wir, dass sie im `if-else` jeweils Funktionalität enthält, die stark mit dem `enum SupportedFrequencies` verbunden ist. Diesen haben wir beim

Erstellen der Unit Tests zur Verbesserung des leicht unhandlichen APIs definiert. Es bietet sich nun an, noch mehr Funktionalität dorthin zu verlagern. Dabei nutzen wir, dass ein `enum` Attribute und Methoden besitzen kann und Letztere sogar überschrieben werden können:

```
public enum SupportedFrequencies
{
   MONTHLY
   {
      public String createTimeStampString(final DateTime start)
      {
         return start.getYear() + "-" + start.getMonthOfYear();
      }
   },

   QUARTERLY
   {
      public String createTimeStampString(final DateTime start)
      {
         return start.getYear() + "-Q" + (1 + (start.getMonthOfYear() - 1) /
3);
      }
   };

   public abstract String createTimeStampString(final DateTime start);
}
```

Die Utility-Klasse enthält nur noch folgende Methode:

```
public static String createTimeStampString(final DateTime start,
                                            final SupportedFrequencies
frequency)
{
    return frequency.createTimeStampString(start);
}
```

Tatsächlich sprechen nur noch Rückwärtskompatibilitätsgründe dafür, die Utility-Klasse überhaupt noch beizubehalten. Ansonsten könnte man die Funktionalität durch direkte Aufrufe an die jeweilige `enum`-Konstante realisieren.

17.4 Der Refactoring-Katalog

Dieser Abschnitt stellt einige Refactorings sowie Transformationen als Schritt-für-Schritt-Anleitungen vor. Diese sollen dabei helfen, ein spezielles Problem auf eine definierte Art und Weise zu beheben. Einige Refactorings lassen sich zu »High-Level-Refactorings« kombinieren. Ein Beispiel dafür ist die als MINIMIERE ZUSTANDSÄNDERUNGEN beschriebene Kombination aus Abschnitt 17.4.5.

Im folgenden Text spreche ich häufig der Einfachheit halber von `get()`- und `set()`-Methoden. Diese müssen nicht immer mit einem solchen Präfix anfangen, sondern es sind ganz allgemein Accessor- und Mutator-Methoden gemeint.

17.4.1 Reduziere die Sichtbarkeit von Attributen

Bekanntlich stellen Attribute den internen Zustand eines Objekts dar. Dieser sollte von außen nicht durch direkte Zugriffe auf diese Attribute sichtbar oder sogar änderbar sein. Dem OO-Grundgedanken der Datenkapselung (vgl. Abschnitt 3.1) folgend, ist es das Ziel bei diesem Refactoring, eine direkte Änderbarkeit von Attributen auf ein Minimum zu reduzieren. Im Idealfall können Details der Implementierung ohne Rückwirkungen auf Nutzer geändert werden. Beispielsweise können Daten entweder als Attribut gehalten, bei Bedarf berechnet oder aus einer externen Quelle gelesen werden. Als Voraussetzung sollte die Sichtbarkeit von Attributen möglichst weit eingeschränkt werden.

Schauen wir dazu auf die in Abbildung 17-2 als Klassendiagramm visualisierte Klasse `Person`, die die zwei Attribute `name` und `age` besitzt und die Ausgangsbasis für dieses Refactoring darstellt. Die Attribute sind zu Demonstrationszwecken `public` ('+') und `protected` ('#').

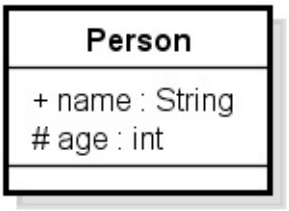


Abbildung 17-2 *Ausgangszustand*

Nachfolgend wird ein `Person`-Objekt erzeugt und direkt auf die Attribute zugegriffen:

```
final Person person = new Person();
person.name = "Meyer";
person.age = 27;

System.out.println("Name='" + person.name + "', Age=" + person.age);
```

Hier findet demnach keine Datenkapselung statt. Um eine solche zu erreichen, sollten möglichst viele Attribute `private` deklariert und über Zugriffsmethoden bereitgestellt werden. Dazu prüfen wir iterativ für jedes Attribut, ob dies möglich ist, indem wir die im Folgenden beschriebenen Schritte ausführen.

Schritt 1: Erzeugen von Zugriffsmethoden

Wir erzeugen Zugriffsmethoden für die Attribute. In Eclipse können wir dazu das Refactoring ENCAPSULATE FIELD aus dem Menü REFACTOR nutzen. Beginnen wir mit dem Attribut `name`. In diesem Fall entstehen die öffentlichen Methoden `getName()` und `setName(String)`, wie es Abbildung 17-3 zeigt.

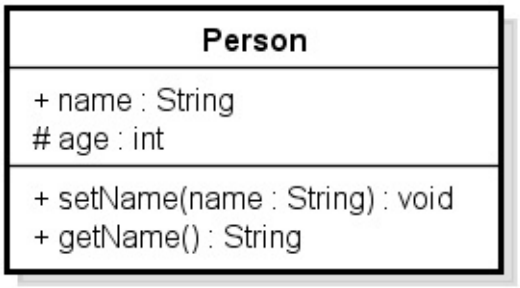


Abbildung 17-3 *Zwischenstand nach dem Einführen der Methoden*

Automatische Korrekturen der Verwendungsstellen

Das Refactoring der IDE passt praktischerweise auch gleich die Aufrufstellen an: Alle Stellen, die bisher lesend auf das Attribut zugegriffen haben, nutzen nun die zuvor erzeugte `getName()`-Methode. Alle schreibenden Zugriffe werden mit der `setName(String)`-Methode ersetzt. Der Sourcecode verändert sich wie folgt:

```
final Person person = new Person();
person.setName("Meyer");
person.age = 27;

System.out.println("Name='" + person.getName() + "', Age=" + person.age);
```

Automatische Anpassung der Sichtbarkeit

Ebenso wie die Zugriffe wird auch die Sichtbarkeit des Attributs `name` auf `private` reduziert, wenn man das Refactoring der IDE nutzt. Das Ergebnis zeigt Abbildung 17-4.

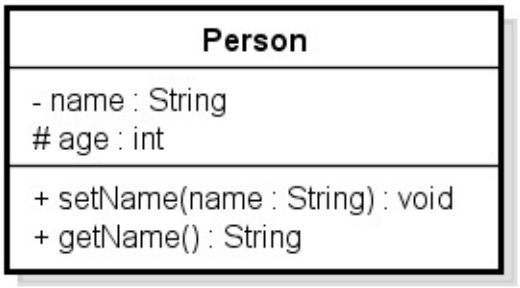


Abbildung 17-4 *Ergebnis von Schritt 1*

Schritt 2: Wiederhole Schritt 1 für alle Attribute

Wiederhole das Vorgehen der Schritte 1 bis 3 für alle Attribute. In diesem Beispiel ist dies lediglich das Attribut `age`. Dadurch entstehen die zwei neuen Zugriffsmethoden `getAge()` und `setAge(int)` mit der Sichtbarkeit `protected`. Deren Einsatz führt zu folgenden Änderungen in nutzendem Sourcecode:

```
final Person person = new Person();
person.setName("Meyer");
person.setAge(27);

System.out.println("Name='" + person.getName() + "', Age=" + person.getAge());
```

Fazit

Diese scheinbar rein kosmetischen Änderungen bewirken eine Verbesserung in der Datenkapselung. Diese erlaubt es, die Attribute zur Speicherung des Namens oder des Alters, wenn nötig, zu verändern. Der Vorteil einer guten Datenkapselung ist, dass sich interne Änderungen nicht in der Schnittstelle für andere Klassen auswirken und man dadurch Folgeänderungen bei Klienten vermeidet. ***Eine bessere Kapselung führt demnach auch zu einer loseren Kopplung***. Wieso ist das gut? Ganz deutlich ist dies für das Attribut `age` erkennbar: Die eingangs gewählte Modellierung des Alters als veränderliches `int`-Attribut ist ungünstig, da man das Alter nicht ändern kann. Es ergibt sich vielmehr aus dem Geburtstag und dem aktuellen Datum. Eine Verbesserung des Designs erreicht man, indem man das Geburtsdatum als Attribut speichert und das Alter dynamisch berechnet, statt dieses in Form eines Attributs zu halten. Dadurch entfällt auch die Methode `setAge(int)`. Stattdessen kann das Geburtsdatum gesetzt werden. Diese Art der Modellierung haben wir bereits in Abschnitt 4.4.2 bei der Besprechung der Klasse `Date` kennengelernt. Abbildung 17-5 zeigt das Klassendiagramm.

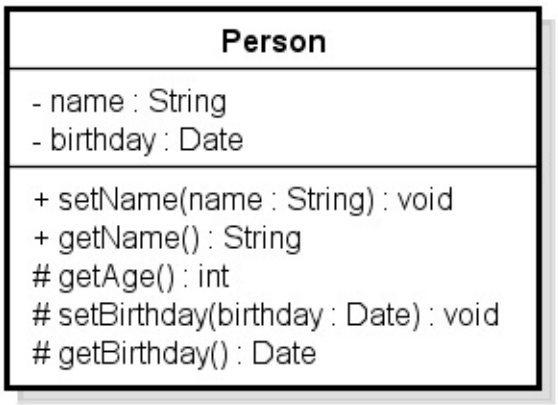


Abbildung 17-5 *Mögliche Designänderungen durch Kapselung*

Sinnvolle weitere Schritte

Die neu entstandenen Zugriffsmethoden sollten in ihrer Sichtbarkeit in einem nachfolgenden Refactoring so weit wie möglich eingeschränkt werden. Das Refactoring REDUZIERE DIE SICHTBARKEIT VON METHODEN stellt in Abschnitt 17.4.3 die Anleitung dazu vor. Häufig können Attribute zudem `final` gemacht werden. Dazu müssen die `set()`-Methoden durch Initialisierungen zum Konstruktionszeitpunkt ersetzt und der Konstruktor um Parameter erweitert werden. Dies beschreibt das Refactoring MINIMIERE VERÄNDERLICHE ATTRIBUTE, das ich nun vorstelle.

17.4.2 Minimiere veränderliche Attribute

Wünschenswert ist es, die Zustandsänderungen eines Objekts möglichst gut zu

kontrollieren und auf ein tatsächlich notwendiges Maß zu beschränken sowie unerwünschte Objektzustände zu vermeiden. Dieses Ziel erreicht man mithilfe dieses Refactorings. Bestenfalls entsteht eine Klasse, die unveränderlich ist.

Die Ausgangssituation in der Realität sieht vielfach jedoch ganz anders aus. Aus Unachtsamkeit werden häufig öffentliche `set()`-Methoden für *eigentlich unveränderliche* Attribute angeboten. Das wurde in Abschnitt 16.2.1 als BAD SMELL: UNNÖTIGERWEISE VERÄNDERLICHE ATTRIBUTE beschrieben. Besonders leicht geschieht dies bei Erweiterungen einer Klasse um neue Attribute: Statt zu deren Initialisierung weitere Parameter in die Konstruktorsignatur aufzunehmen, werden `set()`-Methoden bereitgestellt. Dadurch können aber andere Klassen zu beliebigen Zeitpunkten Änderungen an den Werten der Attribute vornehmen. Das macht es deutlich schwieriger, für einen konsistenten und gültigen Objektzustand zu sorgen. Um die Zustandsänderungen auf wirklich notwendige einzugrenzen, sollte man die Veränderlichkeit von Attributen vermeiden. Dazu muss für alle *privaten* Attribute geprüft werden, ob man sie `final` deklarieren kann, wie es nachfolgend beschrieben wird. Ist dies der Fall, wird die zugehörende `set()`-Methode entfernt und die Konstruktorsignatur erweitert.

Betrachten wir das notwendige Vorgehen anhand der folgenden statischen inneren Klasse `TreeParameter` mit den zwei privaten Attributen `id` und `name` und öffentlichen Zugriffsmethoden. Oftmals muss man für eigene Klassen zuerst mithilfe des Refactorings REDUZIERE DIE SICHTBARKEIT VON ATTRIBUTEN möglichst viele Attribute `private` machen, bevor man dieses Refactoring beginnt.

```
public static final class TreeParameter
{
    private long id = 0;
    private String name = null;

    public TreeParameter()
    {}

    public void setId(final long id)
    {
        this.id = id;
    }

    public long getId()
    {
        return this.id;
    }
    public void setName(final String name)
    {
        this.name = name;
    }
    public String getName()
    {
        return this.name;
    }
}
```

Schritt 1: Prüfe auf Schreibzugriffe – Definiere ein Attribut `final`

Wähle ein privates Attribut aus und definiere dieses zunächst `final`. Ziel ist es, alle Attribute im Konstruktor zu initialisieren. Eine für das entsprechende Attribut vorhandene Defaultinitialisierung wird daher in diesem Schritt auskommentiert und als Zuweisung in den Konstruktor übernommen. Das folgende Listing zeigt Schritt 1 für das Attribut `id`:

```
public static final class TreeParameter
{
    private final long id;          // = 0;
    private String name = null;

    public TreeParameter()
    {
        this.id = 0;
    }
    // …
}
```

Als Folge von Schritt 1 sieht man durch Fehlermeldungen beim Kompilieren, ob innerhalb der Klasse selbst noch schreibende Zugriffe durch Methoden oder Zuweisungen außerhalb des Konstruktors erfolgen. Besonders hilfreich für diesen Schritt ist die automatische Kompilierung in der IDE, weil man so direkt eine Rückmeldung erhält.

Treten keine Kompilierfehler auf, so erfolgte zuvor kein Schreibzugriff auf dieses Attribut und es wurde bislang nur lesend benutzt.[15] Schritt 1 wird dann für das nächste private Attribut ausgeführt. Bei Kompilierfehlern sind zwei Fälle zu unterscheiden. Erfolgt lediglich ein schreibender Zugriff durch eine `set()`-Methode, so ignorieren wir den Fehler zum unerlaubten Schreibzugriff in der `set()`-Methode »`The final field XYZ cannot be assigned`« noch und es geht weiter mit Schritt 2. Finden jedoch Zuweisungen innerhalb mehrerer Methoden statt, so erfolgt eine Rückkorrektur.

Schritt 1 – Rückkorrektur

Man entfernt das zuvor eingeführte `final` wieder. Auch die gegebenenfalls entfernte Defaultinitialisierung wird wieder einkommentiert. Führe als nächstes Schritt 5 aus.

Schritt 2: Prüfe, ob die `set()`-Methode entfernt werden kann

Erreicht man diesen Schritt, so verhindert lediglich eine `set()`-Methode, dass das entsprechende Attribut unveränderlich gemacht werden kann. Man prüft nun, ob überhaupt ein externer Aufruf durch andere Klassen an diese `set()`-Methode erfolgt. Dazu kann man sich entweder über das Tastaturkürzel CTRL+SHIFT+G oder Menü SEARCH –> REFERENCES –> WORKSPACE die Verwendungsstellen ausfindig

machen. Alternativ wird dazu die Methode testweise `private` gekennzeichnet, und man nutzt wieder den Compiler, um Probleme oder Abhängigkeiten aufzudecken[16]:

```
public static final class TreeParameter
{
    private final long id;          // = 0;
    private String name = null;

    public TreeParameter()
    {
        this.id = 0;
    }

    // Verbliebener Schreibzugriff?
    private void setId(final long id)
    {
        this.id = id;
    }
    // …
}
```

Die weiteren Schritte ergeben sich anhand der Ergebnisse der Kompilierung:

- Im besten Fall zeigt der Compiler bzw. die IDE lediglich Fehler in der eigenen Klasse an. Damit hat man externe Aufrufe der `set()`-Methode ausgeschlossen. Aufrufe innerhalb der eigenen Klasse können allerdings noch existieren. Schritt 3 beschreibt das Vorgehen für diesen Fall.
- Gibt es Kompilierfehler außerhalb der eigenen Klasse, so wird die `set()`-Methode von anderen Klassen aufgerufen. Um kein Fass ohne Boden aufzumachen und die Änderungen lokal zu begrenzen, empfiehlt sich die nachfolgend angegebene Rückkorrektur. Mit Schritt 3 kann man nur dann weitermachen, wenn eine sehr kleine Zahl externer Schreibzugriffe existiert, die zudem lediglich Änderungen direkt nach dem Konstruktionsprozess der betrachteten Klasse durchführen, ähnlich zu Folgendem:

  ```
  final TreeParameter treeParams = new TreeParameter();
  treeParams.setId(…);
  treeParams.setName(…);
  ```

  In Schritt 4 wird der Konstruktor noch um einen Übergabeparameter erweitert.

Schritt 2 – Rückkorrektur

Es müssen sowohl die ursprüngliche Sichtbarkeit der Methode wiederhergestellt als auch die Definition des Attributs als `final` entfernt werden. Zudem wird die gegebenenfalls entfernte Defaultinitialisierung wieder eingefügt, d. h., die Kommentarzeichen davor entfernt. Führe als nächstes Schritt 5 aus.

Schritt 3: Prüfen interner Schreibzugriffe

Erfolgen keine externen Zugriffe, oder können die aufrufenden Programmstellen

adäquat angepasst werden, so prüft man abschließend auf Schreibzugriffe innerhalb der Klasse selbst. Gibt es mehrere davon, so ist das Attribut veränderlich und es erfolgt die im Schritt 2 angegebene Rückkorrektur. Existiert jedoch nur ein Schreibzugriff, entfernt man die `set()` -Methode und es folgt Schritt 4.

Schritt 4: Anpassung des Konstruktors

Das Attribut muss jetzt zur korrekten Initialisierung im Konstruktor zugewiesen werden. Die Konstruktorsignatur wird dazu um einen korrespondierenden Übergabeparameter erweitert und der Konstruktorblock um eine entsprechende Zuweisung ergänzt. Diese ersetzt die in Schritt 1 eingeführte Zuweisung im Konstruktor.

```
public static final class TreeParameter
{
    private final long id;
    private String name = null;

    public TreeParameter(final long id)
    {
        this.id = id;
    }

    // …
}
```

Schritt 5: Wiederhole Schritt 1 bis 4 für alle privaten Attribute

Wiederhole das Vorgehen der Schritte 1 bis 4 für alle verbliebenen privaten Attribute. Danach ergibt sich für das Beispiel der Klasse `TreeParameter` folgende Realisierung:

```
public static final class TreeParameter
{
    private final long id;
    private final String name;

    public TreeParameter(final long id, final String name)
    {
        this.id = id;
        this.name = name;
    }

    public long     getId()         { return this.id; }
    public String   getName()       { return this.name; }
}
```

Fazit

Als Ergebnis dieses Refactorings erhalten wir für dieses Beispiel eine Klasse, die keine Zustandsänderungen mehr erlaubt. Die Gefahr von unvorhersehbaren und unerwünschten Objektzuständen konnte vollständig eliminiert werden. Nicht in

jedem Fall erreicht man eine unveränderliche Klasse, jedoch immer eine, die nur noch wirklich notwendige Zustandsänderungen erlaubt.

Nach Abschluss dieses Refactorings sind möglicherweise einige Attribute noch öffentlich über Accessor-Methoden zugänglich, obwohl dies eventuell nicht notwendig ist. Es bieten sich zwei weitere Refactorings im Anschluss an.

Sinnvolle weitere Schritte

Methoden können in ihrer Sichtbarkeit eingeschränkt werden, um eine gute Datenkapselung und eine schlankere Schnittstelle zu erzielen. Man folgt dabei den im nachfolgenden Abschnitt 17.4.3 beschriebenen Schritten des Refactorings REDUZIERE DIE SICHTBARKEIT VON METHODEN. Darüber hinaus empfiehlt es sich, für öffentliche Methoden – zumindest für solche an Schnittstellen zu anderen Systemen oder Komponenten – mithilfe von Parameterprüfungen gültige Eingaben sicherzustellen. Eine Anleitung dazu wird im Refactoring ÜBERPRÜFE EINGABEPARAMETER in Abschnitt 17.5.2 beschrieben.

17.4.3 Reduziere die Sichtbarkeit von Methoden

Der OO-Kerngedanke der Kapselung ist, die Sichtbarkeit von Attributen und Methoden möglichst einzuschränken. Der Zustand eines Objekts kann folglich nicht mehr so sehr von außen bestimmt werden. Veränderungen sollten nur durch eine Reihe von verhaltensdefinierenden Business-Methoden erfolgen. Die Schritte dazu beschreibt das Refactoring ERSETZE MUTATOR- DURCH BUSINESS-METHODE in Abschnitt 17.4.4. Alle nicht in dieser Schnittstelle enthaltenen Methoden versucht man, `private` oder `protected` zu definieren. Dazu sind folgende zwei Vorgehensweisen möglich – bitte beachten Sie aber die später genannten Einschränkungen, bevor Sie überall im Programm die Sichtbarkeit ändern.

1. **Statische Sourcecode-Analyse einsetzen** – Man nutzt Tools zur statischen Sourcecode-Analyse, um unbenutzte Methoden zu ermitteln. Während diese Funktionalität bereits in IntelliJ IDEA integriert ist, bietet es sich für Eclipse an, das Plugin UCDETECTOR[17] im Marketplace nachzuinstallieren. Dieses Tool findet unbenutzten Sourcecode sowie Methoden, deren Sichtbarkeit verringert werden kann. Es ermittelt auch, ob Methoden oder Attribute `final` sein können. Nützlich ist zudem, dass Methoden gefunden werden, die lediglich von Tests aufgerufen werden und damit nicht zur Anwendungsfunktionalität beitragen. Als praktisches Feature werden QUICK-FIX-Vorschläge bereitgestellt.
2. **Methodensichtbarkeit auf Verdacht ändern** – Hierbei gibt es die beiden folgenden Varianten:
   - Ausgehend von der momentanen Sichtbarkeit verringert man diese schrittweise, also etwa von `public` -> `protected` -> Package-private -> `private`, bis

entweder das Kompilieren fehlschlägt oder `private` erreicht ist. Man nimmt die kleinste Sichtbarkeit, bei der kein Kompilierfehler auftritt.

- Man setzt die Methodensichtbarkeit zunächst auf `private`. Bei Bedarf erhöht man schrittweise die Sichtbarkeit, bis das Programm wieder kompiliert.

Anekdote: Positive Auswirkungen dieses Refactorings

Mir wurde ein umfangreicheres Programmpaket übertragen, indem vieles zu kompliziert realisiert war. Das lag unter anderem daran, dass nahezu keine Kapselung stattfand und es vor `public`-Methoden wimmelte. Nach einigen Tagen hatte ich die Sichtbarkeit vieler Methoden gemäß diesem Refactoring reduzieren können. Dadurch konnten ca. 20 % der Methoden als unbenutzt erkannt werden und entfallen.

Aufdecken komplexer und nutzloser Zyklen Beim vorherigen Analysieren der Klassen hatte ich bereits das Gefühl, dass noch mehr Optimierungspotenzial existierte. Daher verfolgte ich die Aufrufe einiger Methoden und erkannte, dass diese eine ringförmige Aufrufhierarchie über einige Delegationen in verschiedenen Klassen enthielten. Durch aufwendiges Nachverfolgen und ein wenig Glück konnte ich etwa weitere 20 % nutzlosen Sourcecode entfernen.

Verbesserungen Die verbliebenen Klassen waren nun wesentlich übersichtlicher. An vielen Stellen konnten Attribute `final` definiert werden, wodurch wiederum die Anzahl der Zustandsänderungen begrenzt und die Klasse somit besser handhabbar wurde.

Einschränkungen

Die Reduktion der Sichtbarkeit kann man nur *sicher* einsetzen, wenn man alle nutzenden Klassen im Zugriff hat und außerdem weiß, dass keine Aufrufe mit Reflection erfolgen. Im ersten Fall würde man durch Änderungen der Sichtbarkeit eventuell Kompilierfehler in externen Komponenten verursachen. Im zweiten Fall kommt es eventuell zu Laufzeitfehlern, da benötigte Informationen über Reflection nicht zugreifbar sind.[18]

Dieses Refactoring ist außerdem nur für solche Methoden sinnvoll, die keine Kernfunktionalität der Klasse darstellen und daher auch nicht in der öffentlichen Schnittstelle für andere Klassen bereitgestellt werden sollen. Wendet man dieses Vorgehen wahllos für alle Methoden der Schnittstelle an, so führt dies nicht unbedingt sofort zu Problemen, weil die Reduktion der Sichtbarkeit momentan noch unbenutzter Methoden keinen Kompilierfehler verursacht. Allerdings wird später für Nutzer der Einsatz der Klasse erschwert, weil einige Methoden dann nicht in der Schnittstelle ansprechbar sind. Deswegen ist dieses Refactoring bevorzugt dann einzusetzen, wenn das System schon (fast) vollständig implementiert ist. Andernfalls weiß man z. B. lediglich, dass eine gegebene Methode momentan nicht außerhalb der Klasse benötigt wird.

17.4.4 Ersetze Mutator- durch Business-Methode

Dieses Refactoring hilft dabei, den Zustand eines Objekts vor feingranularen Änderungen zu schützen, und hat den objektorientierten Gedanken der Definition von Verhalten durch Business-Methoden im Blick und adressiert den Smell FEATURE ENVY.

Benutzen wir hier nochmals grafische Figuren als Beispiel. Zum Setzen der X- und Y-Koordinaten könnte man die Methoden `setX(int)` und `setY(int)` anbieten. Analog dazu könnten die Methoden `setEndX(int)` und `setEndY(int)` Breite und Höhe setzen. Die Schnittstelle ist akzeptabel, suggeriert aber einen Datenbehältercharakter und erlaubt feingranulare Zustandsänderungen von außen. Je nach Anwendungsfall, insbesondere dann, wenn Aufrufer mehrere Methoden am Zielobjekt nahe hintereinander aufrufen, ist es mitunter besser verständlich, Methoden zu bündeln. Im Beispiel würde man Methoden anbieten, mit denen die Startposition oder die Größe einer Figur geändert werden kann (vgl. Abbildung 17-6). Dabei ist aber zu bedenken, dass z. B. in einem Zeichenprogramm die feingranulareren Aufrufe besser geeignet sind.

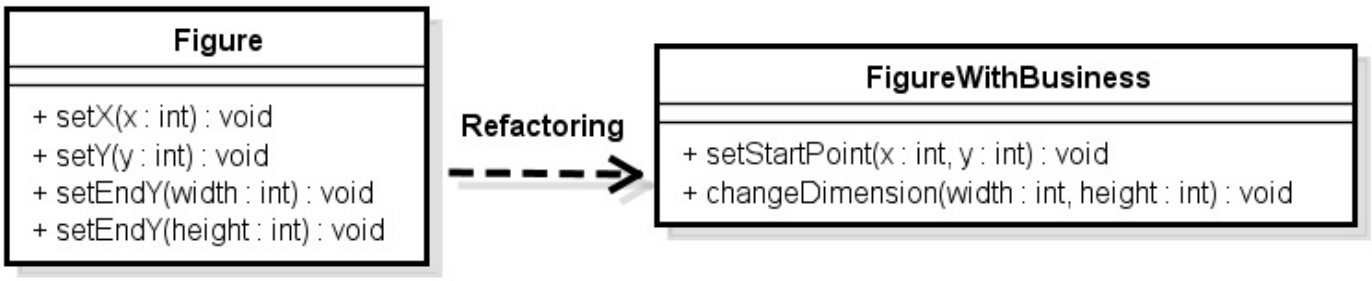


Abbildung 17-6 *Einführen von Business-Methoden*

17.4.5 Minimiere Zustandsänderungen

Je weniger Zustandsinformationen in einem Objekt variabel sind, desto besser lassen sich Modifikationen daran kontrollieren und nachvollziehen. Um die Veränderlichkeit eines Objekts möglichst gering zu halten, kann man die zuvor vorgestellten Refactorings kombinieren und führt folgende Schritte durch:

1. Wir reduzieren die Anzahl und die Sichtbarkeit der veränderlichen Attribute. Dazu nutzen wir die Refactorings REDUZIERE DIE SICHTBARKEIT VON ATTRIBUTEN aus Abschnitt 17.4.1 und MINIMIERE VERÄNDERLICHE ATTRIBUTE aus Abschnitt 17.4.2.
2. Wir halten die Anzahl und die Sichtbarkeit von `set()`-Methoden möglichst gering. Dazu wenden wir das Refactoring REDUZIERE DIE SICHTBARKEIT VON METHODEN aus Abschnitt 17.4.3 auf die `set()`-Methoden an. Zudem können wir Zugriffsmethoden durch Business-Methoden ersetzen, wie dies im Refactoring ERSETZE MUTATOR-DURCH BUSINESS-METHODE in Abschnitt 17.4.4 dargestellt ist.

17.4.6 Führe ein Interface ein

Dieses Refactoring beschreibt, wie wir aus einer bestehenden Klasse ohne Interface eine Klasse machen, die ein Interface anbietet. Bevor ich erläutere, dass dies jedoch nicht in jedem Fall sinnvoll ist, beschreibe ich das Vorgehen bei diesem Refactoring. Dabei bündeln wir die gewünschten öffentlichen Methoden in einem Interface. In Eclipse können wir dazu das Refactoring EXTRACT INTERFACE aus dem Menü REFACTOR nutzen.

In diesem Beispiel enthält das neu entstehende Interface `NewInterface` der Einfachheit halber alle `public`-Methoden der ursprünglichen Klasse `ClassWithoutInterface`. Das zugehörige UML-Diagramm zeigt Abbildung 17-7.

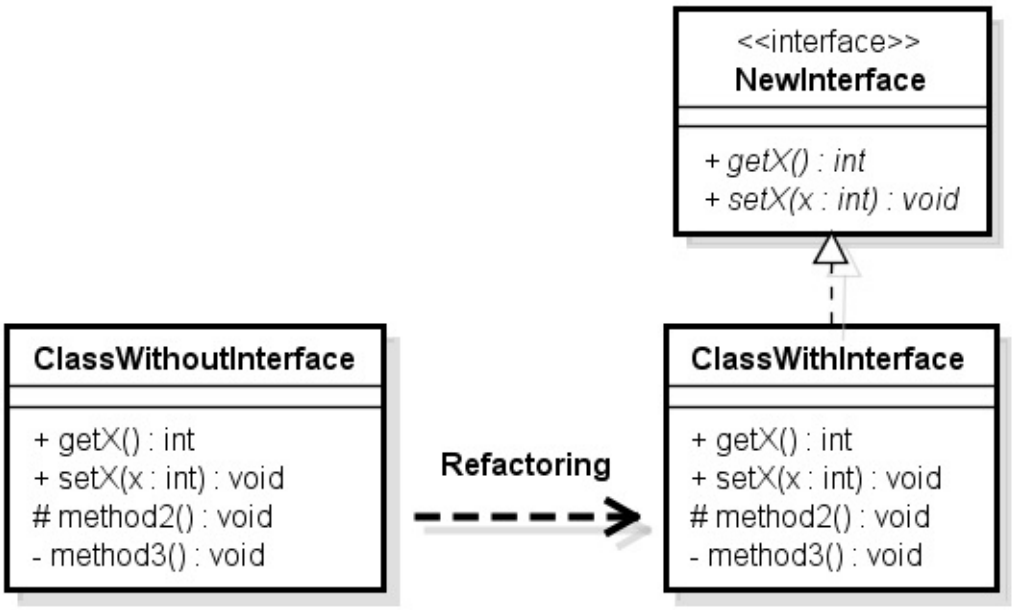


Abbildung 17-7 *Einführen eines Interface*

Für komplexere Klassen ist dies häufig nur ein erster Schritt, um die nach außen sichtbare Funktionalität zu definieren. Ergänzend kann das folgende Refactoring SPALTE EIN INTERFACE AUF genutzt werden, um eine semantische Strukturierung vorzunehmen.

Achtung

Ein Interface ist ***kein Selbstzweck*** und nicht immer hilfreich: Für den Fall, dass eine Klasse nur intern in einem Package benutzt wird, sorgt ein Interface oftmals lediglich für mehr Komplexität und mehr Sourcecode, allerdings ohne größeren Nutzen. An Systemgrenzen oder weil das Design eine Abstraktion erfordert, ***können Interfaces für eine klare Trennung zwischen Angebot und Realisierung von Funktionalitäten sorgen***. Dadurch erhält man mehr Klarheit über die angebotene Funktionalität und erzielt auch eine losere Kopplung. Zudem lässt sich so eine konkrete Realisierung verbergen oder austauschbar gestalten. Vorteilhaft ist dies insbesondere für Klassen,

die aus anderen Packages verwendet werden sollen.

17.4.7 Spalte ein Interface auf

Manchmal stellt man im Verlauf der Entwicklung fest, dass ein Interface besser in zwei oder mehr Interfaces aufgeteilt werden sollte. Dazu dient dieses Refactoring.

In diesem Beispiel soll das Interface `GlobalInterface` aufgespalten werden, da es sowohl Lesezugriffe als auch Business-Funktionalität anbietet. Als Folge entstehen ein Datenbehälter-Interface (`DataProviderInterface`) mit `get()`-Methoden und ein Interface (`BusinessInterface`) mit Business-Methoden (siehe Abbildung 17-8).

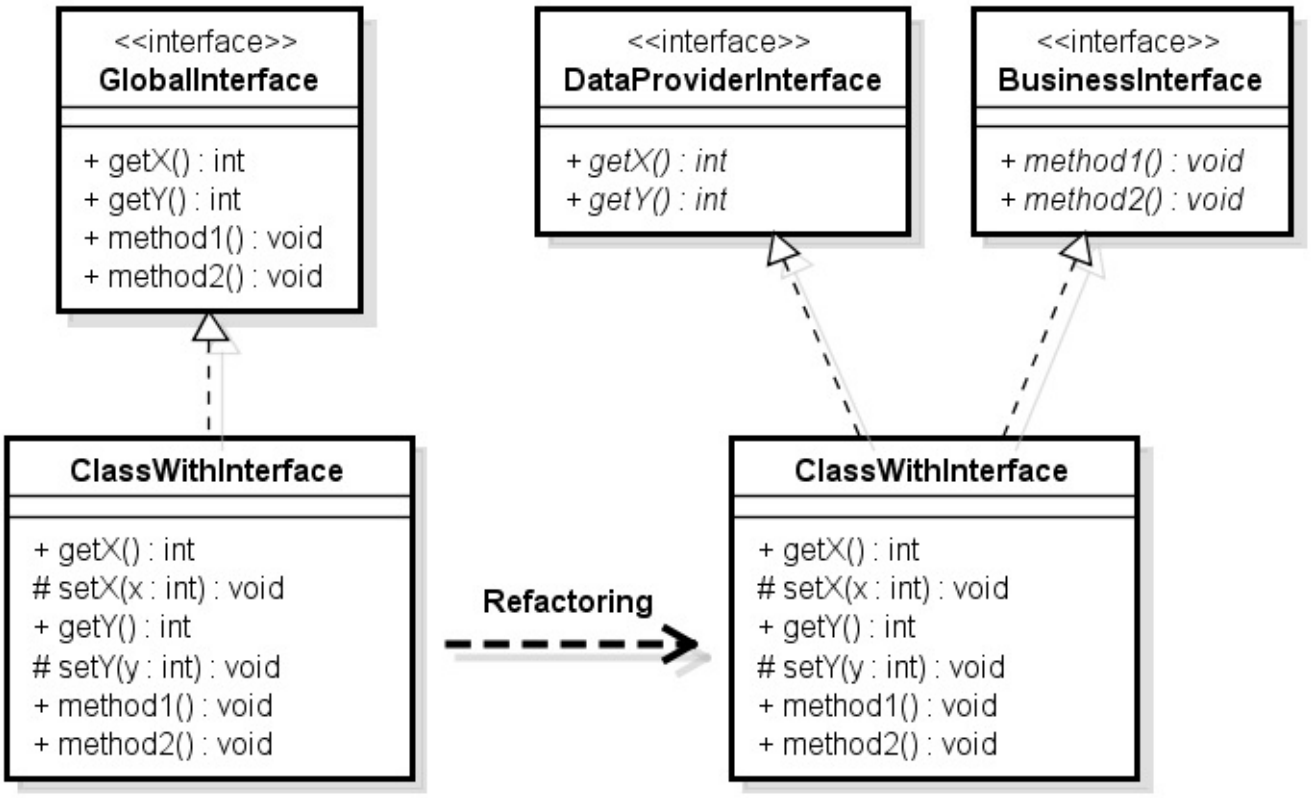


Abbildung 17-8 *Einführen mehrerer Interfaces*

Mit diesem Refactoring kann man eine Trennung von Zuständigkeiten realisieren: Die ehemals auf unterschiedlichen semantischen Ebenen stattfindenden Zugriffe werden dann durch zwei Interfaces mit jeweils verschiedenen Aufgaben ausgedrückt. In diesen Interfaces gruppieren wir die Methoden nach ihrer Zugehörigkeit.

17.4.8 Führe ein Read-only-Interface ein

In komplexeren Systemen kann es zum Schutz vor ungewollten Änderungen durch andere Komponenten sinnvoll sein, ein Read-only-Interface anzubieten. Stellen wir uns folgendes Szenario vor: Eine Klasse wird von Package-externen Klassen lesend genutzt und von Package-internen Klassen schreibend mit Daten versorgt.

Ohne Zugriffsschutz könnten die Modelldaten nicht nur Package-intern, sondern ebenso durch Package-externe Klienten verändert werden. Um dies zu verhindern, kann man ein Read-only-Interface einführen, das durch `get()`-Methoden ausschließlich lesenden Zugriff bietet und den Package-externen Klassen bereitgestellt wird. Die Transformation ist in Abbildung 17-9 visualisiert.

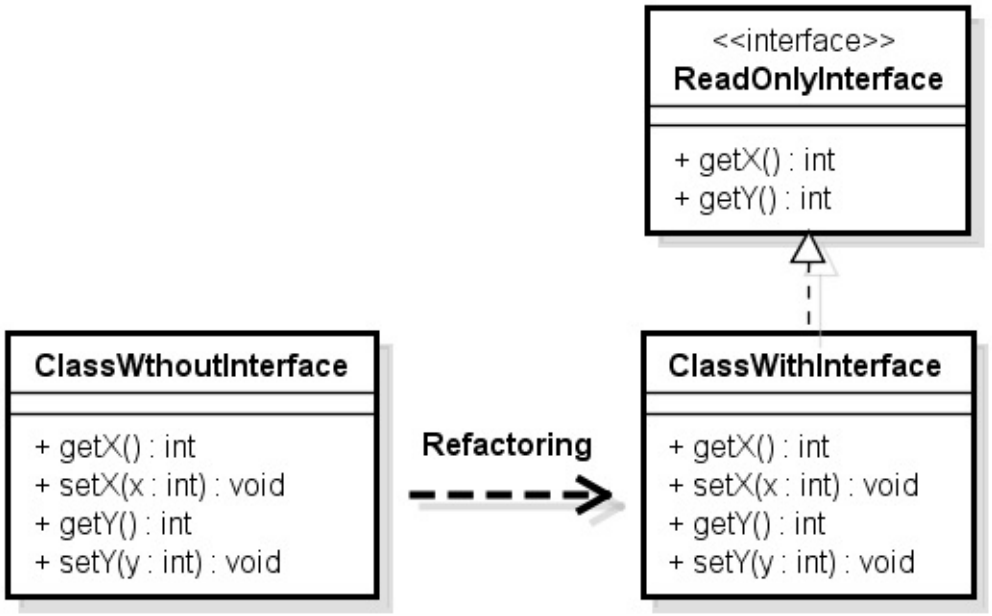


Abbildung 17-9 *Einführen eines Read-only-Interface*

Hinweis

Bedenken Sie, dass es manchmal bereits reicht, wenn man die modifizierenden Methoden Package-private definiert. Das gilt ebenfalls für Methoden, die nicht für Package-externe Klienten zugreifbar sein sollten. Das Read-only-Interface ist insbesondere für lose gekoppelte Komponenten sinnvoll.

17.4.9 Führe ein Read-Write-Interface ein

Wir haben gesehen, dass man mithilfe von Read-only-Interfaces schreibende Zugriffe durch Package-externe Klassen verhindern kann. Manchmal sollen aber einige Komponenten doch schreibend zugreifen, allerdings unter der Prämisse, diesen keinen direkten Zugriff auf konkrete Objekte anzubieten. Dann kann man ein spezielles Interface definieren, das lediglich Schreibzugriffe bereitstellt.

Problematisch an einer strikten Trennung zwischen rein lesenden und rein schreibenden Zugriffen ist, dass dies zu unhandlich wird, weil man dann jeweils eine Referenz unterschiedlichen Typs auf die gleiche Klasse benötigt. Vielfach bietet sich ein Read-Write-Interface an, das das Read-only-Interface um die Möglichkeit von Schreibzugriffen erweitert. Dadurch hat man im Read-Write-Interface immer lesenden und schreibenden Zugriff. Die Verwaltung und Nutzung wird einfacher, da man nur noch eine Referenz auf das Read-Write-Interface benötigt, statt, je nach

Aufgabe, zwei Referenzen auf das Read-only- bzw. Write-Interface. Abbildung 17-10 zeigt diese Varianten.

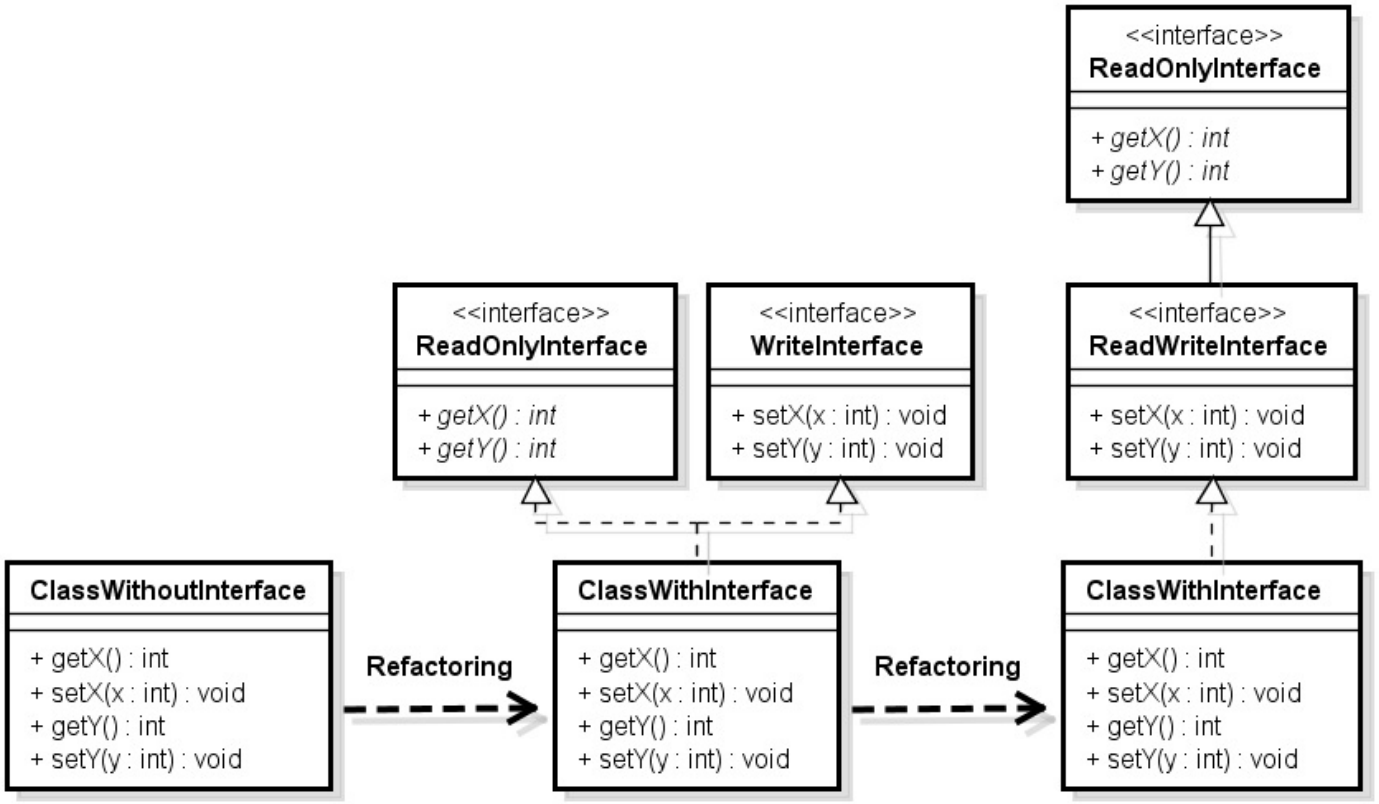


Abbildung 17-10 *Einführen eines Read-Write-Interface*

Einschränkung

Wie schon zuvor argumentiert, benötigt man derartige Konstrukte insbesondere dann, wenn Komponenten möglichst unabhängig voneinander sein sollen und eventuell von verschiedenen Teams oder sogar Firmen entwickelt werden. Dann sind solche Schnittstellen mit klaren Verantwortlichkeiten hilfreich.

17.4.10 Lagere Funktionalität in Hilfsmethoden aus

Dieses Refactoring dient dazu, den Sourcecode zu strukturieren, indem ähnliche und wiederkehrende Programmstücke erkannt und in Hilfsmethoden ausgelagert werden. In Eclipse können wir dazu das Refactoring EXTRACT METHOD nutzen. Nach einer solchen Überarbeitung folgt der Sourcecode dadurch besser dem DRY-Prinzip (»Don't Repeat Yourself«, vgl. »Der Pragmatische Programmierer« [48]).

Die folgende Methode `getFilterInfo(InformationProvider)` prüft verschiedene Bedingungen und baut korrespondierende Informationstexte zusammen:

```
public static String getFilterInfo(final InformationProvider infoProvider)
{
    final StringBuilder builder = new StringBuilder();

    if (isSelected(Option.NAME_CONTAINS))
    {
```

```
        if (builder.length() > 0)
        {
            builder.append(", ");
        }
        builder.append(getLangString("label.name") + "='" +
                       infoProvider.getName() + "'");
        }

        // weitere Auswertungen …

        if (isSelected(Option.JOB_ID_CONTAINS))
        {
            if (builder.length() > 0)
            {
                builder.append(", ");
            }
            builder.append(getLangString("label.jobid") + "='" +
                           infoProvider.getJobID() + "'");
        }

        return builder.toString();
}
```

Die hier genutzte Methode `isSelected(Option)` prüft, ob eine Filterbedingung aktiviert ist. Anschließend werden die Informationen zu dieser Filterbedingung in ein `StringBuilder`-Objekt eingefügt. Die Methode `getFilterInfo(InformationProvider)` ist bereits relativ übersichtlich, besitzt aber kleinere Sprünge in den Abstraktionsebenen. Darüber hinaus existieren einige ähnliche Zeilen.

Schritt 1: Erzeugen von Hilfsmethoden

Wir erkennen zwei immer wiederkehrende Sourcecode-Abschnitte. Zum einen wird jeweils ein Komma und ein Leerzeichen an die Variable `builder` angehängt, sofern dort bereits Zeichen gespeichert sind. Das lässt sich einfach in eine Methode `appendCommaWhenNotEmpty(StringBuilder)` herausfaktorieren:

```
private static void appendCommaWhenNotEmpty(final StringBuilder builder)
{
    if (builder.length() > 0)
    {
        builder.append(", ");
    }
}
```

Zum anderen werden die Informationstexte zu den Filterbedingungen immer wieder auf ähnliche Art und Weise dem `StringBuilder`-Objekt hinzugefügt. Es erfordert etwas mehr Aufwand, diese Funktionalität in eine Methode auszulagern, weil die Unterschiede jeweils im Zugriffsschlüssel auf die Textressourcen und dem entsprechenden Wert liegen. Wir schreiben folgende Methode:

```
private static void appendFilterText(final StringBuilder builder,
                                     final String resourceID,
                                     final Object value)
{
    builder.append(getLangString(resourceID) + "='" + value + "'");
```

```
}
```

Schritt 2: Anpassen der Aufrufstellen

Nutzt man diese beiden Hilfsmethoden, so lässt sich der Sourcecode kürzer und in einer einheitlichen Abstraktionsebene darstellen. Damit verbessert sich die Struktur und die Lesbarkeit des Sourcecodes.

```
public static String getFilterInfo(final InformationProvider infoProvider)
{
    final StringBuilder builder = new StringBuilder();

    if (isSelected(Option.NAME_CONTAINS))
    {
        appendCommaWhenNotEmpty(builder);
        appendFilterText(builder, "label.name", infoProvider.getName());
    }

    // weitere Auswertungen …

    if (isSelected(Option.JOB_ID_CONTAINS))
    {
        appendCommaWhenNotEmpty(builder);
        appendFilterText(builder, "label.jobid", infoProvider.getJobID());
    }

    return builder.toString();
}
```

17.4.11 Trenne Informationsbeschaffung und -verarbeitung

Vielfach ist es vorteilhaft, die Beschaffung von Informationen von ihrer Verarbeitung und Speicherung zu trennen. Diese Teilaufgaben lassen sich im Allgemeinen recht gut in separate Methoden auslagern, wodurch die Orthogonalität (Kombinierbarkeit) und Wiederverwendbarkeit erhöht sowie ungültige Zwischenzustände vermieden werden.

Betrachten wir eine Methode, in der Informationsbeschaffung und -verarbeitung noch miteinander verknüpft sind:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            model.setState(elementVO.getQueueStatus());      // 1
        }
        else
        {
            model.setState(QueueStates.UNKNOWN);             // 2
        }
        model.setQueuedJobs(elementVO.getEntryCount());      // 3
```

```
    }
    else
    {
        model.setState(QueueStates.UNREACHABLE);            // 4
        model.setQueuedJobs(0);                             // 5
    }
}
```

Daten werden aus einem DATA TRANSFER OBJECT (DTO) bzw. VALUE OBJECT (VO) `elementVO` ausgelesen und in einem Modell `model` gespeichert. Der Sourcecode enthält dazu fünf verschiedene Zeilen mit `set()`-Methoden. Die Informationsbeschaffung ist in den restlichen Methoden über `get()`-Aufrufe an `elementVO` verteilt.

Zunächst scheint dies nicht wirklich problematisch zu sein. Bedenken wir aber Folgendes: Prinzipiell kann bei jedem Methodenaufruf ein Fehler auftreten und als Folge eine Exception ausgelöst werden[19]. Dadurch kann es zu Inkonsistenzen durch teilweise initialisierte Objekte kommen. Der folgende Praxistipp geht auf die Thematik genauer ein. Die gezeigte Art der Informationsbeschaffung ist auch beim Einsatz von Multithreading problematisch und kann zu Inkonsistenzen führen (vgl. Kapitel 7).

Unser Ziel ist es, eine bessere Trennung zu erreichen. Im Idealfall können zunächst alle Informationen ermittelt, in einem Immutable-Objekt gespeichert und anschließend verarbeitet werden. Eine Umsetzung in Pseudocode sieht wie folgt aus:

```
final DataValueContainer values = retrieveValues();
processValues(values);
storeValues(values);
```

Achtung: Probleme durch zu feingranulare `set()`-Methoden

Werden für jeden zu verändernden Wert entsprechende `set()`-Methoden einzeln aufgerufen, so kann dies zu Problemen führen.

Einfluss auf Objektintegrität Eine schrittweise Veränderung von Attributen kann zu Zwischenzuständen führen, die keinen gültigen Objektzustand darstellen. Besonders groß ist die Gefahr dafür beim Einsatz von Methoden, die Exceptions auslösen können, etwa in verteilten Systemen mit Remote Calls. Treten Exceptions auf, so kann ein Objekt sehr schnell in einen inkonsistenten Zustand versetzt werden, etwa weil die Verarbeitung durch eine Exception abrupt beendet wird. Eine Trennung von Informationsbeschaffung und -verarbeitung ist dann vorteilhaft: Treten Fehler bei der Informationsbeschaffung auf, kann auf ein Setzen der Werte verzichtet und das Objekt somit in einem konsistenten Zustand belassen werden.

Einfluss auf BEOBACHTER Werden Zustandsänderungen gemäß dem BEOBACHTER-Muster (vgl. Abschnitt 18.3.7) verarbeitet, so bereiten feingranulare `set()`-Methoden neben ständigen Zustandsmeldungen zusätzlich das Problem, dass sie ungültige Zwischenzustände sichtbar machen.

Analogie aus dem realen Leben Warum potenziell teilinitialisierte Objekte nicht

optimal sind, kann man sich mit folgender Analogie aus dem realen Leben verdeutlichen: Morgens verlässt man das Haus ja auch erst, nachdem man sich vollständig angezogen hat und den Rasierschaum oder die Zahnpasta aus dem Gesicht gewischt hat, also nur in einem validen Zustand. Tut man dies nicht, so wirkt das im besten Fall befremdlich auf die Umwelt. Denken Sie an diese Analogie, um sich die Wichtigkeit eines sinnvoll initialisierten Objekts zu vergegenwärtigen und möglichst sparsam zustandsändernde `set()`-Methoden für eigene Klassen anzubieten.

Schritt 1: Erkennen von veränderlichen Elementen und Einführen von Hilfsvariablen

Wir untersuchen den Sourcecode nach veränderlichen Werten und erkennen, dass im `if-` bzw. `else`-Zweig jeweils zwei Variablen durch `set()`-Aufrufe verändert werden. Wir führen daher entsprechende Hilfsvariablen `state` und `entryCount` ein. Dabei hilft uns das Basis-Refactoring EXTRACT LOCAL VARIABLE mit dem Tastaturkürzel ALT+SHIFT+L. Dies erfolgt getrennt für beide Blöcke. Ein derart kleinteiliges Vorgehen wäre hier eventuell nicht notwendig, wenn man schon sehr geübt ist. Aber je komplexer der Ausschnitt des Sourcecodes und je mehr verschiedene Variablen modifiziert werden, desto besser ist es, kleine überschaubare und am besten durch die IDE unterstützte Schritte zu machen. Durch EXTRACT LOCAL VARIABLE werden die direkten Aufrufe der `set()`-Methoden durch Zuweisungen an die neu eingeführten lokalen Variablen und einen anschließenden Aufruf an die jeweilige `set()`-Methode des Modells ersetzt. Danach ordnen wir noch mit ALT+UP/DOWN ein paar Zeilen um und fassen im oberen Block die Variablen für `state` und `entryCount` zusammen:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    if (elementVO != null)
    {
        QueueStates state;
        int entryCount;

        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
            model.setState(state);
        }
        else
        {
            state = QueueStates.UNKNOWN;
            model.setState(state);
        }
        entryCount = elementVO.getEntryCount();
        model.setQueuedJobs(entryCount);
    }
    else
    {
        QueueStates state = QueueStates.UNREACHABLE;
        model.setState(state);
```

```
            int entryCount = 0;
            model.setQueuedJobs(entryCount);
        }
}
```

Trotz dieser umfangreicheren Modifikationen haben wir am Programmverhalten nichts geändert. Der Sourcecode ist allerdings länger geworden. Diesen offensichtlichen Nachteil akzeptieren wir zunächst, da eine entscheidende Verbesserung durch diese Umformung erreicht wurde: ***Die Informationsbeschaffung und der Transfer ins Modell erfolgen nun in zwei Schritten***. Diese Transformation bildet die Grundlage für die weiteren Modifikationen.

Schritt 2: Zusammenfassen der `set()`-Methoden zu Blöcken

Um die Informationsbeschaffung und -verarbeitung weiter zu entkoppeln, wollen wir die Aufrufe der `set()`-Methoden zu Einheiten am jeweiligen Ende eines Blocks zusammenfassen. Dies ist immer dann möglich, wenn der Kontrollfluss den Block vollständig durchläuft. Durch das Verlagern (Zeilen mit ALT+UP/DOWN bewegen) und das Zusammenfassen von `set()`-Methoden erhält man folgenden Zwischenstand:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    if (elementVO != null)
    {
        final QueueStates state;
        int entryCount;

        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
        }
        else
        {
            state = QueueStates.UNKNOWN;
        }
        entryCount = elementVO.getEntryCount();

        model.setState(state);
        model.setQueuedJobs(entryCount);
    }
    else
    {
        final QueueStates state = QueueStates.UNREACHABLE;
        final int entryCount = 0;

        model.setState(state);
        model.setQueuedJobs(entryCount);
    }
}
```

Schwierigkeit: Mehrere Ausgänge aus Blöcken bzw. in einer Methode

Mehrfache `return`-Anweisungen innerhalb komplexer und geschachtelter `if`-Anweisungen erschweren häufig das Verständnis und hätten den gezeigten Schritt deutlich schwieriger – eventuell sogar unmöglich – gemacht. Die negative Aussage zu mehrfachen `return`-Anweisungen gilt jedoch nicht für Shortcut-Returns am Methodenanfang im Rahmen von Fehlerabfragen. Diese tragen fast immer zu einer Vereinfachung der Logik bei.

Schritt 3: Auftrennen von Informationsbeschaffung und -verarbeitung

Zur Trennung von Informationsbeschaffung und -verarbeitung strukturieren wir den Sourcecode um. Wir wollen die Blöcke mit den `set()`-Aufrufen zusammenfassen und definieren dazu Hilfsvariablen im äußersten Block. Nachdem der gesamte Zustand ermittelt wurde, werden dann die `set()`-Methoden ausgeführt. Gleichzeitig nehmen wir eine Vereinfachung vor: Die Initialisierung aus dem `else`-Zweig für den Fehlerfall (`elementVO == null`) nutzen wir als Startbelegung der Variablen:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    QueueStates state = QueueStates.UNREACHABLE;
    int entryCount = 0;

    // Informationsbeschaffung
    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
        }
        else
        {
            state = QueueStates.UNKNOWN;
        }
        entryCount = elementVO.getEntryCount();
    }

    // Verarbeitung
    model.setState(state);
    model.setQueuedJobs(entryCount);
}
```

Schritt 4: Kapseln von Informationsbeschaffung und -verarbeitung

Informationsbeschaffung und -verarbeitung sind nun getrennt und können in eigene Methoden ausgelagert werden. Dazu bietet es sich an, die lokalen Variablen zu Value Objects zusammenzufassen. Doch schauen wir uns dies schrittweise an.

Zunächst nutzen wir das Basis-Refactoring EXTRACT METHOD, um die per Kommentar markierten Teile jeweils als Methode zu extrahieren. Wir beginnen mit

den beiden Zeilen zur Verarbeitung, die wir in die Methode `storeInModel()` auslagern:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    QueueStates state = QueueStates.UNREACHABLE;
    int entryCount = 0;

    // Informationsbeschaffung
    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            state = elementVO.getQueueStatus();
        }
        else
        {
            state = QueueStates.UNKNOWN;
        }
        entryCount = elementVO.getEntryCount();
    }

    // Verarbeitung
    storeInModel(state, entryCount);
}

private void storeInModel(final QueueStates state, final int entryCount)
{
    model.setState(state);
    model.setQueuedJobs(entryCount);
}
```

Wenn wir nun die Zeilen zur Informationsbeschaffung startend vom Kommentar markieren und ebenso extrahieren wollen, dann liefert uns Eclipse die Fehlermeldung, dass der Rückgabewert nicht eindeutig bestimmbar ist und mehr als eine lokale Variable als Kandidat existiert. Was nun?

Wir hatten bereits vor, die beiden semantisch zusammenhängenden Variablen in einer Klasse zu vereinen. Praktischerweise gibt es dazu das Basis-Refactoring INTRODUCE PARAMETER OBJECT. Als Vorbereitung selektieren wir die Methode `storeInModel()` und führen danach das Refactoring aus. Es erscheint ein Dialog. Dort müssen wir lediglich noch den gewünschten Klassen- und Parameternamen wie in Abbildung 17-11 gezeigt eintragen.

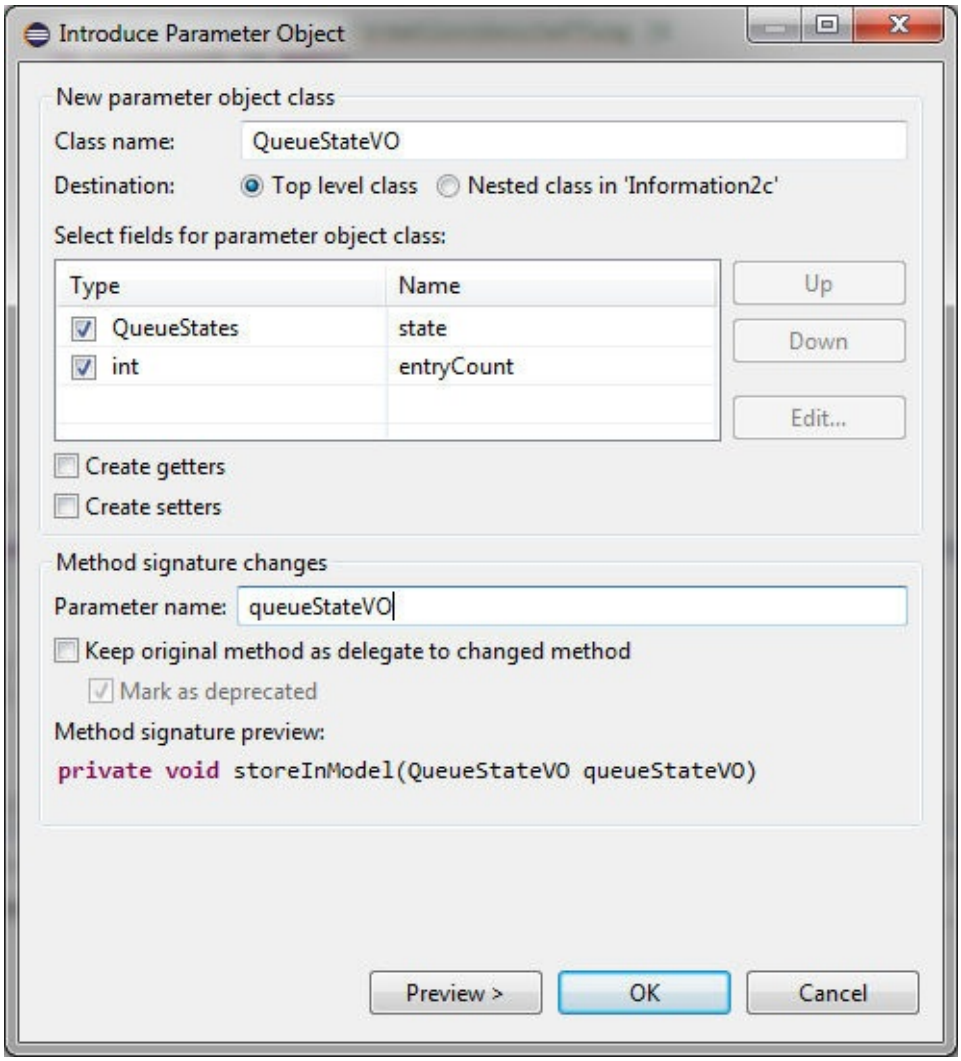


Abbildung 17-11 *Parameter-Objekt einführen*

Es entsteht eine öffentliche Klasse `QueueStateVO` gemäß dem Muster Value Object (vgl. Abschnitt 3.4.5) mit zwei öffentlichen Attributen `state` und `entryCount`. Der Aufruf wird automatisch in Folgendes umgewandelt:

```
storeInModel(new QueueStateVO(state, entryCount));
```

Um uns die Befüllungsarbeit zu erleichtern, extrahieren wir wieder eine lokale Variable per EXTRACT LOCAL VARIABLE:

```
final QueueStateVO queueStateVO = new QueueStateVO(state, entryCount);
storeInModel(queueStateVO);
```

In unserem Kontext benötigen wir keine öffentliche Klasse, sondern eher eine Package-private Hilfsklasse mit ebensolchen Attributen, die direkten Zugriff darauf erlauben. Wir modifizieren die generierte Klasse wie folgt:

```
class QueueStateVO
{
    // ACHTUNG: sind modifizierbar, werden durch andere Klasse geändert
    QueueStates state;
    int         entryCount;

    QueueStateVO(final QueueStates state, final int entryCount)
    {
        this.state = state;
        this.entryCount = entryCount;
```

```
    }
}
```

Nun kann man dann die Zeilen

```
QueueStates state = QueueStates.UNREACHABLE;
int entryCount = 0;
```

durch eine Konstruktoraufruf eines `QueueStateVO`s wie folgt ersetzen:

```
final QueueStateVO stateVO = new QueueStateVO(QueueStates.UNREACHABLE, 0);
```

Der Einsatz der Klasse `QueueStateVO` ist relativ selbsterklärend. Überall dort, wo früher eine Zuweisung an eine lokale Variable erfolgte, verwenden wir nun den Zugriff auf das korrespondierende Attribut des Value Object:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    final QueueStateVO stateVO = new QueueStateVO(QueueStates.UNREACHABLE, 0);

    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
            stateVO.state = elementVO.getQueueStatus();
        }
        else
        {
            stateVO.state = QueueStates.UNKNOWN;
        }
        stateVO.entryCount = elementVO.getEntryCount();
    }

    storeInModel(stateVO);
}
```

Es bietet sich häufig an, Hilfsmethoden zur Datenbeschaffung und Verarbeitung einzuführen, statt nur einige Zeilen umzuordnen. Die Methode `updateModelFromElement(DataElementVO)` ändern wir wie folgt:

```
public void updateModelFromElement(final DataElementVO elementVO)
{
    final QueueStateVO stateVO = createQueueStateVOFromElement(elementVO);

    storeInModel(stateVO);
}
```

Dazu extrahieren wir folgende Methode `createQueueStateVOFromElement(DataElementVO)`, in der wir das `QueueStateVO`-Objekt mit Werten befüllen. Der entsprechende Sourcecode umfasst alle Zeilen zur Informationsbeschaffung der ursprünglichen Methode exklusive des `storeInModel()`-Aufrufs zur Verarbeitung:

```
QueueStateVO createQueueStateVOFromElement(final DataElementVO elementVO)
{
    final QueueStateVO stateVO = new QueueStateVO(QueueStates.UNREACHABLE, 0);

    if (elementVO != null)
    {
        if (elementVO.getQueueStatus() != null)
        {
```

```
            stateVO.state = elementVO.getQueueStatus();
        }
        else
        {
            stateVO.state = QueueStates.UNKNOWN;
        }
        stateVO.entryCount = elementVO.getEntryCount();
    }

    return stateVO;
}
```

Fazit

Durch die Untergliederung in Informationsbeschaffung und Verarbeitung wurde eine Trennung von Zuständigkeiten erreicht. Das führt zu strukturellen Verbesserungen, die sich positiv auf die Lesbarkeit und Verständlichkeit des Sourcecodes auswirken. Zudem kann auch bei möglichen Fehlern während der Informationsbeschaffung ein konsistenter Objektzustand sichergestellt werden.

17.4.12 Wandle Konstantensammlung in `enum` um

Zum Teil findet man im Sourcecode einige Konstanten, die semantisch zusammengehören, aber nicht als eigenständiger Typ definiert sind. Dies wurde als BAD SMELL: ZUSAMMENGEHÖRENDE KONSTANTEN NICHT ALS TYP DEFINIERT in Abschnitt 16.1.3 besprochen. Dieses Refactoring hilft bei der Problemlösung.

Nehmen wir an, in der folgenden Klasse `BusinessClass` wären einige `int`-Konstanten definiert, zu denen teilweise korrespondierende `String`-Konstanten als Beschreibung vorhanden sind. Glücklicherweise sind die Konstanten über die Namenspräfixe `JOBSTATUS` und `ERRORCODE` gegliedert und lassen sich leicht unterscheiden:

```
public class BusinessClass
{
    public static final int     JOBSTATUS_UNDEFINED      = -1;
    public static final int     JOBSTATUS_ACTIVE         = 1;
    public static final int     JOBSTATUS_FINISHED       = 2;

    private static final String JOBSTATUS_UNDEFINED_NAME = "UNDEFINED";
    private static final String JOBSTATUS_ACTIVE_NAME    = "Active";
    private static final String JOBSTATUS_FINISHED_NAME  = "Finished";

    public static final int     ERRORCODE_OK             = 0;
    public static final int     ERRORCODE_WARN           = 1;
    public static final int     ERRORCODE_ERROR          = 2;

    private final BusinessJob job;

    public int getJobState()
    {
        if (job.isActive())
```

```
            return JOBSTATUS_ACTIVE;
        if (job.isFinished())
            return JOBSTATUS_FINISHED;

        return JOBSTATUS_UNDEFINED;
    }
    // …
```

Schritt 1: Sammlung in einer eigenen Konstantenklasse

Anhand der `JOBSTATUS`-Konstanten zeige ich den Ablauf bei diesem Refactoring. Die bisher lose zusammenhängenden `JOBSTATUS`-Konstanten werden in einer eigenen Klasse gesammelt. Bei der Benennung ist es sinnvoll, einen Ober- oder Gliederungsbegriff zu verwenden. Besitzen die Konstanten ein gemeinsames, aussagekräftiges Präfix, ist dies ein guter Anhaltspunkt für den Namen der Konstantenklasse. Wir nutzen hier den Namen `JobStates`. Die Klasse selbst deklarieren wir `final`, da sie nicht als Basis für weitere Klassen dienen soll:

```
public final class JobStates
{
    public static final int    JOBSTATUS_UNDEFINED      = -1;
    public static final int    JOBSTATUS_ACTIVE         = 1;
    public static final int    JOBSTATUS_FINISHED       = 2;

    private static final String JOBSTATUS_UNDEFINED_NAME = "UNDEFINED";
    private static final String JOBSTATUS_ACTIVE_NAME    = "Active";
    private static final String JOBSTATUS_FINISHED_NAME  = "Finished";
}
```

Schritt 2: Erzeugen eines Konstruktors und von Zugriffsmethoden

Die Konstanten bestehen aus einem Wert und einer Beschreibung. Daher führen wir in der Klasse `JobStates` zwei Attribute ein. Die Klasse wird entsprechend dem Muster IMMUTABLE-KLASSE (vgl. Abschnitt 3.4.2) konstruiert. Der private Konstruktor verhindert das Erzeugen weiterer Objekte außerhalb dieser Konstantenklasse. Zudem bieten wir nur lesenden Zugriff auf die Attribute.

```
public final class JobStates
{
    public static final int    JOBSTATUS_UNDEFINED      = -1;
    public static final int    JOBSTATUS_ACTIVE         = 1;
    public static final int    JOBSTATUS_FINISHED       = 2;

    private static final String JOBSTATUS_UNDEFINED_NAME = "UNDEFINED";
    private static final String JOBSTATUS_ACTIVE_NAME    = "Active";
    private static final String JOBSTATUS_FINISHED_NAME  = "Finished";

    private final String        name;
    private final int           value;

    private JobStates(final String name, final int value)
    {
```

```
        this.name = name;
        this.value = value;
    }

    public String getName()          { return this.name; }
    public int getValue()            { return this.value; }
}
```

Der Einsatz dieser Konstantenklasse hilft dabei, die Funktionalitäten der Business-Klasse von der Definition benötigter Konstanten zu trennen. Darüber hinaus können die Konstanten dann von anderen Klassen benutzt werden, ohne dass diese Kenntnis von der Business-Klasse besitzen müssen.

Schritt 3: Umwandlung in eine `enum`-Aufzählung

Als letzten Schritt wandeln wir die Konstantendefinition in eine `enum`-Aufzählung um:

```
enum JobStates
{
    UNDEFINED("UNDEFINED", -1), ACTIVE("Active", 1), FINISHED("Finished", 2);

    private final String name;
    private final int    value;

    private JobStates(final String name, final int value)
    {
        this.name = name;
        this.value = value;
    }

    public String getName()          { return this.name; }
    public int getValue()            { return this.value; }
}
```

Fazit

Die Auslagerung von Konstanten aus Business-Klassen ist häufig bereits ein wichtiger Schritt für mehr Struktur und Verständlichkeit. Die nachfolgende Definition der Aufzählungswerte innerhalb einer `enum`-Aufzählung führt zu Typsicherheit und einer loseren Kopplung – hier besteht dadurch keine Abhängigkeit zur Business-Klasse mehr. Durch eine derartige Realisierung sind die Konstanten außerdem automatisch sortierbar und auch serialisierbar. Darüber hinaus erfolgt eine typsichere, objektorientierte Definition der Konstanten, wodurch Anwendungsfehler wesentlich unwahrscheinlicher als bei der Verwendung von `int`-Werten sind. Auch nachfolgende Refactorings können ohne größere Auswirkungen auf nutzende Klienten durchgeführt werden: Konstanten müssen ausschließlich in der `enum`-Aufzählung gepflegt werden. Der verwendende Sourcecode nutzt lediglich Referenzen auf die Konstanten.

17.4.13 Entferne Exceptions zur Steuerung des Kontrollflusses

Dieses Refactoring dient dazu, den BAD SMELL: EXCEPTIONS ZUR STEUERUNG DES KONTROLLFLUSSES zu korrigieren, wie er in Abschnitt 16.3.3 beschrieben ist.

Betrachten wir dazu die folgende, simple Methode `isAlive()`:

```
private void isAlive() throws DbException
{
    // Teste Existenz in Datenbank
    session.getDbObject(id);
}
```

Diese Methode wird in der folgenden `toString()`-Methode eingesetzt, um die Existenz des eigenen Objekts in der Datenbank vor der Erzeugung einer XMLRepräsentation sicherzustellen. Ist das Objekt dort nicht vorhanden, so wird im `catch`-Block eine Fehlermeldung zurückgeliefert:

```
public String toString()
{
    try
    {
        isAlive();
        return createXmlStringInfo();
    }
    catch (final DbException e)
    {
        return "Object of class " + getClass().getSimpleName() + " is dead!";
    }
}
```

Achtung: Abgrenzung gegenüber Methoden zur Zustandsprüfung

Der Einsatz der Methode `isAlive()` erinnert an Methoden zur Zustandsprüfung. Allerdings gibt es einen entscheidenden Unterschied: `isAlive()` überprüft einen möglichen Zustand des Objekts, für das gilt, dass ein assoziiertes Objekt, hier ein Verweis in die Datenbank, nicht mehr gültig ist. Dieser Zustand ist zwar selten, stellt aber keine unerwartete Situation oder einen ungültigen Objektzustand dar. Daher sollte man hier besser einen booleschen Rückgabewert statt einer Exception verwenden. Eine Zustandsprüfung sichert dagegen einen korrekten und erwarteten Objektzustand ab. Eine Parameterprüfung soll vor unvollständigen Initialisierungen bewahren. Ungültige Objektzustände stellen einen Fehler und eine ungewünschte Ausnahmesituation dar. Deshalb ist dafür der Einsatz von Exceptions sinnvoll.

Schritt 1: Einführen eines Rückgabewerts

Der Methodenname `isAlive()` impliziert die Rückgabe eines booleschen Werts. Daher wandeln wir die Methode dementsprechend um. Dazu fangen wir die zuvor in

der Signatur definierte Exception ab und geben im korrespondierenden `catch`-Block `false` zurück. Eine erfolgreiche Prüfung führt zur Rückgabe des Werts `true`. Abschließend wird die Exception aus der Signatur entfernt:

```
private boolean isAlive()
{
    try
    {
        session.getDbObject(id);   // Teste Existenz in Datenbank
        return true;
    }
    catch (final DbException e)
    {
        return false;
    }
}
```

Es kommt hier zu einer Änderung der Signatur, weil daraus die Exception entfernt wird. In diesem Fall ist dies allerdings möglich, da es sich um eine private Methode handelt und wir dadurch die komplette Kontrolle über den verwendenden Sourcecode haben. Weitere Hinweise liefert folgender Praxistipp.

Achtung: API-Änderungen durch Refactorings

Bei Änderungen an der Methodensignatur muss man vorsichtig vorgehen, um keine Inkompatibilitäten einzuführen. Änderungen an öffentlichen Methoden erfordern Änderungen in nutzenden Klienten. Häufig sind daher Änderungen am API nicht durchführbar. Eine Methode wird dann per Annotation als `@Deprecated` markiert und durch eine Version mit verbessertem API ersetzt. ***Private Methoden können immer ohne Folgen für andere externe Klassen geändert werden*** – manchmal müssen jedoch innere Klassen angepasst werden.

Nach den Transformationen wird offensichtlich, dass der Rückgabewert der `getDbObject()` -Methode nicht ausgewertet wird. Dies ist generell unschön, für dieses Beispiel allerdings unbedeutend.[20]

Schritt 2: Exception Handling durch Auswertung des Rückgabewerts ersetzen

Das Exception Handling mit `try-catch` ist nun überflüssig und wird durch eine `if-else`-Auswertung des Rückgabewerts ersetzt.[21] Dadurch ergibt sich folgender, besser lesbarer Sourcecode:

```
public String toString()
{
    if (isAlive())
    {
        return createXmlStringInfo();
    }
    return "Object of class " + getClass().getSimpleName() + " is dead!";
}
```

Fazit

Die vermeintliche »Fehlerbehandlung« (Abfrage der Existenz) erfolgt nun so lokal wie möglich, nämlich an der Stelle des Datenbankzugriffs in der `isAlive()`-Methode. Für die Aufrufer der Methode bleibt all dies verborgen. Dadurch lässt sich nutzender Sourcecode klarer gestalten. Falls jedoch lokal innerhalb von Verarbeitungsmethoden nicht sinnvoll auf Probleme reagiert werden kann, sollten Exceptions propagiert werden.

17.4.14 Wandle in Utility-Klasse mit statischen Hilfsmethoden um

Utility-Klassen sollten kein Objektverhalten anbieten und daher lediglich aus einer Sammlung von Hilfsmethoden bestehen. Manchmal besitzen diese Klassen aus Unachtsamkeit allerdings doch einen öffentlichen Konstruktor und einige Zustandsattribute sowie nicht statische Hilfsmethoden. Oftmals müssen dem Konstruktor dazu Parameter übergeben werden, die später innerhalb der Hilfsmethoden genutzt werden.

Als Beispiel dient eine Methode `writeDoc(IDocument, String, String)`, die Objekte vom Typ `IDocument` speichern soll. Dazu wird eine Instanz der Klasse `DocumentToFilesystem` genutzt und zuvor erzeugt, um dann eine parameterlose `doSave()`-Methode aufzurufen. Die benötigten Parameter werden aber bereits an den Konstruktor der Klasse `DocumentToFilesystem` übergeben:

```
public void writeDoc(final IDocument document,
                     final String path, final String fileName)
{
    final DocumentToFilesystem docToFileSystem = new DocumentToFilesystem(
          document, path, fileName);

    docToFileSystem.doSave();
}
```

Obwohl es sich bei der Klasse `DocumentToFilesystem` eigentlich um eine reine Utility-Klasse handelt, muss immer (künstlich) ein entsprechendes Objekt erzeugt und korrekt initialisiert werden, um die Hilfsmethoden, hier etwa die Methode `doSave()`, aufrufen zu können. Diese vom Design her missglückte Utility-Klasse – weil sie statt statischer Hilfsmethoden lediglich Objektmethoden anbietet – ist in Abbildung 17-12 als Klassendiagramm dargestellt.

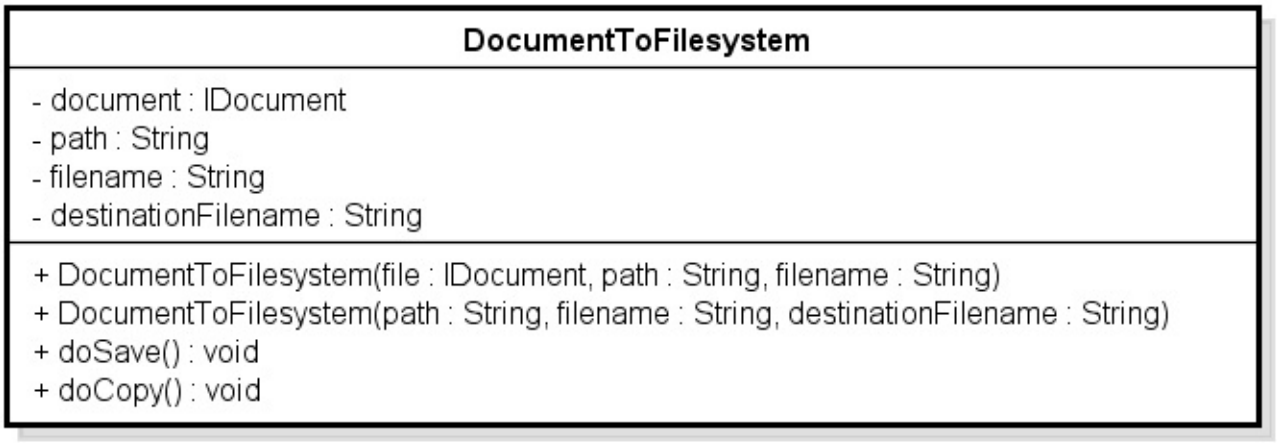


Abbildung 17-12 *Falsche Realisierung einer Utility-Klasse*

Zum Aufruf von Objektmethoden erfolgt zunächst immer eine Parameterübergabe im Konstruktor. Sinnvoller wäre es jedoch, benötigte Parameter direkt an die jeweiligen Methoden zu übergeben. Im Folgenden prüfen wir, ob für eine Utility-Klasse tatsächlich ein Zugriff auf die Zustandsinformationen notwendig ist. Ziel bei diesem Refactoring ist es, die Zustandsinformation aus der Utility-Klasse möglichst vollständig zu entfernen und stattdessen Aufrufparameter an statische Methoden zu verwenden.

Schritt 1: Umwandlung in statische Hilfsmethode

Um eine Methode vom Objektzustand unabhängig zu machen, kann man diese zunächst statisch definieren. In Form von Kompilierfehlern werden automatisch mögliche Abhängigkeiten zu verwendeten Attributen sichtbar. Existieren keine Abhängigkeiten, so kann man eine weitere Methode wählen und erneut diesen Schritt 1 ausführen. Ansonsten löst man die erkannten Abhängigkeiten, wie im Schritt 2 beschrieben, auf.

Schritt 2: Erweiterung der Methodensignatur

Damit die Methode unabhängig von den Attributen einer Instanz der Klasse wird, müssen alle Abhängigkeiten durch Übergabeparameter aufgelöst werden. Man erweitert die Methodensignatur um die zur Ausführung benötigten Parameter. Für die Methode `doSave()` werden die Informationen aus den Attributen `file`, `path` und `filename` benötigt. In Abbildung 17-13 ist dies als Klassendiagramm verdeutlicht.[22]

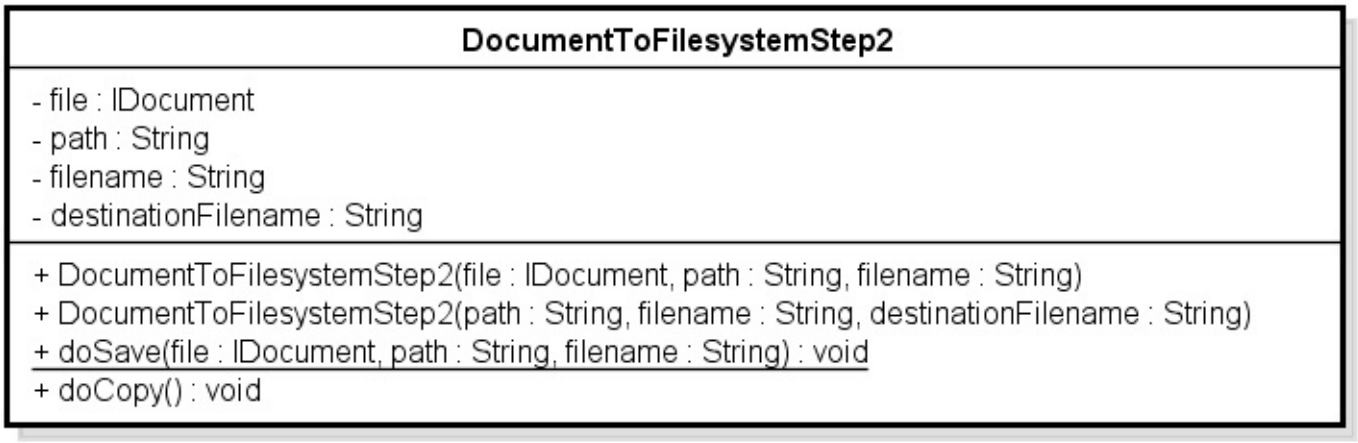


Abbildung 17-13 *Utility-Klasse nach Schritt 2*

Durch die Parameterübergabe entfällt die Notwendigkeit zur Konstruktion der Utility-Klasse, nur um eine Hilfsmethode nutzen zu können. Dies erleichtert die Nutzbarkeit:

```
public void writeDoc(final IDocument document,
                     final String path, final String fileName)
{
    DocumentToFilesystemStep2.doSave(document, path, fileName);
}
```

Schritt 3: Umwandlung aller Methoden in entsprechende Hilfsmethoden

Durch Wiederholen der Schritte 1 und 2 werden alle Methoden bearbeitet und in statische Hilfsmethoden transformiert. Als Ergebnis sind im besten Fall nur noch statische Hilfsmethoden in der Utility-Klasse definiert, wie dies Abbildung 17-14 zeigt.

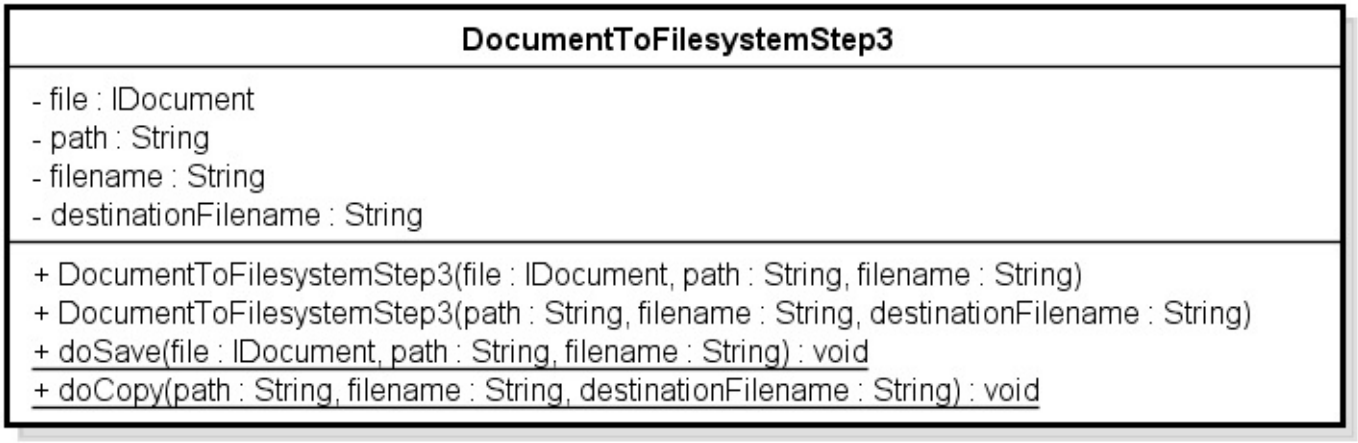


Abbildung 17-14 *Utility-Klasse nach Schritt 3*

Allerdings sind noch die statischen Attribute sowie eine Abhängigkeit zum jeweiligen Konstruktor verblieben. Dies wird, sofern möglich, im letzten Schritt des Refactorings entfernt.

Schritt 4: Entfernen von Zustandsinformationen und Anpassen von Konstruktoren

Wenn alle Methoden erfolgreich bearbeitet wurden, müssen keine Zustandsinformationen mehr gespeichert werden. Sowohl die Konstruktoren mit Parametern als auch die korrespondierenden Attribute sind damit obsolet und können entfallen.

Wir entfernen alle öffentlichen Konstruktoren und erstellen einen privaten Defaultkonstruktor. Dadurch werden Objektkonstruktionen der Utility-Klasse durch andere Klassen unterbunden. Diese Klasse wird anschließend in `DocumentToFilesystemUtils` umbenannt, was zur Lesbarkeit beiträgt. Das in Abbildung 17-15 dargestellte Klassendiagramm verdeutlicht dies.

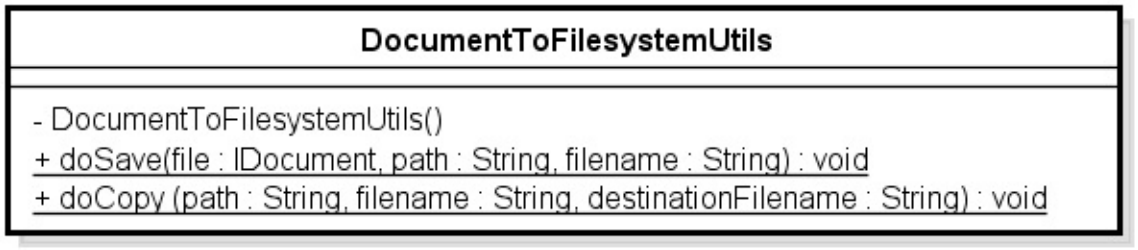


Abbildung 17-15 *Utility-Klasse nach Schritt 4*

Fazit

Durch die erfolgten Transformationen ist nun der Utility-Charakter der Klasse offensichtlich. Zudem werden alle Abhängigkeiten und Parametrierungen klar. Außerdem werden Anwendungsfehler vermieden, die bei einer Konstruktion von Objekten der ursprünglichen Klasse auftreten konnten: Ist beispielsweise die Wertebelegung für eine `doSave()` -Aktion ausgelegt, so ist diese meistens für den Aufruf anderer Hilfsmethoden unpassend und führt zu Fehlern. Gibt es mehrere Instanzen der Utility-Klasse, so steigt die Wahrscheinlichkeit, eine nicht passend initialisierte Instanz einzusetzen.[23]

17.5 Defensives Programmieren

In komplexeren Systemen mit vielen Komponenten und externen Aufrufen empfiehlt es sich, insbesondere an den System- bzw. Komponentengrenzen vorsichtig oder defensiv zu programmieren. Damit ist gemeint, dass man Eingaben validiert und korrekte Initialisierungen und Zustände überprüft.

17.5.1 Führe eine Zustandsprüfung ein

Dieses Umbaumaßnahme dient dazu, eine korrekte Initialisierung von Attributen, die

zum Einsatz der Klasse wichtig sind, sicherzustellen. Dabei nutzen wir das Refactoring LAGERE FUNKTIONALITÄT IN HILFSMETHODEN AUS (vgl. Abschnitt 17.4.10).

Beispiel

Im folgenden Beispiel sehen wir drei Methoden, die jeweils auf ein statisches Attribut `parameterAccess` zugreifen. Nur nach erfolgreicher Initialisierung durch den Aufruf der `initialize()`-Methode kann diese Utility-Klasse korrekt arbeiten. Daher findet in allen Methoden jeweils eine Initialisierungsprüfung statt und es wird eine `IllegalStateException` ausgelöst, falls die benötigte Initialisierung noch nicht erfolgt ist:

```
public static final String getExternalNameServiceName()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly"
+
                " initialized. Use initialize() before any other method
call.");

   return parameterAccessService.getValue("SYSPARAM_EXTERNAL_NAME_SERVICE");
}

public static final String getExternalORBHost()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly"
+
                " initialized. Use initialize() before any other method
call.");

   return parameterAccessService.getValue("SYSPARAM_EXTERNAL_ORB_HOST");
}

public static final String getInternalORBHost()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly"
+
                " initialized. Use initialize() before any other method
call.");

   return parameterAccessService.getValue("SYSPARAM_INTERNAL_ORB_HOST");
}
```

Je mehr Methoden solche identischen Zustandsprüfungen durchführen, desto mehr explodiert der Sourcecode und scheint nur aus der Prüfung der Initialisierung zu bestehen.

Schritt 1: Implementieren einer Prüfmethode

Zur Prüfung der Initialisierung faktorieren wir folgende Methode heraus:

```
private static void checkParameterAccessServiceInitialized()
{
   if (parameterAccessService == null)
     throw new IllegalStateException("ParameterAccessService is not correctly"
+
                " initialized. Use initialize() before any other method
call.");
}
```

Schritt 2: Einsatz der Prüfmethode

Der Einsatz der obigen Methode geschieht gemäß dem DRY-Prinzip und macht die einsetzenden Methoden besser lesbar:

```
public static final String getExternalNameServiceName()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_NAME_SERVICE");
}

public static final String getExternalORBHost()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_EXTERNAL_ORB_HOST");
}

public static final String getInternalORBHost()
{
    checkParameterAccessServiceInitialized();
    return parameterAccessService.getValue("SYSPARAM_INTERNAL_ORB_HOST");
}
```

Fazit

Eine derartige zentrale Initialisierungsprüfung hilft, mehrfach gleichen oder ähnlichen Sourcecode zu vermeiden, und sorgt dadurch für eine bessere Lesbarkeit.

17.5.2 Überprüfe Eingabeparameter

Diese Umbaumaßnahme soll dafür sorgen, dass unerwartete Eingabewerte zurückgewiesen werden. Dadurch bleibt die Objektintegrität erhalten und eine Methodenausführung erfolgt nur unter sicheren Umgebungsbedingungen. Des Weiteren können wir unsere Software im Fehlerfall informativer gestalten. Als Reaktion auf ungültige Parameterwerte sollte man Exceptions nutzen, da diese im Gegensatz zu Rückgabewerten einem Aufrufer eine Menge an Informationen über eine mögliche Fehlerursache mitliefern können.

Zumindest an Systemgrenzen, aber auch für öffentliche Business-Methoden empfiehlt es sich, die eingehenden Parameter auf Gültigkeit zu prüfen. Auf diese Weise vermeidet man, ungültige Werte ins Programm zu lassen und dadurch Merkwürdigkeiten oder Fehler in Berechnungen zu erhalten. Dabei finde ich

folgende Analogie aus dem realen Leben hilfreich: Wenn die dreckigen Schuhe vor der Wohnungstür bleiben, muss man nicht andauernd den Boden der ganzen Wohnung säubern, wenn man mal im Wald spazieren war.[24]

Bei dieser Umbaumaßnahme lassen sich die Fälle der Prüfung von Referenzparametern sowie von Parametern primitiver Datentypen unterscheiden. Für Letztere können wir gültige Werte leichter prüfen, für Referenzparameter ist dies – abgesehen von einer Prüfung der Referenz auf `null` – meistens etwas komplizierter. Da Referenzparameter zum Teil eine Ausnahme bei der Prüfung darstellen, beschränke ich hier die Diskussion auf primitive Datentypen. Der folgende Praxistipp geht auf die Behandlung von `null`-Werten ein.

Stilfrage: `null`-Werte explizit behandeln oder nicht?

Eine Parameterprüfung ist immer dann zwingend notwendig, wenn ansonsten der Fehler »verschleppt« würde oder es zu einem nicht sofort erkennbaren Fehlverhalten käme, etwa zu falschen Berechnungen. Wird direkt auf eine übergebene Referenzvariable zugegriffen, die normalerweise nicht `null` sein darf, so führt dies ohne Zutun zu einer `NullPointerException`. In derartigen Fällen kann auf eine explizite Prüfung eines solchen Parameters verzichtet werden – allerdings wird dann nicht deutlich, ob nur versehentlich keine Prüfung erfolgt. Deshalb prüfe ich aus Gründen der Nachvollziehbarkeit und Konsistenz ich in der Regel alle Parameter und behandle `null`-Referenzen. Ich bevorzuge als Reaktion auf unerwartete `null`-Werte, explizit eine `IllegalArgumentException` zu werfen und weitere Informationen zur Fehlersituation im Text der Exception zu übergeben. Seit JDK 7 ist der Einsatz der Hilfsmethode `Objects.requireNonNull()` empfehlenswert.

Beispiel

Betrachten wir folgenden Konstruktor, in dem drei Eingabeparameter vom Typ `int` entgegengenommen und diese ungeprüft gleichnamigen Attributen zugewiesen werden:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final int registrationStrategy)
{
    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Tatsächlich sind dabei aber folgende Randbedingungen zu beachten:

- Der Wert für `minExecutions` darf nicht negativ sein, d. h., er muss mindestens den Wert 0 haben. Die maximale Anzahl an Ausführungen `maxExecutions` muss kleiner oder gleich einem nicht gezeigten Wert `MAX_EXECUTIONS` sein.

- Der Wert `maxExecutions` muss größer gleich dem Wert `minExecutions` sein.
- Als Übergabewerte für die Registrierungsstrategie sind nur folgende durch `int`-Konstanten definierte Werte zulässig:

```
public static final int REPLACE_OLD_OR_ADD_AS_LAST = 0;
public static final int ADD_AS_FIRST               = 1;
public static final int ADD_AS_LAST                = 2;
```

Schritt 1: Referenz- oder Bereichsprüfung

Durch eine einfache Bereichsprüfung (hier zunächst ohne detaillierte Hinweistexte) eliminieren wir zunächst den in Abschnitt 16.3.8 vorgestellten BAD SMELL: KEINE GÜLTIGKEITSPRÜFUNG VON EINGABEPARAMETERN.

```
public CommandExecutor(final int minExecutions, final int maxExecutions, final
      int registrationStrategy)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions' or " +
                                           "'maxExecutions' is out of valid
range");
    }
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions' must be"
+
                                           " <= 'maxExecutions'");
    }
    // Achtung: Hier korrekter, aber potenziell fehlerträchtiger und fragiler
    // Vergleich, weil das Ganze stark vom Mapping und Werten abhängig ist!
    if (registrationStrategy < REPLACE_OLD_OR_ADD_AS_LAST ||
        registrationStrategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'registrationStrategy'"
+
                                           " is invalid");
    }

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Dieser erste Schritt gibt im Fehlerfall einen vagen Hinweis (auslösender Parameter) auf die Ursache (Fehlerbeschreibung). ***Allerdings sind sowohl die Fehlerbeschreibung als auch die Wertebereichsprüfung fragil.*** Ziemlich unangenehm ist der Einsatz von Vergleichsoperatoren auf willkürlichen Konstanten. Im Speziellen gilt dies hier für die Verwendung der Konstanten der Registrierungsstrategie. Diese hätten theoretisch auch durch beliebige andere Werte abgebildet werden können. Dann wären die Vergleiche eventuell nicht mehr ausreichend oder korrekt gewesen, etwa wenn `ADD_AS_LAST` dem Wert `4` entspricht. In Abschnitt 16.1.3 diskutiert BAD SMELL: ZUSAMMENGEHÖRENDE KONSTANTEN NICHT ALS TYP DEFINIERT dadurch verursachte

Probleme und mögliche Lösungen. Wir nutzen hier das in Abschnitt 17.4.12 beschriebene Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM.

Schritt 2: Einsatz von Aufzählungen, wenn sinnvoll und möglich

Häufig bietet sich für semantisch zusammengehörende Konstanten der Einsatz eines `enum`-Aufzählungstyps oder des ENUM-Musters (vgl. Abschnitt 3.4.4) an. In diesem Fall definiert der Typ `RegistrationStrategy` die Konstanten und wird als Referenzparameter übergeben. Eine Wertebereichsprüfung erfolgt implizit durch die Angabe des Typs. Allerdings ist nun die Referenz auf `null` zu prüfen:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final RegistrationStrategy registrationStrategy)
{
   // … zwei Prüfungen ausgelassen …
   Objects.requireNonNull(registrationStrategy,
                          "parameter 'registrationStrategy' must not be
null");

   this.minExecutions = minExecutions;
   this.maxExecutions = maxExecutions;
   this.registrationStrategy = registrationStrategy;
}
```

Schritt 3: Angabe von übergebenen Werten und von Gültigkeitsbereichen

Anschließend korrigieren wir die unpräzise Fehlermeldung, indem wir den übergebenen Wert in den Exception-Text aufnehmen. Dadurch liefern wir bereits einen guten Hinweis auf mögliche Fehlerursachen. Wir können die Informationen aber noch weiter verbessern, indem wir gegebenenfalls zusätzlich gültige Wertebereiche angeben:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final RegistrationStrategy registrationStrategy)
{
   if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
   {
      throw new IllegalArgumentException("parameter 'minExecutions'=" +
                minExecutions + " or 'maxExecutions'=" + maxExecutions +
                " is out of valid range: [" + 0 + " - " + MAX_EXECUTIONS +
"]");
   }
   if (minExecutions > maxExecutions)
   {
      throw new IllegalArgumentException("parameter 'minExecutions'=" +
                minExecutions + " must be <= 'maxExecutions'=" +
maxExecutions);
   }

   Objects.requireNonNull(registrationStrategy,
```

```
                            "parameter 'registrationStrategy' must not be
null");

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Schritt 4: Herausrefaktorieren von Prüfmethoden

Wie man sieht, wird der Sourcecode der Parameterprüfung immer umfangreicher und komplizierter. Hier dominiert die Wertebereichsprüfung bereits den eigentlichen Programmcode. Ähnliches habe ich im einleitenden Beispiel von Kapitel 16 für Logging-Code als negativ angesprochen. Ein Herausrefaktorieren von Prüfmethoden ist unter dem Aspekt der Wiederverwendbarkeit in anderen Klassen oftmals eher wenig sinnvoll, da die Prüfungen stark vom jeweiligen Kontext abhängig sind. Innerhalb der eigenen Klasse kann sich die Lesbarkeit durch diese Prüfmethoden allerdings enorm erhöhen:

```
public CommandExecutor(final int minExecutions, final int maxExecutions,
                       final RegistrationStrategy registrationStrategy)
{
    assertExecutionsInValidRange(minExecutions, maxExecutions);
    assertMinExecutionsLessOrEqualToMax(minExecutions, maxExecutions);
    Objects.requireNonNull(registrationStrategy,
                           "parameter 'registrationStrategy' must not be
null");

    this.minExecutions = minExecutions;
    this.maxExecutions = maxExecutions;
    this.registrationStrategy = registrationStrategy;
}
```

Die beiden Prüfmethoden werden wie folgt definiert:

```
static void assertExecutionsInValidRange(final int minExecutions,
                                                 final int maxExecutions)
{
    if (minExecutions < 0 || maxExecutions > MAX_EXECUTIONS)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
              minExecutions + " or 'maxExecutions'=" + maxExecutions +
              " is out of valid range: [" + 0 + " - " + MAX_EXECUTIONS + "]");
    }
}

static void assertMinExecutionsLessOrEqualToMax(final int minExecutions,
                                                final int maxExecutions)
{
    if (minExecutions > maxExecutions)
    {
        throw new IllegalArgumentException("parameter 'minExecutions'=" +
              minExecutions + " must be <= 'maxExecutions'=" + maxExecutions);
    }
}
```

Fazit

Die Prüfung von Eingabeparametern verhindert mögliche Inkonsistenzen und sorgt für einen gültigen Objektzustand. Im Fehlerfall werden aussagekräftige Fehlermeldungen produziert, die eine spätere Analyse und Fehlersuche erleichtern.

17.6 Weiterführende Literatur

Das Themengebiet Refactorings lässt sich in einem Kapitel nicht umfassend behandeln. Ich hoffe aber, Ihr Interesse geweckt zu haben. Empfehlenswerte Bücher, die sich ausführlich mit dem Thema Refactorings beschäftigen, sind:

- »**Refactoring: Improving the Design of Existing Code**« von Martin Fowler [27] Einige Tricks und Kniffe, Sourcecode zu verbessern, lernt man von Kollegen oder durch Erfahrung. Martin Fowler fasst dieses Wissen in dem genannten Buch zusammen und stellt ein systematisches Vorgehen zur Sourcecode-Transformation vor.
- »**Refactorings to Patterns**« von Joshua Kerievsky [54] Dieses Buch verknüpft das Standardwerk zu Refactorings von Martin Fowler mit den Ideen der Entwurfsmuster.
- »**Refactorings in großen Softwareprojekten**« von Stefan Roock und Martin Lippert [77] Neben den Refactorings, die einfache Sourcecode-Transformationen beschreiben, kann man Refactorings auch auf Architektur- bzw. Designebene übertragen. Dieses Buch stellt einige Architekturprobleme, Refactorings und Tools vor.
- »**Working Effectively With Legacy Code**« von Michael Feathers [24] In diesem Buch werden Strategien aufgezeigt, wie man mit Altlasten behafteten Sourcecode überarbeiten kann, ohne in einem Wartungsalbtraum zu enden.

18 Entwurfsmuster

Bereits erprobte und regelmäßig eingesetzte allgemeingültige Verfahren zur Lösung eines Entwurfsproblems werden als ***Entwurfsmuster*** oder auch ***Designpattern*** bezeichnet und gewinnen seit einigen Jahren immer mehr an Bedeutung. Populär wurden die Entwurfsmuster durch das Buch »Design Patterns – Elements of Reusable Object Oriented Software« von der sogenannten Gang of Four (GoF): Gamma, Helm, Johnson, Vlissides [29]. Dieses Buch löste Mitte der 90er-Jahre einen Run aus, der bis heute anhält. Viele Bücher sind zu diesem Thema erschienen, ohne jedoch das Original als *die* Referenz zu verdrängen.

Dieses Kapitel stellt einige in der Praxis häufig eingesetzte Entwurfsmuster vor, zeigt aber auch mögliche Probleme auf und entkräftet den Glauben, allein durch Einsatz von Entwurfsmustern erfolgreiche Softwareentwicklung betreiben zu können. Naheliegende Fehlverwendungen werde ich als ***Anti-Pattern*** beschreiben und erläutern.

Die von der GoF vorgenommene Gliederung in die drei Kategorien Erzeugungsmuster, Strukturmuster und Verhaltensmuster übernehme ich in diesem Buch. Die Vorstellung eines Musters beginnt mit einer kurzen Beschreibung und Motivation für dessen Einsatz. Daran schließt sich eine Darstellung der zugrunde liegenden Struktur in Form eines UML-Diagramms sowie eine Beschreibung des jeweiligen Musters anhand vereinfachter Praxisbeispiele mit abschließender Bewertung an.

Abschnitt 18.1 beschreibt, wie der Einsatz von Erzeugungsmustern den Konstrukttionsprozess von Objekten vereinfachen kann. Die in Abschnitt 18.2 vorgestellten Strukturmuster helfen, Funktionalitäten flexibel erweitern und anpassen zu können. Verhaltensmuster werden in Abschnitt 18.3 beschrieben. Durch ihren Einsatz strukturiert und vereinfacht man komplexe Abläufe oder Interaktionen zwischen Objekten.

Design mit Entwurfsmustern

Die durch Entwurfsmuster beschriebenen prototypischen Beispielumsetzungen helfen, ein Problem auf eine dokumentierte Art und Weise zu lösen. Der Einsatz kann zu qualitativ hochwertiger Software führen – es kommt jedoch nicht automatisch zu einem guten Design und sinnvollen Lösungen. Zwar erreicht man häufig ein flexibles Design, allerdings entsteht auf Sourcecode-Ebene oft weitere Komplexität. Der Einsatz von Entwurfsmustern sollte daher immer auf die jeweilige Situation abgestimmt werden.

Jedes Entwurfsmuster trägt einen eindeutigen Namen und beschreibt eine Lösungsidee für ein Entwurfsproblem. Dadurch fällt die Kommunikation von Designentscheidungen unter Entwicklern leichter. Es entsteht eine ***Entwurfssprache***,

die dabei hilft, in Diskussionen länger auf der verständlichen Ebene des Designs mögliche Alternativen besprechen zu können, ohne sich mit Implementierungsdetails beschäftigen zu müssen. Wollte man früher einem anderen Softwareentwickler eine Realisierung beschreiben, so waren dafür viele Worte und Zeichnungen nötig. Etwa: Diese Klasse ist so entworfen, dass sie nur einmal instanziiert werden kann, bzw. diese Klasse ist ein Datenbehälter und definiert für jedes Attribut öffentliche `get()` - und `set()` -Methoden. Heutzutage reicht es, z. B. zu sagen, diese Klasse ist als SINGLETON oder als VALUE OBJECT entworfen. Eine ausführliche Darstellung der Vorteile einer gemeinsamen Designsprache finden Sie in dem ausgezeichneten Buch »Design Patterns Explained« [80] von Alan Shalloway und James R. Trott.

Beispielapplikation: Image-Editor

Einige Entwurfsmuster werden in Form einer Beispielanwendung vorgestellt. Dabei handelt es sich um eine vereinfachte Version einer realen Anwendung. In Gesprächen mit Kunden wurden die im Folgenden beschriebenen Anforderungen aufgenommen.

Anforderungen

Es soll ein grafischer Editor entwickelt werden, der Zeichenfunktionen für grafische Figuren, wie Linien, Rechtecke und Kreise, anbietet. Weiterhin ist der Import und die Anzeige von Bildern gefordert. Die Abmessungen der Zeichenfläche sollen konfigurier- und änderbar sein. Übersteigt deren Größe die Bildschirmabmessungen, so sind Scrollbars gewünscht. Die Applikation soll über Menüs, Kontextmenüs und Toolbars gesteuert werden. Ein erster Skizzenentwurf der Bedienoberfläche kann neben Bleistift und Papier auch mithilfe eines Zeichenprogramms erstellt werden. Häufig ist Letzteres aber mühseliger als eine Freihandskizze. Für Prototyp-Entwürfe von GUIs kann ich das Tool Balsamiq Mockups empfehlen, das als Demoversion unter `http://www.balsamiq.com/` verfügbar ist. Damit entstand der in Abbildung 18-1 dargestellte Entwurf.

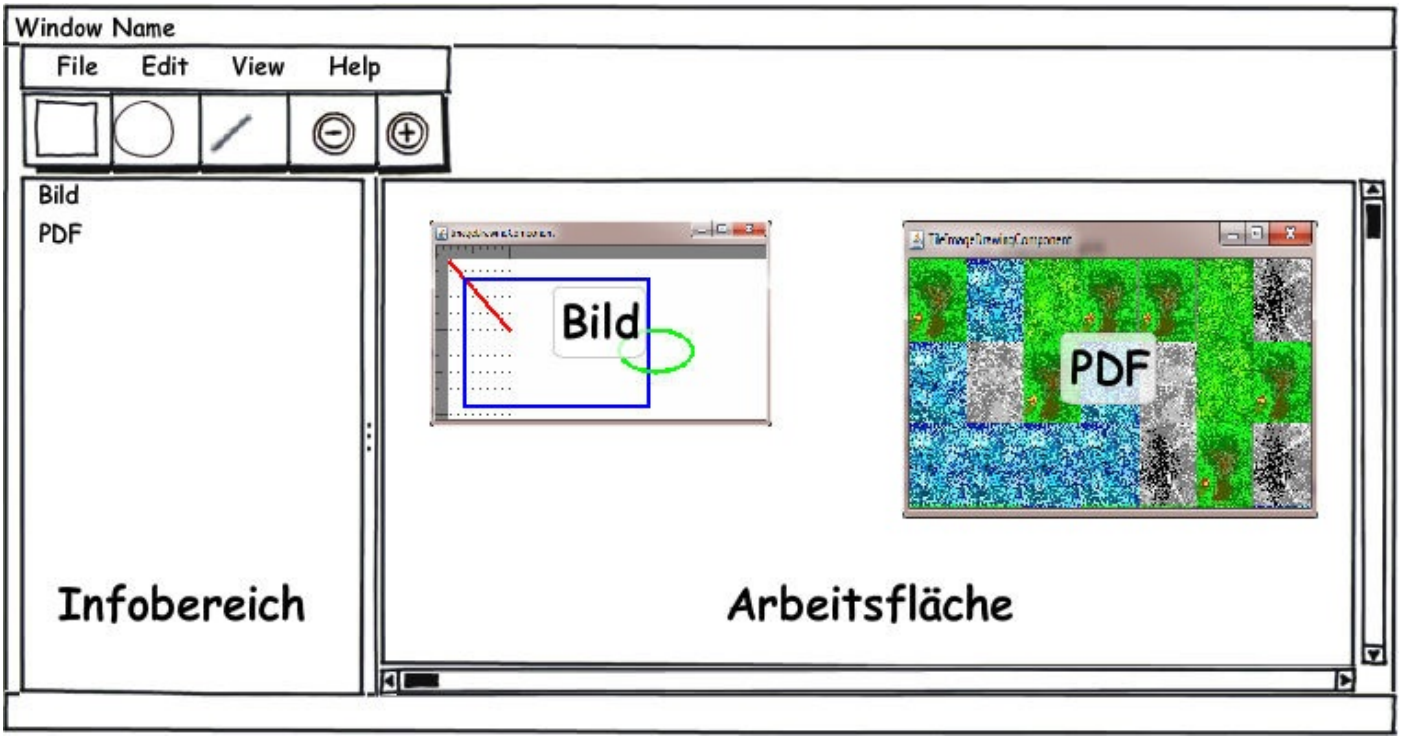


Abbildung 18-1 *UI-Entwurfskizze der Beispielapplikation*

Auf der Arbeitsfläche werden sowohl die Zeichenoperationen ausgeführt als auch Bilder oder PDFs eingefügt und platziert. Als Layouthilfe sind ein Lineal sowie ein Raster gewünscht. Der Infobereich soll eine listenartige Übersicht über die in der Arbeitsfläche enthaltenen und dargestellten grafischen Elemente bieten und kann als Navigationshilfe bei Überlappungen oder Verdeckungen von Elementen und zur Mehrfachselektion dienen. Die Applikation soll sowohl ein deutsches als auch ein englisches GUI bieten. Als Randbedingung sollen alle Aktionen relativ zügig ausgeführt werden bzw. bei länger andauernden Aktionen ein paralleles Weiterarbeiten oder der Abbruch einer Aktion möglich sein. Obwohl diese Anforderung eigentlich natürlich ist und für nahezu jede interaktive Anwendung zutrifft, wird diese Anforderung in diversen Applikationen jedoch nur unzureichend umgesetzt.

Entwurfsideen

Die zu zeichnenden Elemente, also die grafischen Figuren, wollen wir einheitlich behandeln. Daher definieren wir ein gemeinsames Interface sowie eine abstrakte Basisklasse, wie dies in Abschnitt 3.2 vorgestellt wurde.

Wir werden zur Konstruktion des grafischen Editors folgende Muster[1] nutzen:

- **Model-View-Controller** (MVC)[2] – Er werden Views (Infobereich, Arbeitsfläche) mit verschieden Darstellungen gleicher Modelldaten realisiert.
- **Beobachter (Observer)** – Änderungen im Modell werden an die Views propagiert und führen zu Änderungen in der Übersichtsliste und auf der Zeichenfläche.
- **Schablonenmethode (Template-Method)** – Das Zeichnen auf der

Arbeitsfläche erfolgt nach einem genau definierten Algorithmus, der zunächst Lineale sowie Gitterstützpunkte und anschließend Figuren zeichnet. Eine Basisklasse erledigt das Zeichnen der Infrastruktur (Lineal und Raster), definiert aber keine Methode zum Zeichnen der Figuren. Diese wird in einer Spezialisierung realisiert.

- **I**TERATOR – Dieses Muster verwenden wir häufig, etwa zum Durchlaufen des Datenmodells beim Zeichnen verschiedener Figuren.
- **P**ROTOTYP **(P**ROTOTYPE**)** – Dieses Muster hilft dabei, Copy-Paste-Operationen zu erleichtern.
- **B**EFEHL **(C**OMMAND**)** – In Menüs und Toolbars sind häufig Aktionen mit gleicher Funktionalität definiert. Diese Aktionen, wie etwa Laden oder Speichern, können über einen Menüeintrag, Shortcut, Toolbar-Button usw. ausgelöst werden. Um diese Funktionalität nicht mehrfach auszuprogrammieren, also um Sourcecode Duplikation zu vermeiden, werden die Aktionen in Form von speziellen Befehlsobjekten implementiert. Diese werden jeweils an der aufrufenden Stelle erzeugt und ausgeführt. Eine Undo-Funktionalität wird durch Einsatz dieses Musters nachträglich leichter realisierbar.
- **S**INGLETON – Ein zentraler Ressourcenmanager erlaubt den Zugriff auf verschiedene Sprachversionen.
- **E**RZEUGUNGSMETHODE und **F**ABRIKMETHODE **(F**ACTORY METHOD**)** – Beide helfen beim Erzeugen von Objekten.

18.1 Erzeugungsmuster

Erzeugungsmuster dienen dazu, den Konstruktionsprozess von Objekten zu vereinfachen. Wenn Methoden statt Konstruktoren zum Erzeugen von Objekten verwendet werden, so entspricht dies den Ideen der Muster ERZEUGUNGSMETHODE bzw. FABRIKMETHODE. Auch mithilfe des Musters ERBAUER kann man die Objektkonstruktion einfacher gestalten. Durch ein SINGLETON wird sichergestellt, dass lediglich eine Instanz einer Klasse erzeugt und ein definierter Zugriffspunkt darauf angeboten wird. Mit einem PROTOTYP können Objekte leicht aus einem definierten Mustersatz durch Kopie und Parametrierung erzeugt werden.

18.1.1 Erzeugungsmethode

Beschreibung und Motivation

Man kann sich die ERZEUGUNGSMETHODE etwa wie eine Bestellung in einem Restaurant vorstellen. Man sagt, welches Gericht man essen möchte, nennt jedoch nicht jede Zutat. Auch die Art der Zubereitung bleibt verborgen.

Auf ein Programm übertragen bedeutet dies Folgendes: Wenn der Konstruktionsprozess komplex oder fehleranfällig ist, weil beispielsweise viele Parameter übergeben werden müssen, so wird eine Erzeugung durch einen Konstruktoraufruf von außen vermieden. Stattdessen wird die Klasse selbst für die Objekterzeugung verantwortlich gemacht und stellt zur Objektkonstruktion eine spezielle, statische Konstruktionsmethode, die Erzeugungsmethode, bereit. Um Objekterzeugungen von außen ohne Aufruf der Erzeugungsmethode sicher zu verhindern, dürfen lediglich private Konstruktoren bereitgestellt werden.

Struktur

Betrachten wir eine vereinfachte Klasse `DrawingPad`, die entweder als Applet oder als Applikation gestartet werden kann. Dies wird durch den Wert des Konstruktorparameters `runAsApplet` entschieden. Die Signatur des Konstruktors sieht wie folgt aus:

```
DrawingPad(final String title, final boolean runAsApplet,
           final JApplet applet, final ConsoleParams appParams)
```

Die beiden Parameter `applet` bzw. `appParams` sind jeweils nur in einem der beiden Modi sinnvoll zu belegen. Für Applikationen bleibt daher der `applet`-Parameter unbelegt (`null`). Für Applets wird für `appParams` der Wert `null` übergeben. Damit käme es beispielsweise zu folgenden Konstruktoraufrufen:

```
new DrawingPad("Application", false, null, consoleParams);
new DrawingPad("Applet", true, applet, null);
```

Betreiben wir eine kurze Analyse, um einige Probleme bei der Konstruktion eines `DrawingPad`-Objekts aufzudecken.

1. Wirklich lesbar und verständlich sind die gezeigten Konstruktoraufrufe mit `null`-Werten und den booleschen Literalen `true` und `false` nicht.
2. Existieren optionale Parameter, so steigt die Gefahr, an falscher Stelle `null`-Werte zu übergeben und dadurch unerwartete Initialisierungen vorzunehmen.
3. Es ist keine Kontrolle möglich, dass alle Werte konsistent übergeben werden. Im Speziellen können sowohl eine `JApplet`-Referenz als auch eine Referenz auf `ConsoleParams` übergeben werden. Zudem muss darauf geachtet werden, dass der boolesche Wert `runAsApplet` passend übergeben wird. Man kann sich schnell folgende Fragen stellen: Welcher Wert soll Vorrang haben? Sollen beide Werte ausgewertet werden? Und wie drückt man Letzteres durch einen booleschen Parameter aus?
4. Der boolesche Wert `runAsApplet` ist redundant, da er sich anhand der Aussage `applet != null` erschließen lässt. Schlecht an dieser Redundanz ist zweierlei: Erstens ist eine widersprüchliche Eingabe möglich, da `runAsApplet` als `false` in Kombination mit einer gültigen `JApplet`-Referenz übergeben werden kann. Gleiches gilt für die Kombination mit einer `ConsoleParams`-Referenz und dem Wert `true`. Zweitens wird dieses Implementierungsdetail in der öffentlichen

Schnittstelle an alle möglichen Aufrufer kommuniziert. Eine nachträgliche Korrektur würde Änderungen an allen verwendenden Stellen bedingen.

Als Verbesserung bietet sich an, zwei Erzeugungsmethoden zu nutzen: Eine für die Konstruktion von Applets und eine für Applikationen. Diese beiden Methoden bekommen die sprechenden Namen `createAsApplet()` und `createAsApplication()`. Sie werden wie folgt realisiert:

```
private static final boolean RUN_AS_APPLET = true;

public static DrawingPad createAsApplet(final String title,
                                        final JApplet applet)
{
    final ConsoleParams NO_APP_PARAMS = null ;
    return new DrawingPad(title, RUN_AS_APPLET, applet, NO_APP_PARAMS);
}

public static DrawingPad createAsApplication(final String title,
                                             final ConsoleParams appParams)
{
    final JApplet NO_APPLET = null;
    return new DrawingPad(title, ! RUN_AS_APPLET, NO_APPLET, appParams);
}
```

Anhand dieses Beispiels lernen wir eine weitere, nützliche Technik kennen: Statt Magic-Konstanten[3] zu benutzen, werden hier für die Werte `null` und `true` jeweils Konstanten mit sprechendem Namen (hier `NO_APP_PARAMS` bzw. `RUN_AS_APPLET`) definiert. Dadurch erhöht sich die Lesbarkeit und die Intention wird verdeutlicht. Durch Einsatz der Erzeugungsmethoden benötigt man zudem weniger Übergabeparameter für Aufrufe durch Klienten – die zur Objektkonstruktion intern erforderliche Anzahl bleibt natürlich gleich. Allerdings werden die Details der ausgelassenen Parameter und der Redundanz in der Klasse selbst versteckt. Eine spätere Korrektur wird unabhängig von externen Aufrufern möglich.

Erweiterungen

Der Einsatz einer Erzeugungsmethode ermöglicht häufig, die Klasse gemäß dem in Abschnitt 3.4.2 vorgestellten IMMUTABLE-Muster unveränderlich zu machen und dementsprechend Zustandsänderungen nach der Konstruktion zu verhindern. Selbst wenn dies nicht vollständig gelingt, kann eine Reduktion der veränderlichen Attribute auf ein tatsächlich benötigtes Minimum die Übersichtlichkeit fördern. Eine schrittweise Anleitung zum Reduzieren variabler Anteile einer Klasse beschreibt das Refactoring MINIMIERE VERÄNDERLICHE ATTRIBUTE in Abschnitt 17.4.2.

Verwendet man eine Erzeugungsmethode, so ist es bei Bedarf leicht möglich, eine Fehlerbehandlung zu integrieren. Man kann so gewährleisten, dass ein Objekt immer vollständig initialisiert ist, bevor es verwendet wird. Treten Fehler während der Konstruktion auf, kann der Wert `null` oder ein NULL-OBJEKT (vgl. Abschnitt 18.3.2) zurückgeliefert werden. Stellt ein Fehler allerdings eine außergewöhnliche Situation dar, sollte dies besser mit einer Exception ausgedrückt werden. Eine

Diskussion verschiedener Arten der Fehlerbehandlung finden Sie in Abschnitt 4.7.

Bewertung

Der Einsatz einer ERZEUGUNGSMETHODE bewirkt Folgendes:

+ **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt, was für mehr Lesbarkeit sorgt: Die Signatur einer Erzeugungsmethode ist in der Regel kürzer als die Parameterliste des Konstruktors. Dies ist möglich, weil lediglich einige der Parameter des Konstruktors entgegengenommen und die restlichen beim Aufruf des Konstruktors mit Defaultwerten belegt werden. Zudem kann man für eine Erzeugungsmethode einen sprechenden Namen wählen, der die erzeugten Objekte besser charakterisiert als der Konstruktor mit dem Namen der Klasse.
+ **Konstruktionssicherheit** – Es lassen sich vollständig und korrekt initialisierte Objekte erzeugen: Innerhalb einer Erzeugungsmethode können alle Parameter vor Übergabe und Konstruktion des Objekts validiert werden. Dadurch wird es möglich, die Instanzerzeugung abzusichern oder zu erleichtern und angemessen auf Fehlersituationen zu reagieren. Weiterhin kann man eine Art Transaktion bei der Objekterzeugung erreichen: Ein Objekt wird entweder vollständig erzeugt oder im Fehlerfall kein Objekt, sondern `null` zurückgeliefert. Daher leistet eine Erzeugungsmethode bei der Behandlung von Fehlern im Konstruktionsprozess gute Dienste.
○ **Mehraufwand** – Minimal mehr Sourcecode ist erforderlich.

18.1.2 Fabrikmethode (Factory method)

Beschreibung und Motivation

Die Idee hinter dem Muster FABRIKMETHODE ist ähnlich einer Produktion in einer realen Fabrik, die aufgrund einer Bestellung gewünschte Dinge herstellen kann. Bei einer solchen Bestellung sind nur die Artikelnummern oder Bezeichnungen der Teile, nicht aber die konkreten Ausprägungen und Realisierungen bekannt. Es findet demnach analog zum Muster ERZEUGUNGSMETHODE eine Kapselung der Objekterzeugung statt: Objekte werden nicht direkt per Konstruktoraufruf erzeugt, sondern dieser Vorgang wird an ein spezielles Objekt, eine sogenannte Fabrik, delegiert. Die dort definierten Fabrikmethoden erzeugen Objekte verschiedenen Typs.

Der Unterschied zum Muster ERZEUGUNGSMETHODE, das den Konstruktionsprozess einer speziellen Klasse vereinfacht, liegt darin, dass beim Muster FABRIKMETHODE eine andere Klasse, die Fabrik, die Konstruktion von Objekten eines gewünschten konfigurierbaren Typs durchführt.

Struktur

Es wird eine Fabrikklasse `Factory` mit einer Methode `createProduct( ProductType)` zum Erzeugen von Objekten definiert. Dort wird anhand eines Parameters `productType` entschieden, welcher konkrete Typ erzeugt wird. Der gewünschte zu erzeugende Typ wird dabei entweder, wie in diesem Fall, als Parameter übergeben oder aus einer Konfiguration gelesen. Damit überhaupt verschiedene Produkte von einer Methode erzeugt werden können, müssen diese Produkte ein gemeinsames Interface `IProduct`, eine gemeinsame Basisklasse `AbstractProduct` oder beides besitzen. Klienten arbeiten immer mit dieser Abstraktion des konkreten Produkts, wodurch es automatisch zu einer besseren Kapselung und loseren Kopplung zum verwendenden Applikationscode kommt. Abbildung 18-2 zeigt eine mögliche Realisierung als UMLDiagramm.

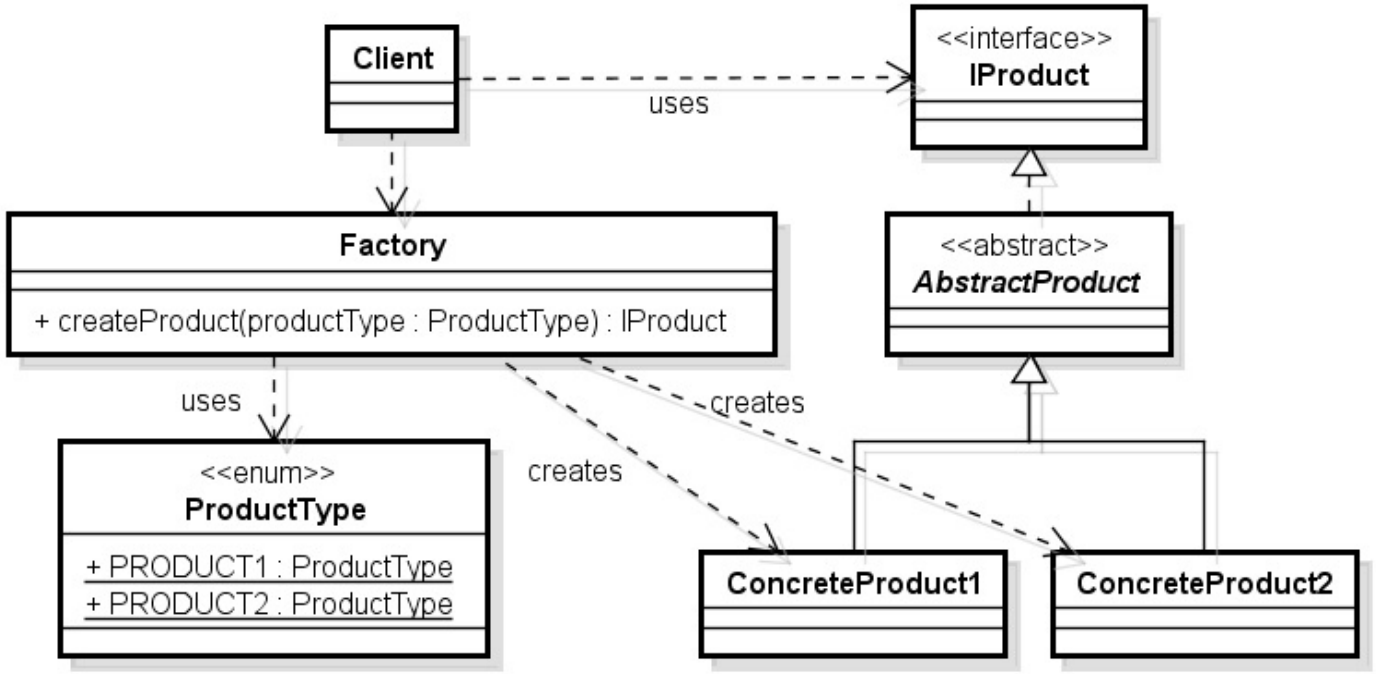


Abbildung 18-2 *Fabrikmethode*

Beispiel

Die Fabrikmethode `createGraphicsElement( ...)` wird in einer Klasse `MenuActionHandler` definiert. Dort wird anhand des übergebenen `enum`-Werts entschieden, von welchem Typ eine neue Instanz erzeugt wird. Abbildung 18-3 zeigt eine mögliche Realisierung eines Teils unserer Grafikanwendung als UML-Diagramm.

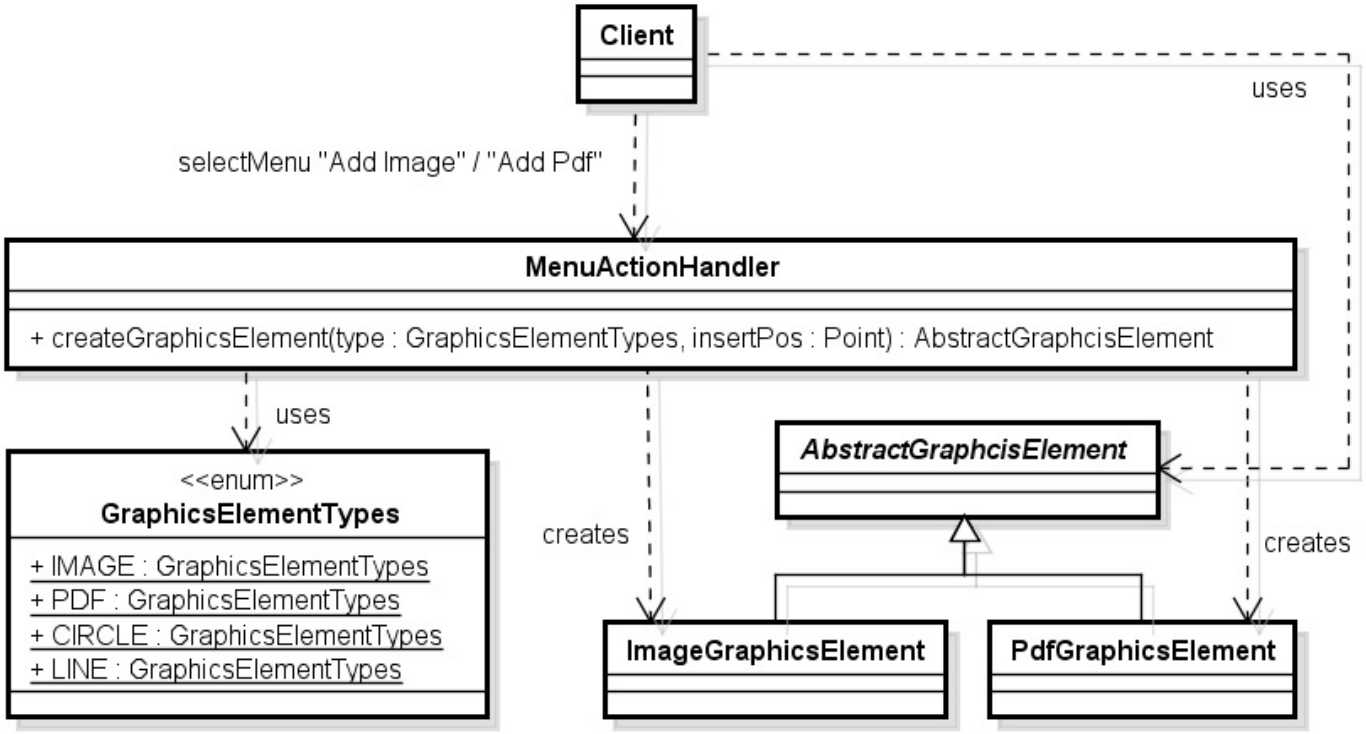


Abbildung 18-3 *Fabrikmethode konkret*

Abhängig von dem übergebenen Aufzählungswert `type` wird ein Objekt eines korrespondierenden grafischen Elements erzeugt, beispielsweise `ImageGraphicsElement`. Dies zeigt der folgende Sourcecode-Ausschnitt:

```
public static AbstractGraphicsElement createGraphicsElement(final
      GraphicsElementTypes type, final Point insertPos)
{
    switch (type)
    {
        case IMAGE:
            return new ImageGraphicsElement(insertPos.x, insertPos.y);
        case PDF:
            return new PdfGraphicsElement(insertPos.x, insertPos.y);

            // …
    }
    throw new IllegalStateException("Unexpected 'GraphicsElementTypes' '" +
                                    type + "'");
}
```

Bewertung

Der Einsatz des Musters FABRIKMETHODE besitzt durch die Ähnlichkeit zum Muster ERZEUGUNGSMETHODE nahezu dieselben Implikationen, weshalb diese in der folgenden Aufzählung nur kurz genannt werden. Als Erweiterung wird beim Muster FABRIKMETHODE allerdings nicht nur der Konstruktionsprozess von Klassen vereinfacht, sondern der Konstruktionsprozess an sich gekapselt: Die Konstruktion von Objekten eines gewünschten, konfigurierbaren Typs wird durch eine externe Klasse realisiert. Es findet somit eine Abstraktion von den konkret erzeugten Klassen statt.

Der Einsatz einer FABRIKMETHODE hat folgende Auswirkungen:

+ **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt, was für mehr Lesbarkeit sorgt.
+ **Abstraktion** – Der konkret erzeugte Objekttyp kann vor dem Aufrufer versteckt werden, indem lediglich eine Referenz auf ein Interface oder eine abstrakte Klasse des erzeugten Objekttyps zurückgegeben wird. Dies verstärkt zudem den Effekt der Kapselung und führt zu einer loseren Kopplung.
+ **Kapselung** – Die Kapselung wird verstärkt: Ein Klient nutzt lediglich eine Schnittstelle zur Erzeugung, wodurch die konkrete Realisierung einer Fabrikklasse, der Erzeuger, ausgetauscht werden kann. Durch die zuvor erwähnte Abstraktion vom konkret erzeugten Objekt können unterschiedliche Fabrikklassen verschiedene konkrete Typen eines gemeinsamen Basistyps zurückliefern.
+ **Konstruktionssicherheit** – Es können immer vollständig initialisierte Objekte erzeugt werden, wodurch eine Konsistenzprüfung und Fehlerbehandlung erleichtert wird.

○ **Mehraufwand** – Es entsteht ein wenig mehr Sourcecode, da hier zumindest eine Fabrikklasse und eine Basisklasse für die Produktklassen benötigt werden.

○ **Mehr Komplexität** – Die weiteren Klassen führen zu etwas mehr Komplexität.

18.1.3 Erbauer (Builder)

Beschreibung und Motivation

Man kann sich das Vorgehen beim ERBAUER-Muster wie bei der Bestellung einer Pizza oder eines Autos vorstellen. Auf einer Bestellliste kann man aus vielen verschiedenen Bestandteilen und Extras genau diejenigen ankreuzen, die man bekommen möchte. Man gibt also eine spezielle Bestellung auf und konfiguriert sich damit sein Objekt mit den gewünschten Eigenschaften. Erst nach Abschluss der Auswahl wird mit der eigentlichen Herstellung begonnen.

Das ERBAUER-Muster kann immer dann sinnvoll eingesetzt werden, wenn ein zu erzeugendes Objekt viele optionale oder komplex zu konstruierende Bestandteile besitzt. Wenn einzelne, zur Objektkonstruktion benötigte Werte iterativ ermittelt werden (etwa aus einem Stream einzeln einlaufen), kann durch den Einsatz des ERBAUER-Musters eine Initialisierung durch einen sukzessiven Aufruf von `set()`-Methoden vermieden werden und einen gültigen Objektzustand sicherstellen (vgl. Abschnitt 3.1.5).

Im Unterschied zum Muster FABRIKMETHODE, bei dem eine abstrakte Bezeichnung, beispielsweise eine Artikelnummer, das zu erzeugende Produkt beschreibt, liegt der Fokus beim Einsatz des ERBAUER-Musters darauf, die einzelnen

Bestandteile eines Objekts auf einfache Weise zu spezifizieren. Dabei werden zudem die Ideen des ERZEUGUNGSMETHODE-Musters aufgegriffen, die eine sichere Konstruktion adressieren. Betrachten wir dies nun etwas konkreter.

Struktur

Eine Klasse `Builder` definiert eine Fabrikmethode `create()`, die später zur Konstruktion des gewünschten `Product`-Objekts aufgerufen werden muss. Um die Erzeugung von Objekten mittels Konstruktoraufruf durch externe Klassen zu verhindern, definiert man den Konstruktor der `Product`-Klasse mit der Sichtbarkeit `private`. Dies zeigt Abbildung 18-4.

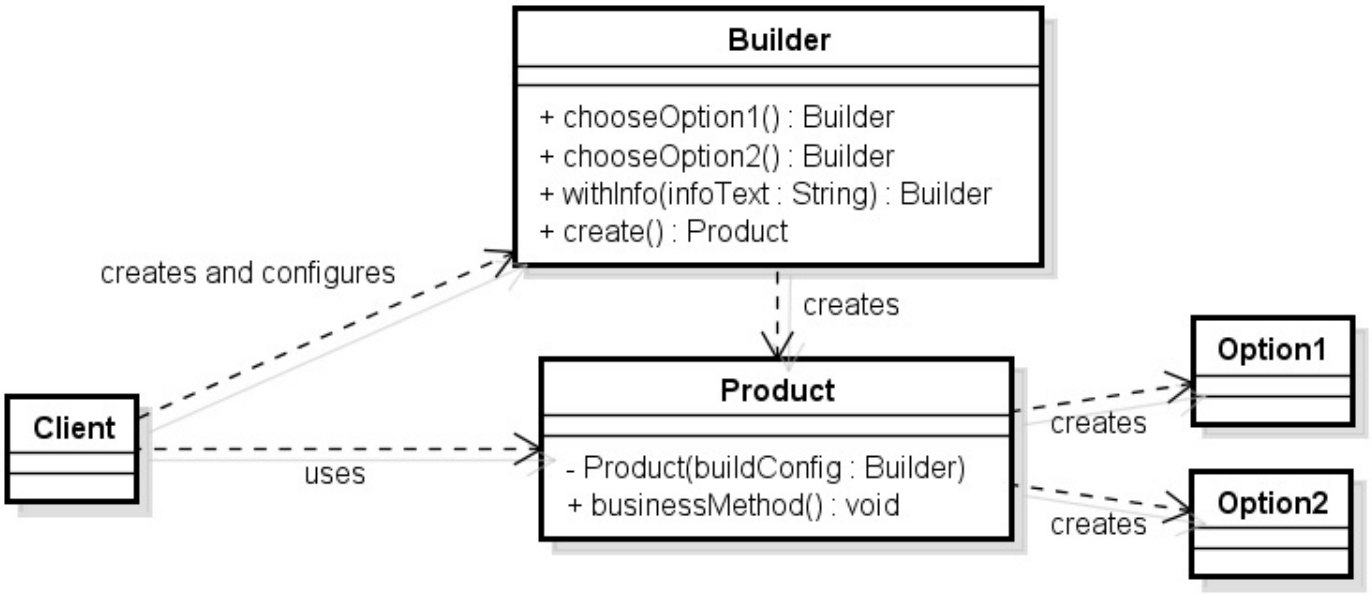


Abbildung 18-4 *Erbauer*

Die Klasse `Builder` sammelt zunächst alle Informationen, um die zu konstruierenden Teile zu ermitteln. Das Auswählen einer Eigenschaft erfolgt durch Aufruf spezieller Methoden mit aussagekräftigem Namen. Nachdem alle gewünschten Eigenschaften spezifiziert wurden, wird die Erstellung des eigentlichen Produkts veranlasst, wodurch auch eine Konstruktion der zu den gewünschten Eigenschaften korrespondierenden Teile erfolgt. Dies geschieht durch Aufruf der `create()`-Methode. Folgendes Sourcecode-Fragment zeigt das zuvor Beschriebene:

```
new Builder().chooseOption1().chooseOption2().create();
```

In diesem Fall ergibt sich durch die Hintereinanderausführung von Methoden eine lesbare und kurze Schreibweise. Voraussetzung dazu ist, dass die Auswahlmethoden jeweils eine Referenz auf die `Builder`-Instanz selbst zurückliefern.

Tipp: Hintereinanderausführung von Methoden

In vielen Situationen ist eine Hintereinanderausführung von Methoden zu vermeiden, da man sich nicht auf korrekte Rückgabewerte verlassen kann und die Verkettung bei`get()`-Methoden eher unleserlich wird.

Für die Methoden im ERBAUER mit sprechenden Namen gilt diese Kritik nicht, da hier garantiert ein gültiges Objekt zurückgeliefert wird und eine Verkettung häufig die Lesbarkeit sogar erhöhen kann.

Beispiel

In dem folgenden Beispiel wird die Klasse `PizzaBuilder` genutzt, um die Eigenschaften zweier verschiedener `Pizza`-Objekte zu spezifizieren und diese anschließend zu konstruieren:

```
public static void main(final String[] args)
{
    final PizzaBuilder builder = new PizzaBuilder();
    builder.mitExtraSardellen();
    System.out.println("Normale Pizza mit extra Sardellen: " +
builder.create());

    final PizzaBuilder builder2 = new PizzaBuilder();
    builder2.mitSalami().small().bitteBeachten("Ohne Mais!");
    System.out.println("Kleine Salami-Pizza ohne Mais: " + builder2.create());
}
```

Listing 18.1 *Ausführbar als* **'BUILDEREXAMPLE'**

Die (statische innere) Klasse `PizzaBuilder` ist ähnlich zum Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) realisiert und enthält die Wunschliste der Optionen:

```
public static final class PizzaBuilder
{
    // Defaultwerte, gelten wenn keine korrespondierende Methode aufgerufen
wird
    boolean mitSalami         = false;
    boolean mitExtraSardellen = false;
    String  info              = "";
    Size    size              = Size.MEDIUM;

    public PizzaBuilder mitSalami()
    {
        this.mitSalami = true;
        return this;
    }

    public PizzaBuilder mitExtraSardellen()
    {
        this.mitExtraSardellen = true;
        return this;
    }

    public PizzaBuilder small()
    {
        this.size = Size.SMALL;
        return this;
    }

    public PizzaBuilder bitteBeachten(final String info)
    {
        this.info = info;
        return this;
```

```
    }

    public Pizza create()
    {
        return new Pizza(this);
    }
    // …
```

Bewertung

Der Einsatz des ERBAUER-Musters hat folgende Auswirkungen:

+ **Lesbarkeit und Vereinfachung** – Die Details des Konstruktionsprozesses werden versteckt, was für mehr Lesbarkeit sorgt: Es werden lediglich gewünschte Eigenschaften spezifiziert. Die eigentliche Konstruktion wird intern geregelt und bleibt verborgen. Zudem kann man sprechende Methodennamen wählen, die die Eigenschaften des zu erzeugenden Objekts besser beschreiben als die Namen von Parametern.
+ **Konstruktionssicherheit** – Es lassen sich vollständig und korrekt initialisierte Objekte erzeugen: Die Parametrierung erfolgt ausschließlich über einige Methoden, die die korrespondierenden Attribute setzen und später als Eingabewerte für den Konstruktor des zu erzeugenden Produkts dienen.
+ **Unterstützung optionaler Attribute** – Der Umgang mit vielen optionalen Attributen wird erleichtert: Defaultwerte müssen nicht explizit gesetzt bzw. im Konstruktor übergeben werden.
- **Mehr Komplexität** – Der Einsatz führt zu mehr Komplexität und Sourcecode, da gewünschte Wertebelegungen von Attributen über den Aufruf von Methoden spezifiziert werden müssen. Der für Aufrufer verborgene Konstruktionsprozess sorgt für mehr Komplexität innerhalb der Klassen des ERBAUER-Musters.

18.1.4 Singleton

Beschreibung und Motivation

Möchte man sicherstellen, dass höchstens eine Instanz einer bestimmten Klasse erzeugt werden kann, so sollte das SINGLETON-Muster zum Einsatz kommen. Außerdem wird ein globaler Zugriffspunkt auf die Instanz bereitgestellt.

Beispiele für Einsatzgebiete sind jede Art von zentraler Registrierung: Wenn ein Programm über Plugins erweiterbar ist, werden diese zweckmäßigerweise über einen zentralen Plugin-Manager verwaltet. Ebenso werden Fabrikklassen häufig gemäß dem SINGLETON-Muster implementiert, da man Klienten nicht zusätzlich die Erzeugung von Fabrikklassen auferlegen möchte.

Struktur

Damit lediglich eine Instanz erzeugt werden kann, ist es wichtig, den Konstruktionsprozess in der Klasse selbst zu kapseln und diese für die Verwaltung ihres einzigen Exemplars zuständig zu machen. Es kommt eine spezielle Form des ERZEUGUNGSMETHODE-Musters zum Einsatz, für die folgende Realisierungsanforderungen existieren:

1. Um eine Objekterzeugung außerhalb der Klasse zu verhindern, verwenden wir einen privaten Konstruktor.
2. Zur Speicherung der einzigen Instanz wird ein statisches Attribut `INSTANCE` verwendet. Der Zugriff auf dieses erfolgt ausschließlich über eine statische Zugriffsmethode, typischerweise `getInstance()` genannt.

Die Umsetzung dieser Forderungen ist in Abbildung 18-5 als Klassendiagramm dargestellt. Dabei werden stellvertretend für beliebige Attribute einer `Singleton`-Klasse zwei Attribute `attribute1` und `attribute2` in die Modellierung aufgenommen.

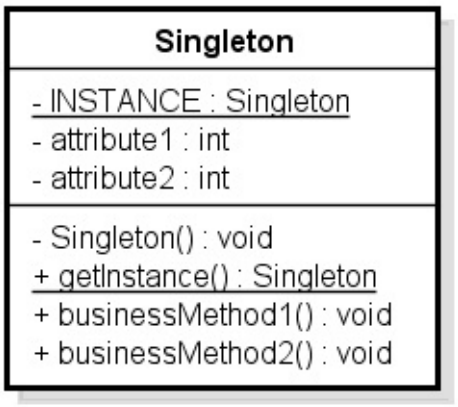


Abbildung 18-5 *Singleton*

Beispiel

Ein erster intuitiver Entwurf, den man häufig so oder ähnlich in Sourcecode findet, sieht wie folgt aus:

```
public final class BadSingleton
{
    private static BadSingleton INSTANCE = null;

    private int attribute1;
    private int attribute2;

    // ACHTUNG: SCHLECHT !!!
    public static BadSingleton getInstance()
    {
        if (INSTANCE == null)
        {
            INSTANCE = new BadSingleton();
        }
```

```
        return INSTANCE;
    }

    private BadSingleton()
    {
    }

    public void businessMethod1() { /* …*/ }
    public void businessMethod2() { /* …*/ }
}
```

Auf den ersten Blick sieht diese Umsetzung schon ganz gut aus. Aber wie bereits in Jon Bentleys »Perlen der Programmierkunst« [6] erwähnt, ist die erste Idee meistens nicht die beste.

In Singlethreading-Umgebungen ist die obige Implementierung akzeptabel. Probleme offenbaren sich erst beim Einsatz in Multithreading-Umgebungen: Wie bereits in Kapitel 7 erwähnt, ist diese Realisierung nicht Thread-sicher. Rufen mehrere Threads nahezu zeitgleich diese `getInstance()`-Methode auf, so kann es zu verschiedenen Race Conditions kommen. Im einfachsten Fall führen mehrere Threads die Prüfung auf `null` durch und werden anschließend unterbrochen. Alle Threads erzeugen als Folge »ihr« eigenes Singleton.

Eine mögliche Lösung ist das Synchronisieren der `getInstance()`-Methode, da durch das Schlüsselwort `synchronized` die Möglichkeit des Zugriffs durch mehrere Threads unterbunden und zusätzlich eine konsistente Sicht aller Threads auf den aktuellen Zustand der Variablen garantiert wird:

```
public static synchronized BetterSingleton getInstance()
{
    if (INSTANCE == null)
    {
        INSTANCE = new BetterSingleton();
    }
    return INSTANCE;
}
```

Bei dieser Realisierung erfolgt bei jedem Zugriff auf die synchronisierte Methode ein gegenseitiger Ausschluss und die Ausführung als kritischer Bereich. Dieses Vorgehen wird eigentlich nur für die erste Initialisierung des statischen Attributs `INSTANCE` benötigt. Für alle weiteren Zugriffe ist dieser Aufwand weder erforderlich noch gewünscht, weil dadurch Nebenläufigkeit verhindert wird: Rufen andere Komponenten häufig die `getInstance()`-Methode auf, so stellt diese einen Flaschenhals dar. Zugriffe können nicht parallel erfolgen, sondern werden sukzessive ausgeführt.

Als SINGLETON realisierte Klassen stellen in der Regel zentrale und vielfach benötigte Komponenten dar. Die zuvor gezeigte Lazy Initialization ist für ein Singleton aber eher fragwürdig, weil sie vor allem für optionale und aufwendig zu konstruierende Komponenten gedacht ist. Außerdem verursacht gerade diese Technik unter Umständen verschiedenste Probleme beim Multithreading.

Tatsächlich lässt sich der Sourcecode viel einfacher, übersichtlicher und kürzer

formulieren, wenn man auf Lazy Initialization verzichtet. Man nutzt stattdessen eine direkte, Thread-sichere, statische Initialisierung des statischen Attributs `INSTANCE`:

```
public final class Singleton
{
    private static final Singleton INSTANCE = new Singleton();

    private int attribute1;
    private int attribute2;

    public static Singleton getInstance()
    {
        return INSTANCE;
    }

    private Singleton()
    {
    }

    public void businessMethod1() { /* …*/ }
    public void businessMethod2() { /* …*/ }
}
```

Thread-Sicherheit ist bei dieser Art der Realisierung dadurch gegeben, dass die Klassenbeschreibung einmal geladen wird und anschließend eine Initialisierung erfolgt, bevor die Klasse für andere Klassen zugreifbar wird.

Folgende Vorteile ergeben sich zusätzlich: Im Unterschied zum ersten Ansatz wird das Objekt immer erzeugt und steht bei der ersten Benutzung der Klasse zur Verfügung. Dadurch entfällt der Bedarf für die Synchronisation. Die Deklaration als `final` sorgt zudem dafür, dass sich die Referenz `INSTANCE` nicht mehr ändern kann und somit auch keine Prüfungen auf `null` notwendig sind.

Besondere Variante basierend auf `enum`

Die zuvor gezeigte Lösung war bis JDK 1.4 die kürzeste und eleganteste Realisierung eines SINGLETONs. Seit JDK 5 kann man eine Besonderheit von `enum`-Aufzählungen nutzen. Diese sind immer eindeutig und nur einmal in einer JVM vorhanden. Somit lässt sich ein SINGLETON als eine einelementige Aufzählung wie folgt definieren:

```
public enum Singleton
{
   INSTANCE;

   private int attribute1;
   private int attribute2;

   public void businessMethod1() { /* …*/ }
   public void businessMethod2() { /* …*/ }
}
```

So elegant diese Lösung auch aussieht, sie besitzt doch ein paar Nachteile. Einer davon ist, dass `enum`-Aufzählungen implizit von der Basisklasse `Enum<E extends Enum<E≫` abgeleitet sind. Für Singletons sind Basisklassen aber in der Regel nicht wünschenswert. Man erbt in diesem Fall nämlich Funktionalität in Form der

Methoden `name()`, `ordinal()` sowie `compareTo(Singleton)`. Im besten Fall ist das überraschend, meistens aber eher verwirrend und lenkt von den eigentlichen Business-Methoden ab. Es kommt noch schlimmer: Für ein Singleton erwartet man nur eine statische Methode, nämlich `getInstance()`. Obige Realisierung basierend auf `enum` bietet aber diverse statische Methoden, etwa `values()`, `valueOf(String)` usw.

Bewertung

Der Einsatz eines SINGLETONs hat folgende Auswirkungen:

+ **Zentraler Zugriffspunkt** – Es gibt einen zentralen Zugriffspunkt auf benötigte Funktionalität, der eine sichere Initialisierung erlaubt.
+ **Strukturierung** – Die Zugriffe auf das SINGLETON werden strukturiert und Implementierungsdetails lassen sich verstecken.
○ **Realisierungsprobleme** – Der Einsatz ist komplizierter, als man meint: Intuitive Lösungen unter Verwendung von Lazy Initialization führen zu Problemen in Multithreading-Umgebungen. In der Vergangenheit wurden dazu komplexe, zum Teil fehlerhafte und schlecht lesbare Realisierungen entwickelt, unter anderem das sogenannte ***Double Checked Locking***.[4] Einfacher und zudem korrekt ist die zuvor vorgestellte Lösung. Ein subtileres Problem lässt sich dadurch jedoch nicht lösen: die im folgenden Punkt beschriebene fehlende Eindeutigkeitsgarantie!
- **Keine Eindeutigkeitsgarantie** – Der Einsatz ist problematisch, wenn man mehrere `ClassLoader`[5] verwendet. Es gibt dann keine Eindeutigkeitsgarantie mehr und es kann dann mehrere Instanzen eines Singletons geben – pro `ClassLoader` eine. Daher müssen zusätzliche Betrachtungen angestellt werden. Dies wird im folgenden Hinweis ausführlicher beschrieben.

Achtung: Fallstricke und Probleme durch mehrere `ClassLoader`

Das SINGLETON-Muster soll dafür sorgen, dass nur eine einzige Instanz einer Klasse innerhalb einer Applikation erzeugt werden kann. Allerdings garantiert der Einsatz dieses Musters selbst innerhalb einer JVM nicht die Einzigartigkeit einer Instanz. ***Das* SINGLETON-*Muster sorgt in Java nur dafür, dass höchstens eine Instanz pro* `ClassLoader` *existiert. In einer JVM kann es also durchaus eben so viele Instanzen des »Singletons« geben, wie es* `ClassLoader` *gibt.***

Das kann z. B. bei einem Webserver der Fall sein, da dort zum Teil mehrere `ClassLoader`-Instanzen eingesetzt werden. Diesen Sachverhalt muss man im Hinterkopf haben, wenn ein SINGLETON eine Ressource kapseln soll, die nur einmal vorhanden ist. In komplexen Applikationen verschärfen sich die Probleme, wenn hier mit mehreren JVMs gearbeitet wird. Dasselbe gilt beim Einsatz von Applikationsservern. Wie man an dieser kurzen Diskussion erkennt, sind beim

Einsatz des SINGLETON-Musters in komplexen Applikationen einige Vorüberlegungen notwendig. Verwendet man allerdings nur einen `ClassLoader`, dann sind die hier angesprochenen Punkte nicht relevant, und man muss lediglich die im Text gegebenen Empfehlungen zur Realisierung beachten.

Achtung: SINGLETON als Anti-Pattern

Ich kenne Applikationen, die das SINGLETON-Muster als Allheilmittel ansehen: Dort ist nahezu jede Klasse als ein solches realisiert. In diesem Fall kann der Einsatz des Musters aber mit einer Ansammlung von statischen Variablen verglichen werden. Es ist aber nicht sinnvoll, überall Zugriff auf alle Klassen anzubieten. Im Normalfall sollte es nur einige wenige Klassen geben, die man zentral bereitstellen muss. Hält man sich nicht daran, so wird das SINGLETON schnell zu einem Anti-Pattern.

18.1.5 Prototyp (Prototype)

Beschreibung und Motivation

Das Vorgehen beim PROTOTYP-Muster besteht darin, basierend auf speziellen Vorlagenobjekten neue Objekte zu generieren. Man nutzt dieses Muster, wenn Instanzen einer Klasse einen großen gemeinsamen Grundstock an gleichen Werten ihrer Attribute haben und durch wenige Modifikationen gebrauchsfertig gemacht werden können. Außerdem kann der Einsatz zur Performance-Steigerung und zur ressourcenschonenden Objekterzeugung verwendet werden, wenn diese per Konstruktor zeitaufwendig ist (z. B. durch erneute Datenbankzugriffe zum Ermitteln der Werte der Attribute) und somit ein Kopieren der Ergebnisse und anschließendes Parametrieren günstiger ist.

Dieses Muster besitzt zwei Varianten: Bei der ***statischen*** Variante dienen eine oder mehrere Kopiervorlagen (Prototypen) als Basis für neue Objekte. Bei der ***dynamischen*** Variante kann sogar der Typ der Kopiervorlage zur Laufzeit unterschiedlich sein.

Struktur

Die Struktur für die statische Variante des Musters wird in Abbildung 18-6 gezeigt. Es existiert ein Objekt vom Typ `Prototype`, das eine `makeCopy()`-Methode anbietet, um sich selbst zu kopieren. Damit kann jedes Objekt dieses Typs als Kopiervorlage agieren. Hier wird bewusst nicht der Name `clone()` verwendet, um eine klare Abgrenzung zum `Clonable`-Interface aus dem JDK zu gewährleisten.

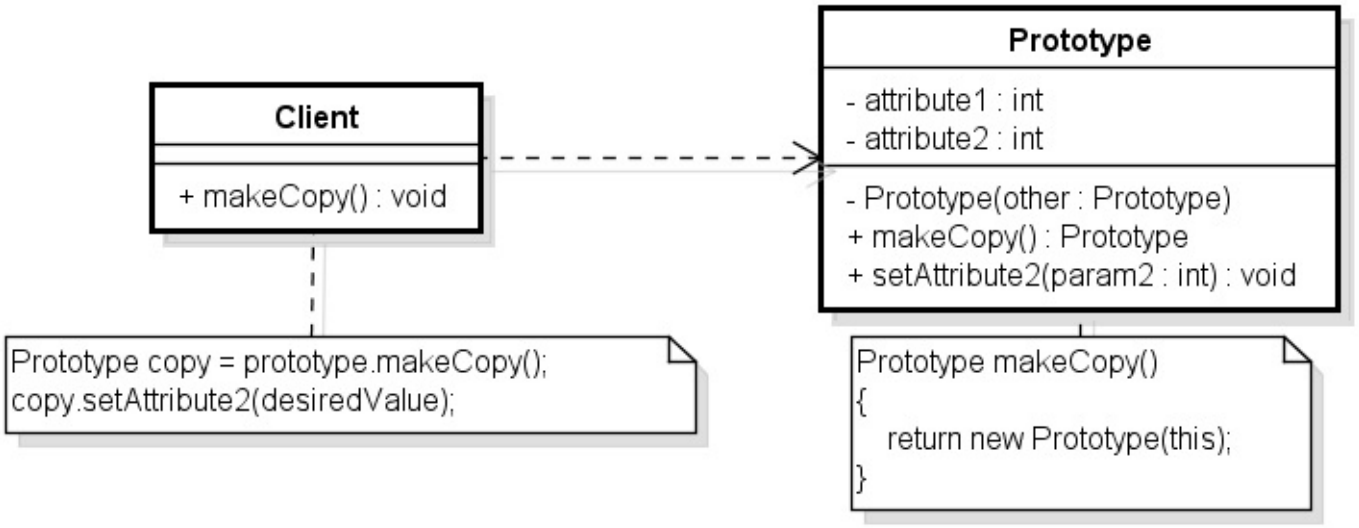


Abbildung 18-6 *Prototyp*

Nach einem Aufruf von `makeCopy()` können bei Bedarf weitere Anpassungen der Attributwerte durch `set()`-Methoden erfolgen, um das neu erstellte Objekt wie gewünscht zu parametrieren.

Beispiel

Die dynamische Variante stelle ich im Folgenden am vereinfachten Beispiel eines grafischen Editors vor, der über Buttons verschiedene Aktionen, wie Figuren zeichnen, kopieren und einfügen, erlaubt. Mehrfache Kopien sind in Abbildung 18-7 für einige Rechtecke gezeigt, wobei die aktuell selektierte Figur fett dargestellt ist.

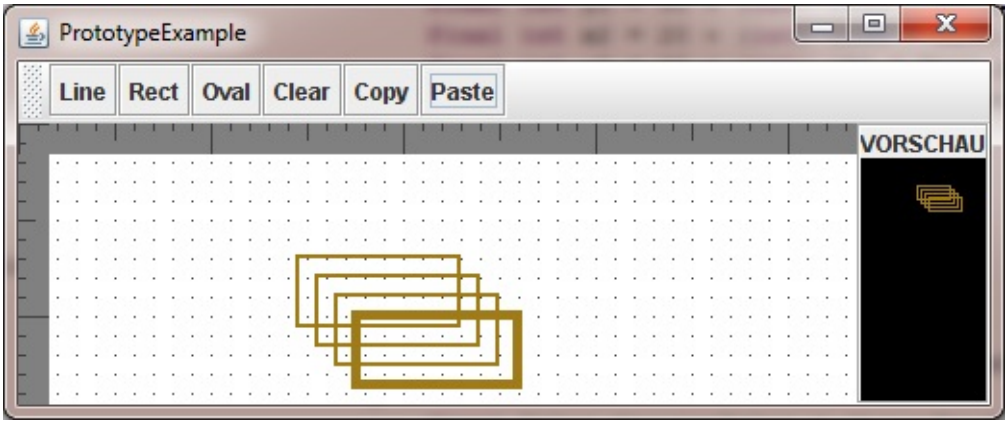


Abbildung 18-7 *Beispielapplikation unter Verwendung des* PROTOTYP*-Musters*

Wird der Copy- bzw. Paste-Button gewählt, so werden spezielle Methoden aufgerufen und das momentan auf der Zeichenfläche selektierte Element kopiert bzw. eingefügt. Die beteiligten Klassen und deren Zusammenwirken zeigt das Klassendiagramm in Abbildung 18-8.

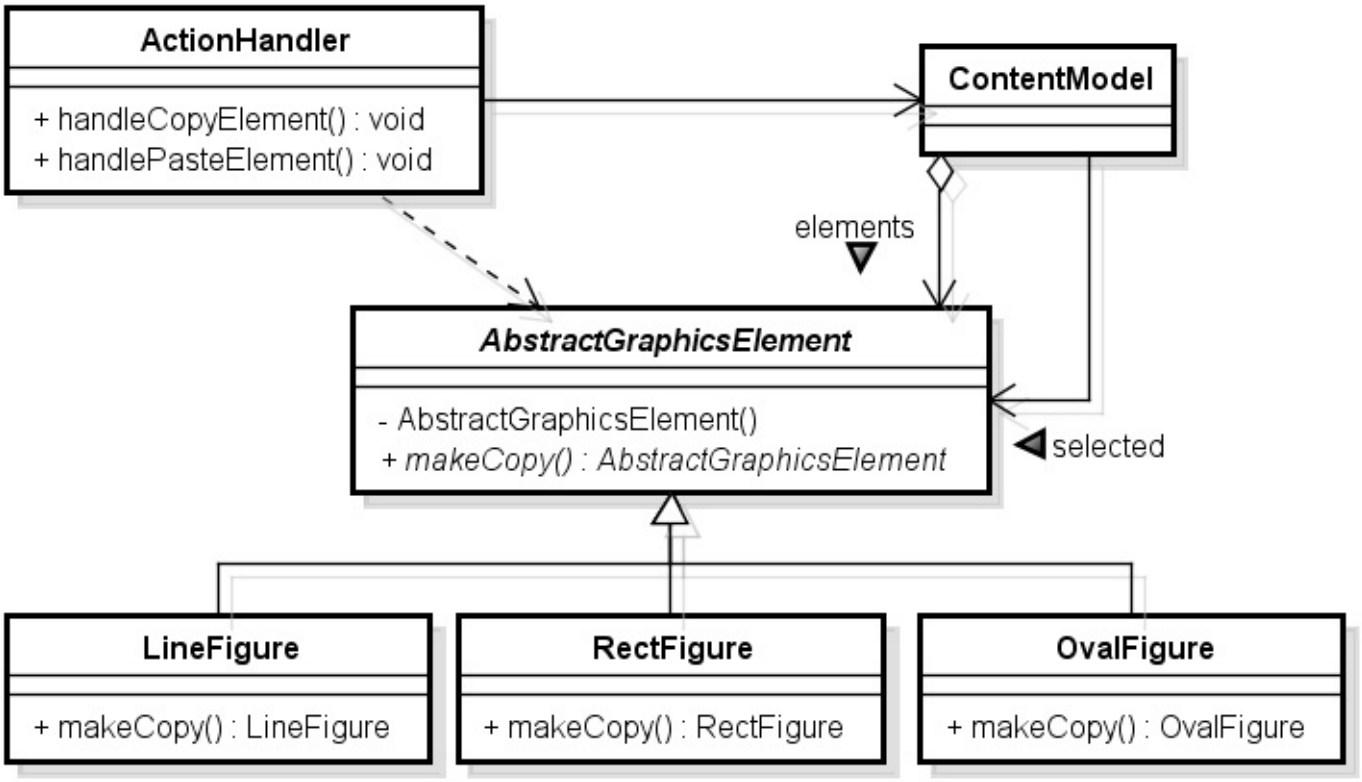


Abbildung 18-8 *Prototyp am Beispiel*

Zur Realisierung der Copy- bzw. Paste-Aktionen werden die Methoden `handleCopy-Element()` und `handlePasteElement()` aufgerufen, um das momentan auf der Zeichenfläche selektierte Element zu kopieren. Dieses kann je nach Auswahl von unterschiedlichen Laufzeittypen sein. Damit eine einheitliche Verarbeitung möglich ist, müssen alle zu verarbeitenden Objekte einen gemeinsamen Typ besitzen. In diesem Beispiel ist dies die abstrakte Klasse `AbstractGraphicsElement` mit den konkreten Subklassen `LineFigure`, `RectFigure` und `OvalFigure`. Eine Copy- bzw. Paste-Aktion nutzt lediglich die abstrakte Basisklasse und muss dadurch die Spezialisierungen nicht kennen. Auf diese Weise können sogar beliebige, abgeleitete Elemente kopiert werden, die zum Kompilierzeitpunkt noch unbekannt sind.

Schauen wir nun auf die Realisierung der Methode `handleCopyElement()`. Es wird hier die zuvor beschriebene Funktionalität umgesetzt, indem eine Kopie des momentan selektierten grafischen Elements erzeugt und im Attribut `clipboardElement` gespeichert wird:

```
public void handleCopyElement()
{
   final AbstractGraphicsElement selected = contentModel.getSelectedElement();

   if (selected != null)
   {
       insertPos = selected.getPosition();
       clipboardElement = selected.makeCopy();
   }
   else
   {
       throw new IllegalStateException("Copy must only be activated if an " +
                                       "element is selected!");
```

```
    }
}
```

Ein Detail dieser Methode ist noch erwähnenswert: Für den Fall, dass im Modell kein Element selektiert ist (`getSelectedElement()` demnach `null` liefert), wird eine `IllegalStateException` geworfen. Zunächst könnte eine solche Reaktion auf diese durchaus normale Situation verwundern. Selbstverständlich darf zu gewissen Zeitpunkten durchaus kein Element selektiert sein. Als Folge sollte eine Statusverwaltung der Aktionen die Copy-Aktion jedoch deaktivieren. Diese Aktion und damit auch der Button sollten nicht anwählbar sein, um einen ansonsten sinnlosen Aufruf dieser Funktionalität zu verhindern. Folgt man dieser Regel, so kann der genannte Fall (`selected == null`) nur dann eintreten, wenn die Statusverwaltung der Aktionen fehlerhaft arbeitet. Der hier gezeigte, offensive Umgang mit unerwarteten Situationen ist ratsam und hilfreich, weil dadurch Denk- oder Realisierungsfehler noch während der Entwicklungsphase gefunden werden können.

Wird eine Paste-Operation ausgeführt, so sollen die meisten Objekteigenschaften beibehalten werden, lediglich die Objektposition wird angepasst und das zwischengespeicherte Objekt als Kopie eingefügt:

```
public void handlePasteElement()
{
    if (clipboardElement != null)
    {
        // Kaskadierende Position sicherstellen
        increaseImageInsertPos();

        // Kopie anpassen …
        final Point insertPos = getImageInsertPos();
        clipboardElement.setPosition(insertPos.x, insertPos.y);

        // … und dem Modell hinzufügen
        contentModel.addElement(clipboardElement);

        // hier nochmal kopieren, damit wir mehrfach einfügen können
        clipboardElement = clipboardElement.makeCopy();
    }

    else
    {
        throw new IllegalStateException("Paste must only be activated if an "
+
                                        "element is selected!");
    }
}
```

Nachdem wir nun einige wichtige Bestandteile der Applikation kennengelernt haben, zeige ich abschließend die `main()`-Methode und das Programm PROTOTYPEEXAMPLE zum Start der Applikation:

```
public static void main(String[] args)
{
    final ContentModel contentModel   = new ContentModel();
    final ActionHandler actionHandler = new ActionHandler(contentModel);
```

```
    final AppFrame appFrame = new AppFrame( "PrototypeExample", contentModel,
          actionHandler);
    appFrame.setVisible( true);
}
}
```

Listing 18.2 *Ausführbar als* **'PROTOTYPEEXAMPLE'**

Bewertung

Der Einsatz des PROTOTYP-Musters besitzt folgende Implikationen:

+ **Vereinfachung** – Die Objektvervielfältigung und -konstruktion wird vereinfacht. Statt neue Objekte mit umfangreicher Parametrierung neu erzeugen zu müssen, wird hier auf eine Vorlage zurückgegriffen, die als Basis dient und lediglich einige Modifikationen zum Einsatz benötigt.

+ **Flexibilität** – Bei der dynamischen Variante ist das Kopieren von zur Kompilierzeit unbekannten Typen möglich. Dies erlaubt das Hinzufügen und Entfernen von kopierbaren Produkten zur Laufzeit.

+ **Nachvollziehbarkeit** – Der Ablauf ist im Gegensatz zum in die JVM integrierten `clone`-Mechanismus (vgl. Abschnitt 8.4) besser nachvollziehbar. In diesem Fall wird explizit eine Implementierung der `makeCopy()`-Methode gefordert. Beim Erfüllen des `Cloneable`-Interface gibt es nur implizite Annahmen über die Existenz von `clone()`-Methoden: `Cloneable` ist lediglich ein Marker-Interface (vgl. Abschnitt 3.4.3) und besitzt keine Methoden.

○ **Fallstrick Referenzsemantik** – Bei Containern ist schwer zu entscheiden, ob durch die Referenzsemantik Probleme entstehen können. Details dazu finden Sie in der Diskussion bezüglich flacher und tiefer Kopien, die bei der Vorstellung der Methode `clone()` in Abschnitt 8.4 geführt wurde.

Tipp: Abgrenzung zu `clone()`

Die Ziele des PROTOTYP-Musters und die Intentionen der Methode `clone()` sind ähnlich. Mit letzterer erzeugt man standardmäßig jedoch nur eine flache Kopie. Dies ist problematisch für alle in einem Objekt verwendeten Referenzvariablen, speziell für Collections. Analog ist eine flache Kopie für alle UI-Komponenten problematisch. In beiden Fällen findet unerwartet eine gemeinsame Nutzung referenzierter Objekte, Collections bzw. Listener usw., statt.

Die Erwartungshaltung beim Anlegen einer Kopie ist jedoch, ein frisches, vollständig eigenständiges Objekt mit einer gewissen Vorbelegung verschiedener Attribute zu erhalten. Durch den Einsatz des PROTOTYP-Musters lässt sich diese Anforderung, insbesondere die Tiefe der Kopieroperation, einfacher erfüllen, da die Objektkonstruktion in weiten Teilen beeinflusst werden kann.

18.2 Strukturmuster

Strukturmuster helfen, Funktionalitäten leichter konfigurierbar oder handhabbar zu machen sowie Verhalten flexibel erweitern und anpassen zu können. Eine FASSADE kapselt und versteckt die Komplexität eines Subsystems. Mit einem ADAPTER lassen sich inkompatible Softwarestücke verbinden. Ein DEKORIERER erlaubt es, Klassen neue Funktionalität hinzuzufügen. Mit dem KOMPOSITUM lassen sich Einzelelemente und Gruppen aus Baumstrukturen einheitlich behandeln.

18.2.1 Fassade (Façade)

Beschreibung und Motivation

Das Entwurfsmuster FASSADE kann dabei helfen, die Komplexität eines Subsystems zu verbergen und den Zugriff darauf für externe Klassen zu vereinfachen bzw. zu steuern. Eine Fassadenklasse definiert dazu ein »High-Level«-Interface, um komplizierte, sehr feingranulare Interaktionen und Beziehungen zwischen Package-internen und -externen Klassen zu vermeiden. Bei Aufrufen durch Klienten delegiert ein Fassadenobjekt dazu Aufrufe entsprechend der Zuständigkeit an spezielle Klassen des Subsystems. Es herrscht eine gerichtete Abhängigkeit: Die Fassadenklasse kennt die Klassen des Subsystems, aber nicht umgekehrt, d. h., keine Klasse des Subsystems kennt die Fassadenklasse.

Struktur

Betrachten wir zunächst ein System mit zwei Klienten `Client1` und `Client2`, die Datenbankzugriffe durchführen wollen. In Abbildung 18-9 erkennt man viele Abhängigkeiten von den eingesetzten Klassen. Dies liegt daran, dass die Logik und Steuerung jeweils in den Klienten erfolgt.

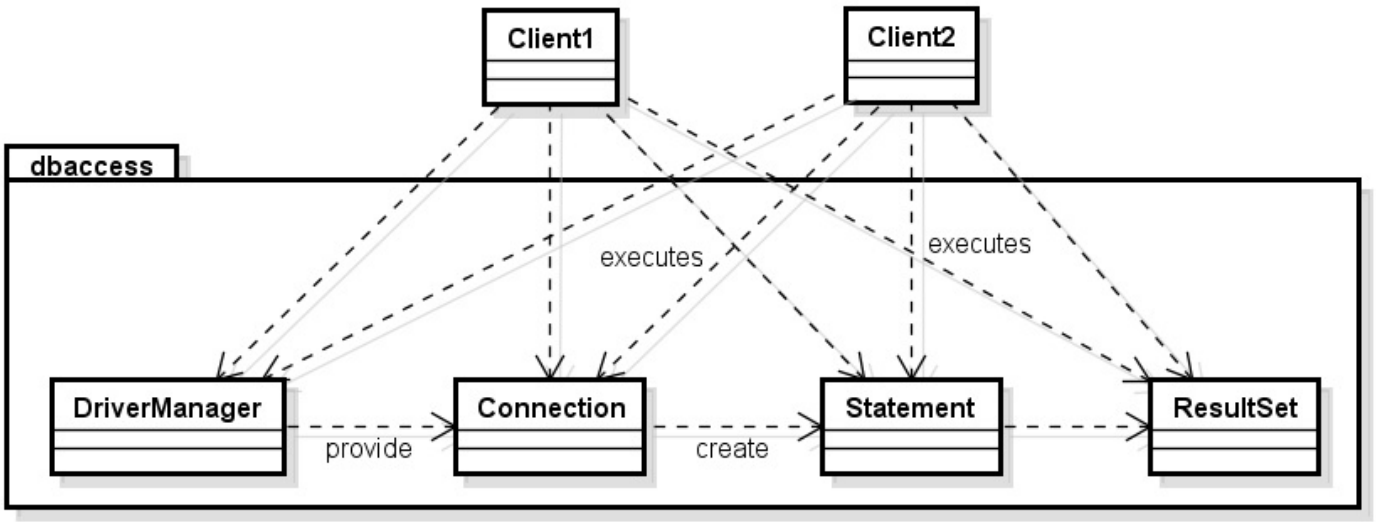


Abbildung 18-9 *Abhängigkeiten ohne Fassade*

Mit der Anzahl der beteiligten Klienten steigen diese Abhängigkeiten immer weiter. Dies ist der Punkt, an dem man über den Einsatz des FASSADE-Musters nachdenken sollte. Führt man im Beispiel eine Fassadenklasse `DBAccess` ein, so lassen sich dadurch die Zugriffe auf die Datenbankzugriffsklassen strukturieren. Abbildung 18-10 zeigt dies eindrucksvoll.

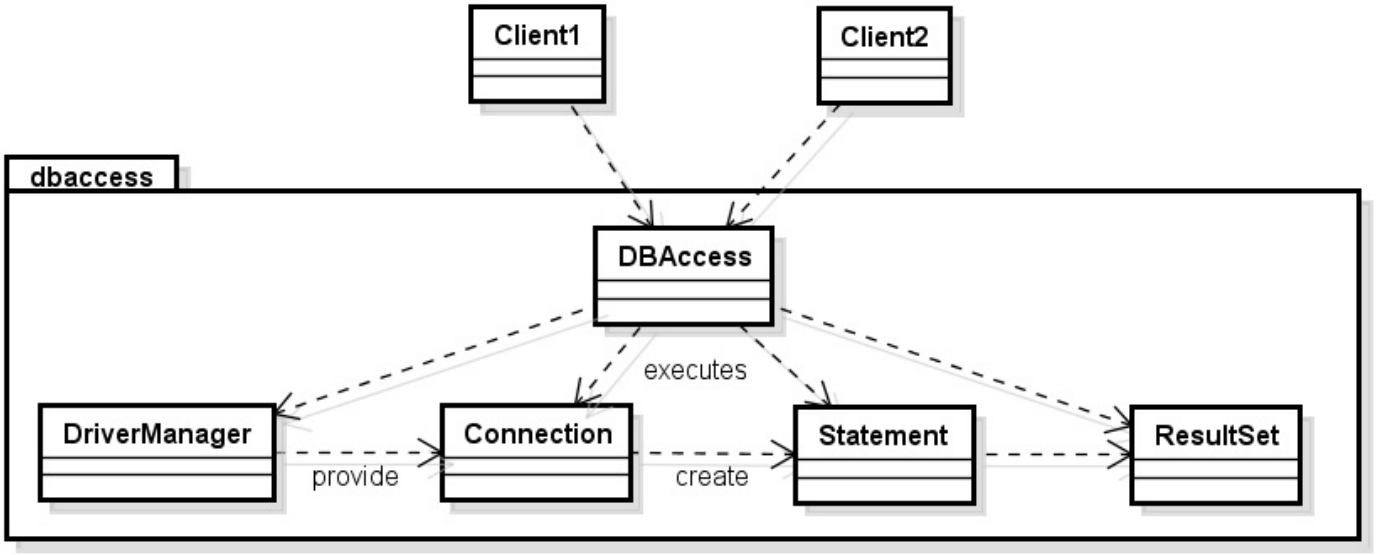


Abbildung 18-10 *Strukturierung durch Fassade*

Die Fassadenklasse verbirgt den Verbindungsaufbau und die Komplexität der Datenbankzugriffe vor aufrufenden Klienten. Diese nutzen nur noch eine Verwaltungsklasse und damit eine einzige Schnittstelle, die es ihnen ermöglicht, ihre Aufgaben auszuführen. Sämtliche Details der Infrastruktur sind für Aufrufer nicht mehr von Interesse. Die meisten Klassen des Subsystems können und sollten folglich Package-private definiert werden, um Implementierungsdetails nach außen auch tatsächlich zu verbergen.

Bewertung

Der Einsatz des FASSADE-Musters hat folgende Auswirkungen:

+ **Vereinfachung des API** – Der Zugriff auf die Funktionalität eines Subsystems fällt leichter, weil eine einfachere Schnittstelle zum Zugriff auf dessen Klassen angeboten wird.
+ **Trennung zwischen Klienten und Subsystem** – Die Zugriffe werden strukturiert und Implementierungsdetails lassen sich verstecken. Klienten müssen somit weniger Kenntnis über die Komponenten des Subsystems haben. Dadurch kommt es zu einer besseren Trennung und loseren Kopplung zwischen Klienten und dem Subsystem.
+ **Zentralisierung von Funktionalität und Updates** – Die Steuerung von sogenannten ***Cross-Cutting Concerns*** (vgl. folgenden Praxistipp »Cross-Cutting Concern und aspektorientierte Programmierung«) erfolgt einheitlich an einer Stelle. Außerdem kann man eine Fassade zu einer Art Transaktionsverwaltung bei

der Kommunikation von einem GUI mit einem komplexeren Modell nutzen. Im besten Fall greift das GUI dann nur über ein Fassadenobjekt auf ein Modell zu. Dies erlaubt es, die Propagation von Modelländerungen nicht feingranular, sondern gebündelt und gegebenenfalls gepuffert durchzuführen. Dadurch vermeidet man beispielsweise ständiges Neuzeichnen und »Geflacker«. Erst nachdem Änderungen vollständig im Modell verarbeitet wurden, wird das GUI durch die Fassadenklasse über Änderungen informiert und dann aktualisiert.

- **Gefahr eines breiten Interface** – Die Kapselung der Funktionalität des Subsystems führt schnell zu einem sehr breiten Interface mit vielen Methoden.
- **Keine Nutzungsgarantie** – Es ist nicht gewährleistet, dass Klienten die Fassadenklasse auch tatsächlich benutzen. Stattdessen können Klienten daran vorbei programmieren, wodurch die zuvor genannten positiven Effekte verloren gehen. Bei der GoF wird dieser Punkt als Feature angesehen, basierend auf folgender Begründung: Für den Regelfall liefert die Fassadenklasse eine gute Standardimplementierung, für Sonderfälle bleibt der Weg zur Spezialimplementierung offen. Dies ist stichhaltig. Allerdings entstehen in der Praxis aus Bequemlichkeit häufig immer wieder neue »Schleichwege« an der Fassadenklasse vorbei. Dadurch leidet die Wartbarkeit, da der Sourcecode unübersichtlich und schlechter nachvollziehbar wird. Um derartige »Fehlverwendungen« aus anderen Packages zu unterbinden, kann man Klassen des Subsystems Package-private definieren.
- **Mehr Indirektionen** – Es werden Weiterleitungen von Methodenaufrufen, sogenannte ***Indirektionen***, eingeführt.

Tipp: Cross-Cutting Concern und aspektorientierte Programmierung

Unter einem sogenannten Cross-Cutting Concern versteht man eine orthogonale, vom eigentlichen Programmcode unabhängige Funktionalität, die in mehreren Programmteilen immer wieder benötigt wird (etwa Logging oder Transaktionsverwaltung) und die durch normale Modularisierungstechniken nicht adäquat abgebildet werden kann. Daher sind diese Funktionalitäten immer wieder an benötigten Stellen aufzurufen oder auszuprogrammieren. Die sogenannte ***aspektorientierte Programmierung*** (***AOP***) adressiert diese Probleme und ermöglicht es, dass derartige Funktionalität nicht direkt in der Anwendung selbst realisiert werden muss. Allerdings unterstützt Java die aspektorientierte Programmierung nicht direkt. Mit AspectJ (`http://www.eclipse.org/aspectj/`) ist eine Erweiterung verfügbar, die den vom Java-Compiler erzeugten Bytecode modifiziert und an speziellen Stellen die durch Aspekte definierte Zusatzfunktionalität einbindet.

18.2.2 Adapter

Beschreibung und Motivation

Dieses Muster lässt sich sehr anschaulich am Beispiel von zwei Steckern oder Puzzleteilen motivieren, die nicht ineinander passen. Man nimmt ein Adapterstück und nutzt dieses zur Verbindung, um die nicht passenden Teile zu verbinden. Bei Software handelt man gleichermaßen: Man verwendet ein Softwarestück, das die Schnittstellen ansonsten inkompatibler Software aufeinander abbildet, um diese miteinander zu verbinden. Der entscheidende Vorteil gegenüber einem direkten Ansprechen einer anderen Klasse ist, dass durch den Einsatz eines Adapters keine Änderungen an den Schnittstellen der bereits vorhandenen Implementierungen nötig sind. Dafür muss allerdings ein neues Softwarestück, der Adapter, entwickelt werden.

Struktur

Ein Klient `Client` nutzt ein Objekt vom Typ `Adapter`, um die Funktionalität einer vorhandenen Klasse `OriginalClass` mit einer für den Klienten passenden Schnittstelle verwenden zu können. Diese wird durch den `Adapter` zur Verfügung gestellt (vgl. Abbildung 18-11). Die Referenz auf die Originalklasse wird häufig `adaptee` genannt.

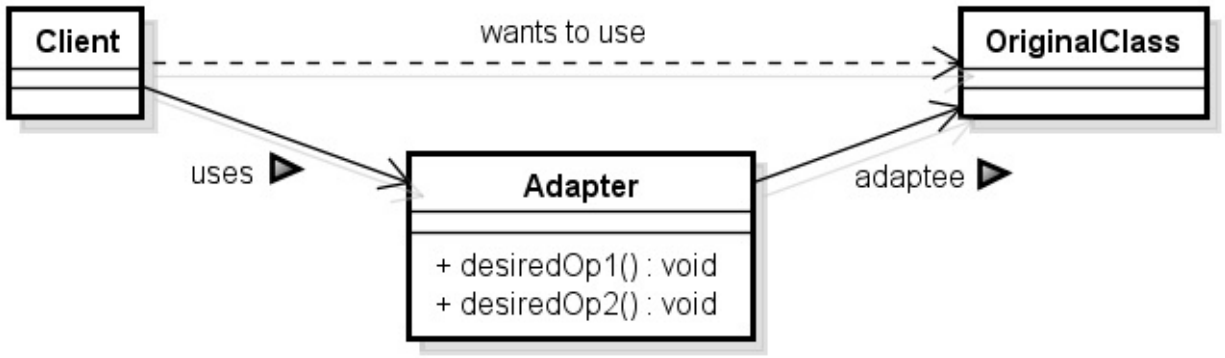


Abbildung 18-11 *Adapter in UML*

Beispiel

In diesem Beispiel ist eine Implementierung eines Adapters gezeigt, die eine beliebige Realisierung des Interface `List<E>` (etwa `ArrayList<E>`, `LinkedList<E>` etc.) mit einem Listenmodell für Swing-Komponenten kompatibel macht.

```
public static class ListToListModelAdapter extends AbstractListModel<Person>
{
    private final List<Person> personsAdaptee = new ArrayList<>();

    ListToListModelAdapter( final List<Person> persons)
    {
        this.personsAdaptee.addAll(persons);
    }

    public int getSize()
    {
        return personsAdaptee.size();
```

```
    }

    public Person getElementAt(final int i)
    {
        return personsAdaptee.get(i);
    }
}
```

Die `OriginalClass` aus dem UML-Strukturdiagramm ist die das `List<Person>`-Interface erfüllende Referenz `persons`. Die gewünschte Klasse bzw. das gewünschte Interface wird durch die abstrakte Klasse `AbstractListModel<E>` und das Interface `ListModel<E>` repräsentiert. Die verbindende Adapterklasse ist die Klasse `ListToListModelAdapter`. Diese nutzt die abstrakte Klasse `AbstractListModel<E>`, um viele Standardaufgaben aus dem Interface `ListModel<E>` zu erfüllen: Dadurch müssen nur an zwei Stellen Abbildungen von Funktionalitäten von `List<E>` auf `ListModel<E>` vorgenommen werden. Die zur Anpassung notwendigen Methoden `getSize()` und `getElementAt(int)` werden auf die Methoden `size()` bzw. `get(int)` des Interface `List<E>` abgebildet. Dazu wird ein Zugriff auf die `OriginalClass` benötigt. In der Regel wird dies durch Speicherung als Attribut in der Adapterklasse gelöst. In diesem Fall ist dies die Referenz `personsAdaptee`, die eine flache Kopie der Eingabeliste erstellt.

Bewertung

Der Einsatz des ADAPTER-Musters hat folgende Auswirkungen:

+ **Sicherstellung von Kompatibilität** – Zwei unabhängige Klassen (oder Systeme) werden durch den Adapter kompatibel zueinander gemacht, wodurch es möglich wird, die Funktionalität einer anderen Klasse zu nutzen, ohne deren Implementierung ändern zu müssen. Zwar könnte man dies auch durch Änderungen in den bestehenden Klassen erreichen, dadurch könnten aber Inkompatibilitäten entstehen, und die Wahrscheinlichkeit für Fehler steigt.
+ **Keine Änderungen am bestehenden System** – Es ist nur die Realisierung des Adapters erforderlich. Dadurch wird das Risiko für Fehler minimiert, da die bisherige (getestete) Funktionalität unverändert bleibt.
○ **Aufwand durch Delegation** – Es wird Delegation genutzt und es ist zum Teil einiges an Codierungsaufwand notwendig.

18.2.3 Dekorierer (Decorator)

Beschreibung und Motivation

Mithilfe des DEKORIERER-Musters kann man vorhandenes Objektverhalten um zusätzliches Verhalten erweitern, ohne dass dafür der Sourcecode der ursprünglichen

Klasse modifiziert werden muss. Dies ist insbesondere für den Fall sehr praktisch, dass entweder eine zu erweiternde Klasse nicht als Sourcecode vorliegt oder dieser nicht verändert werden darf. Ein erster Gedanke ist häufig, eine Subklasse zu bilden und dort die gewünschten Erweiterungen vorzunehmen. Teilweise löst diese Art der Realisierung, wie bereits in Abschnitt 3.3.2 diskutiert, eine kombinatorische Explosion von Klassen aus, wenn verschiedenste Funktionalitäten miteinander verknüpft werden sollen. Das DEKORIERER-Muster bietet hierzu eine Alternative: Erweiterungen in der Funktionalität werden wie ein Mantel um die vorhandene Funktionalität gelegt. Jede Dekoriererklasse realisiert nur einen Teil der Gesamtfunktionalität. Daher wird das Muster manchmal auch ***Wrapper*** genannt. Eine Realisierung gemäß diesem Muster ist einer puren statischen Vererbung überlegen, da eine Ummantelung sogar dynamisch zur Laufzeit erfolgen kann.

Struktur

Grundlage dieses Musters ist ein gemeinsamer Basistyp (etwa ein Interface oder eine abstrakte Klasse). Dieser Typ definiert die öffentliche Schnittstelle für Dekoriererklassen und für zu dekorierende Klassen. In diesem Beispiel geschieht dies durch das Interface `CommonIF`. Dieses definiert eine spezielle Funktionalität, die durch die zwei konkreten Realisierungen `ConcreteImpl1` bzw. `ConcreteImpl2` erfüllt wird. Soll nun weitere Funktionalität zur Verfügung gestellt werden, so kann dies mithilfe der beiden Dekoriererklassen `ConcreteDecorator1` bzw. `ConcreteDecorator2` geschehen. Zur Realisierung der Ummantelung hält jedes Dekoriererobjekt eine Referenz auf das zu dekorierende Objekt in Form des Interface `CommonIF`. Werden verschiedene Dekoriererklassen definiert, um unterschiedliche Erweiterungen unabhängig einsetzen zu können, bietet sich die Einführung einer abstrakten Basisklasse `AbstractDecorator` an. Diese hält eine Referenz auf das Interface `CommonIF` und kann damit entweder konkrete Ausprägungen oder wiederum Dekoriererklassen referenzieren. Abbildung 18-12 zeigt das entsprechende Klassendiagramm.

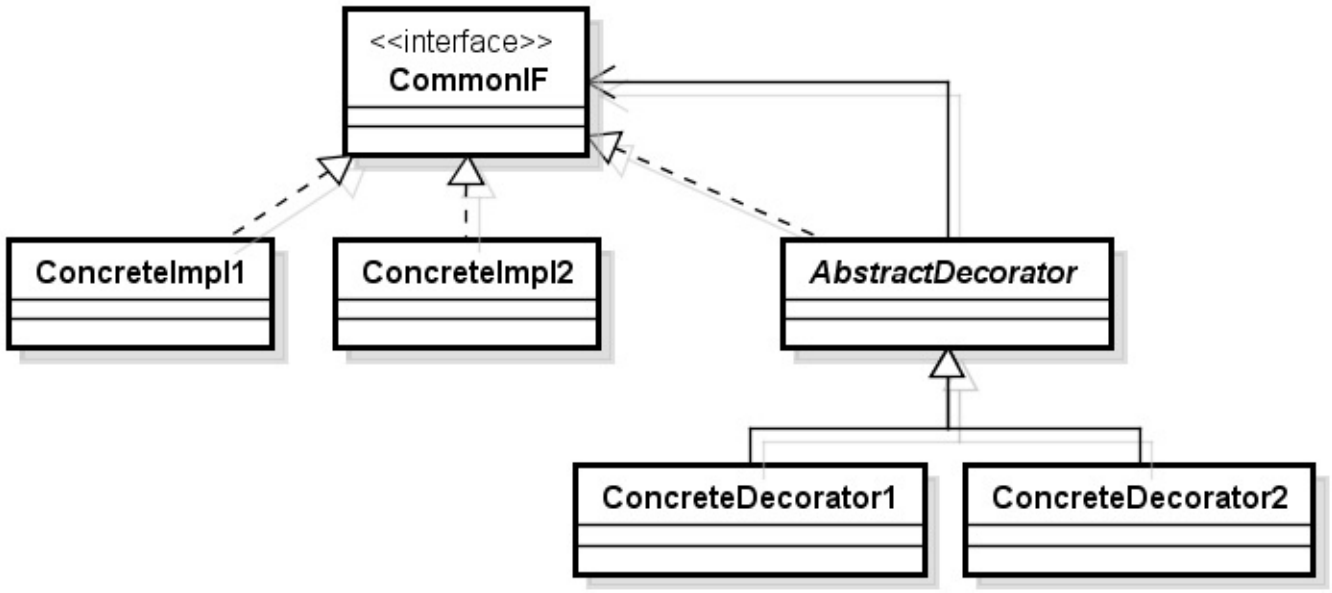


Abbildung 18-12 *Klassendiagramm mit mehreren Dekoriererklassen*

Bei Aufruf einer Methode des Dekoriererobjekts nutzt dieses eine gleichlautende Methode des referenzierten, zu dekorierenden Objekts und delegiert somit diesen Auftrag, um vorher oder nachfolgend gewünschte Erweiterungen durchzuführen. Problemlos können auch mehrere Dekoriererobjekte hintereinander geschaltet werden, d. h., es findet dann eine mehrfache Ummantelung statt.

Existiert lediglich eine Dekoriererklasse bzw. ist nur eine geplant, kann man auf die Basisklasse verzichten. Eine Mehrfachummantelung ist in einem solchen Fall lediglich mit gleicher Funktionalität möglich. Dies ist meistens nicht sinnvoll.

Beispiel

Ein Beispiel für dieses Muster ist die Umkehrung einer Sortierreihenfolge, bestimmt durch spezielle Realisierung des Interface `Comparator<T>`. Wir definieren eine Dekoriererklasse `ReverseComparator<T>`, die das Interface `Comparator<T>` implementiert und in deren `compare(T, T)`-Methode die Eingabeparameter tauscht:

```
public final class ReverseComparator<T> implements Comparator<T>
{
    private final Comparator<T> originalComparator;

    public ReverseComparator(final Comparator<T> originalComparator)
    {
        this.originalComparator = Objects.requireNonNull(originalComparator,
                                    "originalComparator must not be null!");
    }

    @Override
    public int compare(final T o1, final T o2)
    {
        return originalComparator.compare(o2, o1);
    }
}
```

In diesem Fall besteht die Bearbeitung durch das Dekoriererobjekt demnach lediglich im Tausch der Eingabeparameter und dem Aufruf von `compare(o2, o1)`. Das zugehörende Klassendiagramm ist in Abbildung 18-13 dargestellt.

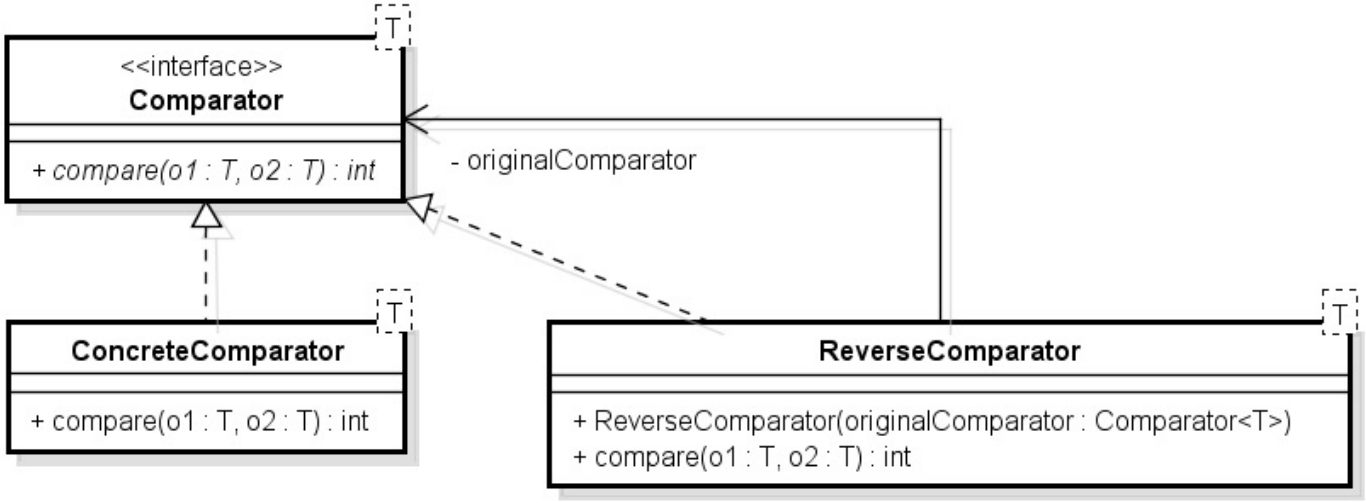


Abbildung 18-13 *Klassendiagramm des ReverseComparator*

Bewertung

Der Einsatz des DEKORIERER-Musters hat folgende Auswirkungen:

+ **Transparent zusätzliche Funktionalität** – Zusätzliche Funktionalität kann transparent für einen Aufrufer hinzugefügt werden. Dies ist sogar zur Laufzeit möglich.
+ **Hintereinanderschaltung** – Mehrere Dekoriererobjekte können hintereinander geschaltet werden, um eine komplexere Funktionalität zu realisieren.
+ **Flexibilität** – Die zu dekorierende Klasse ist nicht festgelegt, da lediglich gegen eine gemeinsame Schnittstelle programmiert wird. Dekoriererklassen können somit für verschiedene zu dekorierende Klassen genutzt werden und ermöglichen damit Wiederverwendung. Statische Vererbung erlaubt das nicht.
+ **Vereinfachung von Vererbungshierarchien** – Komplexe und unübersichtliche Vererbungshierarchien können durch Einsatz dieses Musters vermieden werden.
○ **Gemeinsamer Basistyp benötigt** – Damit eine Ummantelung von Klassen möglich ist, müssen alle im DEKORIERER-Muster beteiligten Klassen einen gemeinsamen Basistyp besitzen (Interface oder abstrakte Klasse), der die öffentliche Schnittstelle für Dekoriererklassen und für zu dekorierende Objekte definiert.
○ **Fehlende Kontrolle** – Eine Kontrolle, wer welche Funktionalität wie und wann hinzufügt und ob dies sinnvoll ist, bleibt der Disziplin des Entwicklers überlassen. Eine mehrfache Hintereinanderschaltung von Objekten derselben Dekoriererklasse, beispielsweise `BufferedInputStream` oder `ReverseComparator<T>`, ist somit möglich, aber meistens nicht sinnvoll.
- **Zugriff auf Spezialisierungen schwieriger möglich** – Die Funktionalität wird durch Dekoriererobjekte transparent hinzugefügt, wodurch ein Aufrufer nicht direkt darauf zugreifen kann, da dieser nur eine Referenz auf die allgemeine

Dekoriererklasse hält. Als Lösung kann man eine Referenz auf eine konkrete Dekoriererklasse speichern, um deren Zusatzfunktionalität explizit nutzen zu können.

- **Implementierungsaufwand** – Enthält das zu ummantelnde Interface relativ viele Methoden, so müssen all diese implementiert werden.

18.2.4 Kompositum (Composite)

Beschreibung und Motivation

Das KOMPOSITUM-Muster ermöglicht es, in hierarchischen Datenstrukturen sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln. Damit kann man in den meisten Fällen den Unterschied zwischen Einzelobjekten und Kompositionen außer Acht lassen.

Struktur

Um sowohl Einzelobjekte als auch Kompositionen einheitlich behandeln zu können, definiert eine abstrakte Basisklasse (oder alternativ auch ein Interface) `Component` die gemeinsame Schnittstelle für alle Akteure. Diese Schnittstelle wird jeweils sowohl von Einzelobjekten `Leaf` als auch vom Kompositum `Composite` implementiert. Das Kompositum wird in der Regel noch weitere Methoden besitzen, um beispielsweise die hierarchische Struktur aufzubauen. Abbildung 18-14 stellt das Klassendiagramm für das KOMPOSITUM-Muster dar.

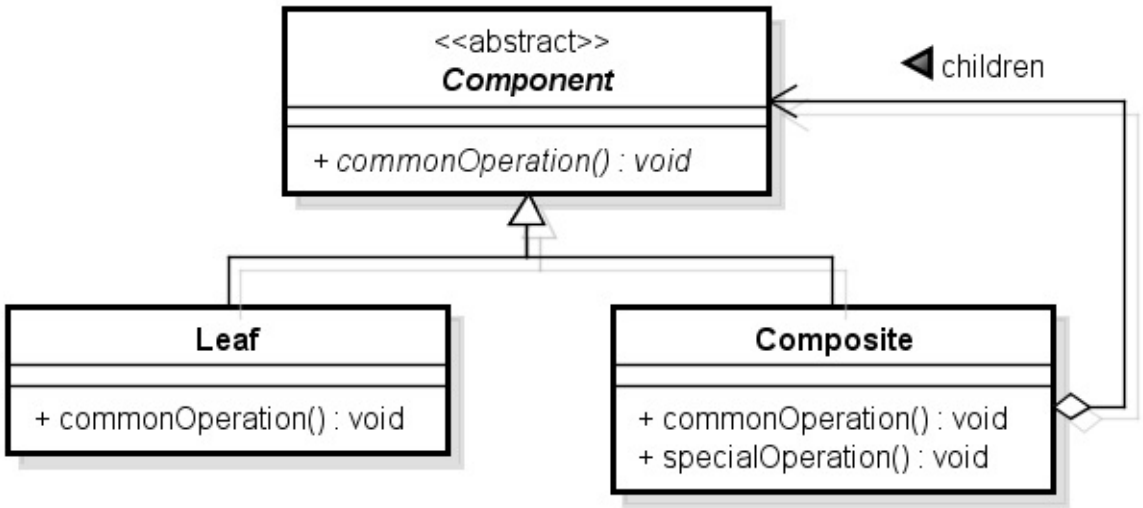


Abbildung 18-14 *Grundstruktur des* KOMPOSITUM*-Musters*

Besonderheiten

Bei der GoF sind die speziellen Operationen, unter anderem diejenigen, die zum Aufbau der Hierarchie notwendig sind, Bestandteil der gemeinsamen Schnittstelle.

Durch die Definition innerhalb der Basisklasse müssen diese Methoden in der Klasse der Einzelobjekte implementiert werden. Dies kann entweder in Form eines leeren Methodenrumpfs geschehen oder aber, indem dort eine `UnsupportedOperationException` ausgelöst wird. Beide Varianten können problematisch sein. ***Meiner Ansicht nach sollten daher Spezialoperationen nicht Bestandteil der gemeinsamen Schnittstelle sein.***

Für viele Anwendungsfälle, etwa die später gezeigten Berechnungen, wird keine Unterscheidung benötigt, sondern sie ist sogar unerwünscht. Die Aufnahme der Spezialmethoden des Kompositums in die Klasse der Einzelobjekte verletzt zwar dieses Transparenzkriterium nicht, ist aber schlechtes Design, da sich diese Methoden in den Einzelobjekten nicht sinnvoll implementieren lassen.

In grafischen Oberflächen wird für einige Aktionen die Unterscheidung zwischen Einzelobjekt und Kompositum benötigt, etwa um in Kontextmenüs Aktionen auf Gruppen zu aktivieren oder Aktionen für Einzelelemente zu deaktivieren. Für diesen Anwendungsfall bietet es sich an, die gemeinsame Schnittstelle um eine Methode `isComposite()` zu erweitern. Die Implementierung für Einzelobjekte gibt immer den Wert `false` zurück, die für Komposita den Wert `true`. Nur im letzteren Fall kann man dann einen expliziten Cast auf die Kompositum-Klasse anwenden, um dafür spezifische Operationen auszuführen.

Beispiele

Die Klasse `DefaultMutableTreeNode` aus dem Package `javax.swing.tree` realisiert dieses Muster in kompakter Form: Diese Klasse kann sowohl ein Einzelobjekt als auch ein Kompositum selbst modellieren. Die Unterscheidung zwischen beiden ist durch die Methoden `hasChildren()` und `isLeaf()` möglich.

Im Folgenden stelle ich ein vereinfachtes Beispiel aus der Praxis vor, das mithilfe des KOMPOSITUM-Musters die Projektkosten hierarchisch organisierter Projekte berechnen soll. Ziel ist es, die Kosten unabhängig von der Hierarchieebene mit dem gleichen Aufruf ermitteln zu können. Eine Projekthierarchie besteht aus Einzelprojekten (Klasse `Project`) oder aus Projektgruppen (Klasse `ProjectGroup`), die mehrere Einzelprojekte oder untergeordnete Projektgruppen verwalten können. Diese Struktur wird mithilfe des KOMPOSITUM-Musters wie folgt modelliert (Abbildung 18-15):

- Ein Einzelprojekt entspricht einem Blatt (`Project`).
- Eine Projektgruppe entspricht dem Kompositum (`ProjectGroup`). Unterprojekte werden mit der Methode `add(ProjectComponent)` hinzugefügt.
- Die abstrakte Basisklasse `ProjectComponent` definiert die gemeinsame Schnittstelle. Dazu zählt inbesondere die abstrakte Methode `calcCosts()`.

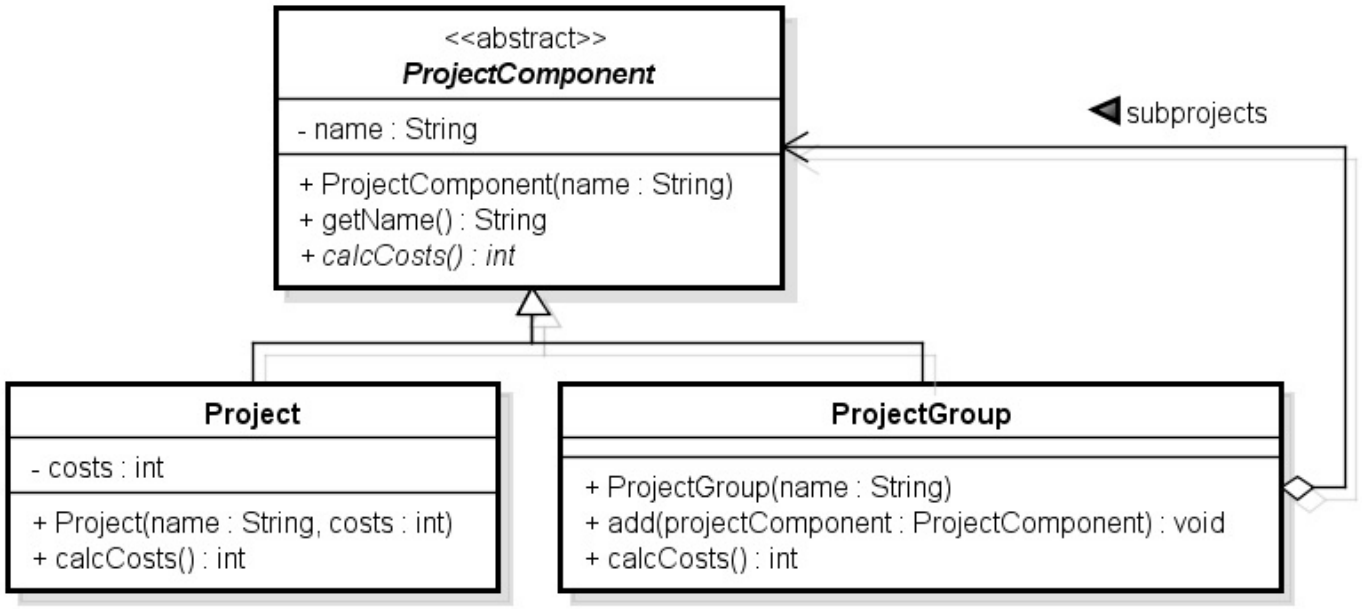


Abbildung 18-15 *Kompositum am Beispiel einer Projektstruktur*

Einzelprojekten sind bestimmte Kosten zugeordnet (als Vereinfachung im Beispiel per Konstruktorparameter übergeben). Projektgruppen ermitteln ihre Kosten durch Addition der Kosten ihrer Unterprojekte (Einzelprojekte oder Unterprojektgruppen), die sie transparent über die Schnittstelle der Basisklasse `ProjectComponent` verwalten. Mit dieser Modellierung können die Kosten auf jeder Hierarchieebene berechnet werden. Die Ermittlung dieser Kosten lässt sich somit einfach rekursiv wie folgt formulieren:

```
@Override
public int calcCosts()
{
    int costs = 0;
    for (final ProjectComponent current : subprojects)
    {
        costs += current.calcCosts();
    }

    return costs;
}
```

Im folgenden Sourcecode wird ein Hauptprojekt `mainProject` betrachtet, das zwei Projektgruppen `projectGroup` und `projectGroup2` als Unterprojekte enthält. Diese umfassen wiederum Einzelprojekte mit den Kosten 7, 4 und 5 Tage.

```
public static void main(final String[] args)
{
    final int SEVEN_DAYS = 7;
    final int FOUR_DAYS = 4;
    final int FIVE_DAYS = 5;
    final ProjectComponent project1 = new Project("Seven", SEVEN_DAYS);
    final ProjectComponent project2 = new Project("Four", FOUR_DAYS);
    final ProjectComponent project3 = new Project("Five", FIVE_DAYS);

    final ProjectGroup projectGroup = new ProjectGroup("Group 1: 7+4");
    projectGroup.add(project1);
    projectGroup.add(project2);
    final ProjectGroup projectGroup2 = new ProjectGroup("Group 2: 5");
```

```
    projectGroup2.add(project3);

    final ProjectGroup mainProject = new ProjectGroup("Main");
    mainProject.add(projectGroup);
    mainProject.add(projectGroup2);

    final ProjectComponent[] components = { project1, project2, project3,
                                projectGroup, projectGroup2, mainProject };
    for (final ProjectComponent current : components)
        printCosts(current);
}

private static void printCosts(final ProjectComponent projectComponent)
{
    System.out.println("Cost of '" + projectComponent.getName() + "': " +
                       projectComponent.calcCosts());
}
```

Listing 18.3 *Ausführbar als* **'KOMPOSITUMEXAMPLE'**

Die verschiedenen Aufrufe der Ausgaberoutine `printCosts(ProjectComponent)` zeigen sehr schön, wie durch den Einsatz des KOMPOSITUM-Musters eine Abstraktion von der Datenstruktur erfolgt und Einzelelemente und Gruppen auf allen Hierarchieebenen gleich behandelt werden können. Das Programm KOMPOSITUMEXAMPLE errechnet folgende Projektkosten:

```
Cost of 'Seven': 7
Cost of 'Four': 4
Cost of 'Five': 5
Cost of 'Group 1: 7+4': 11
Cost of 'Group 2: 5': 5
Cost of 'Main': 16
```

Bewertung

Der Einsatz des KOMPOSITUM-Musters hat folgende Auswirkungen:

+ **Vereinfachung** – Der Zugriff auf die Funktionalität einer komplexeren Datenstruktur wird vereinfacht, weil ein Klient nicht wissen muss, welche Art von Element er anspricht.
+ **Strukturierung** – Zugriffe werden strukturiert und Implementierungsdetails lassen sich verstecken.
○ **Erschwerter Zugriff auf Spezialisierungen** – Durch die Gleichbehandlung über eine gemeinsame Schnittstelle wird ein Zugriff auf Erweiterungen und Eigenschaften von Spezialisierungen von Blättern oder Kompositum verhindert. Beide können weitere Operationen anbieten, die sich jedoch nur mit Aufwand ansprechen lassen.

Achtung: Bäume, Graphen, Zyklen und das KOMPOSITUM-Muster

Sogenannte ***Bäume*** sind dadurch gekennzeichnet, dass sie streng hierarchisch strukturiert sind und dadurch wie bei realen Bäumen auch kein Ast wieder zurück in den Stamm wächst. Ein Ende eines Asts wird als ***Blatt*** bezeichnet, das ein Spezialfall

eines Verästelungspunkts ist, den man ***Knoten*** nennt. Die verbindenden Äste heißen ***Kanten***.

Die durch ein Kompositum beschriebene Datenstruktur stellt jedoch nicht zwangsläufig lediglich einen Baum dar, sondern kann auch einen sogenannten ***Graphen*** modellieren. Graphen können sogenannte ***Zyklen*** enthalten, d. h., es kann eine Verbindung eines Unterelements auf ein beliebiges anderes Element existieren. Ist dies im Speziellen ein Oberelement, so kommt es damit zu einer Rückkopplung. Ein Baum ist also ein Spezialfall eines Graphen, nämlich einer ohne Zyklen.

Enthielte das obige Beispiel einen Zyklus, könnte die intuitive rekursive Berechnung der Methode `calcCosts()` in einer endlosen Ausführung münden.[a] Da zyklische Graphen einen seltenen Spezialfall für das KOMPOSITUM-Muster darstellen, wird hier keine Lösung gezeigt. Im Buch »Design Patterns Java Workbook« [65] von Steven J. Metsker findet man eine Behandlung dieses Themas.

[a] Tatsächlich kommt es durch die Begrenzungen der Aufrufhierarchie von Methoden zu einem `java.lang.StackOverflowError`.

18.3 Verhaltensmuster

Verhaltensmuster strukturieren oder vereinfachen komplexe Abläufe. Ein ITERATOR bietet eine einheitliche Möglichkeit, verschiedene Datenstrukturen zu durchlaufen und dabei die Details zu verstecken. Ein NULL-OBJEKT repräsentiert einen `null`-Wert als Objekt und hilft, Zustandsabfragen zu vermeiden. Eine SCHABLONENMETHODE erlaubt es, gewisse Teilschritte eines Algorithmus vorzugeben und an anderen Stellen Varianten zuzulassen. Die STRATEGIE hilft dabei, Varianten von Algorithmen zu definieren. Das COMMAND-Muster fasst Methodenaufrufe zu Befehlsaktionen zusammen. Ein PROXY agiert als Stellvertreter für ein anderes Objekt und kontrolliert den Zugriff darauf. Ein BEOBACHTER ermöglicht es, Zustandsänderungen anderer Objekte zu beobachten und darauf zu reagieren. Vor allem in grafischen Applikationen nutzt man dazu die sogenannte MODEL-VIEW-CONTROLLER-Architektur (MVC) zur Trennung von Zuständigkeiten und zur Darstellung verschiedener Sichten auf gleiche Daten.

18.3.1 Iterator

Beschreibung und Motivation

Das Entwurfsmuster ITERATOR bietet eine allgemeingültige Möglichkeit für Klienten, alle Elemente einer Datenstruktur zu besuchen, ohne dafür die internen Details für den Durchlauf kennen zu müssen. Es findet eine Abstraktion von der zu traversierenden Datenstruktur statt, wodurch keine Kenntnis über deren Aufbau nötig ist: Bäume lassen sich auf ähnliche Weise durchlaufen wie Listen oder Mengen.

Dies ist ein gutes Beispiel für die Trennung von Zuständigkeiten. Die Datenstruktur verwaltet lediglich die Daten, jedoch nicht die Art, wie diese durchlaufen werden.

Struktur

Eine Iteratorklasse bietet mindestens die Möglichkeit, zu testen, ob ein nächstes Element existiert, und erlaubt es, ein solches gegebenenfalls zu ermitteln. In der Regel werden die beiden dazu notwendigen Methoden `hasNext()` und `next()` bzw. `hasMoreElements()` und `nextElement()` [6] oder ähnlich genannt. In der in Abbildung 18-16 dargestellten UML-Struktur erlaubt der dort gezeigte `ContainerIterator` ein Durchlaufen eines `Container`-Objekts unter Verwendung dieser Methoden.

Die dort gezeigte Lösung verwendet eine konkrete Realisierung eines Iterators für eine konkrete Containerklasse. Normalerweise existieren analog zur obigen Klasse `Container` weitere Datenstrukturen. Dann wäre es für deren Nutzer recht unpraktisch, dafür jeweils spezielle Iteratorklassen mit möglicherweise geringfügig unterschiedlicher Methodensignaturen kennen zu müssen. Zudem würde es die Austauschbarkeit der verwendeten Containerklassen erschweren, da der Sourcecode von Klienten auf deren spezielle Iteratortypen verweisen müsste. Für diese Fälle wäre es praktisch, wenn man einen Basisiteratortyp für alle nutzen könnte. Dies nennt man auch einen polymorphen Iterator, den wir uns nun kurz anschauen.

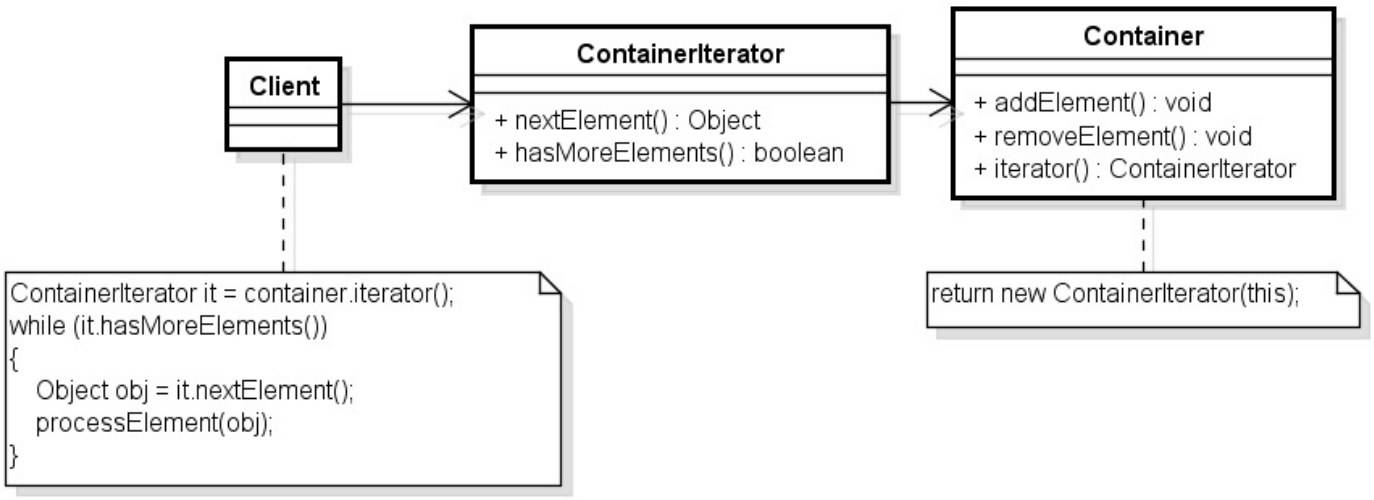


Abbildung 18-16 *Die Klasse* `Container` *mit zugehörigem* `ContainerIterator`

Polymorphe Iteratoren

Nehmen wir dazu vereinfachend an, es gäbe eine weitere selbst definierte Containerklasse namens `Set`. Beide Containerdatenstrukturen sollen selbstverständlich mithilfe von Iteratoren durchlaufen werden können. Im folgenden Beispiel implementieren die beiden Realisierungen `SetIterator` und `ContainerIterator` diese Funktionalität und verwalten die aktuelle Position während der Traversierung der Datenstrukturen. Gemäß der in Abschnitt 3.2.1 besprochenen Technik des gemeinsamen Interface nutzen wir als polymorphen

Iterator das Interface `CommonIterator`.

Dessen konkrete Implementierungen werden jeweils durch die Containerklassen erzeugt, indem sie ein Objekt der passenden Iteratorklasse zurückgeben. Klienten arbeiten jedoch nur gegen die allgemeine Schnittstelle `CommonIterator`. Abbildung 18-17 zeigt das entsprechende UML-Diagramm.

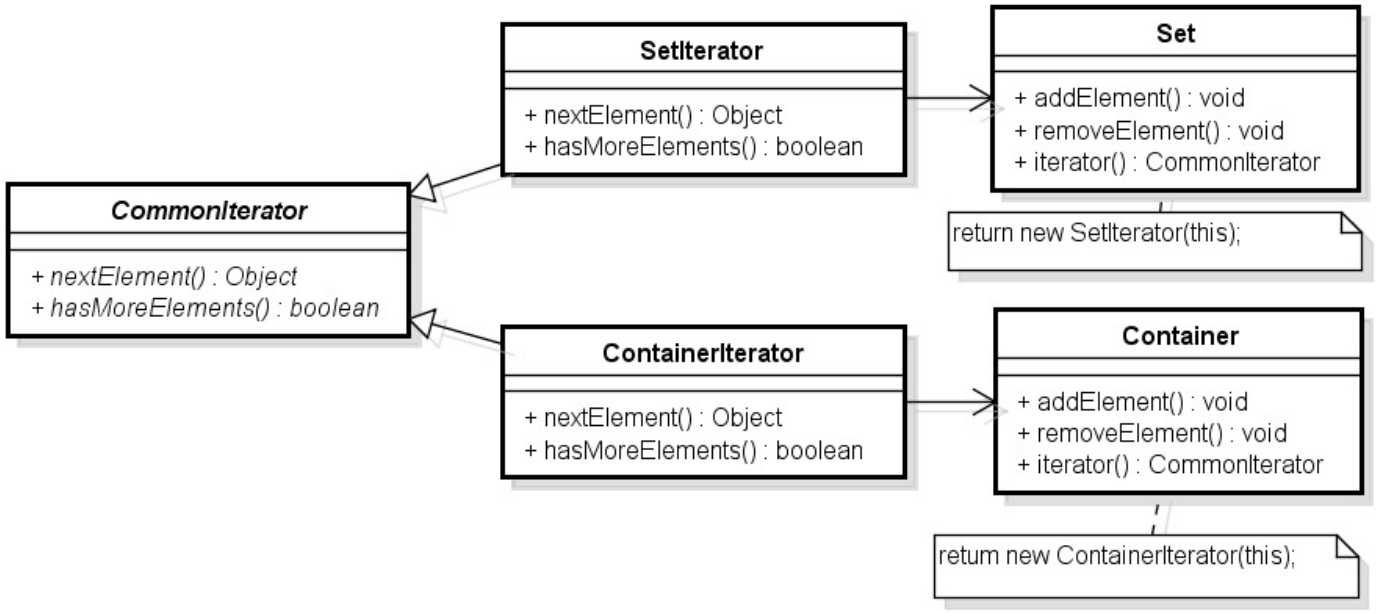


Abbildung 18-17 *Allgemeiner Iterator*

Beispiel

Für Java-Programmierer ist dieses Muster aus dem Collections-Framework bereits bekannt. Alle dort definierten Container lassen sich über einen `java.util.Iterator<E>` traversieren. Dabei handelt es sich um einen polymorphen Iterator.

Bewertung

Der Einsatz des ITERATOR-Musters besitzt folgende Auswirkungen:

+ **Abstraktion und Kapselung** – Die zu durchlaufende Datenstruktur wird gekapselt und kann bei Bedarf im Nachhinein verändert werden, ohne dass dies Änderungen aufseiten von Klienten verursacht.
+ **Vereinfachung** – Die Realisierung eigener Iteratorklassen für komplexere Datenstrukturen kann dabei helfen, eine ansonsten schwierige Implementierung des Iterationsprozesses zu kapseln, aus der Programmlogik herauszulösen und diese zu vereinfachen.
+ **Bessere Lesbarkeit** – Die Lesbarkeit ist viel besser als bei einer `for`-Schleife mit indiziertem Zugriff. Es wird das Durchlaufen auf einer konzeptionellen Ebene statt auf einer syntaktischen Ebene abgebildet.

- **Existenz einer alternativen Schreibweise in der Sprache** – Seit Java 5 existiert mit der for-each-Schleife eine Möglichkeit, die Lesbarkeit von Iterationen zu verbessern. Es können beliebige Datenstrukturen durchlaufen werden, die das Interface `Iterable<T>` implementieren. Dies wird implizit mit einem `Iterator<E>` implementiert – versteckt aber dieses Detail.

18.3.2 Null-Objekt (Null Object)

Beschreibung und Motivation

Die Idee beim Einsatz des NULL-OBJEKT-Musters ist, einen Platzhalter für `null`-Referenzen zu verwenden und so ein nicht vorhandenes Objekt zu modellieren. Es wird ein spezielles Objekt bereitgestellt, das kein eigenes Verhalten im fachlichen Sinn definiert. Es kann überall dort eingesetzt werden, wo ansonsten `null` als semantische Aussage für »kein Objekt« dient. Damit vereinfacht sich die Behandlung für Klienten: Sie können auf `null`-Prüfungen und weitere Spezialbehandlungen im Applikationscode verzichten.

Struktur

Um als »funktionsloser« Ersatz dienen zu können, muss ein Null-Objekt die Schnittstelle der zu vertretenden Klasse erfüllen. Die Implementierung erfolgt entweder durch Vererbung oder durch Realisierung eines Interface. Abbildung 18-18 zeigt diese beiden Varianten.

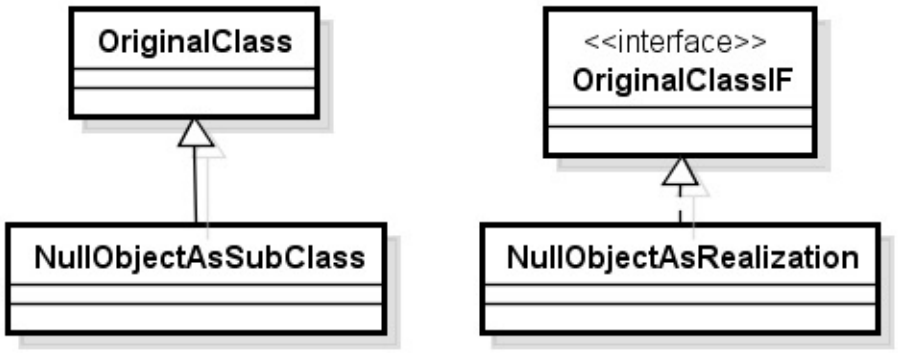


Abbildung 18-18 *Implementierungsvarianten des* NULL-OBJEKT-*Musters*

Implementierungsvarianten

Nutzt man Vererbung, so ist die Null-Objekt-Klasse `NullObjectAsSubClass` eine Spezialisierung der Klasse `OriginalClass`, soll aber keine Anwendungsfunktionalität zur Verfügung stellen. Demnach muss hier künstlich Funktionalität der Basisklasse wieder entfernt werden, indem Methoden überschrieben werden und eine (nahezu) leere Implementierung besitzen. Da die Null-Objekt-Klasse jedoch standardmäßig die ganze in der Basisklasse

implementierte Funktionalität erbt, sind (versehentliche) Aufrufe an die Basisklasse möglich und nicht auszuschließen.

Implementiert die Klasse `OriginalClass` ein Interface, so sollte dieses bevorzugt zur Realisierung des Null-Objekts genutzt werden. Es ist sinnvoll, ein solches Interface direkt zu implementieren, d. h. ohne von der Basisklasse abzuleiten. Dadurch sind Implementierungsdetails der Basisklasse nicht mehr im Zugriff. Unwahrscheinliche, aber mögliche Fehlaufrufe sind somit ausgeschlossen. Im Beispiel ist dies durch die Klasse `NullObjectAsRealization` angedeutet.

Implementierung der Methoden

Um die »Null-Eigenschaft« auszudrücken, müssen sinnvolle Implementierungen und Rückgabewerte für Methoden gefunden werden. Folgende Aufzählung nennt gute Kandidaten für Rückgabewerte:

- Alle `void`-Methoden werden leer implementiert.
- `boolean`-Methoden geben häufig `false` zurück (z. B. für `hasNext()`).
- Für `int` bieten sich oft `0` oder `-1` als Rückgabewerte an (z. B. für `getSize()`).
- Gleiches gilt für weitere Zahlentypen (`float`, `double` usw.).
- Für Strings bietet sich ein Leerstring als Rückgabe an.
- Für Objektreferenzen kann entweder rekursiv das NULL-OBJEKT-Muster angewendet oder aber der Wert `null` zurückgeliefert werden. Manchmal bietet sich auch das Auslösen einer `NoSuchElementException` an.
- Für untypisierte Container sind im Collections-Framework die NULL-OBJEKTKonstanten `EMPTY_SET`, `EMPTY_LIST` oder `EMPTY_MAP` definiert. Für generische Typen existieren die Methoden `emptySet()`, `emptyList()` und `emptyMap()`, die typsichere, leere und unmodifizierbare Container zurückliefern.
- Arrays sollten mit `new ArrayType[0]` als leeres Array zurückgegeben werden.
- Für Methoden, deren Rückgabewert für ein Null-Objekt nicht sinnvoll gewählt werden kann, sollte eine `UnsupportedOperationException` geworfen werden.

Beispiel

Beim Durchlaufen einer komplexeren Baumstruktur, die über das KOMPOSITUM-Muster realisiert ist, kann die Anwendung des NULL-OBJEKT-Musters helfen, wenn man den Iterationsprozess für Blätter und Aggregationen gleichartig behandeln möchte. Jedes Element im Kompositum kann dafür einen Iterator zum Durchlaufen seiner Unterelemente anbieten: Für Container ist dies einfach durch Rückgabe der Iteratoren aus dem Collections-Framework zu realisieren. Blattelemente besitzen

definitionsgemäß keine Unterelemente. Eine Rückgabe von `null` wäre denkbar, würde jedoch Sonderbehandlungen im Iterationsvorgang bedingen. Vermeiden kann man dies, wenn man eine spezielle Klasse `NullIterator<E>` definiert, die das Interface `Iterator<E>` erfüllt, aber keine Elemente zurückliefert. Eine mögliche Umsetzung zeigt folgendes Listing:

```
public final class NullIterator<E> implements Iterator<E>
{
    public boolean hasNext()
    {
        return false;
    }

    public E next()
    {
        throw new NoSuchElementException("NullIterator provides no
elements!");
    }

    public void remove()
    {
        throw new UnsupportedOperationException("NullIterator does not " +
                                                "implement remove!");
    }
}
```

Durch Einsatz dieses speziellen Iterators vermeidet man, dass beim Iterieren jeweils Abfragen auf Blatt oder Container erfolgen müssen, weil Blattelemente normalerweise keinen Iterator für Unterelemente liefern. Nutzt man dagegen den `NullIterator<E>`, so werden keine Spezialbehandlungen beim Iterieren mehr benötigt.

Natürlich könnte man Kritik an der obigen Realisierung mit Exceptions üben, da Aufrufe einer der Methoden `next()` bzw. `remove()` eine Exception auslösen. Beide Methoden verhalten sich aber gemäß der Spezifikation im Interface `Iterator<E>`. Für den Aufruf von `next()` gilt im Speziellen, dass dieser niemals erfolgen sollte, ohne vorher mit `hasNext()` die Existenz eines weiteren Elements abgesichert zu haben.

Bewertung

Der Einsatz des NULL-OBJEKT-Musters bewirkt Folgendes:

+ **Bessere Lesbarkeit und keine Spezialbehandlungen** – Der Anwendungscode kann auf Spezialbehandlungen verzichten und ist dadurch klarer lesbar.

o **Konzeptionelle Probleme** – Für einige Anwendungsfälle ist es zur Repräsentation von `null`-Werten schwierig, eine sinnvolle Leerimplementierung zu finden. Manchmal gibt es tatsächlich keine geeignete. Dies ist im obigen Beispiel bei der Realisierung der `next()`-Methode der Fall, weswegen auf das Werfen einer Exception als Hilfsmittel zurückgegriffen wird. Auch eine Implementierung der Methode `remove()` lässt sich schwerlich finden.

- **Fehlerverschleierung möglich** – In manchen Situationen ist es erforderlich, die Fälle »es gibt *ein* Objekt« und »es gibt *kein* Objekt« deutlich voneinander unterscheiden zu können. Der Einsatz dieses Musters kann in solchen Fällen zu Problemen führen und Fehler eher verschleiern. Bei der Besprechung der Technik Lazy Initialization werden wir dies in Abschnitt 22.3.2 genauer kennenlernen. Basierend auf dieser Argumentation muss abgewogen werden, ob eine Prüfung auf `null` oder der Einsatz eines Null-Objekts angebrachter ist.

18.3.3 Schablonenmethode (Template method)

Beschreibung und Motivation

Das Muster SCHABLONENMETHODE definiert den grundsätzlichen Ablauf eines Algorithmus und erlaubt es, an einigen Stellen durch Subklassen spezielle Funktionalität einzubringen. Dazu wird ein Algorithmus zunächst in eine festgelegte Abfolge verschiedener Schritte aufgeteilt. Die Abfolge der Schritte wird in einer speziellen Methode der Basisklasse realisiert, die ***Schablonenmethode*** genannt wird. Sie ist in der Regel `final` definiert, um die grundsätzliche Abfolge vor Veränderungen zu schützen. Einige der Berechnungsschritte des Algorithmus sind in der Basisklasse noch undefiniert und werden durch abstrakte Methoden modelliert, die von Subklassen implementiert werden müssen. Möchte man Subklassen dagegen *optional* eine Veränderungsmöglichkeit anbieten, so kann man mit einer Leerimplementierung einer Methode in der Basisklasse arbeiten, die als ***Hook*** bezeichnet wird. Jede Subklasse kann durch Überschreiben dieser Methode bei Bedarf eigene Funktionalität ausführen.

Das beschriebene Vorgehen stellt sicher, dass die Struktur eines Algorithmus unverändert bleibt, Subklassen jedoch Möglichkeiten der Einflussnahme gegeben wird.

Struktur

Der grundsätzliche Ablauf und Algorithmus ist in der Basisklasse `BaseClass` in der Methode `templateMethod()` realisiert und umfasst die Teilschritte 1 bis 2 und einen abschließenden Hook. Diese Funktionalität wird durch die korrespondierenden Methoden `step1()`, `step2()` sowie `afterHook()` realisiert. Die Methode `template-Method()` ist `final` und die Methoden `step1()` und `step2()` sind abstrakt. Die Methode `afterHook()` ist in der Basisklasse leer implementiert, kann aber in Subklassen mit Funktionalität gefüllt werden. Die Subklasse `SubClass` implementiert den variablen Teil des Algorithmus durch die Methoden `step1()` sowie `step2()` und kann so beliebige Funktionalität im Ablauf des Algorithmus beisteuern. In Abbildung 18-19 ist das zugehörige Klassendiagramm gezeigt.

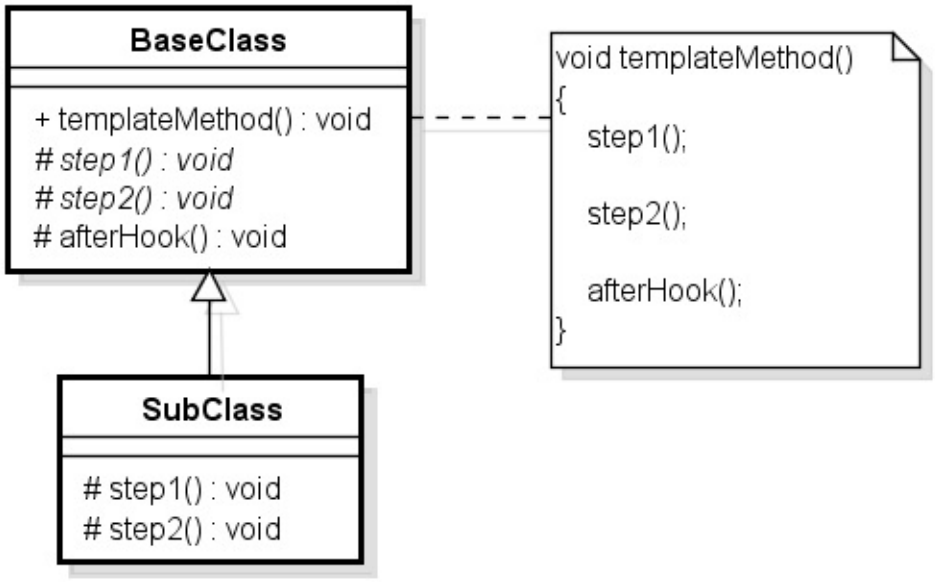


Abbildung 18-19 *Klassendiagramm des* SCHABLONENMETHODE-*Musters*

Beispiel

In der folgenden Klasse `BaseDrawingComponent` ist die Methode `paint(Graphics)` die Schablonenmethode. Der Vorgang des Zeichnens folgt dabei einem festgelegten Ablauf: Zunächst wird der Hintergrund mit `drawBackground(Graphics2D)` gezeichnet. Anschließend erfolgt ein Aufruf der abstrakten Methode `drawContent(Graphics2D)`, die den variablen Teil des Algorithmus definiert, der von Subklassen realisiert werden muss. Abschließend kann optional durch Implementieren der Methode `postDraw(Graphics2D)` eine beliebige Aktion nach dem Zeichnen ausgeführt werden:

```
public abstract class BaseDrawingComponent extends JComponent
{
    public final void paint(final Graphics g)
    {
        super.paint(g);

        final Graphics2D g2d = (Graphics2D) g;

        drawBackground(g2d);
        drawContent(g2d);

        postDraw(g2d);          // hook
    }

    private void drawBackground(final Graphics2D g2d)
    {
        final Dimension componentSize = getSize();
        g2d.setColor(Color.GRAY);
        g2d.fillRect(0, 0, componentSize.width, componentSize.height);
    }

    protected abstract void drawContent(final Graphics2D g2d);

    protected void postDraw(final Graphics2D g2d)
    {
```

```
    }
}
```

Von dieser Basisklasse `BaseDrawingComponent` sind die zwei Klassen `TileImageDrawingComponent` und `ImageDrawingComponent` abgeleitet, die den variablen Anteil des Algorithmus auf ganz unterschiedliche Weise implementieren, wie wir dies im Folgenden sehen werden.

Zufällige Bilder mit Kacheln: `TileImageDrawingComponent`

Die Klasse `TileImageDrawingComponent` zeichnet eine Landschaft, die aus Kacheln besteht. Ein Beispiel ist in Abbildung 18-20 gezeigt.

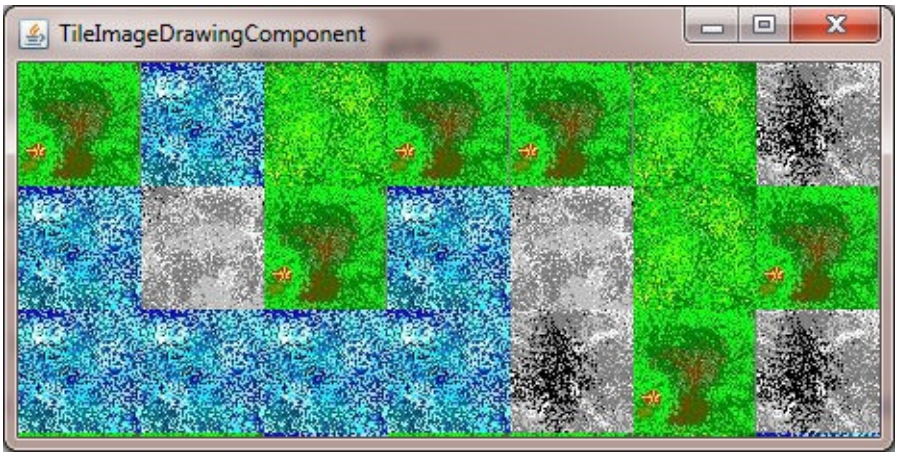


Abbildung 18-20 *Landschaft erzeugt mit der Klasse* `TileImageDrawingComponent`

Bei jeder Größenänderung des Fensters wird per Zufall eine neue Landschaft generiert und durch die Methode `drawContent(Graphics2D)` wie folgt gezeichnet:

```
@Override
public void drawContent(final Graphics2D g2d)
{
    for (int x = 0; x < getSize().width; x += TILES_WIDTH)
    {
        for (int y = 0; y < getSize().height; y += TILES_HEIGHT)
        {
            final int tileIndex = (int) (Math.random() * tileImages.length);
            g2d.drawImage(tileImages[tileIndex], x, y, null);
        }
    }
}
```

Listing 18.4 *Ausführbar als* **'TILEIMAGEDRAWINGCOMPONENT'**

Minizeichenprogramm: `ImageDrawingComponent`

Ein simples Zeichenprogramm wird durch die Klasse `ImageDrawingComponent` realisiert und nutzt wieder die Basisklasse `BaseDrawingComponent`. Abbildung 18-21 zeigt die Miniapplikation. Die Methode `drawContent(Graphics2D)` wird so realisiert, dass dort eine Zeichenfläche mit `drawSheet(Graphics2D)` sowie ein

Raster mit `drawGrid(Graphics2D)` gezeichnet wird. Die verschiedenen grafischen Figuren werden mit der Methode `drawFigures(Graphics2D)` gemalt. Als Letztes wird dort die (im Listing nicht gezeigte) Methode `drawRuler(Graphics2D)` zum Zeichnen zweier Lineale aufgerufen, wodurch gewährleistet ist, dass diese immer über allen anderen Elementen gezeichnet werden und damit in jedem Fall sichtbar sind.

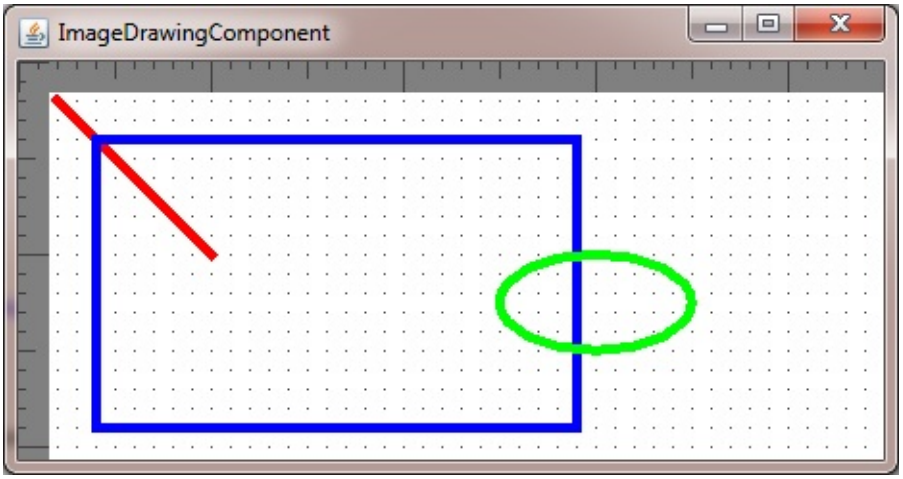


Abbildung 18-21 *Minizeichenprogramm* `ImageDrawingComponent`

```
public void drawContent(final Graphics2D g2d)
{
    drawSheet(g2d);
    drawGrid(g2d);

    drawFigures(g2d);

    drawRuler(g2d);
}

private void drawSheet(final Graphics2D g2d)
{
    g2d.setColor(Color.WHITE);
    g2d.fillRect(0, 0, getSize().width, getSize().height);
}

private void drawGrid(final Graphics2D g2d)
{
    g2d.setColor(Color.DARK_GRAY);

    for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
    {
        for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
        {
            g2d.drawLine(x, y, x, y);
        }
    }
}

private void drawFigures(final Graphics2D g2d)
{
    final Stroke oldStroke = g2d.getStroke();
    g2d.setStroke(new BasicStroke(2.0f));

    for (final AbstractGraphicsElement graphicsFigure :
                                        contentModel.getElements())
```

```
    {
        graphicsFigure.draw(g2d);
    }
    g2d.setStroke(new BasicStroke(5.0f));

    final AbstractGraphicsElement selectedElement =
                                    contentModel.getSelectedElement();
    if (selectedElement != null)
        selectedElement.draw(g2d);

    g2d.setStroke(oldStroke);
}
```

Listing 18.5 *Ausführbar als* **'IMAGEDRAWINGCOMPONENT'**

An diesem Beispiel wird der Hauptunterschied zwischen Schablonenmethode, Hook-Methode und einem Schritt im Algorithmus klar: Eine Hook-Methode ***kann*** überschrieben werden, um optionale Funktionalität zu realisieren. Eine Algorithmus-Methode ***muss*** zwingend überschrieben werden, um den Algorithmus mit Leben zu füllen. Und schließlich: Die Schablonenmethode ***darf nicht*** überschrieben werden.

Bewertung

Der Einsatz des SCHABLONENMETHODE-Musters hat folgende Auswirkungen:

+ **Definition von Erweiterungsstellen** – Es wird ein Algorithmus vorgegeben. Subklassen können an speziellen Stellen eigene Funktionalität einbringen.

○ **Weniger Flexibilität** – Man ist darin eingeschränkt, den Algorithmus zu modifizieren. In diesem Fall ist diese Eigenschaft explizit gewünscht. Soll der Algorithmus jedoch variiert werden können, so ist das Muster SCHABLONENMETHODE nicht die geeignete Wahl. Alternativ kann man mit dem im Folgenden vorgestellten STRATEGIE-Muster zwar den Algorithmus variieren, dafür aber keine Teilschritte vorgeben.

18.3.4 Strategie (Strategy)

Beschreibung und Motivation

Das STRATEGIE-Muster ermöglicht es, das Verhalten eines Algorithmus an ausgesuchten Stellen anzupassen. Im Unterschied zum Muster SCHABLONENMETHODE werden die variablen Bestandteile eines Algorithmus durch eigene Klassen statt durch überschriebene Methoden realisiert. Eine erste Implementierung unterschiedlicher Strategien könnte die Wahl von Alternativen über `if`-Anweisungen steuern. Ein solches Vorgehen ist allerdings schlecht erweiterbar und wird schnell unübersichtlich. Beim STRATEGIE-Muster werden daher die variablen Teile eines Algorithmus jeweils in eigenen Klassen (mit gemeinsamer Basis) gekapselt und sind dadurch austauschbar. Das konkrete Verhalten kann bei Bedarf sogar erst zur Laufzeit durch Wahl einer beliebigen vorhandenen Realisierung

festgelegt werden.

Struktur

Die abstrakte Klasse `AbstractStrategy` definiert eine Schnittstelle mit benötigten und zu variierenden Methoden. Statt einer abstrakten Klasse kann dies alternativ durch ein Interface erfolgen. Spezielle Funktionalität wird in den Subklassen `ConcreteStrategy1` und `ConcreteStrategy2` implementiert. Abbildung 18-22 stellt dies dar.

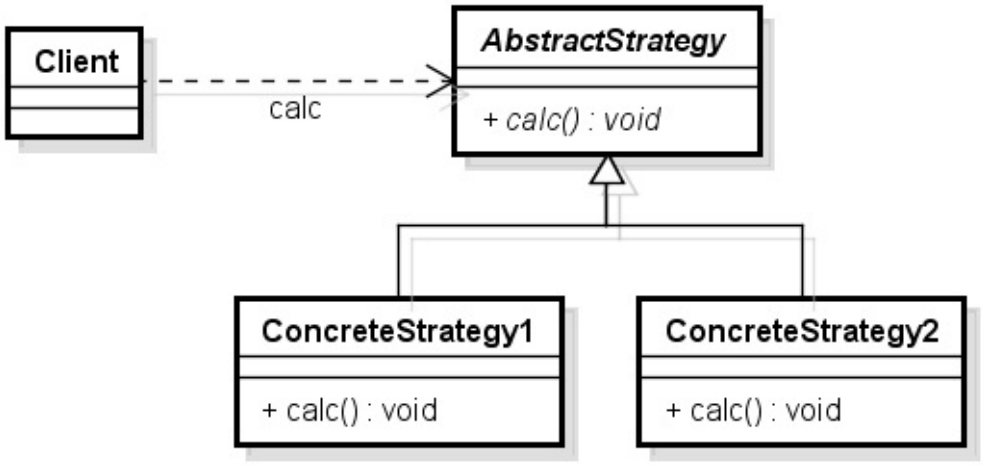


Abbildung 18-22 *Struktur des* STRATEGIE-*Musters*

Beispiel

Die in den `sort()`-Methoden des Collections-Frameworks genutzten Realisierungen des Interface `Comparator<T>` sind Beispiele des STRATEGIE-Musters.

Im Folgenden stelle ich zur Demonstration des STRATEGIE-Musters einen Algorithmus zum Filtern vor: Aus einer Liste von Werten sollen einige Werte nach speziellen Kriterien herausgefiltert werden. Die hier gezeigte Variante beschränkt sich auf das Filtern von `int`-Werten. Weil Generics nicht für primitive Typen anwendbar sind, wird hier zur Speicherung eine Liste von `Integer`-Objekten verwendet.

Erste, intuitive Lösung ohne STRATEGIE-Muster

Es soll eine Menge von Zahlen gefiltert werden, wobei der erlaubte Wertebereich durch zwei `int`-Werte entweder in Form eines geschlossenen oder offenen Intervalls angegeben werden kann. Die Art der Filterung wird mithilfe einer `enum`-Aufzählung `FilterType` mit den Werten `OPEN_INTERVAL` und `CLOSED_INTERVAL` bestimmt. Innerhalb einer Schleife werden die Eingabewerte geprüft und gegebenenfalls in die Ergebnismenge aufgenommen. Die folgende erste Implementierung setzt diese Anforderungen funktional korrekt um:

```
enum FilterType { OPEN_INTERVAL, CLOSED_INTERVAL }

// ..

public static List<Integer> filterAll(final List<Integer> inputs,
                                      final FilterType filterStrategy,
                                      final int lowerBound,
                                      final int upperBound)
{
    final List<Integer> filteredList = new LinkedList<>();

    if (filterStrategy == FilterType.CLOSED_INTERVAL)
    {
        for (final Integer value : inputs)
        {
            if (value >= lowerBound && value <= upperBound)
                filteredList.add(value);
        }
    }
    if (filterStrategy == FilterType.OPEN_INTERVAL)
    {
        for (final Integer value : inputs)
        {
            if (value > lowerBound && value < upperBound)
                filteredList.add(value);
        }
    }

    return resultSet;
}

public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    System.out.println("Filtering values for Intervall 2-7");
    System.out.println("Using CloseInterval [2,7] " + filterAll(inputs,
                                                    FilterType.CLOSED_INTERVAL, 2,
7));
    System.out.println("Using OpenInterval ]2,7[ " + filterAll(inputs,
                                                    FilterType.OPEN_INTERVAL, 2, 7));
}
```

Listing 18.6 *Ausführbar als* **'STRATEGYFILTERBASEEXAMPLE'**

Führen wir das Programm aus, so erhalten wir erwartungsgemäß folgende Ausgabe:

```
Filtering values for Intervall 2-7
Using CloseInterval [2,7] [2, 3, 4, 5, 6, 7]
Using OpenInterval ]2,7[ [3, 4, 5, 6]
```

Funktional können wir also zufrieden sein. Was ist aber zum Design zu sagen? Eine kurze Analyse deckt folgende Probleme auf:

1. Der Sourcecode zum Filtern für geschlossenes bzw. offenes Intervall ist fast 1:1 dupliziert.
2. Jede weitere Filterstrategie erfordert Erweiterungen in der `filterAll()`-Methode. Dadurch reduziert sich schnell die Lesbarkeit.
3. Werden weitere Auswahlkriterien gewünscht, etwa halboffene Intervalle, so muss

die ursprüngliche Klasse um `if`-Anweisungen zur Auswahl der entsprechenden Strategie und deren Realisierung erweitert werden. Zudem müssen neue Aufzählungswerte zur Unterscheidung definiert werden.

4. Die realisierte Filterung setzt jeweils die Angabe von zwei Grenzwerten voraus. Wollte man eine vollständig andersartige Auswahlstrategie anwenden, so wäre dies mit dieser Implementierung extrem aufwendig, vielleicht sogar unmöglich: Eine Filterung auf eine Menge erlaubter Werte ließe sich so nicht realisieren.

Lösung mit dem STRATEGIE-Muster

Es bietet sich an, das STRATEGIE-Muster einzusetzen. Die Basis bildet ein Interface `FilterStrategy` und die dort definierte Methode `acceptValue(int)`. Zur Intervallprüfung werden zwei konkrete Strategieklassen `OpenInterval` und `ClosedInterval` definiert (vgl. Abbildung 18-23).

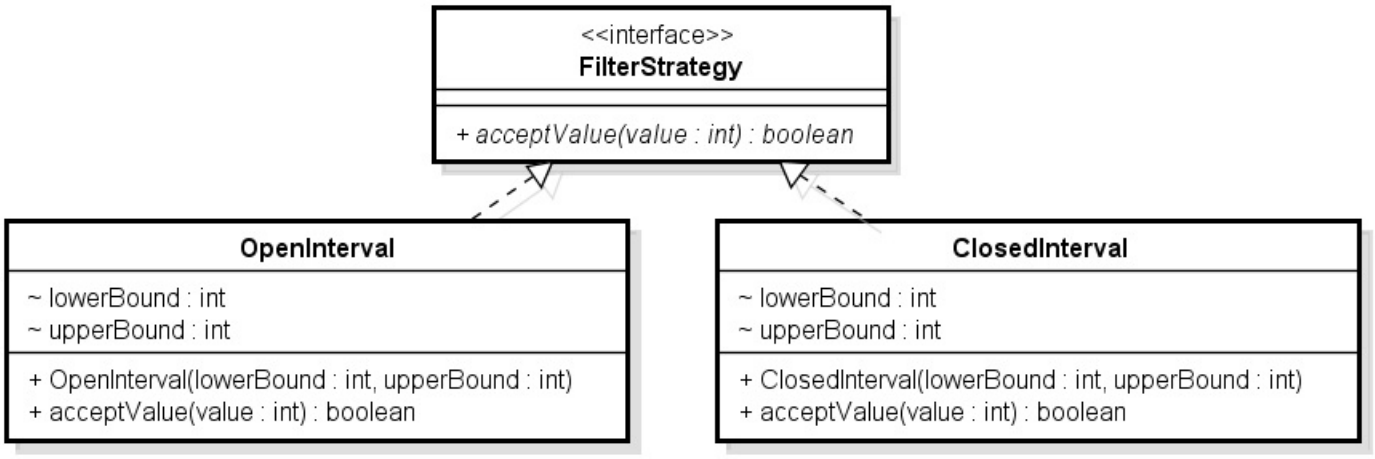


Abbildung 18-23 *Das Interface* `FilterStrategy` *und zwei Realisierungen*

Das grundsätzliche Vorgehen ist mit der zuvor vorgestellten Lösung identisch: Zum Filtern wird über die Eingabe iteriert und für jeden Wert überprüft, ob dieser in das Ergebnis aufgenommen werden soll. Allerdings wird diese Prüfung nicht mehr in der `filterAll()`-Methode realisiert, sondern an die dafür zuständigen Filterklassen delegiert. Exemplarisch zeige ich hier nur die Umsetzung des geschlossenen Intervalls, da die Realisierung des offenen Intervalls analog geschieht.

```
public static class ClosedInterval implements FilterStrategy
{
   private final int lowerBound;
   private final int upperBound;

   public ClosedInterval(final int lowerBound, final int upperBound)
   {
       if (upperBound < lowerBound)
          throw new IllegalArgumentException("lowerBound must be >=
upperBound");

       this.lowerBound = lowerBound;
       this.upperBound = upperBound;
   }
```

```
    @Override
    public boolean acceptValue(final int value)
    {
        return lowerBound <= value && value <= upperBound;
    }

    @Override
    public String toString()
    {
       return "ClosedInterval [" + lowerBound + ", " + upperBound + "]";
    }
}
```

Betrachten wir nun die konkrete Implementierung der Filterung:

```
public final class StrategyFilterExample
{
   public static List<Integer> filterAll(final List<Integer> inputs,
                                          final FilterStrategy filterStrategy)
   {
      final List<Integer> result = new LinkedList<>();
      for (final Integer value : inputs)
      {
         if (filterStrategy.acceptValue(value))
            result.add(value);
      }
      return result;
   }

   public static void main(final String[] args)
   {
      final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

      System.out.println("Filtering values for Intervall 2-7");
      System.out.println("Using " + closedInterval + " " + filterAll(inputs,
                                                    new ClosedInterval(2,
7)));

      System.out.println("Using " + openInterval + " " + filterAll(inputs,
                                                    new OpenInterval(2, 7)));
   }
}
```

Listing 18.7 *Ausführbar als* **'STRATEGYFILTEREXAMPLE'**

Führt man das Programm aus, so stimmen die gelieferten Ergebnisse mit denen des ursprünglichen Programms überein. Funktional ist also kein Vorteil erzielt worden. Zudem ist der Sourcecode-Umfang und die Anzahl der eingesetzten Klassen und Interfaces gestiegen. Die Lösung mit dem STRATEGIE-Muster ist also länger, umfangreicher und komplexer. Wieso ist diese Lösung aber trotzdem in der Regel besser? Die Antwort ist einfach: Diese Lösung ist modularer und leichter erweiterbar. Der Vorgang der Filterung und der tatsächlichen Auswahl sind voneinander unabhängig und lediglich über das Interface `FilterStrategy` verbunden. Eine Einschränkung auf eine spezielle Art der Filterung ist lediglich durch die `acceptValue(int)`-Methode gegeben. Zudem ist die Lösung deutlich objektorientierter, da nun eigenständige Klassen die Auswahl realisieren und weitere Funktionalität, beispielsweise die Aufbereitung einer aussagekräftigen Stringrepräsentation mithilfe der Methode `toString()` zur jeweiligen

Filterstrategie anbieten können. Ohne Strategieobjekte wären für eine derartige textuelle Ausgabe der gewählten Filterstrategie jedes Mal umfangreiche Prüfungen mit diversen `if`-Anweisungen durchzuführen, die ähnlich der Auswahl des Filterkriteriums selbst sind. Dies ist fehlerträchtig. Durch Strategieobjekte wird diese unnötige Komplexität vermieden. Jedes einzelne Objekt überschreibt die `toString()`-Methode und liefert so eine informative Ausgabe.

Falls Sie noch nicht überzeugt sein sollten, kommt das Beste zum Schluss: Durch Einsatz des STRATEGIE-Musters können komplett neue Filterstrategien implementiert werden, die insbesondere auch nicht auf die ursprüngliche Schnittstelle mit den zwei `int`-Werten zur Bereichsprüfung eingeschränkt sind. Ein erstes Beispiel dafür ist etwa eine Filterung auf alle geraden Zahlen:

```
public static class EvenFilter implements FilterStrategy
{
    @Override
    public boolean acceptValue(final int value)
    {
        return value % 2 == 0;
    }

    @Override
    public String toString()
    {
        return "EvenFilter";
    }
}
```

Algorithmisch etwas anspruchsvoller ist eine Filterung von Primzahlen:

```
public final class PrimeFilter implements FilterStrategy
{
    @Override
    public boolean acceptValue(final int value)
    {
        return isPrime(value);
    }

    private boolean isPrime(final int value)
    {
        if (value < 2)
            return false;

        if (value == 2)
            return true;

        for (int i = 2; i <= Math.sqrt(value); i++)
        {
            // Test auf Teilbarkeit
            if ((value % i) == 0)
                return false;
        }
        return true;
    }

    @Override
    public String toString()
    {
```

```
        return "PrimeFilter";
    }
}
```

Wären diese Erweiterungen in der ursprünglichen Klasse zu realisieren gewesen, dann wäre dort bereits ein ziemliches Chaos entstanden.

Noch stärker wäre der Effekt, wenn Erweiterungswünsche aufkommen: Stellen Sie sich vor, dass die inverse Auswahloperation realisiert werden soll. In der ursprünglichen Lösung würde diese Forderung spätestens jetzt zu sehr unübersichtlichem Sourcecode führen: Jede weitere Änderung ist fehlerträchtig und bestenfalls noch durch eine große Anzahl an Unit Tests zu beherrschen. Auch hierbei hilft der Einsatz des STRATEGIEMusters, um den Implementierungs- und Testaufwand erheblich zu reduzieren. Man nutzt zudem das aus Abschnitt 18.2.3 bekannte DEKORIERER-Muster und implementiert eine allgemeingültige inverse Filterstrategie `InverseFilter`. Damit ergibt sich das in Abbildung 18-24 gezeigte Klassendiagramm.

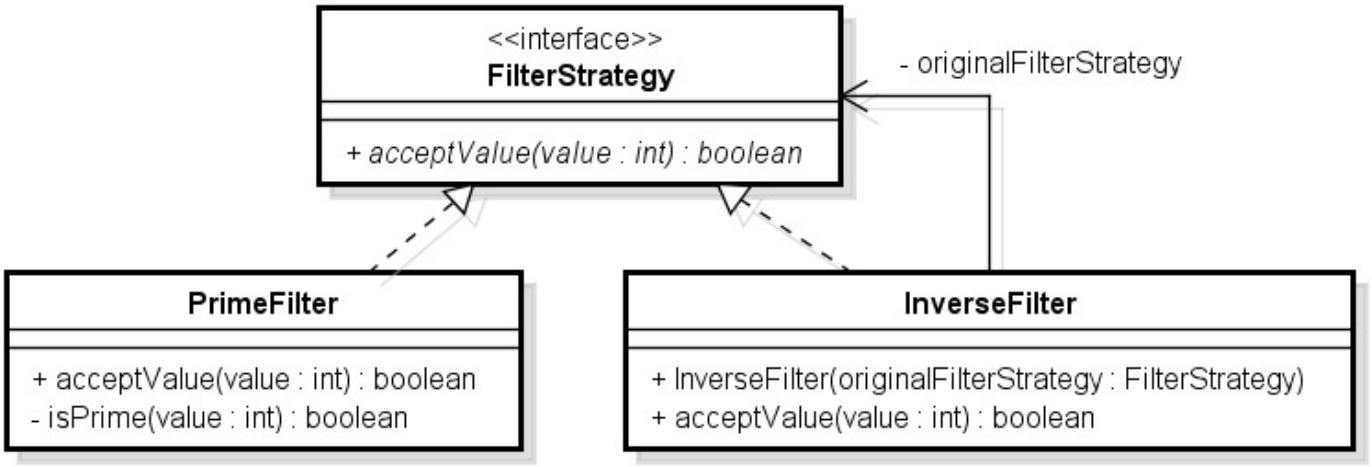


Abbildung 18-24 *Das Interface* `FilterStrategy` *und das* DEKORIERER*-Muster*

Die Implementierung eines inversen Filters durch die Klasse `InverseFilter` geschieht wie folgt:

```
public final class InverseFilter implements FilterStrategy
{
    private final FilterStrategy originalFilterStrategy;

    public InverseFilter(final FilterStrategy originalFilterStrategy)
    {
        this.originalFilterStrategy =
                    Objects.requireNonNull(originalFilterStrategy,
                    "parameter 'originalFilterStrategy' must not be null!");
    }

    @Override
    public boolean acceptValue(final int value)
    {
        return !originalFilterStrategy.acceptValue(value);
    }

    @Override
    public String toString()
    {
```

```
        return "InverseFilter " + originalFilterStrategy;
    }
}
```

Betrachten wir nun, mit wie wenig Aufwand die inversen Filter »ungerade Zahl« und »keine Primzahl« aus den ursprünglichen Filtern `EvenFilter` und `PrimeFilter` realisiert werden können, wenn man die gerade entwickelte Dekoriererklasse `InverseFilter` einsetzt:

```
public static void main(final String[] args)
{
    final List<Integer> inputs = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    // EvenFilter und OddFiler (InverseEvenFilter)
    final FilterStrategy evenFilter = new EvenFilter();
    System.out.println(evenFilter + ": " + filterAll(inputs, evenFilter));

    final FilterStrategy oddFilter = new InverseFilter(evenFilter);
    System.out.println(oddFilter + ": " + filterAll(inputs, oddFilter));

    // PrimeFilter und InversePrimeFilter
    final FilterStrategy primeFilter = new PrimeFilter();
    System.out.println(primeFilter + ": " + filterAll(inputs, primeFilter));

    final FilterStrategy inversePrimeFilter = new InverseFilter(primeFilter);
    System.out.println(inversePrimeFilter + ": " + filterAll(inputs,
          inversePrimeFilter));
}
```

Listing 18.8 *Ausführbar als* **'STRATEGYFILTEREXAMPLE2'**

Startet man das Programm StrategyFilterExample2, so kommt es zu folgender Ausgabe:

```
EvenFilter: [2, 4, 6, 8]
InverseFilter EvenFilter: [1, 3, 5, 7, 9]
PrimeFilter: [2, 3, 5, 7]
InverseFilter PrimeFilter: [1, 4, 6, 8, 9]
```

Bewertung

An diesem Beispiel sieht man deutlich, welche Vorteile der bewusste und gezielte Einsatz der für ein zu lösendes Problem passenden Entwurfsmuster bringen kann. Der Einsatz des Strategie-Musters bewirkt Folgendes:

+ **Erhöhte Flexibilität** – Die Auswahl aus verschiedenen Implementierungen kann zur Laufzeit erfolgen. Dadurch erhöht sich die Flexibilität und die Wiederverwendbarkeit.
+ **Bessere Erweiterbarkeit** – Die Alternativen besitzen keine Abhängigkeiten untereinander, sodass sich verschiedene Anforderungen (Bereichsgrenzen, Primzahlen usw.) unabhängig realisieren lassen. Zudem lassen sich zusätzliche Anforderungen, wie für die inverse Abbildung gezeigt, leicht und gezielt integrieren.
○ **Kopplung an konkrete Strategien** – Klienten müssen die konkreten

Strategieklassen kennen und instanziieren, um die gewünschte Funktionalität auszuführen. Eine Kombination mit einer FABRIKMETHODE (vgl. Abschnitt 18.1.2) bietet sich zur Kapselung und loseren Kopplung an.

- **Erhöhter Umfang** – Die Anzahl der Klassen erhöht sich.

18.3.5 Befehl (Command)

Beschreibung und Motivation

Das Muster BEFEHL vereinfacht die Ausführung komplexerer Aufgaben. Da dieses Muster nicht so leicht zu verstehen ist, möchte ich es anhand eines Beispiels aus der Realität, einer Bestellung in einem Restaurant, verdeutlichen. Ein Gast gibt eine Bestellung auf, etwa: »Ein Steak medium mit Pommes und ein Bier«. Der Kellner notiert diese Bestellung und übergibt sie der Küche und der Theke als Arbeitsanweisungen zur Ausführung. Die ausführenden Einheiten können anhand der Bestellung verschiedene konkrete kleinere Handlungen durchführen (Braten, Frittieren, Bier zapfen usw.), die für den Gast aber uninteressant sind (etwa die Temperatureinstellung am Herd).

Auf die Software übertragen erkennen wir folgende Akteure bei diesem Muster: Ein Klient (Gast) gibt eine Bestellung (einen oder mehrere (abstrakte) Befehle) auf, die von einer Komponente (Kellner) registriert und von anderen Komponenten (Küche und Theke) ausgeführt wird. Ein klassisches Beispiel sind Aktionen eines Editors, die durch Menüs, Kontextmenüs oder Toolbars ausgelöst werden und beispielsweise zum Speichern eines Dokuments oder zum Erzeugen einer grafischen Figur führen.

Struktur

Dieses Muster existiert in verschiedenen, komplexen Ausprägungen, jedoch findet immer eine Kapselung von Methodenaufrufen in Befehlsobjekten statt, um eine Trennung von einem Befehlswunsch und dessen Ausführung zu erreichen. Die Basis bildet normalerweise ein Interface `ICommand`, das standardmäßig eine Methode `execute()` definiert. Jedes konkrete Befehlsobjekt implementiert das Interface `ICommand` und bietet somit eine sehr einfache und für alle Befehlsklassen einheitliche Schnittstelle zur Ausführung an. Jede konkrete Realisierung eines Befehls, im nachfolgenden Beispiel `PrintTextCommand` bzw. `WaitCommand`, implementiert die Methode `execute()` entsprechend der gekapselten Aufgabe, indem dort die Verarbeitung durch feingranulare Methodenaufrufe stattfindet. Dies zeigt Abbildung 18-25.

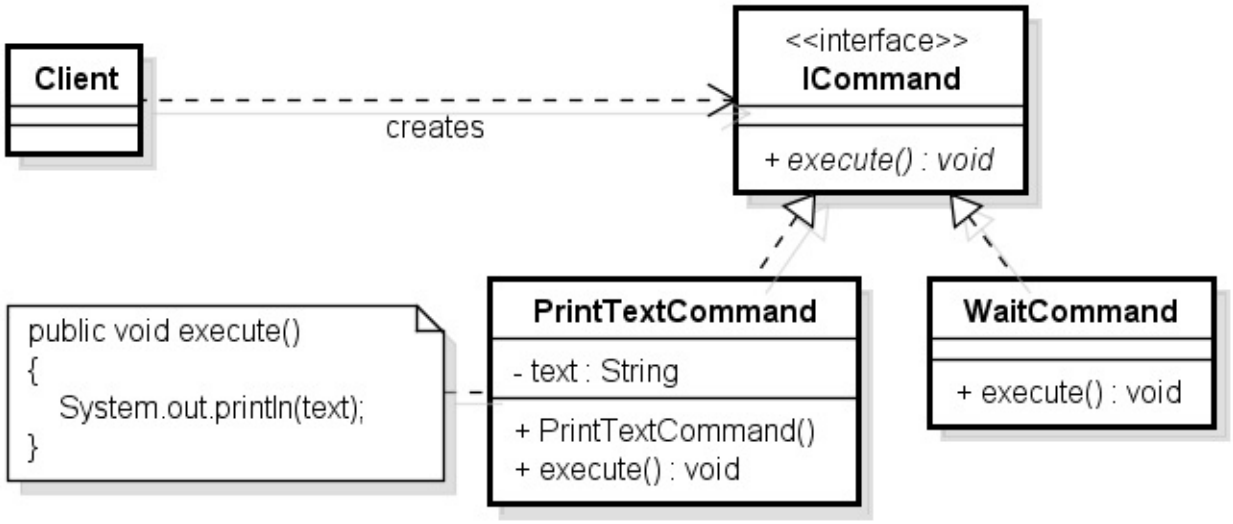


Abbildung 18-25 *Kommando in UML*

Die Komplexität der im Befehlsobjekt realisierten konkreten Aufgabe ist für Klienten zum Anstoßen einer Befehlsausführung irrelevant. Ein Klient benötigt darüber hinaus auch kein Wissen über das konkrete API der im Befehlsobjekt verwendeten Programmteile, da diese dort gekapselt werden. Daher kann man dieses Muster gut zur Entkopplung von Vorgängen (sprich des befehlsaufrufenden und des befehlsempfangenden Programmteils) einsetzen.

Beispiel

Ich stelle das Muster BEFEHL zunächst in seiner einfachsten Form dar, um es schrittweise zu erweitern und auf mögliche Unzulänglichkeiten und Verbesserungen aufmerksam zu machen.

Einfachste Form

Der folgende Sourcecode zeigt einen einfachen Ablauf. Es werden zunächst drei Befehlsobjekte erzeugt und anschließend ausgeführt:

```
public static void main(final String[] args)
{
    // Erzeugen der Command-Objekte
    final ICommand command1 = new PrintTextCommand("Dies ist ein Text");
    final ICommand command2 = new WaitCommand();
    final ICommand command3 = new PrintTextCommand("Der Test ist beendet!");

    // Ausführen der Command-Objekte
    command1.execute();
    command2.execute();
    command3.execute();
}
```

Listing 18.9 *Ausführbar als* **'SIMPLECOMMANDEXAMPLE'**

Betrachten wir dieses sehr einfache Beispiel, so sehen wir, dass lediglich Funktionalität in den Befehlsklassen gekapselt wird. Dies ist bereits gut, könnte allerdings auch durch die Extraktion einer Methode erreicht werden. In diesem Fall

wird zudem der Befehl direkt vom Klienten ausgeführt. Dies kann in einigen wenigen Situationen sinnvoll sein. In der Regel rufen jedoch Klienten die `execute()`-Methode nicht direkt auf. Vielmehr übergeben Klienten Befehle zur Ausführung an eine separate Verarbeitungseinheit, die dann `execute()` aufruft.

Erweiterung um eine Ausführungskomponente

Wir werden nun dieses einfache Modell dahingehend erweitern, dass eine ausführende Komponente hinzugefügt wird. Klienten führen hier die Befehle nicht mehr selbst aus, sondern übergeben diese an die Ausführungskomponente. Diese speichert die auszuführenden Befehle in einer listenähnlichen Struktur. Die im folgenden Listing gezeigte Klasse `CommandExecutor` nutzt dazu eine `LinkedBlockingDeque<E>` zur Thread-sicheren Bearbeitung. Die Implementierung der Klasse `CommandExecutor` als `Runnable` ermöglicht es, diese in einem Thread abzuarbeiten. Parallel dazu findet eine Abarbeitung von Klienten statt. Diese stellen bei Bedarf Befehle zur Ausführung ein.

```
public final class CommandExecutor implements Runnable
{
   private final Deque<ICommand> commands = new LinkedBlockingDeque<>();

   private static final ICommand NULL_COMMAND = new ICommand()
   {
       @Override
       public void execute()
       {
       }

       @Override
       public String toString()
       {
           return "NULL_COMMAND";
       }
   };
```

Im vorherigen Listing wird zudem ein statisches Attribut `NULL_COMMAND` gemäß dem NULL-OBJEKT-Muster (vgl. Abschnitt 18.3.2) erzeugt, das wir im Folgenden nutzen werden, um bei der Bearbeitung der Befehlsliste `null`-Abfragen zu vermeiden.

Das folgende Listing setzt das Beispiel fort und zeigt, wie Befehlsobjekte von Klienten mithilfe der Methode `registerCommand(ICommand)` als Bearbeitungswunsch eingetragen werden können. Parallel dazu werden Befehlsobjekte in der Methode `run()` sequenziell abgearbeitet. Dort wird das nächste auszuführende Befehlsobjekt mithilfe der Methode `getAndRemoveNextCommand()` ermittelt und per Aufruf von `execute()` ausgeführt. Ist kein solches Befehlsobjekt verfügbar, erspart der Einsatz des `NULL_COMMAND` eine Spezialbehandlung in der Abarbeitung:

```
public void registerCommand(final ICommand commandToExecute)
{
    Objects.requireNonNull(commandToExecute, "Passed command must not be
```

```
null");

    commands.offer(commandToExecute);
}

public void run()
{
    while (!Thread.currentThread().isInterrupted())
    {
        final ICommand commandToExecute = getAndRemoveNextCommand();
        commandToExecute.execute();

        SleepUtils.safeSleep(50); // Vermeide CPU-Belastung
    }
}

private ICommand getAndRemoveNextCommand()
{
    final ICommand commandToExecute = commands.poll();
    if (commandToExecute == null)
        return NULL_COMMAND;

    return commandToExecute;
}
```

Im folgenden Programmausschnitt wird der `CommandExecutor` genutzt, um die aus dem vorherigen Beispiel bekannten drei Befehle abzuarbeiten. Im Unterschied zur simplen Variante erkennt man hier, dass kein Aufruf von `execute()` durch den Klienten erfolgt, sondern lediglich ein Bearbeitungswunsch durch `registerCommand(ICommand)` ausgedrückt wird:

```
public static void main(final String[] args)
{
    // Die Abarbeitung der Kommandos erfolgt nebenläufig
    // zu den folgenden Aktionen
    final CommandExecutor executor = new CommandExecutor();
    new Thread(executor).start();

    // Client erzeugt Kommandos
    final ICommand command1 = new PrintTextCommand("Dies ist ein Text");
    final ICommand command2 = new WaitCommand();
    final ICommand command3 = new PrintTextCommand("Der Test ist beendet!");

    // Client übergibt Kommandos an Executor
    executor.registerCommand(command1);
    executor.registerCommand(command2);
    executor.registerCommand(command3);
}
```

Listing 18.10 *Ausführbar als* **'COMMANDEXECUTOR'**

Nach der Erweiterung um den `CommandExecutor` sieht das Klassendiagramm wie in Abbildung 18-26 aus.

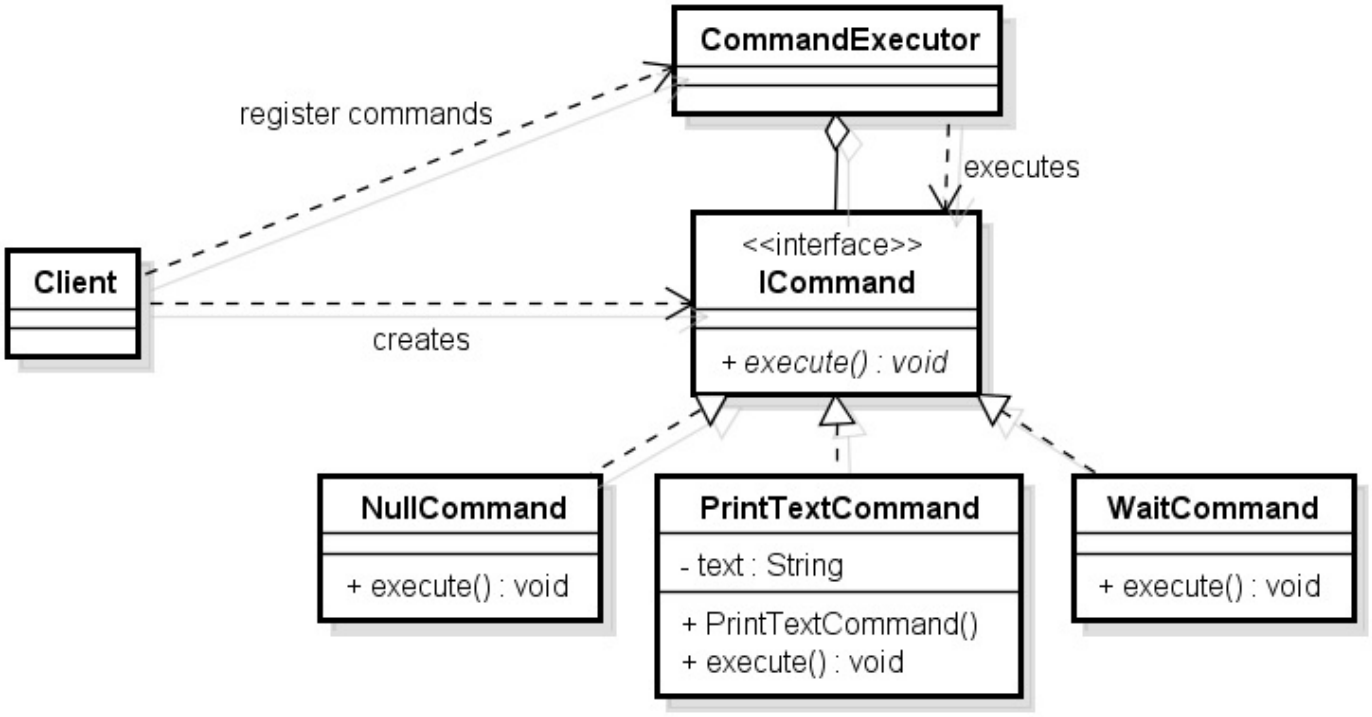


Abbildung 18-26 *Command und Executor in UML*

Ausführungskontext

Bislang sind die vorgestellten `execute()`-Methoden funktional sehr einfach. In der Praxis benötigen Befehle meistens einen Kontext oder Eingabeparameter zu ihrer Ausführung bzw. Daten, um auf diesen zu operieren. Selbst für das einfache `PrintTextCommand` ist ein Parameter zur Übergabe des auszugebenden Texts erforderlich.

Viele zur Ausführung benötigte Informationen stehen bereits zum Erzeugungszeitpunkt der Befehlsobjekte fest und können als Parameter an deren Konstruktor übergeben werden. Mit diesen Parametern legt man häufig die Rahmenbedingungen und Eigenschaften fest. Bezogen auf das Beispiel einer Bestellung im Restaurant sind dies beispielsweise die Namen des Gerichts und des Getränks.

Einige Informationen, etwa die konkrete Umgebung der Ausführung oder die ausführende Einheit, sind eventuell noch unbekannt. Dies sind Detailinformationen, die zur tatsächlichen Abarbeitung eines Befehls, also von dessen `execute()`-Methode, benötigt werden. Bezogen auf das Restaurantbeispiel kennt der Gast wesentliche Details der Zubereitung nicht: Welches Stück Fleisch gebraten wird und welcher Koch die Zubereitung übernimmt, wird erst in der Küche entschieden.

Aus diesen beiden Randbedingungen folgt, dass es in der Regel zwei Arten von Parametrierungen eines Befehls gibt:

1. **Spezifikationsdaten** – Einige Parameter werden genutzt, um eine konkrete Spezifikation des Befehls und damit eine Präzisierung des Auftrags vorzunehmen: Es wird das »Was« beschrieben, etwa: »Steak medium mit Pommes«. Diese Daten

werden von Klienten festgelegt.

2. **Ausführungsdaten** – Um den gewünschten Auftrag auszuführen und aus der zuvor angegebenen Spezifikation eine konkrete Abarbeitung des Befehls zu ermöglichen, werden in der Regel weitere Daten und Akteure benötigt, die an den Befehl übergeben werden müssen. Es wird das »Wie« und »Wer« beschrieben, etwa: »Welcher Koch die Arbeit übernimmt, welches Steak zubereitet wird, welcher Grill verwendet wird.« Auf diese Daten hat ein Klient in der Regel keinen Einfluss.

Die Daten zur Präzisierung des Auftrags sollten von Klienten möglichst zum Konstruktionszeitpunkt eines Befehlsobjekts übergeben und in dessen Attributen gespeichert werden.

Wie bereits angedeutet, wird ein Befehl zur Ausführung seiner `execute()`-Methode Zugriff auf diverse andere Objekte und Informationen benötigen. Zur Vereinfachung ist es sinnvoll, die erforderlichen Daten als Laufzeitkontext in Form einer Value-Object-Klasse (vgl. Abschnitt 3.4.5) `ExecutionContext` zu bündeln. Zur Aufbereitung dieser Informationen ist ein Zugriff auf alle beteiligten Klassen nötig, um deren Referenzen im Value Object speichern zu können. Wann und wo können diese Informationen dem Befehlsobjekt übergeben werden? In einigen Applikationen kann man diese den Befehlsobjekten bereits zu ihrem Konstruktionszeitpunkt übergeben. Häufig stehen die benötigten Informationen den Erzeugern der Befehlsobjekte (z. B. Menüeinträgen) jedoch nicht ohne Weiteres zur Verfügung. Dann müssten die Daten für die Erzeuger nur deshalb zugänglich gemacht werden, um die Befehlsobjekte zum Konstruktionszeitpunkt mit dem notwendigen Kontext zu versorgen; die Anzahl der Parameter und Abhängigkeiten nehmen damit zu. Das stört Kapselung und Wiederverwendbarkeit. Wenn ein Befehlsobjekt von verschiedenen Programmteilen erzeugt werden kann (z. B. über ein Menü, ein Kontextmenü, einen Button oder Shortcut), muss jedes dieser Programmteile dann all diese – ansonsten nicht benötigten – Informationen bereitstellen können.

Einfacher und klarer ist es deshalb meistens, diese Aufgabe der Informationsbereitstellung durch ein Objekt vom Typ `CommandExecutor` erledigen zu lassen. Es bietet sich folgender Trick an, der die Ideen der sogenannten ***Dependency Injection*** aufgreift: Der Methode `execute()` werden die Kontextdaten als Klasse `ExecutionContext` übergeben. Diese stellt dann alle zur Ausführung benötigten Informationen zur Verfügung. Die Konstruktion von Befehlsobjekten kann nun unabhängig von der Bereitstellung der Ausführungsdaten erfolgen.

Tipp: Bereitstellung von Ausführungsdaten

Werden Applikationen komplexer oder sollen Befehle in mehreren Kontexten ausgeführt werden, so bietet sich die Übergabe eines `ExecutionContext` an die Methode `execute()` an. Nur so ist eine Unabhängigkeit von Befehlserzeugung und deren Ausführung möglich. Das gilt insbesondere dann, wenn Befehle in einer JVM erzeugt und in einer anderen JVM ausgeführt werden sollen.

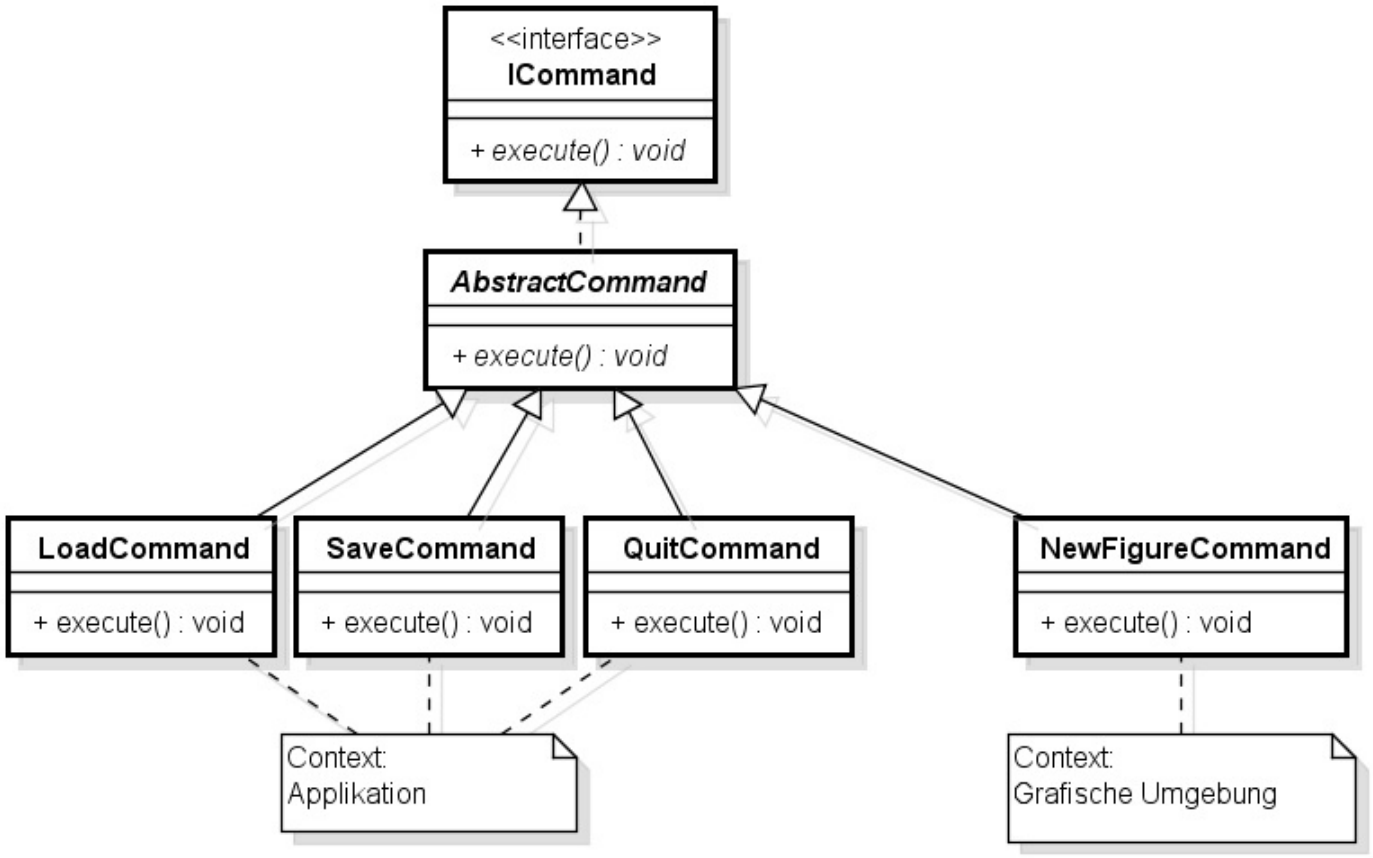


Abbildung 18-27 *Das Muster* BEFEHL *in einem grafischen Editor*

Erweiterungen

Auch eine Undo-Funktionalität ist mithilfe des Musters BEFEHL leichter zu realisieren als ohne. Im einfachsten Fall speichert man bei jedem Ausführen eines Befehls zuvor den aktuellen Zustand des Gesamtsystems ab und kann diesen bei einem Undo wiederherstellen. Das ist aber in der Regel zu aufwendig. Daher wird bevorzugt mit relevanten, von dieser Aktion betroffenen Ausschnitten des Zustands gearbeitet. Undo ist daher in realen Systemen immer mit einigem Aufwand verbunden.

Leichter als eine Undo-Funktionalität lassen sich Befehle als Makros (Kommandofolgen) zusammenfassen und als Einheit ausführen. Durch eine Speicherung einer Abfolge von Befehlen kann man eine Art Befehlsrekorder bauen, der ein späteres Abspielen ähnlich eines Videorekorders erlaubt. Die Makros können wir mithilfe des KOMPOSITUM-Musters (vgl. Abschnitt 18.2.4) und durch eine Klasse `MacroCommand` realisieren.

Bewertung

Der Einsatz des Musters BEFEHL bewirkt Folgendes:

+ **Lose Kopplung** – Durch die Kapselung von Aktionen in Form von Objekten können diese von beliebigen Programmteilen erzeugt und von anderen Programmteilen ausgeführt werden. Funktionalität kann auf verschiedenen

Wegen verfügbar gemacht werden, etwa als Reaktion auf Menüs, Buttons oder andere Eingaben, ohne dass es zu einer Kopplung an konkrete Elemente der Benutzerschnittstelle kommt: Aufrufer und ausführende Einheit sind nicht miteinander verbunden, wie dies bei einem Methodenaufruf der Fall wäre.

+ **Bessere Abstraktion** – In einem Befehl werden verschiedene auszuführende Methodenaufrufe an einer Stelle gebündelt. Dies ist ähnlich zu der Extraktion einer Methode, geht allerdings einen Schritt weiter, da ein neues, in anderen Kontexten wiederverwendbares Befehlsobjekt entsteht.

+ **Wiederverwendbarkeit** – Ein Befehl stellt eine Verhaltensbeschreibung, eine Art semantische Klammer über verschiedene Methoden, dar und ist vielseitig einsetzbar. Ein Befehl kann von unterschiedlichen Auftraggebern (Menü, Kontextmenü, Toolbar usw.) an verschiedene ausführende Einheiten weitergereicht und dort in diversen Kontexten (Einzelselektion, Mehrfachselektion usw.) ausgeführt werden.

+ **Fernsteuerung** – Eine Speicherung und spätere Ausführung von Befehlen ist möglich: Befehlsobjekte können wie alle anderen Objekte auch verarbeitet oder in Methodenaufrufen verwendet werden. Im Speziellen könnte man diese serialisieren und zu späteren Zeitpunkten erneut ausführen. Daher kann dieses Muster gut zur zeitgesteuerten Fernsteuerung eingesetzt werden. Tatsächlich können auch andere Programme die Befehle verarbeiten.

+ **Undo/Redo-Fähigkeit** – Eine Undo/Redo-Fähigkeit ist leichter zu realisieren. Zu definierten Zeitpunkten können Zwischenstände des zugrunde liegenden Modells gesichert werden, die bei Bedarf wiederhergestellt werden können.

- **Erhöhter Umfang** – Die Anzahl der Klassen erhöht sich.

18.3.6 Proxy

Beschreibung und Motivation

Das PROXY-Muster dient zum Verlagern der Kontrolle über ein Objekt auf ein Stellvertreterobjekt, den sogenannten ***Proxy***. Ein Klient spricht den Stellvertreter an und kennt das eigentliche Objekt nicht. Dadurch kann der Proxy sowohl die Erzeugung als auch den Zugriff auf das eigentliche Objekt kontrollieren und somit eine Zugangskontrolle realisieren oder Remote Calls verstecken. Auch zur Vereinfachung von Lazy Initialization kann ein Proxy eingesetzt werden.

Struktur

Die Basis dieses Musters ist eine gemeinsame Schnittstelle `ServiceIF`. Diese stellt sicher, dass sowohl der Proxy in Form der Klasse `ServiceProxy` als auch das eigentliche Objekt vom Typ `Service` nach außen gleich behandelt werden können.

Zur Durchführung der eigentlichen Aufgaben verwaltet der Proxy eine Referenz auf das Objekt. Falls angebracht, kann man sogar die Erzeugung des Objekts an den Proxy delegieren, die dann beim ersten Aufruf einer Methode an den Proxy erfolgt.

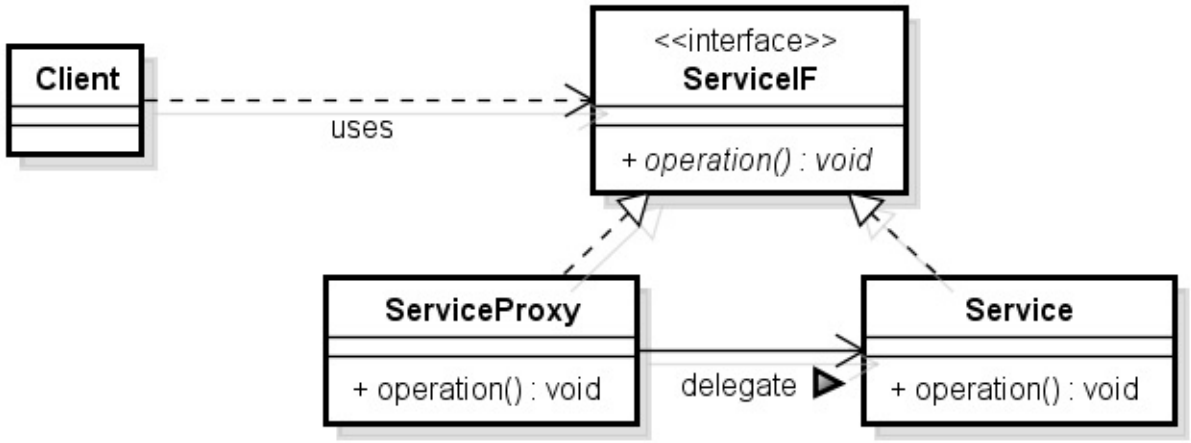


Abbildung 18-28 *Proxy in UML*

Varianten

Das PROXY-Muster existiert in der Realität unter anderem in folgenden konkreten Ausprägungen, die verschiedene Intentionen verfolgen:

- **Remote Proxy** – Kann den Eindruck erwecken, ein entferntes Objekt wäre ein lokales. Dies ist zum Verbergen des Einsatzes von RMI oder Webservices praktisch.
- **Lazy Init Proxy** – Vermittelt den Eindruck, ein Objekt stehe schon zur Verfügung, bevor es tatsächlich erzeugt wurde. Objektbestandteile werden erst in dem Moment konstruiert, in dem diese auch tatsächlich benutzt werden sollen. Optionale Funktionalität oder besonders aufwendig zu konstruierende Daten kann man mit einem solchen Proxy kapseln. Dies kann zu signifikanten Performance-Verbesserungen führen und wird in Abschnitt 22.3 detailliert beschrieben.
- **Access Control Proxy** – Kann den Zugriff auf gewisse Daten steuern. Beispielsweise könnte vor jeder Methodendelegation eine Rechteprüfung erfolgen, die die Eingabe eines Passworts verlangt.

Beispiel

Dieses Beispiel demonstriert einen Access Control Proxy für den Zugriff auf eine Map. Die folgende Proxy-Klasse `RestrictedAccessMap` implementiert das Interface `Map<K, V>` und erweitert die ursprüngliche Realisierung um eine Prüfung auf Zugriffsrechte. Es wird vor jeder Methodendelegation an die zugrunde liegende Map über eine Methode `ensureAccessGranted()` geprüft, ob der gewünschte Zugriff erlaubt ist. Hat ein Benutzer keine Zugriffsrechte, so erfolgt kein Zugriff auf die Map und es wird eine selbst definierte `InvalidAccessRightsException`

ausgelöst. Diese muss vom Typ `RuntimeException` sein, da in den Signaturen der Methoden des Interface `Map<K, V>` keine Checked Exceptions definiert sind.

```
public class RestrictedAccessMap<K, V> implements Map<K, V>
{
    private final Map<K, V> underlyingMap = new HashMap<>();

    public void ensureAccessGranted() throws InvalidAccessRightsException
    {
        // …
    }

    @Override
    public V put(final K key, final V value)
    {
        ensureAccessGranted();
        return underlyingMap.put(key, value);
    }

    @Override
    public V remove(final Object key)
    {
        ensureAccessGranted();
        return underlyingMap.remove(key);
    }
    // …
```

Bewertung

Der Einsatz des PROXY-Musters bewirkt Folgendes:

+ **Steuerung von Funktionalität** – Ähnlich zum DEKORIERER-Muster kann die eigentliche Anwendungsfunktionalität um weitere Funktionen ergänzt werden. Bei diesem Muster steht jedoch der steuernde Charakter im Vordergrund.

○ **Aufwand durch Delegation** – Besitzt ein Originalobjekt viele Methoden, so ist die Realisierung des Proxy-Objekts durch die vielen, notwendigen Delegationen aufwendig.

18.3.7 Beobachter (Observer)

Beschreibung und Motivation

Der Einsatz des BEOBACHTER-Musters ermöglicht es, Änderungen am Zustand eines speziellen Objekts durch andere interessierte Objekte beobachten zu können, ohne dass sich ein Objekt und seine Beobachter direkt kennen müssen. Beobachter melden sich dazu als Interessenten bei einem Objekt an und »abonnieren« damit Informationen über Veränderungen, die ihnen das Objekt in der Folge mitteilt. Sind Interessenten nicht mehr an Änderungen interessiert, so können sie sich abmelden.

Man kann die diesem Muster zugrunde liegende Idee mit dem Abonnement einer

Zeitschrift vergleichen. Hat man sich als Abonnent registriert, so erhält man regelmäßig die neuesten Ausgaben, ohne ständig am Kiosk nachschauen zu müssen. Dieses Muster ist auch unter dem Namen ***Publisher/Subscriber*** oder ***Listener*** bekannt.

Ereignisbehandlungen in grafischen Oberflächen basieren in der Regel auf diesem Prinzip – häufig durch Einsatz einer Model-View-Controller-Architektur, die in Abschnitt 18.3.8 beschrieben wird.

Struktur

Um Änderungen an einem beobachteten Subjekt `ObservedSubject` mitgeteilt zu bekommen, können sich andere, das Interface `ChangeListener` erfüllende Objekte als Änderungsinteressenten bei dem `ObservedSubject` anmelden. Ein zugehöriges Klassendiagramm ist in Abbildung 18-29 gezeigt.

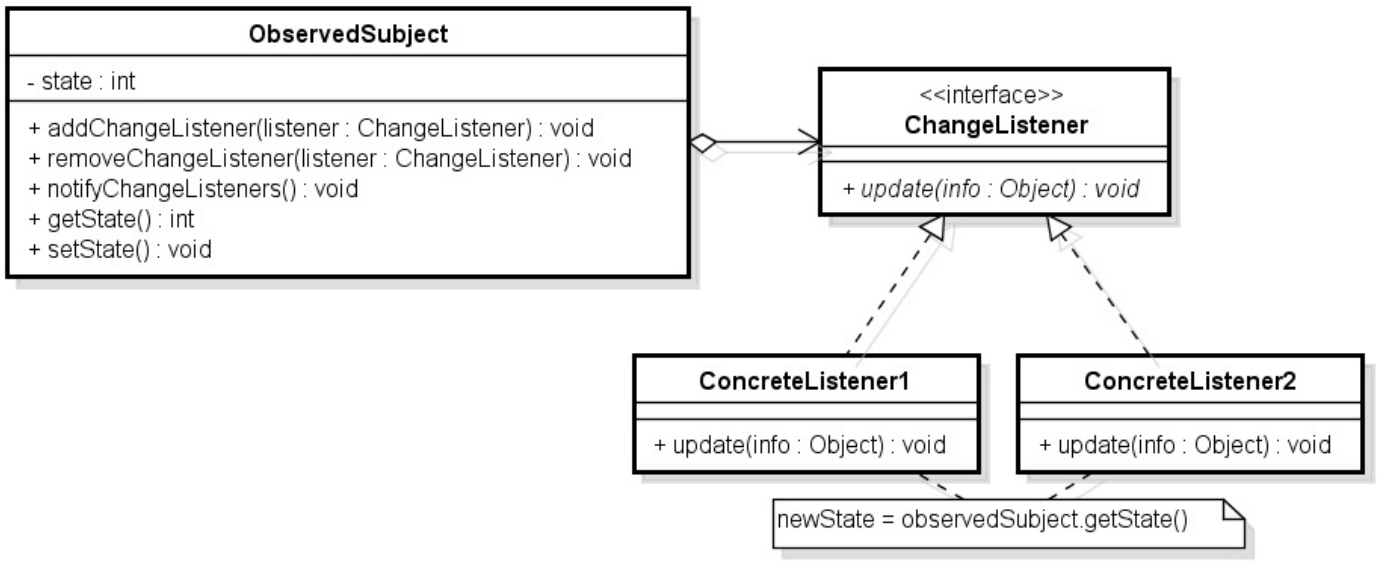


Abbildung 18-29 BEOBACHTER-*Muster mit zwei Beobachtertypen*

Sobald sich der relevante Zustand des `ObservedSubjects` verändert, löst dieses über die Methode `notifyChangeListeners()` eine Benachrichtigungsmitteilung an jeden angemeldeten Beobachter vom Typ `ChangeListener` aus. Dazu wird die Änderung in Form eines Methodenaufrufs propagiert: In den folgenden einführenden Beispielen übernehme ich hier zunächst vereinfachend den in der Literatur gebräuchlichen Namen `update()` für diese Methode. Im Verlauf der Beschreibung dieses Musters werde ich begründen, warum ein solch allgemeiner Name nicht optimal ist, und dann Alternativen mit besser lesbaren und informativeren Methodennamen vorschlagen.

Bei einer Änderungsnachricht variiert deren Inhalt in der Art und Anzahl der übertragenen Daten. Man unterscheidet zwischen den im folgenden Abschnitt beschriebenen Varianten ***Pull*** und ***Push***. Nachdem ein Beobachter über eine Änderung informiert wurde, kann dieser weitere Informationen von dem `ObservedSubject` holen. Dazu existieren in der Regel diverse `get()`-Methoden – hier exemplarisch die Methode `getState()`.

In der Model-View-Controller-Architektur sorgt das beschriebene Vorgehen bei einer Datenänderung dafür, dass als Beobachter angemeldete Views automatisch aktualisiert werden können. Ein möglicher Ablauf ist in Abbildung 18-30 in Form eines Sequenzdiagramms für das Modell `ObservedSubject` und zwei registrierte Views `ConcreteListener1` bzw. `ConcreteListener2` visualisiert.

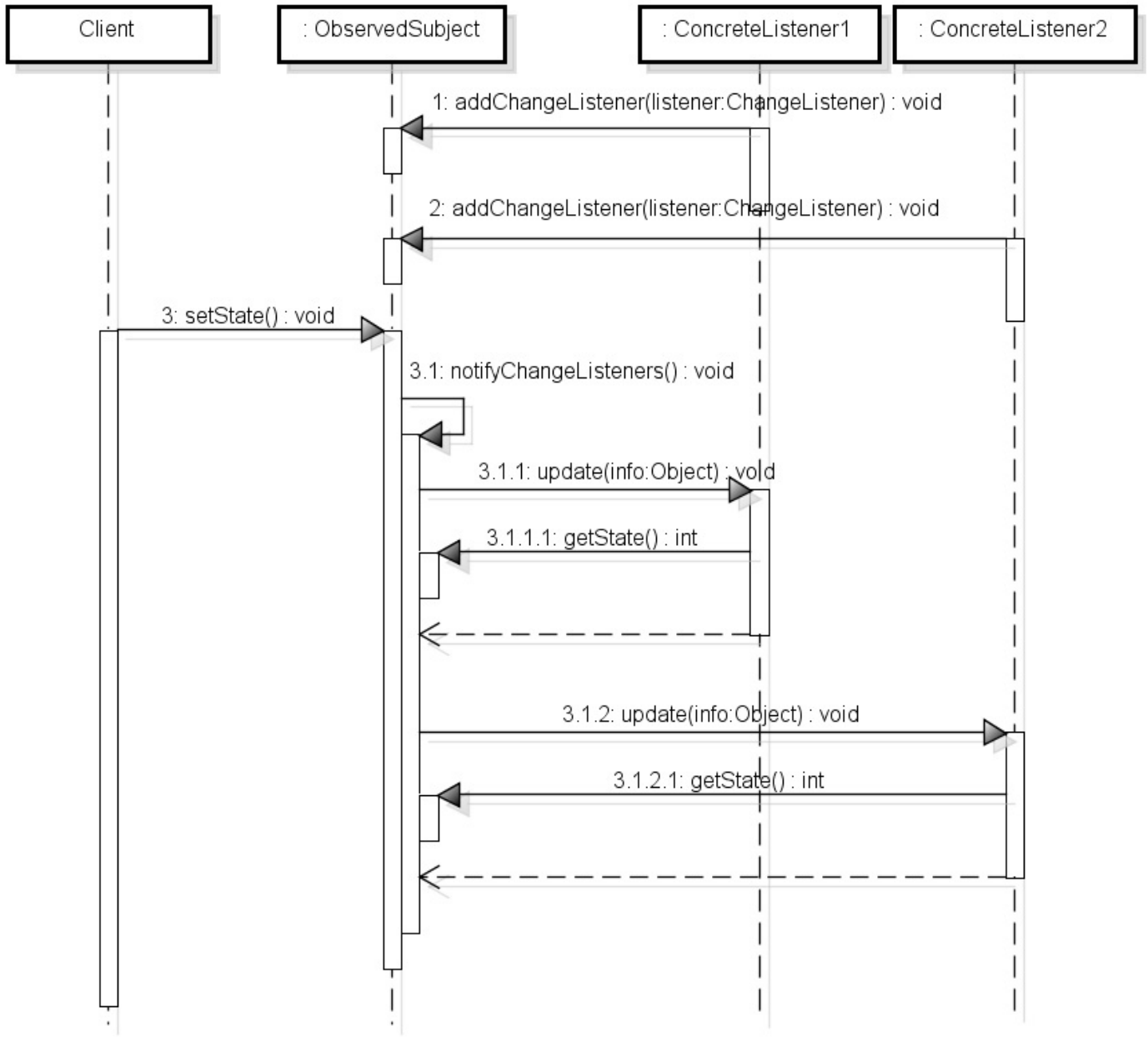


Abbildung 18-30 *Ablauf im* BEOBACHTER*-Muster*

Anmerkungen

Nachdem wir den groben Ablauf kennengelernt haben, müssen wir das BEOBACHTER-Muster genauer betrachten, da es einige Fallstricke bereithält, die man unbedingt kennen sollte.

Varianten der Information über Zustandsänderungen

Um Beobachter über Zustandsänderungen zu informieren, wird eine `update()`-

Methode aufgerufen. Dabei gibt es zwei mögliche Varianten zur Übermittlung benötigter Zustandsdaten:

- **Pull** – Jeder einzelne Beobachter holt sich benötigte Zustandsdaten beim beobachteten Subjekt selbst ab.
- **Push** – Alle Zustandsdaten des beobachteten Subjekts werden bei Aufruf der Methode `update()` mitgeschickt.

Bei der Pull-Variante dient die `update()`-Methode lediglich als Hinweis, dass eine Änderung stattgefunden hat. Zur Ermittlung und zum Abgleich von Zustandsinformationen werden durch die jeweiligen Beobachter in der Regel diverse `get()`-Methoden zur Zustandsabfrage des `ObservedSubjects` aufgerufen. ***Dies bedingt allerdings, dass jeder Beobachter eine Rückreferenz auf das beobachtete Subjekt besitzt.*** Im Idealfall ist dies lediglich ein Interface, das aus einem Übergabeparameter der `update()`-Methode stammt und nicht im Beobachter gespeichert wird (vgl. Diskussion im Absatz »Mythos lose Kopplung«). Ein Vorteil des Pull-Verfahrens ist, dass der Zustand nur bei Bedarf durch den Beobachter auch tatsächlich nachgefragt wird. Für gewisse Darstellungen kann es bereits ausreichend sein, zu wissen, dass überhaupt eine Änderung stattgefunden hat, um etwa ein '*' für eine veränderte Datei anzuzeigen.

Bei der Push-Variante werden der `update()`-Methode alle benötigten Zustandsinformationen als Parameter mitgegeben. Dadurch werden keine Rückrufaktionen durch die Beobachter erforderlich, wodurch die Kopplung gelöst wird: Beobachter müssen sich lediglich bei »ihrem« Subjekt als Interessent anmelden, dessen sonstige Schnittstelle jedoch nicht kennen. Nachteil dieser Lösung ist allerdings, dass man alle möglicherweise benötigten Informationen ermitteln und versenden muss. ***Da keine Annahmen über Beobachter und von diesen benötigte Anteile des Subjektzustands im Voraus bekannt sind, muss immer der komplette Zustand mitgeteilt werden. Das zu übertragende Datenvolumen kann unter Umständen recht umfangreich sein.***

Beobachtung mehrerer Subjekte

Es ist möglich, dass sich ein Beobachter bei mehreren Subjekten als Interessent registriert. Zur Unterscheidung der Quelle der Änderungen ist es daher wichtig, dass eine Referenz auf den Urheber der Änderung in der `update()`-Methode mitgesendet wird. Arbeitet man jedoch nur mit einer simplen `update()`-Methode ohne einen solchen Quellenparameter, so weiß der Beobachter nicht, bei welchem Subjekt er den aktuellen Zustand nachfragen soll.

Benachrichtigungsmechanismus bei Multithreading

Im Fall einer Zustandsänderung sollen alle angemeldeten Beobachter informiert werden. Dies klingt viel einfacher, als es tatsächlich ist. Betrachten wir dazu folgendes Szenario: Zu beliebigen Zeitpunkten können sich Beobachter nebenläufig an- oder

abmelden. Dies könnte theoretisch auch während einer laufenden Benachrichtigung geschehen. Sofort fragt man sich: Wie sorgt man für eine konsistente Beobachterliste bei Multithreading-Zugriffen? Im Prinzip muss man alle Zugriffe auf die Beobachterliste synchronisieren, um Probleme zu verhindern. Das bedeutet allerdings auch, dass Beobachter sich während eines Benachrichtigungsvorgangs weder an- noch abmelden können, wenn die Realisierung synchronisiert wie in der folgenden Methode erfolgt:

```
private synchronized void notifyChangeListeners()
{
    final Iterator<ChangeListener> it = listeners.iterator();
    while (it.hasNext())
    {
        final ChangeListener listener = it.next();

        listener.update(this);
    }
}
```

In Kapitel 7 haben wir dieses Problem detailliert analysiert. Bei Bedarf nach mehr Parallelität bietet sich der Einsatz `CopyOnWriteArrayList<ChangeListener>` zur Speicherung der Beobachter an. Eine explizite Synchronisierung wird dadurch überflüssig.

Tipp: Weitere Probleme von Benachrichtigungen

Bei der Realisierung von Benachrichtigungsmethoden gibt es weitere, bisher nicht genannte Probleme:

1. Soll ein während eines Benachrichtigungsvorgangs neu hinzugefügter Beobachter über eine Änderung informiert werden?
2. Müssen sich abmeldende Beobachter, die bereits benachrichtigt wurden, Zustandsänderungen rückgängig machen?

Für beide Fragen gibt es keine allgemeingültige Antwort. Das Verhalten ist applikationsspezifisch. Diese Probleme werden nur extrem selten auftreten, weil An- und Abmeldungen normalerweise nicht während der Beobachtung ausgeführt werden. Die obigen Probleme werden dann häufig akzeptiert, da eine Behandlung nur zu viel mehr Komplexität für relativ wenig Nutzen führen würde. Bezogen auf das Zeitschriftenabonnement hieße dies: Wenn man sich am Tag vor der Versendung von Zeitschriften als Abonnent anmeldet, erhält man in der Regel die aktuelle Ausgabe nicht mehr. Ähnliches gilt für eine kurzfristige Abmeldung, man wird höchstwahrscheinlich die aktuelle Ausgabe noch zugestellt bekommen.

Verzögerung durch Bearbeitung der Listener

Bei der Benachrichtigung der Beobachter besteht weiterhin das Problem, dass es sich beim Aufruf der Methode `update()` um synchrone Aufrufe an die Beobachter handelt (vgl. Abbildung 18-30). Das bedeutet auch, dass eine weitere Verarbeitung

durch Aufruf der `notifyChangeListeners()`-Methode so lange blockiert wird, bis alle Beobachter informiert wurden. Dies kann unter Umständen einige Zeit dauern, wenn einer oder mehrere Beobachter lang andauernde `update()`-Methoden realisieren. Auch hier gibt es zwei Möglichkeiten zur Lösung: Man kann die gesamte Benachrichtigung asynchron in einem eigenen Thread ablaufen lassen oder jeden einzelnen `update()`-Aufruf asynchron ausführen. Beides führt aber schnell zu weiteren Synchronisationsproblemen, weil es zu konkurrierenden Lesezugriffen kommt, die dann eventuell auf bereits von anderen Threads aktualisierte Daten zugreifen.

Mythos lose Kopplung

Bei Realisierungen analog zum eingangs gezeigten Beispiel wird oftmals keine lose Kopplung erreicht. Schauen wir auf die Gründe.

Zum einen gibt eine Abhängigkeit durch die Speicherung von Referenzen der einzelnen `ChangeListener` im beobachteten Subjekt. Da dies jedoch in der Regel in Form eines Interface, also gekapselt, erfolgt, ist das meistens nicht störend.

Zum anderen kommt es zu einer Kopplung von Beobachtern in Richtung Subjekt. Diese ist in der Regel unangenehmer, da Beobachter häufig eine Referenz auf den konkreten Typ des Subjekts nutzen, um nach Änderungsmitteilungen auf den Zustand des Subjekts zuzugreifen. Es kommt deshalb zu einer starken, unidirektionalen Kopplung zwischen Beobachter und Subjekt. Ein Beobachter kann dadurch sämtliche Methoden der Schnittstelle seines Subjekts aufrufen. ***Eine Verbesserung erreicht man durch Einsatz eines Interface `IDataProvider`, das vom Subjekt zu erfüllen ist und lediglich Zugriff auf die für Beobachter relevanten Daten erlaubt: Nutzen die Beobachter nur noch Referenzen auf dieses Interface, so löst man die Kopplung***. Abbildung 18-31 zeigt eine mögliche Umsetzung.

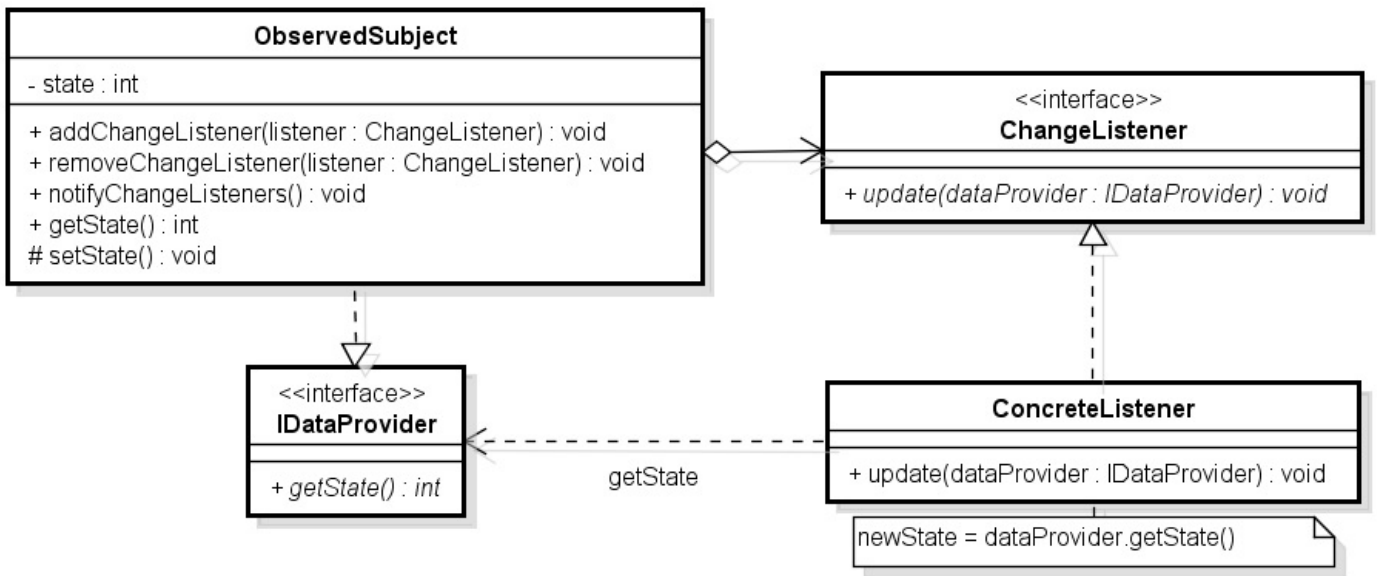


Abbildung 18-31 BEOBACHTER*-Muster mit* `IDataProvider`*-Interface*

Beispiel: Warum ist Java-Built-In-Observer ungünstig?

Die Entwickler von Sun haben in das JDK mehrere BEOBACHTER-Muster integriert. Die Realisierungen durch die Klassen `ActionListener` und `EventListener` sind gelungen. Leider kann man das nicht für die Realisierung in Form der Klasse `Observable` und des Interface `Observer` sagen. Die Klassenhierarchie ist in Abbildung 18-32 dargestellt.

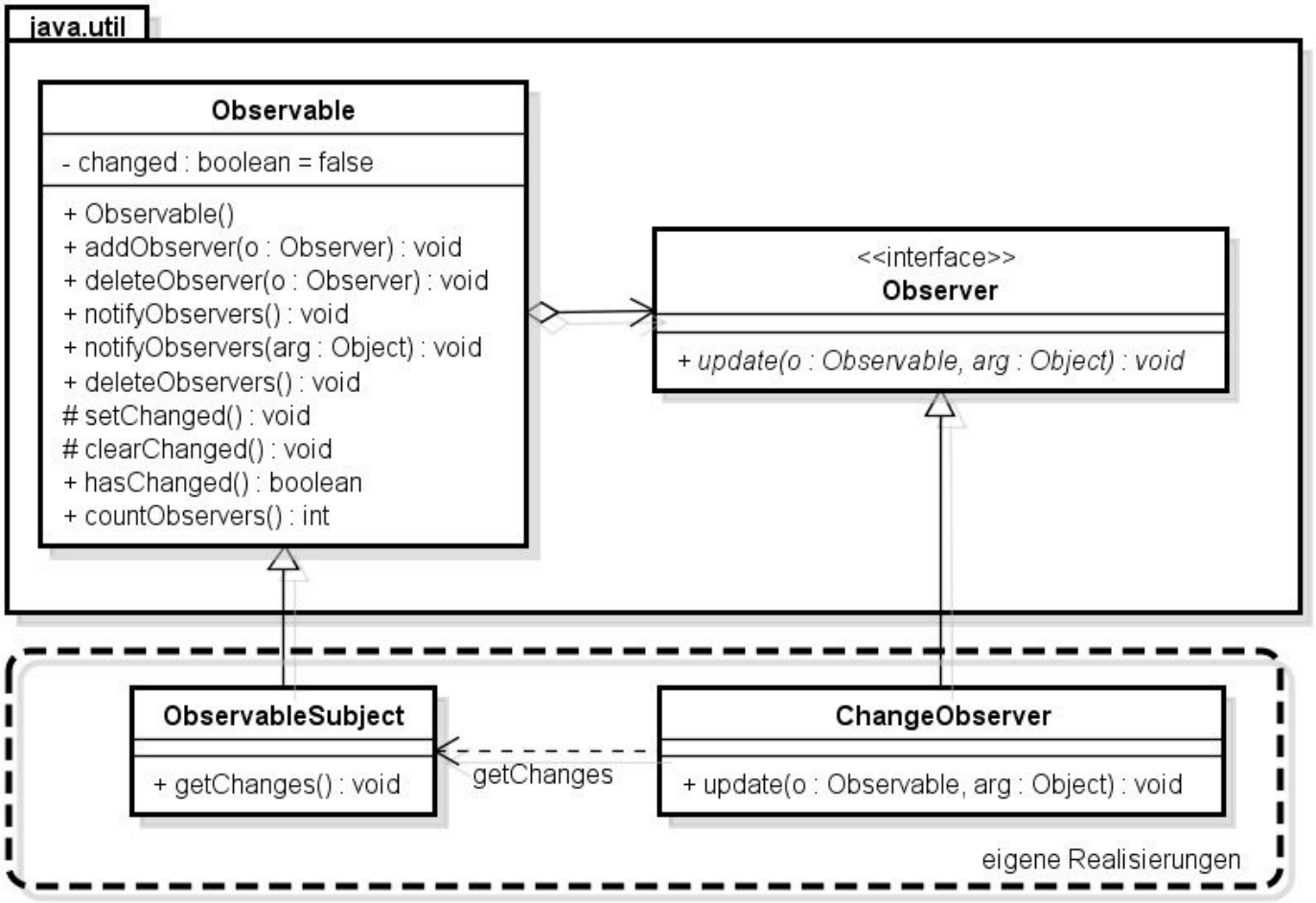


Abbildung 18-32 *Nutzung der Observer-Funktionalität aus dem JDK*

Die Realisierung des BEOBACHTER-Musters durch diese Klassenhierarchie verstößt gegen einige OO-Gedanken. Beobachtbar zu sein ist eine Rolle und keine Erweiterung einer Basisklasse. Um diese Funktionalität nutzen zu können, erfordert die JDK-Realisierung eine technisch bedingte Ableitung von der Basisklasse `Observable` (Implementierungsvererbung). ***Interessanterweise widerspricht zudem die gewählte Namensgebung allen Konventionen im JDK:*** Die Endung `-able` wird ansonsten immer für Namen von Interfaces verwendet, hier jedoch zur Benennung einer Klasse. Erschwerend kommt hinzu, dass durch den Zwang, von einer konkreten Klasse abzuleiten, eine Nutzung dieser vorgefertigten Realisierung in einer eigenen Klassenhierarchie unmöglich wird: Besitzt eine eigene Klasse bereits eine Basisklasse, so kann sie nicht mit dem Java-Built-In-Observer genutzt werden.[7] Daher muss man in der Regel die gesamte Funktionalität erneut selbst realisieren.

Weiterhin ist die Methode `update()` zu unspezifisch. Erstens hat ihr Name wenig Aussagekraft. Zweitens ist unklar, welche Teile des geänderten Zustands als

Übergabeparameter verwendet werden sollten. Für eine solche generische Lösung ist damit der Datentyp der mitzuteilenden Daten unbekannt. Daher muss in diesem Fall ein Parameter vom Typ `Object` übergeben werden. Um sinnvoll mit den Daten arbeiten zu können, muss jeder Beobachter daraus die benötigten Informationen zurückgewinnen. Dies erfordert eine explizite Typprüfung per `instanceof`, um einen ansonsten nicht typsicheren Cast auf den erwarteten Typ der Zustandsinformation zu vermeiden.

Anhand dieser kurzen Diskussion erkennt man leicht, dass eine universelle, generische `update()`-Methode, wie sie im Built-In-Observer von Java realisiert ist, wenig Sinn macht.

Verbesserungen

Lediglich eine allgemeine `update()`-Methode zur Benachrichtigung anzubieten, ist schlecht lesbar und provoziert Missverständnisse. Häufig ist es sinnvoller, in das Interface `ChangeListener` mehrere spezielle Methoden aufzunehmen, mit denen jeweils spezifische, mögliche Zustandsänderungen propagiert werden können. Dies hat zudem den Vorteil, dass man mit sprechenden Methodennamen arbeiten kann und sich damit Änderungen wesentlich besser nachvollziehen lassen als bei einer unspezifischen `update()`-Methode. Als Beispiel sehen wir hier einen Beobachter `IModelListener`, der auf Änderungen in einem Modell durch drei Methoden informiert wird:

```
public interface IModelListener
{
   public void imageElementsChanged(final List<AbstractGraphicsElement>
images);
   public void pdfElementsChanged(final List<AbstractGraphicsElement> pdfs);

   public void nameChanged(final String newName);
}
```

Als logische Erweiterung kann man statt eines allgemeinen Beobachters mehrere spezialisierte Beobachter für ausgewählte Teilaspekte des Objektzustands definieren. Eine Änderung der Selektion könnte etwa durch einen weiteren, speziellen `ISelectionListener` und dessen `selectionChanged()`-Methode behandelt werden:

```
public interface ISelectionListener
{
   void selectionChanged(final List newSelection);
}
```

Abbildung 18-33 zeigt ein mögliches Klassendiagramm, das die beiden Beobachter nutzt und aufgrund der eingesetzten Push-Variante keine Referenzen und Zugriffe auf das Datenmodell `ContentModel` benötigt. Allerdings würde es hier durch die Trennung der Interfaces verschiedener Beobachter möglich, recht spezifische Informationen in den Benachrichtigungsmethoden zu kommunizieren.

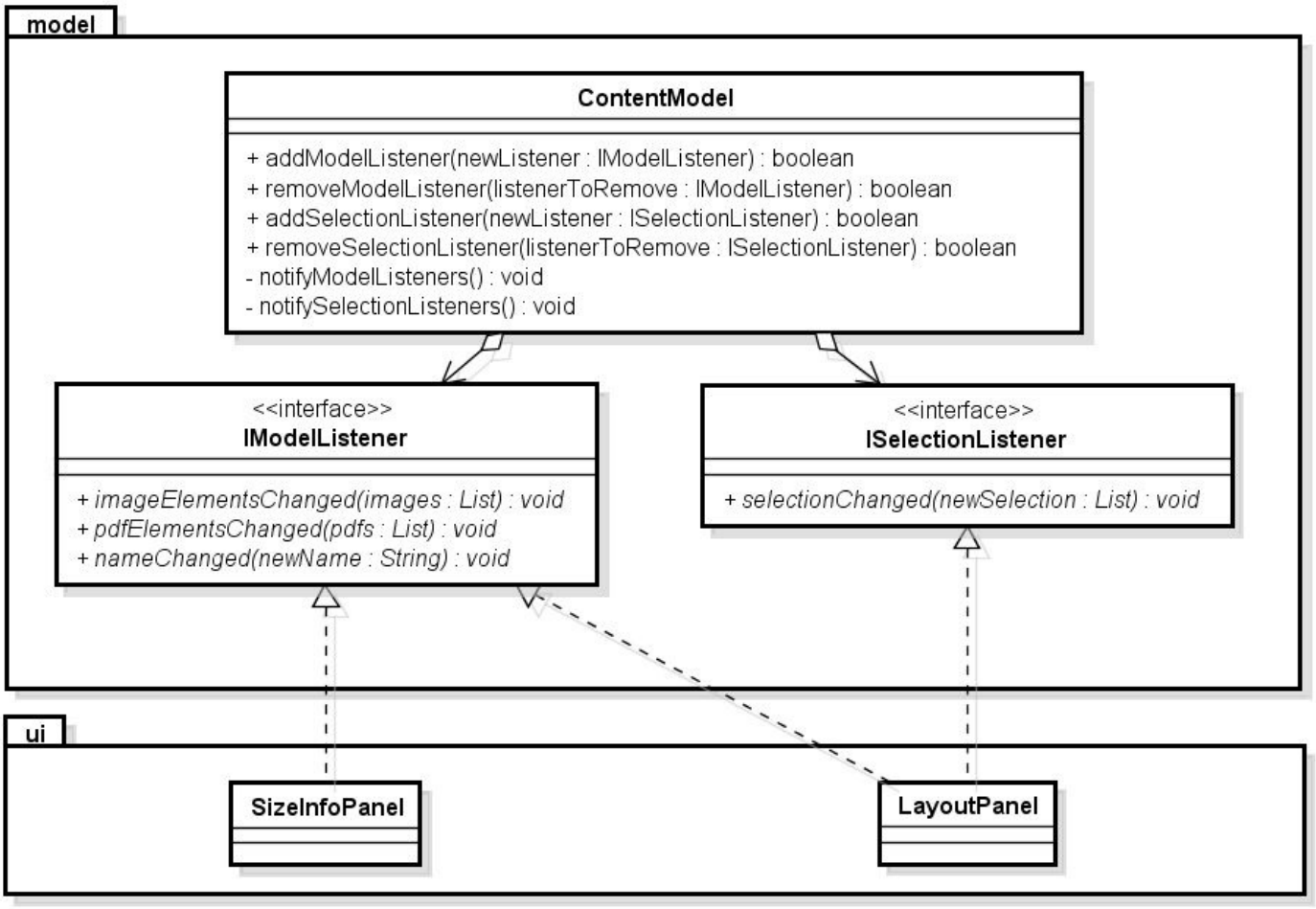


Abbildung 18-33 *Beispiel für mehrere Beobachter*

Bewertung

Der Einsatz des BEOBACHTER-Musters hat folgende Auswirkungen:

+ **Lose Kopplung** – Im Idealfall kennen sich Subjekt und registrierte Beobachter nicht direkt, wenn eine Kapselung über Interfaces erfolgt. Durch die Trennung von Subjekt und Beobachtern sind Erweiterungen in beiden Klassen problemlos möglich, solange die Schnittstelle unverändert bleibt.

+ **Flexibilität** – Die Anzahl der Interessenten kann zur Laufzeit verändert werden. Hinzukommende oder später wieder entfernte Beobachter beeinflussen im Idealfall andere Beobachter nicht (siehe auch den Punkt »Probleme der Statusänderung«).

○ **Fehlende Reaktion** – Falls einen Beobachter gewisse Meldungen nicht interessieren, kann er diese gegebenenfalls ignorieren: Es bleibt dem Beobachter überlassen, ob und wie er mit Nachrichten umgeht. Im Extremfall kann ein Beobachter die Änderungsmitteilungen einfach nicht beachten.

○ **Probleme der Statusänderung** – Das Subjekt weiß nicht, welche Aktionen die Beobachter bei einer Änderungsmitteilung ausführen. Komplexe Aktionen können die Abarbeitung der Benachrichtigung verzögern oder sogar blockieren.

- **Komplexität** – Es stellt eine große Herausforderung dar, den Benachrichtigungsvorgang konsistent und Thread-sicher durchzuführen. In der Regel lebt man mit gewissen Kompromissen, etwa dem, dass Beobachter, die sich während einer Benachrichtigung neu anmelden, erst bei zukünftigen Änderungen informiert werden.
- **Nachvollziehbarkeit** – Beim Einsatz dieses Musters werden häufig sehr viele Benachrichtigungen erzeugt, was die Übersicht erschwert. Die Ursachen solch unerwünschter Updates sind schwierig zu finden.
- **Gefahr von Zyklen** – Werden Beobachter wieder von anderen Klassen beobachtet, so kommt man schnell in ein Benachrichtigungschaos oder sogar in einen Zyklus.

Achtung: BEOBACHTER als Anti-Pattern

Das BEOBACHTER-Muster kann sehr schnell zum Anti-Pattern werden, wenn Beobachter ihrerseits selbst ein beobachtetes Subjekt sind. Sind Abhängigkeitsbeziehungen nicht klar definiert, kommt es schnell zu schwer nachvollziehbaren Updates. Eine scheinbar harmlose Zustandsänderung auf einem Subjekt kann eine Kaskade von Updates an Beobachter und deren abhängiger Beobachter zur Folge haben. In einem solchen System können kleinste Änderungen dazu führen, dass das System nicht mehr konvergiert. Nur minimale Variationen der Anfangsparameter führen möglicherweise zu einem komplett anderen Systemverhalten.

Es kann sehr schnell eine Lawine von nicht mehr beherrschbaren Ereignissen ausgelöst werden, die im schlimmsten Fall zu Endlosschleifen führen, jedoch zumindest eine häufige Ursache für eine schlechte Performance sind.

Tipp: Auswirkung von Business-Methoden auf das BEOBACHTER-Muster

Das Einführen von Business-Methoden hilft, ein mögliches Benachrichtigungschaos beim Einsatz des BEOBACHTER-Musters (vgl. Abschnitt 18.3.7) zu vermeiden. Wenn lediglich Business-Methoden Zustandsänderungen propagieren, sorgt dies für mehr Klarheit und Transparenz.

18.3.8 MVC-Architektur

Die sogenannte Model-View-Controller-Architektur teilt ein zu modellierendes System in die drei Bestandteile Daten (Model), Darstellung (View) und Business- bzw. Kontrolllogik (Controller). Jeder Teil wird möglichst unabhängig von den anderen realisiert und über Schnittstellen gekapselt.

Die Idee, Daten und ihre Repräsentation zu trennen, ist insofern sinnvoll, als dass

man mehrere Darstellungsformen anbieten kann. Weiterhin erreicht man durch diese Trennung eine klarere Struktur. Die Trennung bedingt allerdings, dass Änderungen im Modell an die Views kommuniziert werden müssen. Abschnitt 18.3.7 beschreibt das BEOBACHTER-Muster, das dafür eingesetzt werden kann.

Anmerkungen

Wenn man es auf sehr feingranularer Ebene betrachtet, verstößt die MVC-Architektur gegen die Idee der Objektorientierung, die besagt, dass ein Objekt all seine Aspekte selbst behandeln sollte. Eine interessante Darstellung dazu finden Sie im Buch »Holub on Patterns« von Allen Holub [42]. Auf der anderen Seite kann man es auch als Fortführung des Gedankens der Kapselung sehen. Ein Objekt wird dabei semantisch in mehrere Objekte aufgespalten. Jeweils eins ist für den Zustand (Model) und das Verhalten (View und Controller) zuständig.

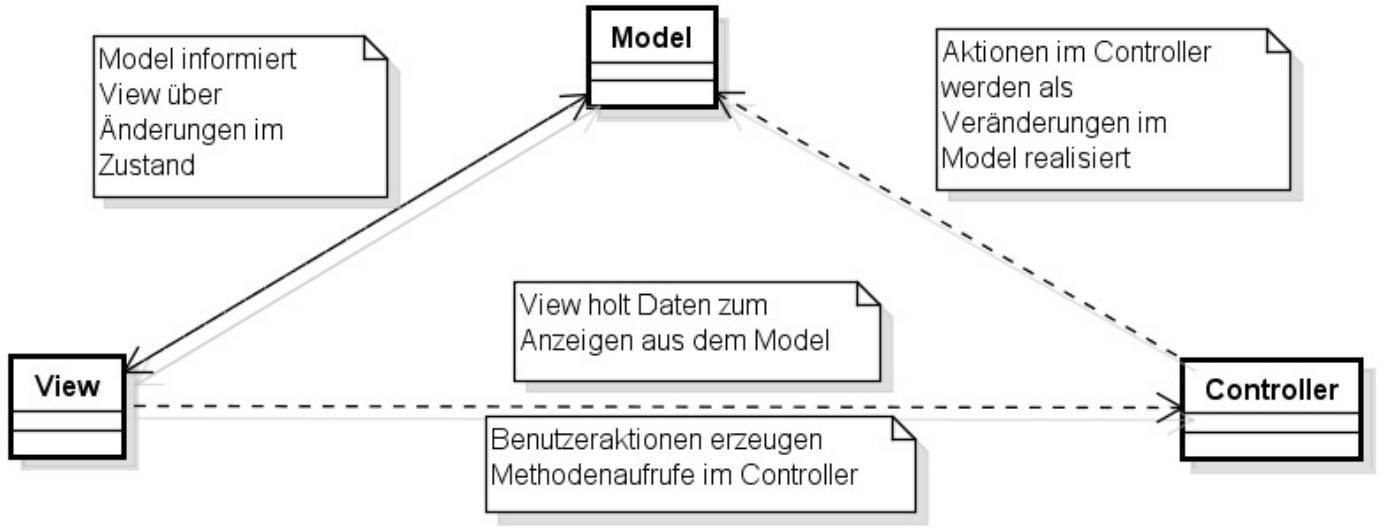


Abbildung 18-34 *Model-View-Controller*

Bewertung

Der Einsatz der MVC-Architektur besitzt folgende Auswirkungen:

+ **Lose Kopplung** – Kommunizieren die einzelnen Komponenten ausschließlich über Interfaces, so sind alle Komponenten nur lose miteinander verbunden.
+ **Trennung von Zuständigkeiten** – Jede Komponente repräsentiert einen speziellen Aspekt des Gesamtsystems. Es findet eine Trennung von Zuständigkeiten statt.
+ **Flexibilität** – Mehrere Darstellungen oder Ansichten (z. B. Balkendiagramm und Tortendiagramm) auf dieselben Daten lassen sich leicht realisieren. Die einzelnen Komponenten können als eine Klasse oder als mehrere Klassen oder sogar in Form eines eigenständigen Programms realisiert sein.
+ **Konsistente Darstellung** – Durch die zentrale Datenhaltung im Modell und die

Änderungsbenachrichtigungen mithilfe des BEOBACHTER-Musters können alle Darstellungen (Views) konsistente Daten anzeigen (sofern die Views korrekt auf Änderungsmitteilungen reagieren).

- **Overengineering** – In einer Auslegung der Objektorientierung sollte ein Objekt alle seine Belange regeln. Nicht immer ist eine Aufteilung wie bei der Model-View-Controller-Architektur sinnvoll und notwendig. Wie bei allen Techniken sollte man darauf achten, dass man nicht mit Kanonen auf Spatzen schießt.

- **Erhöhte Komplexität** – Die Anzahl der benötigten Klassen führt zu etwas höherer Komplexität.

- **Gefahr vieler Änderungsmitteilungen** – Werden viele feingranulare Änderungen im Modell vorgenommen und sofort an die Ansichten kommuniziert, so kommt es zu diversen Aktualisierungen der Ansichten. Das kann unerwünscht sein und sich negativ auf die Performance auswirken.

- **Gefahr enger Kopplung** – Werden die Komponenten nicht durch Schnittstellen voneinander abgekoppelt, so führt man ungewollt direkte Abhängigkeiten ein und die Wiederverwendbarkeit der Einzelkomponenten sinkt.

18.4 Weiterführende Literatur

Zu dem Thema Entwurfsmuster ist unzählige Literatur erschienen. In der folgenden Aufzählung empfehle ich einige Bücher, die jeweils unterschiedliche Aspekte und Herangehensweisen bei der Vorstellung von Entwurfsmustern verfolgen.

- »**Design Patterns – Elements of Reusable Object Oriented Software**« bzw. »**Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software**« von der sogenannten Gang of Four (GoF): Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides [29] bzw. [30]

  Das Standardwerk der GoF habe ich 1998 kennen- und viele der Muster schätzen gelernt. Manche Beschreibungen sind etwas trocken und formal. Die dahinter steckenden Ideen sind jedoch beim Entwurf guter Software sehr hilfreich.

- »**Design Patterns Explained**« von Alan Shalloway und James R. Trott [80]

  Dieses exzellente Buch erklärt die von der GoF beschriebenen Entwurfsmuster auf eine verständlichere Weise anhand von Praxisbeispielen. Es wird zudem ein sehr guter, fundierter Einstieg in den OO-Entwurf gegeben.

- »**Head First Design Patterns**« von Eric Freeman, Elisabeth Freeman, Kathy Sierra und Bert Bates [28]

  Dieses unterhaltsame Buch beschreibt Entwurfsmuster weniger formal und ermöglicht einen guten und unterhaltsamen Einstieg. Die Muster werden sehr verständlich und anschaulich in Form von Text und Bildern dargestellt.

- »**Holub on Patterns**« von Allan Holub [42]

  Dieses Buch betrachtet Entwurfsmuster aus dem Blickwinkel der Praxis. Anhand zweier Beispielapplikationen werden alle GoF-Muster vorgestellt und verdeutlicht. Bei der Vorstellung wird großer Wert auf sauberes objektorientiertes Design gelegt.

- »**Design Patterns Java Workbook**« von Steven J. Metsker [65]

  Dieses Buch beschreibt alle von der GoF vorgestellten Muster und ist so konzipiert, dass jedes Muster durch Übungsaufgaben erlernt und das jeweilige Wissen vertieft werden kann.

Teil V

Qualitätssicherungsmaßnahmen

19 Programmierstil und Coding Conventions

In meiner Tätigkeit als Softwareentwickler hat es sich immer wieder als Vorteil erwiesen, gewisse Konventionen beim Programmieren einzuhalten. In diesem Kapitel beginne ich mit allgemeinen Hinweisen zum Programmierstil in Abschnitt 19.1. Auch das Layout des Sourcecodes kann großen Einfluss auf dessen Verständlichkeit haben. Das wird in Abschnitt 19.2 diskutiert. Im Anschluss daran stellt der nach Themen gegliederte Katalog der Codierungsregeln (Coding Conventions) in Abschnitt 19.3 Leitsätze, Regeln und Tipps vor, die dabei helfen, Fehler provozierenden Sourcecode zu vermeiden und übersichtlichere und verständlichere Programme zu schreiben. Man könnte diese Techniken auch die »Good Smells« nennen – im Gegensatz zu den »Bad Smells« aus Kapitel 16. Werden diese Techniken zur Gewohnheit, so vermeidet dies Probleme und hilft dabei, Fehler leichter zu finden, ohne dafür viel Mühe zu investieren. Man erspart sich dadurch einigen Aufwand, der in unübersichtlichen Programmen betrieben werden muss, um Softwaredefekte im Nachhinein aufzuspüren und zu beheben.

Abschnitt 19.4 geht auf das Thema Sourcecode-Prüfung ein und stellt Bewertungskriterien, sogenannte Metriken, zur Beurteilung der Güte von Sourcecode vor. Anschließend werden verschiedene Tools betrachtet, um die Einhaltung der vorgestellten Programmierrichtlinien zu überprüfen.

19.1 Grundregeln eines guten Programmierstils

Wenn sich die Anforderungen an Programme über die Zeit wandeln, ist es für eine Wartung von großem Vorteil, wenn der Sourcecode übersichtlich und gut lesbar ist. Dies erleichtert das Verständnis. Für den Computer ist das Kriterium der Lesbarkeit allerdings vollkommen unbedeutend, solange das Programm syntaktisch korrekt ist. Diese syntaktische Korrektheit sagt aber noch gar nichts aus. Selbst wenn das Programm auch noch semantisch korrekt ist, kann es immer noch fürchterlich strukturiert sein und damit jegliche Wartung zur Qual werden lassen. Martin Fowler schreibt dazu Folgendes: »Any fool can write code that a computer can understand. Good programmers write code that humans can understand.« Übersetzt: ***»Jeder Dummkopf kann Code schreiben, der für Computer verständlich ist. Gute Programmierer schreiben Code, den Menschen verstehen können.«*** [27].

19.1.1 Keep It Human-Readable

Guter Sourcecode liest sich (zumindest auf der Ebene der öffentlichen Methoden) fast wie ein Roman. Es ist ein klarer Handlungsstrang (Kontrollfluss) erkennbar und jeder Akteur (Klasse, Objekt oder Methode) macht nur das, was seiner Aufgabe entspricht. Die gute Lesbarkeit hat verschiedene Aspekte. Als Basis dienen

aussagekräftige und sprechende Namen für Klassen, Methoden und Variablen. Erklärende Kommentare erleichtern das Verständnis und geben Kontextinformationen. Wichtig ist aber auch das Layout des Sourcecodes. Sind Einrückungen und Klammerungen nicht zu erkennen, so erschwert dies die Lesbarkeit ungemein.

19.1.2 Keep It Simple And Short (KISS)

Halten Sie Ihr Design und den Sourcecode einfach. Denken Sie an Albert Einsteins Spruch: »Everything should be made as simple as possible, but no simpler.« Frei übersetzt: »Alles sollte so einfach wie möglich gemacht werden, aber nicht einfacher.« Man spricht auch vom KISS-Prinzip (Keep It Simple And Short). Versuchen Sie also nicht, andere durch komplizierte Konstrukte zu beeindrucken – denn erfahrungsgemäß sind gerade diese scheinbar besonders cleveren Lösungen oftmals schlecht lesbar, wenig verständlich und führen deswegen später häufig zu einem Wartungsalbtraum.

Je weniger Komplexität in Methoden und Klassen steckt, desto übersichtlicher, test-und wartbarer werden diese. Im besten Fall lassen sie sich in anderen Zusammenhängen wiederverwenden. Wird jedoch zu viel Funktionalität in eine Klasse oder Methode eingebracht, so stört dies die ***Orthogonalität***. Methoden und Klassen lassen sich dann nicht mehr so gut zu neuen Einheiten kombinieren. Dies adressiert auch das Single Responsibility Principle (SRP), das besagt, dass Klassen und Methoden nur eine Zuständigkeit besitzen und somit auch möglichst einfach und kurz sein sollten (vgl. Abschnitt 3.5.3).

19.1.3 Keep It Natural

Bevor Sie mit dem Entwurf Ihrer Software beginnen, sollten Sie die Anforderungen daran recht gut verstanden haben, da ansonsten die Gefahr groß ist, etwas zu entwerfen, was die Bedürfnisse Ihrer Kunden nicht (korrekt) erfüllt. Am besten schreiben Sie wichtige Dinge nieder, weil sich dadurch Ihr Verständnis für das zu lösende Problem verbessert. Das hilft dabei, ein besseres Bild zu gewinnen und eine angemessene und natürliche Lösung zu finden.

Denken Sie beim Design in klaren Strukturen und Zuständigkeiten und diskutieren Sie Ihre Entwürfe mit anderen, wobei UML-Diagramme hilfreich sein können. Verwenden Sie zur Strukturierung immer dann Entwurfsmuster, wenn es von Nutzen ist. Verfallen Sie aber nicht jedem neuen Hype und versuchen Sie nicht, etwas krampfhaft einsetzen zu wollen: Als Mitte der 90er-Jahre Entwurfsmuster bekannt wurden, habe ich Kollegen erlebt, die diese ohne Hinterfragen selbst für einfache Problemstellungen genutzt haben. Ähnliches gilt für Optimierungen: Befassen Sie sich nicht zu früh damit, denn diese führen teilweise zu merkwürdigen

Umsetzungen.

19.1.4 Keep It Clean

Halten Sie unbedingt immer Ordnung in Ihrem Sourcecode. Es gilt das Gesetz der Entropie: Wo etwas Unordnung herrscht, kommt (automatisch) neue hinzu. Man kennt dies auch als »Broken-Windows-Theorie«.

```
public void updateXMLState(final InputStream inStream) throws IfsException,
                                                              IfsException
```

Sie halten dies für ein konstruiertes Beispiel? Nein, weit gefehlt. Das ist tatsächlich Sourcecode aus einem realen Programm. Man fragt sich, wie das passieren konnte. Nachträglich kann man sich das schwer erklären. Durch ein wenig Disziplin, Ordnung und Achtsamkeit können solche Unschönheiten meist schnell aufgedeckt werden.

Der Ratschlag »Keep It Clean« folgt dem DRY-Prinzip (»Don't Repeat Yourself«), das besagt, dass man Duplikation in Sourcecode möglichst vermeiden sollte (vgl. [48]).

19.2 Die Psychologie beim Sourcecode-Layout

Jeder kennt es: Es müssen Änderungen im Sourcecode durchgeführt werden, aber bereits das Verständnis des Sourcecodes bereitet Probleme, da auf den ersten Blick keine klare Struktur erkennbar ist. Vielmehr sieht das Programm nach einer wild zusammengewürfelten Methodensammlung aus.

In der Wahrnehmungspsychologie gibt es für das Erkennen der Zusammengehörigkeit von Elementen verschiedene sogenannte Gestaltgesetze. Zwei für die SourcecodeFormatierung wichtige stelle ich im Folgenden vor.

19.2.1 Gesetz der Ähnlichkeit

Das sogenannte ***Gesetz der Ähnlichkeit*** beschreibt, wie wir Dinge miteinander in Verbindung bringen, wenn sich diese ähneln, z. B. in ihrer Farbe oder Form. Verschiedene Merkmale führen zu unterschiedlicher Gruppierung (vgl. Abbildung 19-1).

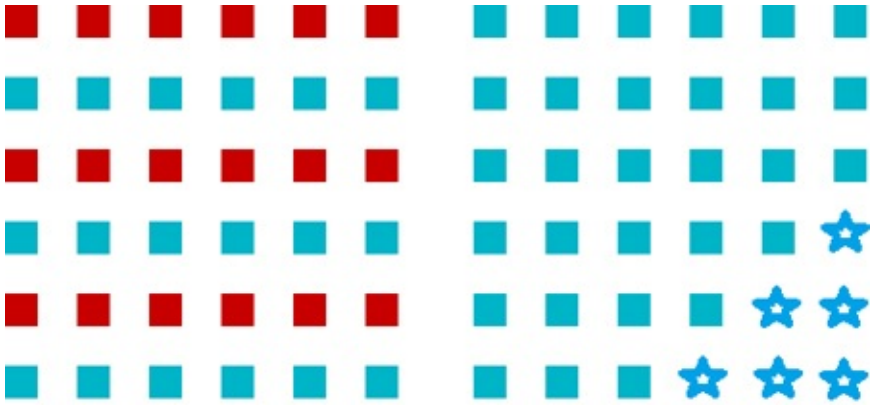


Abbildung 19-1 *Gesetz der Ähnlichkeit*

Auf Sourcecode-Ebene übertragen, kann man den Grad an Ähnlichkeit anhand von Abstraktionsebenen ausdrücken. Es bietet sich an, etwa folgende drei zu unterscheiden:

1. Auf hoher Abstraktionsebene werden lediglich Business-Methoden aufgerufen.
2. Auf mittlerer Abstraktionsebene finden normale Methodenaufrufe statt.
3. Auf niedriger Abstraktionsebene werden einzelne Anweisungen kombiniert.

Was bedeutet das für die Praxis, wenn man in einer Methode verschiedene Abstraktionsgrade findet? Zunächst scheint das nur kosmetischer Natur zu sein – es steht aber ein größeres Problem dahinter: Die unterschiedlichen Abstraktionsebenen erschweren das Verständnis sowie das Erkennen von Ähnlichkeiten und Zusammenhängen.

Beispiel

Schauen wir uns die Problematik konkret am Beispiel der folgenden Methode `paint(Graphics)` an:

```
public void paint(final Graphics graphics)
{
    if (showGrid)
    {
        graphics.setColor(Color.DARK_GRAY);

        // Raster zeichnen
        for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
        {
            for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
            {
                graphics.drawLine(x, y, x, y);
            }
        }

    }

    paintFigures(graphics);
}
```

Diese Methode ist schon recht kurz und übersichtlich. Man kann das Ganze noch

verbessern, indem man (private) Hilfsmethoden – in diesem Fall eine Methode zum Zeichnen des Rasters `paintGrid(Graphics)` – einführt. Damit erreicht man eine bessere Strukturierung des Sourcecodes. Außerdem kann die Implementierung der jeweiligen Methode nun auf einheitlichen Abstraktionsebenen erfolgen, wie nachfolgend gezeigt:

```
public void paint(final Graphics graphics)
{
    if (showGrid)
    {
        paintGrid(graphics);
    }

    paintFigures(graphics);
}

private void paintGrid(final Graphics graphics)
{
    graphics.setColor(Color.DARK_GRAY);

    for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
    {
        for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
        {
            graphics.drawLine(x, y, x, y);
        }
    }
}
```

Die gezeigte Strukturierung erleichtert das Nachvollziehen. Für öffentliche Methoden (`paint(Graphics)`) kann man in Konzepten denken. Für private Methoden (`paintGrid(Graphics)`) bleibt man immer auf Implementierungsebene. Der Vorteil einer einheitlichen Abstraktionsebene innerhalb der Implementierung einer Methode ist, dass nicht ständig gedanklich zwischen den Ebenen gewechselt werden muss. Das kommt dem Lesefluss und der Verständlichkeit zugute.

19.2.2 Gesetz der Nähe

Eine schlechte Sourcecode-Formatierung kann sich negativ auf das Verständnis auswirken. Das sogenannte ***Gesetz der Nähe*** beschreibt, dass eng benachbarte Elemente als Teil eines größeren Ganzen wahrgenommen werden, wie dies Abbildung 19-2 zeigt, wo die acht Linien zu vier Linienpaaren kombiniert werden.

Abbildung 19-2 *Gesetz der Nähe*

Beispiel

Wenn wir semantisch zusammengehörende Elemente gruppieren, so wird der Sourcecode besser lesbar und dessen logische Struktur ist leichter zu erkennen. Als Negativbeispiel dient ein über mehrere Zeilen verstreuter Kommentar, der gemäß dem Gesetz der Nähe nur mit viel Mühe als ein zusammenhängender Kommentar erkennbar ist:

```
private boolean showContent = false;  // Don't

// show
// content
// first

private final Object lockObject = new Object();
```

Die verschiedenen Kommentarzeilen zum Attribut `showContent` sind unübersichtlich und kaum zusammenhängend. Das Gesetz der Nähe lässt uns darin spontan zwei Kommentare sehen und Teile davon fälschlicherweise dem Attribut `lockObject` zuordnen.

Eine Verbesserung erreicht man durch den Einsatz eines einzeiligen Kommentars oder bevorzugt durch eine Javadoc-Kommentierung. Dadurch wird eine klare visuelle Trennung erzielt und auch eine Zuordnung der Kommentare zum Attribut `lockObject` vermieden. Folgendes Listing zeigt dies:

```
/** Don't show content first*/
private boolean showContent = false;  // Don't show content first

private final Object synchronizationObject = new Object();
```

19.3 Coding Conventions

Hinweise zur Gestaltung des Sourcecodes werden durch ***Coding Conventions*** beschrieben. Als Ausgangsbasis für ein eigenes Regelwerk können die Regeln von Oracle[1] oder die (von mir bevorzugten) Regeln[2] von Scott Ambler [1] dienen. In diesem Abschnitt werden diese Basisregeln um weitere »Best Practices« ergänzt, die dabei helfen sollen, gewisse bekannte Fallstricke zu vermeiden.

Vor allem im Team profitiert man von »gelebten« Coding Conventions: Ein nahezu einheitlicher Programmierstil ermöglicht, dass sich jeder Entwickler sofort auch in »fremdem« Sourcecode zurechtfindet, da alles ähnlich und vertraut ist. Persönliche Vorlieben und Eigenarten werden auf ein Minimum reduziert, wodurch auch die Durchführung von Codereviews und Pair Programming erleichtert wird.

Definition: Codereviews und Pair Programming

Unter ***Codereviews*** versteht man Meetings mehrerer Entwickler (in der Regel zumindest der Sourcecode-Produzent selbst sowie ein oder mehrere Begutachter), die einige Sourcecode-Abschnitte kritisch betrachten und auf potenzielle Schwachstellen oder Fehler analysieren. Weitere Informationen liefert Kapitel 21.

Mit ***Pair Programming*** bezeichnet man das Schreiben von Programmen durch zwei Entwickler, die gemeinsam vor einem Rechner sitzen. Der eine ist aktiv und programmiert, der andere hat die Rolle des aktiven Beobachters und Reviewers. Er beobachtet, denkt mit, stellt Fragen und gibt Kommentare ab oder bringt Verbesserungsvorschläge ein, wodurch sich sukzessive ein gemeinsames Verständnis entwickelt. Nach einer gewissen Zeit wechselt dann die jeweilige Aufgabe.

Eine große Akzeptanz der Coding Conventions erreicht man, indem man vor der Einführung möglichst alle Entwickler an Entwürfen des neuen Regelwerks mitwirken lässt. Nichtsdestotrotz erfordert die Einhaltung der Regeln anfangs immer etwas Disziplin. Es sollte zudem eine dynamische Entwicklung des Regelwerks vollzogen werden, um dessen Akzeptanz weiter zu erhöhen. Bisher nicht adressierte Programmierprobleme können als Regel neu aufgenommen werden. Eher hinderliche oder überkritische Regeln können nach gründlicher Prüfung und sinnvoller Begründung – nicht nur aufgrund des persönlichen Missfallens einzelner Entwickler – aus dem Regelwerk entfernt werden. Bevor man allerdings vorschnell eine Regel entfernt, kann man besser in einigen Ausnahmesituationen gegen einzelne Regeln verstoßen: ***Nicht jede Regel ist gleichermaßen gut auf jede Situation anzuwenden, sodass es in Einzelfällen durchaus sinnvoll ist, gegen diese zu verstoßen. Allerdings sollte dies gut begründet und dokumentiert werden***.

Meiner Erfahrung nach ist es sehr hilfreich, die Coding Conventions mit einem Sourcecode-Checker zu überprüfen. Wird die Einhaltung der Regeln nur über Codereviews geprüft, so sind meistens diejenigen die »Buhmänner« und Kritiker, die das Regelwerk am besten kennen. Weiterhin kann es zum »Schwarzen Peter«-Syndrom kommen: Der Reviewte fühlt sich persönlich angegriffen und die Reviewer mögen kaum noch weitere Kritik äußern. Als Folge werden dann aber nicht mehr alle Softwaredefekte und Regelverstöße angemerkt. Somit werden Sinn und Nutzen des durchgeführten Codereviews fraglich. Diesem Dilemma kann man durch den Einsatz von Tools zur Sourcecode-Prüfung entgegenwirken. Diese übernehmen die Rolle des Kritikers, jedoch ohne emotionale und persönliche Probleme zu verursachen. Diese sozialen Aspekte sollte man im Hinterkopf behalten, wenn man Coding Conventions einführen und ergänzend dazu Codereviews durchführen möchte. Das Buch »Soft

Skills für Softwareentwickler« von Uwe Vigenschow und Björn Schneider [89] enthält weitere Informationen zu psychologischen Aspekten der Arbeit in Teams.

19.3.1 Grundlegende Namens- und Formatierungsregeln

Es ist sinnvoll, die bereits erwähnten Coding Conventions bezüglich Formatierung und Namensgebung als Basis für eigene Regeln zu verwenden. Als wesentliche Punkte für eine gute Übersicht und Lesbarkeit möchte ich folgende zwei Punkte besonders hervorheben, die in diesem Buch genutzt werden:

1. Klammern und Anweisungen sollten in jeweils eigenen Zeilen stehen, pro Zeile also möglichst nur eine Anweisung bzw. eine Variablendeklaration.
2. Zeilen sollten eine gewisse Länge nicht überschreiten. Ein Wert zwischen 100 und 130 Zeichen hat sich als praktikabel erwiesen.

Das folgende Beispiel zeigt sowohl die Anwendung der Namenskonventionen als auch der Formatierungsregeln:

```
public final class FormatingExample
{
    private static final Logger log = Logger.getLogger("FormatingExample");

    public static String asHex(final byte[] tele)
    {
        log.info("asHex(" + Arrays.toString(tele) + ")");

        final StringBuffer sb = new StringBuffer("0x");

        for (int i = 0; i < tele.length; i++)
        {
            final String hex = Integer.toHexString(tele[i]);
            sb.append(hex);
        }

        return sb.toString();
    }
    // …
}
```

Ausnahmen für die Formatierung

Nur in wenigen Fällen sind Abweichungen von der Standardformatierung sinnvoll. Dies gilt insbesondere für die Forderung »ein Statement pro Zeile«. Gegen diese Regel kann man beispielsweise dann verstoßen, wenn nur eine Menge einfacher `get()`-Methoden angeboten werden. Wenn die Methodenrümpfe – wie nachfolgend gezeigt – hinter den Methodennamen geschrieben werden, lässt sich die Lesbarkeit erhöhen.

```
public final String getName()        { return name; }
```

```
public final String getCity()       { return city; }
public final int    getAge()        { return age; }
```

Wie man leicht sieht, ist diese Schreibweise kurz und elegant und fokussiert die Aufmerksamkeit auf die Schnittstelle und nicht die Implementierung, die sowieso trivial ist. Die Lesbarkeit des Sourcecodes wird positiv beeinflusst.

Namensregeln

Eine konsistente Namensgebung von Variablen, Methoden und Klassen hilft, auf einen Blick diese Elemente erkennen und voneinander unterscheiden zu können. Tabelle 19-1 liefert Anregungen. Für Namen gilt in der Regel die sogenannte ***CamelCaseSchreibweise***, d. h., jedes neue Wort wird wieder mit einem Großbuchstaben begonnen.

Tabelle 19-1 *Namenskonventionen*

Typ	Präfix	Postfix	Beispiel
Packages	-	-	`mypackage.subpackage`
Interfaces	`I`	`IF`	`IFileInfo`, `FileInfoIF`
Abstrakte Klassen	`Base`, `Abstract`	-	`AbstractProcessStep`
Klassen	-	-	`JobAssistant`
Testklassen	-	`Test`	`ProcessStepTest`
Methoden	-	-	`getFileName()`
Konstanten	-	-	`MAX_VALUE`
Lokale Variablen	-	-	`imageWidth`
Membervariablen	-, `m_`, `this.`	-	`name`, `m_name`, `this.name`

Für die obigen Beispiele ist manchmal sowohl ein Präfix als auch ein Postfix angegeben. Über den Einsatz für Interfaces und Membervariablen gibt es kontroverse Meinungen mit guten Argumenten dafür als auch dagegen – ansatzweise wird dies im folgenden Praxistipp beleuchtet.

Für Interfaces existiert mit dem Postfix `IF` eine alternative Notationsform zum Präfix `I`. Testklassen sollten mit dem Postfix `Test` enden und nicht damit starten, weil so die Testklasse leichter zu einer bestehenden Klasse gefunden werden kann. Wie bereits in Abschnitt 2.4.1 angedeutet, würden durch Einsatz von `Test` als Präfix alle Testklassen damit beginnen und man fände den »Wald vor lauter Bäumen nicht.«

Hinweis: Stilfragen

Position der geschweiften Klammern Die Position der öffnenden, geschweiften Klammer löst häufig kontroverse Diskussionen aus. Mir helfen Klammern in einer eigenen Zeile, Fehler zu vermeiden und die Struktur besser zu erkennen.

Präfixe für Attribute Die Diskussion um Präfixe für Attribute ist wahrscheinlich genauso endlos zu führen wie die um die richtige Position der öffnenden Klammer. Allerdings können Präfixe manchmal helfen, Fehler zu vermeiden und die Übersicht zu erhöhen. Dies gilt vor allem bei der Verwendung von Übergabeparametern, die wie Attribute heißen. Folgende Methode `setCounter( int)` zeigt diesen Fehler:

```
public void setCounter( int count)
{
    count = count;    // Achtung: Sinnlose Selbstzuweisung!
}
```

Hier findet nicht die gewünschte Änderung im Objektzustand statt, da die Variable an sich selbst zugewiesen wird. Als Abhilfe existieren folgende Möglichkeiten:

1. ***Aktivierung entsprechender Style-Checks in der IDE, sodass solche Zuweisungen angemerkt werden.***
2. Den Übergabeparameter `final` deklarieren, wodurch ein Kompilierfehler bei einer solchen versehentlichen Zuweisung erzeugt wird.
3. Die Angabe von `this.` verhindert die Zuweisung und man erreicht eine bessere visuelle Trennung von Attributen, lokalen Variablen und Parametern.

Alle Abhilfen lassen sich kombinieren. Nachfolgend sind Abhilfen 2 und 3 eingesetzt:

```
public void setCounter( final int count)
{
    this. m_count = count;
}
```

Typpräfixe für Attribute Das Präfix `m_` für Attribute führt zu einer starken visuellen Trennung – geht aber mit einer etwas schlechteren Lesbarkeit einher. Manchmal findet man zusätzliche Präfixe, etwa `str` für Strings, `is` für boolesche Variablen. Auch hier scheiden sich wieder die Geister. *Auf jeden Fall sollte dies mit Bedacht genutzt werden*. In der Regel sollte nicht der Typ einer Variablen im Namen stecken, sondern bevorzugt ihre semantische Bedeutung.

Verstoß der Empfehlungen gegen Oracle-Regeln Meine Empfehlungen widersprechen zwar einigen Regeln von Oracle. Ein Blick in den Sourcecode des

JDKs zeigt aber, dass die dort genutzten recht kurzen Zeilenlängen und die Formatierung nicht immer für gute Lesbarkeit sorgen. Längere Zeilen und Klammern in eigenständigen Zeilen tragen meiner Ansicht nach aber stark zur besseren Lesbarkeit bei. Schlussendlich ist das Ganze aber ein kontrovers diskutiertes Thema.

19.3.2 Namensgebung

Namensregeln lassen sich nur bedingt durch Tools prüfen. Zwar können diese gerade noch eine regelkonforme Schreibweise testen, aber für ein Tool ist der Name `map` oder `list` genauso gut oder schlecht wie die verständlichen Namen `nameToPersonMap` oder `deviceList`. Daher ist es sehr sinnvoll, hier ein wenig Selbstdisziplin walten zu lassen. Auch Pair Programming und Codereviews bieten eine gute Hilfe beim Finden von sinnvollen Namen.

Vermeide Namenskürzel

Aussagekräftige Namen machen ein Programm besser lesbar. Abkürzungen oder Namenskürzel sind zu vermeiden, wenn es sich nicht gerade um die gebräuchlichen Abkürzungen `sb` für `StringBuffer` oder `it` für Objekte vom Typ `Iterator` handelt. Im Extremfall verkümmert ein Variablenname zu einer kryptischen Zusammenstellung aus den Anfangsbuchstaben der Teilworte. Für gängige Abkürzungen (HTML, DB, XML) kann auch dies vernünftig sein. In vielen Fällen leidet aber die Lesbarkeit und Verständlichkeit ungemein. Als »Leckerbissen« hat mir ein Kollege die Variable `AAA` zugespielt … nein, es hat nichts mit Batterien zu tun! Dies ist eine misslungene Abkürzung für eine booleschen Variable mit der Bedeutung »**A**lle **A**nschlüsse **A**nzeigen«.

Schauen wir auf ein weiteres Negativbeispiel, das Kommentare nutzt, um den ansonsten unleserlichen Sourcecode zu beschreiben:

```
// enthält den Maximalwert
int val = -1;

// Durchlaufe alle vorhandenen Tabellenzeilen
for (int i = 0; i < 50; i++)
{
    val = Math.max(val, values[i].getValue());
}
```

Die beiden Kommentare sind trivial, kaum hilfreich und sogar ein wenig irreführend. Natürlich vermutet man, dass es sich bei `i` um die Zeile handelt und dass `val` den Maximalwert ermitteln soll, aber warum schreibt man es dann nicht gleich wie folgt?

```
final int PERSON_TABLE_ROW_COUNT = 50;

int maxAge = -1;
for (int rowIndex = 0; rowIndex < PERSON_TABLE_ROW_COUNT; rowIndex++)
{
```

```
    maxAge = Math.max( maxAge, persons[ rowIndex].getAge());
}
```

Durch die sinnvolle Benennung der Variablen (z. B. `maxAge` statt `val`, `persons` statt `values`) wird dann nicht nur der Algorithmus (Berechnung des Maximums), sondern auch der dahinter liegende Sinn (maximales Alter der aufgelisteten Personen ermitteln), offensichtlich. Gut gewählte Namen sorgen also nicht nur für Verständlichkeit, sondern erlauben es auch, auf eine »Pseudokommentierung« zu verzichten.

Vermeide Variablennamen, die nur den Typ wiederholen

Manchmal findet man Sourcecode, in dem Variablennamen aus deren Typen hergeleitet werden. Derartige Variablennamen sind oftmals wenig verständlich, da sie keinen Hinweis auf den Einsatzzweck enthalten. Betrachten wir folgendes Beispiel:

```
final Vector<File> vector = new Vector<>();

final List<File> list = new ArrayList<>();
final List<File> list1 = new ArrayList<>();
final List<File> list2 = new ArrayList<>();
```

Im Listing erkennt man, dass es für einzelne temporäre Variablen zum Teil schwierig ist, einen sinnvollen Namen zu finden – eine (reine) Nutzung des Typs ist dann tolerierbar. Es gibt aber Ausnahmen: Problematisch wird dies etwa, wenn mehrere Objekte gleichen Typs verwendet werden: Zur Unterscheidung muss man die Objekte irgendwie kennzeichnen. Meistens werden diese dann einfach durchnummeriert. Dabei stellt sich die Frage nach der Namenskonsistenz: Wird das erste Objekt mit einer »`0`« oder einer »`1`« gekennzeichnet oder erhält der Name, wie im obigen Beispiel, keinen Zusatz?

Verwende sinnvolle, konsistente Namen

Wie bereits angemerkt, helfen sinnvoll gewählte Namen, ein Programm nachvollziehbar zu machen, lassen sich aber leider nicht durch Tools prüfen. Für die Lesbarkeit und die Semantik ist man als Entwickler demnach selbst verantwortlich. Helfen Sie sich und Ihren Kollegen durch passend gewählte Namen.

Greifen wir das obige Beispiel wieder auf. Nehmen wir an, die drei Listen würden zur Modellierung von Veränderungen des Inhalts eines Verzeichnisses genutzt. Folgende Namensgebung macht diesen Sachverhalt intuitiv klar:

```
final List<File> newFiles = new ArrayList<>();
final List<File> changedFiles = new ArrayList<>();
final List<File> removedFiles = new ArrayList<>();
```

Eine Verwechselung von `list1` und `list2` aus dem vorherigen Beispiel ist leicht möglich. Bei der Namensgebung `changedFiles` und `removedFiles` ist eine Verwechselung nahezu ausgeschlossen.

Nicht immer ist ein Problem auf den ersten Blick zu erkennen, wenn man nicht direkt die Definition der Variablen und des zugehörigen Typs sieht:

```
final PersonService personDAO = ServiceFactory.getPersonService();
```

Betrachtet man den verständlichen Variablennamen `personDAO`, so könnte man meinen, ein Data Access Object (DAO) zu referenzieren. Erst ein zweiter, genauerer Blick auf die Definitionsstelle offenbart die Inkonsistenz zwischen Name und Inhalt (Referenz auf einen Service). Wir erkennen daran, dass die konsequente Benennung von Variablen ein weiterer wichtiger Schritt in Richtung Nachvollziehbarkeit ist. Dazu gehört auch, einen Zähler namens `counter` nicht an anderer Stelle `cnt` zu nennen.

Benenne Klassen nach ihrer Funktion

Klassen sollten nach ihrer Funktion benannt werden und nicht nach dem Package, in dem sie gespeichert sind. Es gibt Projekte, in denen einige Entwickler dem Klassennamen den Package-Namen voranstellen. Das führt zu seltsamen Klassennamen:

```
JC_JOB
TEDI_Figure
PIDMessageHandler
```

Wird eine Klasse mit Package-Präfix in ein anderes Package verschoben, so müsste als Folge konsequenterweise der Klassenname angepasst werden – unterbleibt dies, so kommt es zu Inkonsistenzen. Wird eine Klasse von vielen weiteren Klassen referenziert, so müssen überall dort die Namensänderungen nachgezogen werden. Zum Glück geschieht dies meistens automatisch durch die IDE. Das führt wiederum zu vielen geänderten Dateien und erhöht die Gefahr für Konflikte beim Einchecken ins Repository.

Verwende für Containerklassen einen Pluralnamen

Nutzt man Pluralnamen für Collections oder Arrays, erhöht dies die Lesbarkeit und erleichtert die Unterscheidung von normalen Referenzen:

```
// Einzelelemente
final GraphicObject graphicObject = new GraphicObject();
final String predefinedName = "Name";

// Collections
final List<GraphicObject> graphicObjects = new ArrayList<>();
final String[] predefinedNames = new String[] { "Name 1", …, "Name n" };
```

Bei Containern kann der Typ der Containerklasse im Namen wiederholt werden und als Kontextinformation über die verwendete Datenstruktur (z. B. `List<E>` oder `Map<K, V>`) dienen. Dadurch verringert sich jedoch teilweise die Lesbarkeit:

```
final List<GraphicObject> listOfGraphicObjects = new ArrayList<>();
final List<GraphicObject> graphicObjectsList = new ArrayList<>();
```

```
final String[] predefinedNamesArray = new String[] { "Name 1", …, "Name n" };
```

Wie bereits zuvor beschrieben, sollte man vermeiden, lediglich den Typ der Collection ohne weitere Beschreibung als Namen zu verwenden, etwa `map` oder `list`. Damit würde nur die Deklaration wiederholt, ohne den Einsatzzweck zu beschreiben.

Hilfsvariablen können die Lesbarkeit erhöhen

Manchmal kann man durch die Definition einer Hilfsvariablen mit sprechendem Namen den Sourcecode besser verständlich gestalten. Betrachten wir folgendes Beispiel:

```
private final TreeSet<String> timeStampSet = new TreeSet<>(Collections.
                                                    reverseOrder());
```

Wenn wir diese Zeile anschauen, erkennen wir, dass eine umgekehrte Sortierreihenfolge realisiert wird. Das ist recht klar, aber was bedeutet das konkret? Versuchen wir den Sinn durch eine Hilfsvariable zu verdeutlichen und sehen wir uns an, wie gut der folgende Sourcecode dies kommuniziert:

```
private final Comparator<String> mostCurrentFirst =
Collections.reverseOrder();
private final TreeSet<String> timeStampSet = new TreeSet<>(mostCurrentFirst);
```

Hinweis: Vorteil sprechender Namen

Achten Sie darauf, möglichst gut lesbaren Sourcecode zu schreiben. Der Grund ist einfach folgender: Sie werden Sourcecode z. B. im Rahmen von Erweiterungen oder Bugfixes viel häufiger lesen und verstehen müssen, als Dinge neu zu schreiben.

19.3.3 Dokumentation

Verwende kurze, aussagekräftige Kommentare

Ist ein Abschnitt des Sourcecodes trotz guter Formatierung und sinnvoller Namensgebung von Variablen und Methoden noch nicht ausreichend verständlich, so sollten kurze Kommentare als Überschrift eingefügt werden.

Vermeide Kommentare, die nur den Ablauf beschreiben

Manchmal wird die vorherige Regel falsch ausgelegt und es werden selbst *einfache* Programmstellen kommentiert: Man findet dann häufig Beschreibungen zum Ablauf bzw. zur Ausführung des Sourcecodes, nicht aber zu dessen Ziel. Das haben wir schon im Beispiel der Namensgebung gesehen und wird hier nochmals aufgegriffen:

```
// SCHLECHTER KOMMENTAR: 365 Tage durchlaufen
```

```
for (int tag = 0; tag < 365; tag++)
{
    gesamtUmsatz += tagesUmsatz[tag];
}
```

Es ist es viel sinnvoller, zu beschreiben, warum hier eine zwar durch die Variablennamen naheliegende, aber zunächst willkürliche Grenze von 365 Durchläufen verwendet wird, statt nur die offensichtliche Abbruchbedingung der Schleife zu kommentieren. Eine Verbesserung des Kommentars und der Implementierung, die auch Schaltjahre korrekt behandeln kann, könnte so aussehen – der Kommentar ist jetzt nahezu überflüssig:

```
// KORREKTUR: Gesamtumsatz aus den Tagesumsätzen eines Jahres berechnen
final int tageImJahr = tagesUmsatz.length;
for (int tag = 0; tag < tageImJahr; tag++)
{
    gesamtUmsatz += tagesUmsatz[tag];
}
```

Dokumentiere alle Methoden im öffentlichen Interface der Klasse

Im Sinne der Wiederverwendbarkeit ist es vorteilhaft, Klassen, Methoden sowie vor allem Interfaces möglichst gut zu dokumentieren. Der Einsatz des JavadocKommentarstils für alle im öffentlichen Interface von Klassen sichtbaren Methoden (`public`, `protected`) bietet sich an. Für interne Methoden kann in vielen Fällen auf eine Kommentierung verzichtet werden, wenn die Methoden entsprechend kurz sind, einen aussagekräftigen Namen haben und nur wenige Übergabeparameter besitzen.

Ein Kommentar sollte zumindest den Zweck einer Methode, die Übergabeparameter sowie den Rückgabewert beschreiben. ***Die Rückgabe von `null` sollte ausdrücklich dokumentiert werden, wenn dies ein gewollter und möglicher Rückgabewert ist***. Dadurch schützt man Aufrufer vor einer sonst nicht erwarteten `NullPointerException` beim Zugriff auf den Rückgabewert. Ebenso sollte man auf unangenehme Seiteneffekte hinweisen, falls sich diese nicht anderweitig vermeiden lassen.

```
/**
* copies all contents from the passed input stream into the given output
* stream using buffered copy.
*
* @param is    source input stream
* @param os    destination output stream
*
* @return the amount of bytes copied
*
* @throws java.lang.NullPointerException if parameters are <code>null</code>
* @throws java.io.IOException if access to streams fails
*
* @see java.io.InputStream, java.io.OutputStream,
*      java.lang.NullPointerException, java.io.IOException
*/
```

```
public static int copyBuffered(final InputStream is, final OutputStream os)
      throws IOException
// …
```

Tipp: Dokumentation mit Javadoc

Die Erfahrung zeigt, dass eine vom Sourcecode unabhängige Dokumentation sehr schnell veraltet und außerdem häufig nicht die für Entwickler wichtigen Informationen enthält. Die Idee hinter Javadoc, dem Dokumentationswerkzeug des JDKs, ist es, aus Kommentaren im Java-Sourcecode automatisch HTML-Dokumentationen erstellen zu können. Aufgrund der Nähe zur Entwicklung besteht außerdem die Hoffnung, mit Javadoc eine aktuelle und konsistente Dokumentation für Klassen, Methoden usw. erzeugen zu können.

Der Sourcecode wird bei der Aufbereitung der Dokumentation durch das Javadoc-Tool nach Kommentaren im Javadoc-Stil (startend mit `/**` und abgeschlossen mit `*/`) durchsucht. Alle Informationen der dort enthaltenen Javadoc-Tags (beginnend mit `@`) werden für die Ausgabe in HTML ausgewertet. Eine Übersicht gebräuchlicher Javadoc-Tags liefert Tabelle 19-2. Weitere Informationen finden Sie unter
`http://docs.oracle.com/javase/8/docs/technotes/tools/windows/`

Mit Eclipse ist es möglich, über das Menü PROJECT –> GENERATE JAVADOC eine Javadoc-Generierung durchzuführen. Noch einfacher ist es, diesen Schritt durch einen Aufruf im Build-Prozess zu automatisieren (vgl. Abschnitt 2.7.3).

Tabelle 19-2 *Gebräuchliche Javadoc-Tags*

Javadoc-Tag & Wert	Zweck
`@author <name>`	Beschreibt den Autor.
`@version <versionInfo>`	Erzeugt einen Versionseintrag.
`@param <name> <description>`	Parameterbeschreibung einer Methode.
`@return <description>`	Beschreibung des Rückgabewerts einer Methode.
`@throws <exception> <description>`	Beschreibung einer Exception, die von dieser Methode geworfen werden kann.
`@deprecated <description>`	Markiert eine veraltete Methode, die nicht mehr eingesetzt werden sollte. Zusätzlich sollte immer die `@Deprecated`-Annotation verwendet werden.
`@inheritDoc`	Kopiert die Beschreibung aus der überschriebenen Methode. Erspart damit das früher notwendige Copy-Paste für alle Kommentare aus Interfaces.
`@see reference`	Erzeugt einen Verweis auf ein anderes Element der Dokumentation.
`@since <version>`	Gibt an, ab welcher Version eine bestimmte Funktion enthalten ist. Das ist insbesondere bei Bibliotheken hilfreich.

19.3.4 Programmdesign

Halte den Sourcecode sauber

Unbenutzte Programmteile und auskommentierte Blöcke blähen den Sourcecode nur unnütz auf und machen ein Programm schlechter lesbar. Diese »Vorratsdatenspeicherung« von ehemals sinnvollen Zeilen ist durch den Einsatz einer Versionsverwaltung überflüssig. Allerspätestens nach ein paar Tagen sollten solche Programmteile gesäubert werden. Wird dieser »Frühjahrsputz« immer wieder

verschoben, können Sie im Nachhinein über Kommentare wie den folgenden schmunzeln: »`Quickfix. Muss nach der Auslieferung unbedingt korrigiert werden. 7.7.2001`«. Heutzutage würde man solche Stellen mit einem Kommentar `TODO` oder `FIXME` versehen, wodurch automatisch eine Markierung und eine Aufnahme in eine Task-Liste in der IDE erfolgt.

Vermeide Seiteneffekte

Seiteneffekte sind neben Race Conditions bei Multithreading unangenehm und können einem als Programmierer die Laune verderben. Ich habe einige Methoden kennenlernen müssen, die als `get()`-Methode »getarnt« Schreibzugriffe auf andere Variablen durchgeführt haben. Ein absolut extremes Beispiel ist mir in Form von Datenbank-Updates in einer `get()`-Methode begegnet. So etwas sollte man unbedingt vermeiden, damit das Programm dem »Prinzip des geringsten Erstaunens« (»Principle of Least Astonishment« (POLA)) folgt.

Vermeide übermäßige Debug- und Log-Ausgaben

Logging kann dabei helfen, Fehler zu ermitteln, wenn keine Möglichkeit des Debuggings besteht oder ein Fehler im Nachhinein analysiert werden muss. Zum Teil sieht man jedoch Sourcecode, der nur aus der Aufbereitung von Debug- und Log-Ausgaben zu bestehen scheint. Die eigentliche Programmlogik tritt in den Hintergrund. Das ist häufig ein Zeichen dafür, dass das zu lösende Problem nicht verstanden wurde. Solcher Sourcecode sollte überarbeitet werden, um diesen Ballast loszuwerden.

Vermeide Ausgaben per `System.out`

Ausgaben mit `System.out` können für einfache Anwendungen und zum schnellen Testen durchaus eingesetzt werden. Für professionelle Programme ist dies aber wenig hilfreich, da keine Konfiguration oder Persistierung der Ausgabe erfolgen kann. Man benutzt daher besser ein Logging-Framework (vgl. Abschnitt 6.4).

Verwende `final`

Über das Schlüsselwort `final` können wir steuern, ob der Wert einer Variablen konstant ist oder ob er nach der ersten Zuweisung veränderbar ist. In vielen Fällen können wir Variablen `final` machen und uns dadurch effektiv davor schützen, dass diese für andere Dinge zweckentfremdet werden. Je weniger Zustandsinformationen in einem Objekt veränderlich sind, desto übersichtlicher und wartbarer wird der Sourcecode.

Vermeide Magic Numbers und Strings

Die Verwendung fest codierter Zahlenwerte, Texte und boolescher Werte im Sourcecode erschwert sowohl dessen Lesbarkeit als auch Refactorings. Die Bedeutung eines Zahlenwerts lässt sich meistens sinnvoller durch eine benannte Konstante ausdrücken und erhöht gleichzeitig die Verständlichkeit und die Lesbarkeit. Außerdem wird eine spätere Wartung erleichtert, falls ein solcher Wert einmal geändert werden muss. Für Strings und boolesche Werte gilt Ähnliches.

Bevorzuge wenige `return`-Anweisungen

Umfangreiche Methoden mit vielen `return`-Anweisungen sind oftmals fehlerträchtig und schwierig nachzuvollziehen. Eine Methode mit nur einem definierten Ausgangspunkt (oder wenigen) hilft, den Ablauf besser verfolgen zu können. Diese Empfehlung gilt allerdings nur dann, wenn die Methode länger als etwa eine Bildschirmseite ist, weil dann mehrere `return`-Anweisungen ein Problem darstellen können. Denn je umfangreicher eine Methode wird, desto mehr erschweren viele `return`-Anweisungen das Herausfaktorieren von Methoden und das Nachvollziehen des Programmablaufs. Beachten Sie bitte folgende Ausnahmen: Insbesondere Shortcut-Returns am Methodenanfang für mögliche Fehlerabfragen sind fast immer eine Vereinfachung. Auch gilt: Sind Methoden kurz und übersichtlich, so ist es durchaus sinnvoll, mehrere `return`-Anweisungen zu verwenden, da eine Methode dann häufig ohne zusätzliche Hilfsvariablen und Verschachtelungen auskommt. Für diese Fälle ist es sogar schlimmer, krampfhaft zu versuchen, mehrere `return`-Statements zu vermeiden, weil dann künstliche Hilfsvariablen eingeführt werden (müssen).

Prüfe Eingabeparameter auf Gültigkeit

Alle eingehenden Parameter aller öffentlichen Methoden sollten auf Gültigkeit geprüft werden, um fehlerhafte Werte zu erkennen und zurückzuweisen. Interne Methoden können sich somit auf einen gültigen Objektzustand verlassen und müssen nicht zwingend weitere Parametertests durchführen. Eine Parameterprüfung soll also verhindern, dass ungültige Parameterwerte an andere Methoden weitergegeben werden und dort Schaden anrichten. Dies folgt den Gedanken von »***Design by Contract***« (vgl. gleichnamigen Praxistipp in Abschnitt 3.1.5). Dadurch werden falsch initialisierte Objekte vermieden.

Verwende Assertions oder Exceptions zur Absicherung für Pre- und Post-Conditions

Teilweise müssen gewisse Annahmen über den Zustand des Programms bzw. die Wertebelegung von Variablen getroffen werden. Zur Auswertung kann man das

Schlüsselwort `assert` nutzen, manuell die gewünschte Bedingung als booleschen Ausdruck prüfen und gegebenenfalls bei Verstoß eine Exception auslösen oder aber die Prüfung mithilfe von Google Guavas `Preconditions` (vgl. Abschnitt 6.2.4) erledigen. Bei Problemen im Programmablauf und Inkonsistenzen im Zustand wird ein etwaiger Denkfehler sofort sichtbar. Bedenken Sie aber, dass die Auswertung von Assertions mit `assert` standardmäßig ausgeschaltet ist und explizit aktiviert werden muss (vgl. Abschnitt 4.7.5), weshalb Exceptions zur Absicherung zu bevorzugen sind.

Vermeide die sorglose Rückgabe von `null`

Die Rückgabe von `null` zwingt Aufrufer, Sonderbehandlungen durchzuführen. Wird dies vergessen, so kommt es zu einer `NullPointerException`. Ist `null` jedoch als gültige Rückgabe vorgesehen, sollte dies ausdrücklich in der Methodendokumentation erwähnt werden. Häufig ist es möglich, das NULL-OBJEKT-Muster (vgl. Abschnitt 18.3.2) anzuwenden, statt `null` zurückzuliefern.

Vermeide `null` wenn möglich, prüfe Referenzen auf `null`

Innerhalb des eigenen Programms sollte man möglichst sparsam `null` nutzen, sondern bevorzugt gültige Referenzen. Befolgt man diesen Hinweis nicht, dann sieht man überall im Programm `null`-Prüfungen, ob sinnvoll oder auch nicht. Fakt ist jedoch, dass man Sourcecode dadurch nur aufbläht und ihn unleserlich macht. Zudem verschleiert diese übervorsichtig Prüfung auch Fehler, die besser während der Implementierungsphase des Programms aufgetreten wären. Es empfiehlt sich, mithilfe statischer Sourcecode-Analyse einige Verstöße aufdecken zu lassen. Abschnitt 19.4 stellt einige Tools dazu vor.

Für den (hoffentlich) seltenen Fall, dass eine Referenz einen `null`-Wert enthalten kann, sollte man vor einem Zugriff darauf prüfen. Eine Applikation sollte nicht mit einer unerwarteten und unbehandelten `NullPointerException` abstürzen. In solchen Situationen arbeitet man besser fehlertolerant, gibt aber für den unerwarteten `null`-Fall eine Warnmeldung in eine Log-Datei aus.

Prüfe Rückgabewerte und behandle Fehlersituationen

Rückgabewerte von Methoden sollten vom Methodenaufrufer ausgewertet werden, damit angemessen auf eine mögliche Fehlersituation reagiert werden kann. Geschieht dies nicht, so werden Fehler verschleiert oder sie machen sich entweder gar nicht oder nur durch merkwürdiges Programmverhalten bemerkbar. Im UI wird beispielsweise eine Erfolgsmeldung ausgegeben, obwohl eine Aktion nicht vollständig durchgeführt werden konnte, etwa aufgrund eines Problems bei einem Datenbank- oder Remote-Zugriff.

Tipp: Sorglose Reaktion auf Fehler

Fehler durch den sorglosen Umgang mit Rückgabewerten haben mich einiges an Arbeit gekostet. Teilweise ist es viel schwieriger, solche Fehler zu lokalisieren, als diese zu beheben. Oftmals hätte zumindest ein Hinweis in einer Log-Ausgabe eine Fehlersuche (enorm) verkürzen können.

Wenn man – aus welchen Gründen auch immer – momentan nicht in der Lage ist, eine angemessene Behandlung für eine Fehlersituation oder einen unerwarteten Rückgabewert (etwa `null`) in den Sourcecode einzubauen, muss wenigstens im Log-Level `warn` ein Hinweis in eine Log-Datei geschrieben werden. Ansonsten kommt es zu dem sogenannten »***Silent Fail***« oder »verschluckten Fehlern«, die extrem schwer zu lokalisieren sind: Im Normalfall läuft das Programm einwandfrei, aber im Fehlerfall erhält man keinen Hinweis auf ein Problem.

Behandle auftretende Exceptions wenn möglich

Exceptions zu ignorieren, ist problematisch: Wenn der Programmcode etwas Sinnvolles zur Fehlerbehandlung beitragen kann, dann sollte er das auch tun. Wenn dem nicht so ist, sollte eine Exception weiter propagiert werden, um anderen Programmteilen zu ermöglichen, eine Fehlersituation adäquat zu behandeln.

Hilfreich ist es, auftretende Exceptions in einer Log-Datei zu protokollieren. Nur so kann später nachvollzogen werden, dass ein Fehler im Programm vorlag. Werden zusätzliche Kontextinformationen, etwa der Stacktrace, die Belegung lokaler Variablen und von Attributen, ausgegeben, so erleichtert dies häufig eine spätere Fehlersuche.

Vermeide `catch( Exception ex)` und `catch ( Throwable th)`

Exceptions möglichst spezifisch abzufangen, hilft dabei, diese sinnvoll behandeln zu können. Anhand des Typs einer auftretenden Exception sollte dann eine passende Fehlerbehandlung erfolgen. Werden dagegen Exceptions vollkommen unspezifisch mit `catch ( Exception ex)` oder `catch ( Throwable th)` abgefangen, führt dies dazu, dass alle möglichen Fehlersituationen mit diesem `catch`-Block bearbeitet werden. Treten unerwartet weitere Exceptions auf, werden diese dann so behandelt wie erwartete Exceptions. Unerwartete Exceptions sollten allerdings besser an andere Programmteile weiter propagiert werden, um sie dort in spezifischen `catch`-Blöcken angemessen behandeln zu können.

Im Extremfall werden auch `RuntimeExceptions` und `Errors` abgefangen. Diese stellen jedoch schwerwiegende Fehler dar, die man als Programmierer in der Regel nicht in einem `catch`-Block behandeln sollte.

Vermeide `return` in `catch`/`finally`-Blöcken

Manchmal kann ein `return` sinnvoll in `catch`-Blöcken eingesetzt werden. Allerdings besteht die Gefahr, dass beim Hinzufügen eines `finally`-Blocks der zurückgelieferte Wert verändert wird. Daher sollte auch `finally` nicht mit `return` beendet werden. Folgendes konstruierte Programm zeigt dies:

```
public final class TestReturnInCatch
{
    public static String test()
    {
        try
        {
            throw new IllegalStateException( "Exception simulieren! ");
        }
        catch ( final IllegalStateException ex)
        {
            return "Im Fehlerfall";
        }
        finally
        {
            return "Tatsächlich";
        }
    }

    public static void main( final String[ ] argv)
    {
        System. out. println( test());
    }
}
```

Listing 19.1 *Ausführbar als* **'TESTRETURNINCATCH'**

Der `finally`-Block wird immer *nach allen* anderen Aktionen, also als letzte Anweisungsfolge abgearbeitet. Es kommt hier zur Ausgabe von "`Tatsächlich`", und das obwohl der `catch`-Block und das dortige `return` ausgeführt werden. Das `return` im `finally` übersteuert aber alle vorherigen `return`-Anweisungen.

19.3.5 Klassendesign

Bevorzuge lokale Variablen gegenüber Attributen

Wenn möglich sollte man lokale Variablen anstelle von Attributen verwenden: Lokale Variablen sind immer Thread-sicher, da sie niemals von mehreren Threads gleichzeitig geschrieben bzw. gelesen werden können. Außerdem verringern sie die Anzahl möglicher Objektzustände und machen ein Programm dadurch oftmals verständlicher.

Vermeide statische Attribute

Statische Attribute erinnern an globale Variablen aus nicht objektorientierten Zeiten. Statische Attribute haben die unangenehme Eigenschaft, dass sie vom Garbage Collector nicht freigegeben werden können, da erst nach Programmende die letzte

Referenz darauf erlischt.[3] Diese »Dauerreferenzierung« ist in den meisten Fällen unerwünscht. Nützlich sind statische Variablen jedoch für Metadaten von Klassen, etwa zur Bereitstellung von Loggern.

Schlimmer als statische Attribute sind statische Collections. Diese können ungehindert während der Programmlaufzeit wachsen, da sie nicht in ihrer Größe beschränkt sind. Programmierfehler können so leicht Out-of-Memory-Situationen provozieren.

Greife auf Attribute bevorzugt über Methoden zu

Die Forderung, auf Attribute nicht direkt, sondern über Methoden zuzugreifen, folgt dem OO-Gedanken der Datenkapselung. Innerhalb von Klassen kann man beim Zugriff auf eigene Attribute auf den Einsatz von Zugriffsmethoden verzichten. Dies gilt vor allem, wenn man dort auf private Attribute zugreift. Auf jeden Fall sollte man bei privaten Attributen entweder konsequent auf `get()`-Methoden verzichten oder diese konsequent einsetzen. Mischt man den Zugriff, fragt man sich als Betrachter der Klasse immer, worin der Unterschied zwischen einem direkten Zugriff und einem `get()`-Aufruf liegt.

Vermeide feingranulare Änderungen am Objektzustand

Klassen mit vielen öffentlichen `set()`-Methoden erlauben eine feingranulare Änderung des internen Zustands von außen und widersprechen damit dem objektorientierten Gedanken, dass ein Objekt seinen Zustand kapseln und nach außen möglichst nur fachliche Operationen (meistens in Form von Business-Methoden) anbieten sollte.

Minimiere Zustandsänderungen

Der Einsatz von Read-only-Interfaces (vgl. Abschnitt 3.4.1) kann vor ungewollten Veränderungen von außen schützen. Manchmal ist es zusätzlich möglich, den Objektzustand mithilfe des IMMUTABLE-Musters (vgl. Abschnitt 3.4.2) vollständig gegen Modifikationen abzusichern. Auf jeden Fall sollten all diejenigen Attribute `final` definiert werden, die nach ihrer Initialisierung unveränderlich sein sollen. Ungewollte und inkonsistente Änderungen am Objektzustand werden dadurch vermieden.

Liefere keine Referenzen auf interne Datenstrukturen zurück

Wie schon gerade eben angedeutet, sollte ein Objekt seinen Zustand kontrollieren. Dies ist nahezu unmöglich, wenn Referenzen auf interne Datenstrukturen herausgegeben werden. Stattdessen sollte man entweder Kopien davon zurückgeben oder die `unmodifiable`-Wrapper des Collections-Frameworks nutzen. Selbst wenn

nur Daten lesende Methoden angeboten werden, lauern noch versteckte Probleme durch die Referenzsemantik von Java: Es können Änderungen am Zustand der in Containerklassen gespeicherten Objekte erfolgen.

Abstraktion: Programmiere gegen Interfaces oder abstrakte Klassen

Sofern Abstraktion, Austauschbarkeit und Wiederverwendbarkeit wichtige Kriterien beim Design sind, empfiehlt sich der Einsatz von Interfaces und abstrakten Klassen, um konkrete Implementierungen zu verstecken.

Ein gängiges Beispiel ist die Nutzung des Interface `List<E>` statt der konkreten Klassen `ArrayList<E>` oder `LinkedList<E>`. Damit wird neben einer loseren Kopplung zudem sichergestellt, dass wirklich nur die Methoden des angebotenen Interface aufgerufen werden (können). Wenn lose Kopplung und Abstraktion so gut sind, könnte man auf die Idee kommen, nahezu immer ein noch allgemeineres Interface, im Beispiel also `Collection<E>`, zu verwenden. ***Eine Generalisierung sollte aber nur so weit fortgesetzt werden, wie es keinen Aussagekraftverlust für den gewünschten Einsatzzweck gibt***. Schauen wir zur Verdeutlichung auf eine Verwaltung von Listenern, wobei jeder von diesen nur einmal angemeldet werden soll, um doppelte (bzw. potenziell mehrfache) Benachrichtigungen zu unterbinden. Als ungünstige Variante ist einführend eine Definition als `Collection<Subscribers>` gezeigt:

```
// kommuniziert Mengeneigenschaft nicht
private final Collection<Subscriber> subscribers = new HashSet<>();
```

Durch diese starke Abstraktion verliert man jedoch die wichtige Information über die Mengeneigenschaft. Daher verwendet man passenderweise ein `Set<Subscriber>`. Die folgende Variante bietet eine gute Abstraktion und liefert ein für den Anwendungsfall zugeschnittenes Interface:[4]

```
private final Set<Subscriber> subscribers = new HashSet<>();
```

Halte das Interface übersichtlich

Ein Interface sollte möglichst nur Methoden für genau einen klar abgegrenzten Aufgabenbereich definieren. Steigt die Anzahl der in einem Interface deklarierten Methoden über einen gewissen Wert (etwa fünf bis zehn Methoden), so deutet dies darauf hin, dass das Interface besser in mehrere separate, voneinander unabhängige Interfaces aufgespalten werden sollte. Das adressiert auch das Interface Segregation Principle (ISP) – eines der SOLID-Prinzipien für guten OO-Entwurf (vgl. Abschnitt 3.5.3).

Sorge für Lesbarkeit und die richtige Abstraktionsebene

Innerhalb von öffentlichen Methoden sollten nicht zu viele Implementierungsdetails gezeigt werden. Schauen wir uns folgende beiden Methodenaufrufe an, wobei RBL eine branchenübliche Abkürzung für ein Rechner-basiertes Leitsystem ist:

1. `systemStateService.isSystemAlive(SystemType.RBL);`

2. `isRBLAlive();`

Der erste Methodenaufruf erfolgt auf Realisierungsebene und zeigt viele Implementierungsdetails. Er macht die beteiligte Klasse sowie den Methodenaufruf sichtbar. Die zweite Variante ist von ihrer Abstraktionsebene viel höher angesiedelt. Die Realisierung auf dieser fachlichen Ebene zeigt keine Details und versteckt die dahinterliegende Komplexität. Öffentliche Methoden sollten auf einer solch hohen Abstraktionsebene als verhaltensdefinierende Business-Methoden formuliert werden. Mit strengerer Sichtbarkeit nimmt die Abstraktion von technischen Details gewöhnlich ab: Private Methoden dürfen demnach Implementierungsdetails zeigen.

Verstecke Implementierungsdetails

Benutze bevorzugt eine möglichst kleine Sichtbarkeit (und das ist nicht `PUBLIC` ;-)). Attribute sollten `private` oder Package-private definiert werden. Für Hilfsmethoden gilt dies ebenfalls. Business-Methoden, die nach außen sichtbare Funktionalität anbieten, sind dagegen meistens `public`. Nur wenn Klassen so entworfen sind, dass sie eine Erweiterbarkeit durch Überschreiben von Methoden erlauben, sollten auch Methoden mit der Sichtbarkeit `protected` existieren.

Definiere abstrakte Methoden möglichst `protected`

Zur späteren Spezialisierung von Verhalten in Subklassen dienen abstrakte Methoden in Basisklassen. Werden diese Methoden `protected` definiert, können abgeleitete Klassen auf sie zugreifen – Zugriffe aus demselben Package sind auch möglich.

Beachte die maximale Methodenlänge von ca. 30 – 50 Zeilen

Eine Methode sollte genau eine kleine Teilaufgabe durchführen. Häufig sieht man eine Verquickung von Informationsbeschaffung und -verarbeitung. Das stört die Orthogonalität und Wiederverwendbarkeit. Methoden können dann nicht wie kleine Bausteine zu neuen, größeren Methoden kombiniert werden.

Tipp: Methodenlänge

Man findet leider immer wieder Methoden, deren Länge die 100 Zeilen mehrfach übersteigt. Wiederholt sind mir Methoden mit über 500 Zeilen aufgefallen. Derartige Monstren sind nahezu unwartbar. Man sollte versuchen, sie durch das Herausfaktorieren von Hilfsmethoden beherrschbar zu machen. Der Vorgang wird

so lange wiederholt, bis die Methode eine vernünftige Struktur und Länge besitzt.

Beachte die maximale Klassenlänge von ca. 500 – 1000 Zeilen

Eine Klasse sollte nur für einen Aufgabenbereich zuständig sein. Wird eine Klasse immer länger, so deutet dies darauf hin, dass sie für zu viele verschiedene Dinge verantwortlich ist. Wie bereits für die Methodenlänge angedeutet, stört dies die Orthogonalität und Wiederverwendbarkeit – in diesem Fall der Klasse.

Vermeide zu viele Referenzen auf andere Klassen

Werden von einer Klasse viele andere Klassen referenziert (mehr als ca. sieben), kommt es schnell zu »Objekt-Spaghetti«. Eine solche Situation entsteht vielfach dadurch, dass die Aufteilung von Funktionalität auf Klassen nicht sinnvoll durchgeführt wurde.

19.3.6 Parameterlisten

Halte die Parameterliste kurz

Aus der Psychologie ist bekannt, dass ein Mensch sich etwa 7 ± 2 Dinge im Kurzzeitgedächtnis merken kann. Daher sollte man maximal sieben Parameter verwenden. Übersichtlicher sind natürlich weniger Parameter, am besten nur ein bis vier. Durch das Einführen von Parameter Value Objects (vgl. Abschnitt 3.4.5) kann man Parameter zu Gruppen zusammenfassen und erreicht so kürzere, typsichere und verständlichere Signaturen.

Vermeide überflüssige Parameter

Zum Teil sieht man Methoden, die Parameter in Form eines Objekts übergeben bekommen und zusätzlich separat noch einen Parameter, der aus dem Objekt stammt. Das folgende Listing verdeutlicht dies anhand der Methode `handleMsg( byte[], Device, String)`, die auch Informationen aus dem Objekt `device` erhält:

```
getTelegramHandler().handleMsg( tmpBytes, device, device.getAddress());
```

Der über `device.getAddress()` ermittelte Wert könnte als Parameter entfallen und in der Methode selbst aus dem übergebenen `device` ermittelt werden.

Vermeide mehrere gleiche Typen aufeinander folgend in der Parameterliste

Mehrere gleiche Typen in einer Parameterliste können problematisch sein, da die Gefahr der Verwechselung von Positionen besteht. Je mehr Parameter gleichen Typs verwendet werden, desto stärker macht sich das Problem bemerkbar. Nachfolgende Signaturen provozieren geradezu Fehler:

```
public void fillRect(int x1, int y1, int x2, int y2, long colorRGB)
public void fillRect(int red, int green, int blue,
                     int x, int y, int width, int height)
```

Auch hier leisten Parameter Value Objects gute Dienste, um typsichere und verständlichere Signaturen zu erreichen und Anwendungsfehler nahezu auszuschließen:

```
public void fillRect(Point point1, Point point2, long colorRGB)
public void fillRect(Color fillColor, Point point, Dimension size)
```

Halte die Reihenfolge von Parametern bei Methodenaufrufen konsistent

Besitzen verschiedene Methoden gleiche Parameter und rufen sich auf, so sollte die Reihenfolge möglichst einheitlich gehalten werden. Folgendes Listing zeigt ein Negativbeispiel, in dem die Parameter `a` und `b` für die Methoden `doThis(TypeA, TypeB)` und `doThat(TypeB, TypeA)` in ihrer Reihenfolge getauscht sind:

```
public int doThis(TypeA a, TypeB b)
{
    return doThat(b,a);
}

public int doThat(TypeB b, TypeA a)
{
    // …
}
```

Warum ist das ungünstig? Die Begründung ist einfach: Ändert sich die Reihenfolge der Parameter ständig, so wird das Verwenden der Funktionen mühselig, da man sich jedes Mal wieder Gedanken über die Aufrufreihenfolge machen muss. Zudem kann durch Vertauschen der Position eines Parameters diesem eine andere semantische Bedeutung gegeben werden. Solche Fehler sind schwierig zu finden.

19.3.7 Logik und Kontrollfluss

Vermeide explizite `true`-/`false`-Literale in `if`-/`while`-Anweisungen

Verwendet man in Abfragen von booleschen Variablen explizit die Schlüsselwörter `true` oder `false`, so wird dies schnell unleserlich. Hier ein Beispiel:

```
if (attributeValue.isNullValue() == false)
    if (isEditable == true)
```

Besser lesbar werden die Abfragen durch Weglassen der booleschen Konstanten und die Kombination zu einer einzeiligen Bedingung:

```
if (!attributeValue.isNullValue() && isEditable)
```

Verwende möglichst wenige `else if`-Anweisungen

Komplexe Bedingungen mit mehreren `else if`-Varianten machen Programme recht schnell unübersichtlich. Mit zunehmender Anzahl wird es immer schwieriger, die Bedingungen so zu formulieren, dass sich diese tatsächlich gegenseitig ausschließen. Folgende Zeilen zeigen ein Negativbeispiel:

```
if (x >= 3 && y < 11 && z == 300 && command.equals("one"))
{
    System.out.println("1");
}
else if (x < 3 && (y > 10 || y < 10))
{
    System.out.println("2");
}
// für x == 3 und z == 300 Überschneidung mit Fall 1
else if (x <= 3 && z == 300)
{
    System.out.println("3");
}
```

Einfache Bedingungen mit mehreren `else if`-Anweisungen sind akzeptabel:

```
if (command.equalsIgnoreCase("cd"))
{
    executeCommand(new CdCommand());
}
else if (command.equalsIgnoreCase("list"))
{
    executeCommand(new ListCommand());
}
```

Allerdings lässt sich auch hier eine elegantere Lösung finden – etwa mit einem gemeinsamen Interface und Polymorphie.

Verwende nicht mehr als drei bis vier Logikauswertungen in einem `if`

Werden sehr viele, mitunter auch komplexe Bedingungen in einer `if`-Anweisung geprüft, so leidet die Übersichtlichkeit, wie dies folgendes Beispiel zeigt:

```
if (hasSpecialTexts()  // Sondertext
    || !deviceIsActive  // Anlage deaktiviert
    || !displayIsActive  // Anzeiger deaktiviert
    || !myDisplay.isEnabled()  // Anzeiger gesperrt
    || noDeps  // keine Abfahrten
    || ((getFallbackCode() == FailureConstants.NO_RADIOCONNECT) && !
          showWithinFallback)
    || ((getFallbackCode() == FailureConstants.NO_AVMCONNECT) && !ready)
    || (getFallbackCode() == FailureConstants.NO_DISPCONNECT)
    || (getFallbackCode() == FailureConstants.NO_PIDCONNECT))
```

Das Einführen von einigen Prüfmethoden kann die Lesbarkeit ungemein erhöhen:

```
if (    hasSpecialTexts()
    ||  deviceOrDisplaysInactive()
```

Verwende den Conditional-Operator mit Bedacht

Einfache `if`-Bedingungen mit einigen Zeilen Sourcecode lassen sich manchmal elegant mit dem Conditional-Operator formulieren. Folgende Zeilen dienen als Basis:

```
final String strType;
if (isFolder)
    strType = "directory";
else
    strType = "file";
```

Der Einsatz des Conditional-Operators vereinfacht den Sourcecode wie folgt:

```
final String strType = (isFolder ? "directory" : "file");
```

Folgender Conditional-Operator ist durch die Methodenaufrufe in den Auswertungen (schon fast) zu kompliziert, um leicht verständlich zu sein und eingesetzt zu werden:

```
return oldState != null ? oldState.compareTo(newState) == 0 :
                          newState.equals(null);
```

Wirklich verwirrend und komplex ist aber folgendes Konstrukt:

```
final Double value = value1 == null ? value2 : value2 == null
                                     ? value1 : new Double(value1 + value2);
```

19.4 Sourcecode-Prüfung mit Tools

In den vorherigen Abschnitten wurden diverse Regeln aufgestellt, deren Einhaltung die Qualität des Sourcecodes verbessern kann. Eine manuelle Prüfung umfangreicher Projekte ist durch Sourcecode-Inspektion und Codereviews jedoch mühevoll und extrem zeitaufwendig. Das macht die Unterstützung durch Tools wünschenswert.

In den folgenden Abschnitten gehe ich zunächst auf die Grundlagen der Auswertung durch Tools ein. Anschließend werden einige Kennzahlen zur Bewertung der Güte von Programmen vorgestellt. Nachfolgend wird dann die automatisierte SourcecodePrüfung durch Tools thematisiert. Dort werden unter anderem Checkstyle, FindBugs und PMD kurz beschrieben und gezeigt, wie man diese in den automatischen Build-Lauf integrieren kann.

Sourcecode-Checker und statische Analyse

Die Auswertungen von Sourcecode-Checkern basieren auf einer sogenannten ***statischen Analyse***, die den Sourcecode bzw. den vom Java-Compiler generierten Bytecode untersucht. Es ist demnach keine Ausführung des Programms notwendig,

wie dies bei der sogenannten ***dynamischen Analyse***, beispielsweise der eines Profilers, der Fall ist. Durch eine statische Analyse können somit keine Aussagen zum Laufzeitverhalten gemacht werden. Es lassen sich jedoch unter anderem folgende Schwachstellen erkennen:

- Duplikation von Sourcecode
- lange Klassen und Methoden
- umfangreiche Parameterlisten
- unbenutzte Methoden und Variablen

Viele solcher Probleme auf Sourcecode-Ebene lassen sich mithilfe von Codereviews und einer Sammlung von Coding Conventions aufwendig, aber kaum vollumfänglich feststellen, wie das automatisiert durch Tools möglich ist. Architektur- und Designprobleme sind dagegen meistens schwieriger zu erkennen und können zum Teil nur über die Auswertung der in Abschnitt 19.4.1 vorgestellten Kennzahlen, sogenannten ***Metriken***, ermittelt werden. Das Tool JDepend kann diese Kennzahlen berechnen. Anhand dieser lassen sich beispielsweise folgende Probleme aufdecken:

- viele Abhängigkeiten zwischen Packages und Klassen
- schwache Kohäsion innerhalb von Klassen
- starke Kopplung zwischen Klassen
- tiefe oder breite Vererbungshierarchien

Nicht alle von den Tools aufgedeckten Schwachstellen sind tatsächlich auch problematisch. Es handelt sich dabei eher um Sourcecode-Abschnitte, die möglicherweise Probleme enthalten und daher genauer untersucht werden sollten. Erfolgt eine Integration des Prüftools in die IDE, so kann man die Stellen direkt anspringen. Eine Auswertung und Bearbeitung ist damit ohne Verlassen der IDE möglich. Das erleichtert eine Fehlersuche bereits während der Programmierung.

Mit Tools können festgestellte Mängel als Warnung oder Kompilierfehler klassifiziert werden. Dies kann man auch zur Einhaltung einiger wichtiger Regeln nutzen, wie es der folgende Praxistipp beschreibt.

Tipp: Einhaltung von Codierungsregeln

Will man die Einhaltung einer speziellen Regel forcieren, so kann man einen ***Verstoß dagegen als Fehler werten*** und somit ein erfolgreiches Kompilieren verhindern. Dieser Trick hat sich als hilfreich erwiesen, ***um gewissen Regeln die gewünschte Aufmerksamkeit zu verleihen***. Allerdings sollte dieses Vorgehen nicht überstrapaziert werden, also lediglich für einige als wichtig angesehene Regeln angewendet werden.

Findet ein Entwickler nach der Aktivierung von neuen Regeln zu viele Fehler vor, so ist er schnell entmutigt. Wichtig ist es daher, dass die Regeln allen Entwicklern gut bekannt und auch von allen akzeptiert werden – einführende Schulungen und

Workshops tragen dazu bei. Zudem hat sich eine schrittweise Einführung und Verschärfung der Regeln als vorteilhaft erwiesen. Die Grundlage bilden mehrere unterschiedlich strenge Regelsätze. Dabei kann die strengste Auslegung nahezu jeden Regelverstoß als Fehler ahnden. Die niedrigste Stufe dient als Einstieg in die Sourcecode-Prüfung und erleichtert die Akzeptanz solcher Maßnahmen bei Skeptikern.

19.4.1 Metriken

Wünschenswert ist es, die Qualität eines Programms anhand einiger weniger Kennzahlen ablesen zu können. Dazu wird der Sourcecode analysiert und unter verschiedenen Aspekten betrachtet, etwa der Anzahl der Zeilen oder der Anzahl der Klassen bzw. Interfaces. Die ermittelten Werte kann man auf jeweils eine eigene sogenannte ***Metrik*** abbilden. Diese liefert eine bewertbare Kennzahl für eine spezielle Eigenschaft eines Softwaresystems. Definiert man erlaubte Wertebereiche für die jeweiligen Metriken, so kann man durch fortlaufende Prüfungen des Sourcecodes eines Projekts die Güte der Einhaltung und Trends ablesen. Abweichungen vom gewünschten Ziel können leicht erkannt und durch entsprechende Maßnahmen korrigiert werden. Anhand der ermittelten Kennzahlen können dann Auswirkungen von Änderungen bewertet werden. Zudem lassen sich durch Metriken eventuelle Schwachstellen leichter lokalisieren. ***Allerdings ist eine abschließende Bewertung, ob tatsächlich ein Problem vorliegt, häufig nur durch einen erfahrenen Entwickler möglich, und nicht allein aufgrund einer Kennzahl.***

Folgende Aufzählung nennt einige Bewertungskriterien und Metriken zur Beurteilung der Güte des Sourcecodes. Weitere Informationen finden Sie im Buch »Object-Oriented Metrics: Measures of Complexity« von Brian Henderson-Sellers [39].

- **Lines Of Code (LOC)**

  Die Anzahl der Zeilen im Sourcecode ist wohl die bekannteste Messgröße und kann ein Maß für die Komplexität eines Softwaresystems sein. Eine Weiterentwicklung dieser Metrik besteht darin, lediglich Zeilen mit Anweisungen, also weder Zeilen mit Kommentar noch Leerzeilen, zu zählen. Diese Metrik wird als NCSS (Non Commented Sourcecode Statements) oder auch MLOC (Method Lines of Code) bezeichnet und ermittelt normalerweise eine aussagekräftigere Kennzahl als LOC.

  Bewertung: In der Regel gilt ein einfacher Zusammenhang: Je mehr Zeilen ein Programm besitzt, desto komplizierter, fehleranfälliger und unübersichtlicher ist es. Die Zeilenmetriken geben aber nur einen groben Überblick über den Umfang eines Programms, jedoch nicht über dessen Qualität. Somit besitzen diese Metriken nur eine beschränkte Aussagekraft.

- **Cyclomatic Complexity / McCabe-Metrik**

Die Cyclomatic Complexity oder McCabe-Metrik misst die Anzahl alternativer Wege durch ein Programm. Komplexe `if-else`-Konstrukte und hohe Schachtelungstiefen erhöhen diesen Wert. Vielfach steigt mit zunehmender Komplexität, z. B. durch eine schlecht gewählte Strukturierung, der Aufwand, den Programmablauf nachzuvollziehen. Als Faustregel sollte die Cyclomatic Complexity möglichst kleiner als 15 sein. Werte über 50 kann man als schwierig wartbar klassifizieren.

Bewertung: Diese Metrik ist mit Vorsicht zu genießen, da zum Teil für übersichtliche, aber mehrfach aufeinander folgende `if-` oder `case`-Anweisungen hohe Kennzahlen berechnet und damit durch diese Metrik negativ bewertet werden. Nichtsdestotrotz ist mit dieser Metrik in der Regel eine bessere Aussage über die Qualität des Sourcecodes zu erzielen als mit den Zeilenmetriken wie LOC etc.

- **Dokumentation**

  Aus der Anzahl und der Güte der Kommentare innerhalb eines Programms kann man Rückschlüsse auf dessen Wartbarkeit ziehen. Das ist dadurch begründet, dass Kommentare normalerweise zur Strukturierung und Lesbarkeit eines Programms beitragen und damit ein späteres Nachvollziehen (auch von anderen Entwicklern) erleichtern. Eine fehlende Dokumentation[5] von Klassen und öffentlichen Methoden zeugt meistens von einem unausgereiften Design oder von Zeitdruck. ***Wenn man nicht in der Lage ist, die Aufgaben von (öffentlichen) Methoden und Klassen in Worte zu fassen, fehlt häufig das Verständnis für das zu lösende Problem***. Dementsprechend unstrukturiert ist dann oftmals auch die Implementierung. Es gibt auch den Fall der übermäßigen oder überflüssigen Dokumentation. Nicht jedes Stück Sourcecode muss kommentiert werden. Private Methoden können manchmal bereits ohne Kommentar allein durch aussagekräftig gewählte Methoden- und Variablennamen sowie durch eine kurze Methodenlänge verständlich sein. Guter Sourcecode ist meistens (nahezu) selbsterklärend. Dann sollte auf eine eher »künstliche« Kommentierung einer privaten Methode verzichtet werden.

  Bewertung: Die Diskussion zeigt, dass diese Metrik mit Vorsicht zu genießen ist: Ein Tool kann sinnvolle Kommentare nicht von nichtssagenden oder automatisch generierten Kommentaren unterscheiden. Weiterhin besteht das Problem der Konsistenz zwischen Dokumentation und Sourcecode. Selbst wenn die Voraussetzung für ein einfaches Verständnis sowohl durch einen aussagekräftigen Kommentar als auch durch gut lesbaren Sourcecode vorliegt, kann es Verwirrung auslösen, wenn das Programm etwas anderes macht, als es der Kommentar ausdrückt.

- **OO-Metriken nach Chidamber und Kemerer**

  OO-Metriken messen hauptsächlich Ursachen für Probleme im Design, die ich bereits in Kapitel 3 im Zusammenhang mit Kohäsion, Kapselung und Vererbung vorgestellt habe. Je stärker z. B. die Abhängigkeiten zwischen verschiedenen

Klassen sind, desto mehr kann man von »Objekt-Spaghetti« sprechen. Durch eine fehlende Kapselung pflanzen sich Änderungen meistens unangenehm fort. Dies wird unter anderem durch die bereits 1994 von Shyan R. Chidamber und Chris F. Kemerer [13] in einem Artikel vorgestellten Metriken erfasst. Das sind etwa folgende:

- **Depth in Inheritance Tree (DIT)**

  Die Metrik DIT beschreibt die Tiefe des Ableitungsbaums, also die Anzahl der Oberklassen einer betrachteten Klasse. Einerseits wird dadurch die Wiederverwendung gemessen, andererseits kann ein hoher Wert aber auch ein Hinweis auf falsche Abstraktionen sein. In Java ist der minimale Wert für DIT der Wert zwei, da jede Klasse von der Basisklasse `Object` erbt. Sinnvolle Werte für DIT liegen im Bereich von zwei bis vier. Für Frameworks und einige Swing-Komponenten kommt durch Vererbung von Basiskomponenten sogar einen DIT von fünf bis acht zustande.

- **Number Of Children (NOC)**

  Die Metrik NOC gibt die Anzahl der direkten Subklassen einer Klasse an und misst demnach die Breite des Ableitungsbaums. Diese kann als ein Indiz für die Wichtigkeit und Wiederverwendung einer Klasse angesehen werden: Bei einem hohen Wert erfordern Änderungen an Basisklassen mit einiger Wahrscheinlichkeit auch Folgeänderungen in Subklassen. Generell sollten Basisklassen zwar mehrere Subklassen besitzen und damit einen höheren Wert für NOC: Je tiefer man sich in der Ableitungshierarchie befindet, desto unwahrscheinlicher ist es aber, dass von den Spezialisierungen wiederum viele weitere Spezialisierungen existieren. Ähnlich wie bei DIT liegen sinnvolle Werte für NOC im Bereich von fünf bis acht, häufiger eher im Bereich von zwei bis vier. Für viele Klassen, etwa solche, die `final` sind, ist der Wert für NOC 0.

- **Coupling Between Objects (CBO)**

  Die Metrik CBO bestimmt die Anzahl der Klassen, mit denen die betrachtete Klasse gekoppelt ist, d. h. auf deren Methoden oder Attribute sie zugreift. Zudem gehen auch alle Zugriffe und Aufrufe von anderen Klassen in die betrachtete Klasse mit in die Kennzahl ein. Ein hoher Wert zeigt eine starke Kopplung an, was wahrscheinlich zu einem erhöhten Aufwand bei Änderungen und nachfolgenden Tests führen wird. Der ermittelte Wert sollte möglichst kleiner als fünf bis zehn sein. Selten sind Werte bis 20 in Ordnung.

- **Number Of Methods (NOM)**

  Die Metrik NOM beschreibt die Anzahl in einer Klasse definierter Methoden. Ein hoher Wert deutet darauf hin, dass die Klasse für zu viele Dinge verantwortlich ist. Hier gelten die gleichen Schlussfolgerungen wie bei der Metrik CBO. Bevorzugt sollte der Wert im Bereich von 10 bis 30

liegen. Bei komplexeren Klassen können ausnahmsweise auch höhere Werte akzeptiert werden.

Durch Überprüfen der obigen Kennzahlen kann man zwar einige Fehlerquellen reduzieren, eine korrekte Realisierung der gewünschten Funktionalität lässt sich so allerdings nicht sicherstellen. Zudem existieren weitere für die Qualität entscheidende Bewertungskriterien, die sich jedoch kaum durch Tools prüfen lassen. Dies sind unter anderem die folgenden:

- **Lesbarkeit / Übersichtlichkeit**

  Normalerweise geht eine gute Lesbarkeit mit verbesserten Werten anderer Bewertungskriterien einher und stellt daher eine der wichtigsten Messgrößen dar. Die Lesbarkeit lässt sich für wenige Klassen relativ gut durch stichprobenartige Blicke in den Sourcecode beurteilen. Bei umfangreicheren Projekten ist eine derartige Prüfung kaum noch sinnvoll möglich. Problematisch ist zudem, dass sich die Lesbarkeit nur schwierig formalisieren lässt und dadurch auch nicht durch Tools geprüft werden kann. Am einfachsten lässt sich dieses Kriterium erfüllen, wenn vor Projektbeginn ein Regelwerk ähnlich zu den vorgestellten Coding Conventions vereinbart und beim Programmieren auf dessen Einhaltung geachtet wird.

- **Abstraktionsgrad**

  Die Wahl geeigneter Abstraktionsgrade in Programmkomponenten ist eine durch Menschen intuitiv erfassbare Messgröße, die sich wiederum lediglich stichprobenartig prüfen lässt. Deren Einhaltung erkennt man in der Regel daran, dass öffentliche Methoden gut lesbar und verständlich sind, da sie auf einem hohen Abstraktionsniveau formuliert sind und sich aus Aufrufen von anderen Methoden geringerer Sichtbarkeit (`private`, Package-private und `protected`) zusammensetzen. Diese (Hilfs-)Methoden bilden in der Regel die Bausteine der öffentlichen Methoden. Wichtig ist es, dass Klassen ihre Methoden auf dem niedrigstmöglichen Sichtbarkeitslevel bereitstellen und somit das API, d. h. die Menge der öffentlichen Methoden, möglichst nur noch aus verhaltensdefinierenden Business-Methoden besteht und keinesfalls aus nur in der Klasse selbst benötigten Hilfsmethoden. Befolgt man das Gesagte, so führt dies meistens zu einer klaren Aufrufhierarchie von `public` nach `private`. ***Erfolgen dann durch private Methoden noch Aufrufe an öffentliche Methoden, so sollte man diese genau prüfen, da dies ein Zeichen schlechten Designs und falsch gewählter Abstraktionsebenen sein kann***. Es gibt jedoch verschiedene Anwendungsfälle, in denen in privaten Methoden bestimmte Funktionalität aus dem API benötigt wird. Dann kann ein Aufruf öffentlicher Methoden aus einer privaten Methode sinnvoll und unkritisch sein, da insbesondere das öffentliche API in der Regel stabil bleibt. Wichtig dabei ist allerdings, dass man ein Aufrufchaos vermeidet und das lässt sich am einfachsten über die Einhaltung der oben genannten strengen Hierarchie erreichen. Eine Möglichkeit zur Vermeidung eines Aufrufs einer öffentlichen Methode aus einer privaten besteht

darin, eine zusätzliche private Verarbeitungsmethode mit der gewünschten Funktionalität zu erzeugen, die dann sowohl von öffentlichen Methoden als auch privaten Methoden aufgerufen werden kann.

- **Fehlersicherheit und -toleranz**

  Bei der Fehlersicherheit und -toleranz kann man zwischen dem extern beobachteten Verhalten und den tatsächlichen internen Abläufen unterscheiden. Werden Fehler nicht nach außen sichtbar, so bedeutet das zunächst wenig. Es kann durchaus sein, dass diese »verschluckt« werden: Man spricht dafür auch von ***Fehlermaskierung***. Viel wichtiger ist demnach die Reaktion auf Fehler im Programm selbst, die man nicht von außen sieht. Durch eine Analyse des Sourcecodes kann man zumindest feststellen, ob Parameter geprüft und angemessen auf Exceptions oder Fehlersituationen reagiert wird, etwa um Ressourcen auch im Fehlerfall wieder freizugeben.

19.4.2 Sourcecode-Prüfung im Build-Prozess

Wie bereits erwähnt, ist es sinnvoll, die Qualität des Sourcecodes regelmäßig zu analysieren und dabei die Einhaltung gewisser Standards zu beachten. In einem ersten Schritt kann man auf die in Eclipse integrierte Sourcecode-Prüfung zurückgreifen. Umfangreichere Tests sollten mit Tools zur statischen Analyse wie Checkstyle, PMD und FindBugs erfolgen, die in den folgenden Abschnitten beschrieben werden.

Das Tool Checkstyle

Das Tool Checkstyle ermöglicht, die Einhaltung von Formatierungsvorgaben und Coding Conventions zu überprüfen. Unter `http://eclipse-cs.sf.net/` steht ein Eclipse-Plugin frei zur Verfügung.

Hilfreich beim Einsatz von Checkstyle ist, dass sich die gewünschten Codierungsregeln feingranular konfigurieren lassen, wodurch nahezu alle möglichen Coding Conventions realisiert werden können. Praktischerweise existiert ein vordefinierter Regelsatz für die Coding Conventions von Sun.

Die Auswertung kann Arbeitspunkte in eine To-do-Liste einfügen und auch Reports im HTML-Format erzeugen. Weiterführende Informationen sind online unter `http://checkstyle.sourceforge.net/` verfügbar.

Integration in den Build-Prozess

Es ist sinnvoll, die Prüfung des Sourcecodes als Bestandteil des Build-Prozesses auszuführen. Checkstyle lässt sich bequem in den Gradle-Build-Lauf integrieren, indem man das passende Plugin einbindet und selbst bei festgestellten Verstößen (`ignoreFailures`) auch noch weitere Prüftools ausführt:

```
apply plugin: 'checkstyle'

checkstyle {
    ignoreFailures = true
}
```

Zur Konfiguration der Regeln benötigt Checkstyle aber ein Verzeichnis `config/checkstyle`. Die Prüfung schlägt fehl, wenn dieser Ordner nicht existiert. Dort wird nach einer Datei `checkstyle.xml` gesucht, in der die Regeln definiert sind. Eine minimale Datei als Ausgangsbasis kann etwa wie folgt aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>

<module name="Checker">
    <module name="TreeWalker">
        <module name="AvoidStarImport"/>
        <module name="ConstantName"/>
        <module name="EmptyBlock"/>
    </module>
</module>
```

Die Checkstyle-Seite `http://checkstyle.sourceforge.net/config.html` liefert weitere Infos. Eine ausführliche Prüfung, die von Google verwendet wird, findet man unter `https://code.google.com/p/google-api-java-client/source/browse/checkstyle.xml?repo=samples`.

HTML-Report erzeugen

Wie schon erwähnt, ist es möglich, ein HTML-Dokument zu erzeugen (vgl. Abbildung 19-3).

Allerdings muss man dazu noch ein wenig in der Gradle-Build-Datei nachhelfen, da Checkstyle zunächst nur einen XML-Report generiert. Wir definieren folgenden Zusatz-Task `checkstyleBuildHtmlReport`, der von Hand gestartet werden muss:

```
task checkstyleBuildHtmlReport << {
    if (file("$buildDir/reports/checkstyle/main.xml").exists())
    {
        ant.xslt(in: "$buildDir/reports/checkstyle/main.xml",
                 style: "config/checkstyle/checkstyle-noframes-sorted.xsl",
                 out: "$buildDir/reports/checkstyle/checkstyle_main.html")
    }
}
```

Die dort zur Transformation genutzte Datei `checkstyle-noframes-sorted.xsl` ist frei unter `https://svn.apache.org/repos/asf/hive/trunk/checkstyle/checks noframes-sorted.xsl` verfügbar und muss ebenfalls im Verzeichnis `config/checkstyle` abgelegt werden.

CheckStyle Audit

Designed for use with CheckStyle and Ant.

Summary	
Files	**Errors**
4	68

Files	
Name	**Errors**
F:\eclipse_workspace\AntTestProject\src\util\StringUtils.java	23
F:\eclipse_workspace\AntTestProject\src\ui\AntTestProject.java	20
F:\eclipse_workspace\AntTestProject\src\util\Log4jSupport.java	18
F:\eclipse_workspace\AntTestProject\src\ui\WindowClosingHandler.java	7

File F:\eclipse_workspace\AntTestProject\src\ui\AntTestProject.java	
Error Description	**Line**
Missing package-info.java file.	0
Javadoc-Kommentar fehlt.	9
'{' sollte in der vorhergehenden Zeile stehen.	10
Zeile länger als 80 Zeichen	11
Javadoc-Kommentar fehlt.	11
Zeile länger als 80 Zeichen	13

Abbildung 19-3 *Checkstyle-Report*

Das Tool PMD

PMD ist ein Tool zur statischen Analyse und untersucht den Sourcecode. PMD lässt sich in diverse IDEs integrieren. Das Eclipse-Plugin ist online unter `http://pmd.sf.net/eclipse` frei verfügbar.

Integration in den Build-Prozess

PMD lässt sich mit wenigen Zeilen in den Gradle-Build-Lauf einbinden. Das hatte ich bereits in Abschnitt 2.7.3 kurz beschrieben. Hier wiederhole ich die wesentlichen Angaben:

```
apply plugin: 'pmd'

pmd
{
    toolVersion = '5.1.1'
    ruleSets = ["java-basic", "braces", "design"]
    ignoreFailures = true
}
```

Konfiguration von PMD in Eclipse

PMD bietet eine Konfigurationsmöglichkeit vorhandener Regeln als Dialog unter WINDOW –> PREFERENCES –> PMD –> RULES CONFIGURATION.

Zudem gibt es eine Erkennung von dupliziertem Sourcecode (»Copy-Paste«). Dieses Feature ist besonders nützlich, um Redundanzen und kopierte Abschnitte leichter erkennen und die dadurch betroffenen Sourcecode-Zeilen reduzieren zu können.

Befunde mit PMD

Abbildung 19-4 zeigt die von PMD gefundenen Probleme für einen kurzen Sourcecode-Abschnitt mit einigen bewusst integrierten Schwachstellen. Wie man an den Fehlermeldungen sieht, macht PMD auch Anmerkungen zu Designentscheidungen:

- `UseCollectionIsEmpty` – Benutze für Collections einen Aufruf der Methode `isEmpty()` statt der Abfrage `size() == 0`, weil Ersteres besser lesbar ist.
- `MethodArgumentCouldBeFinal` – Ein Methodenparameter sollte `final` definiert werden, um eine mögliche Zuweisung an einen Parameter zu verhindern.
- `UseSingleton` – Eine Definition gemäß dem SINGLETON-Muster (vgl. Abschnitt 14.1.4) empfiehlt sich hier, anstatt nur statische Methoden einzusetzen.

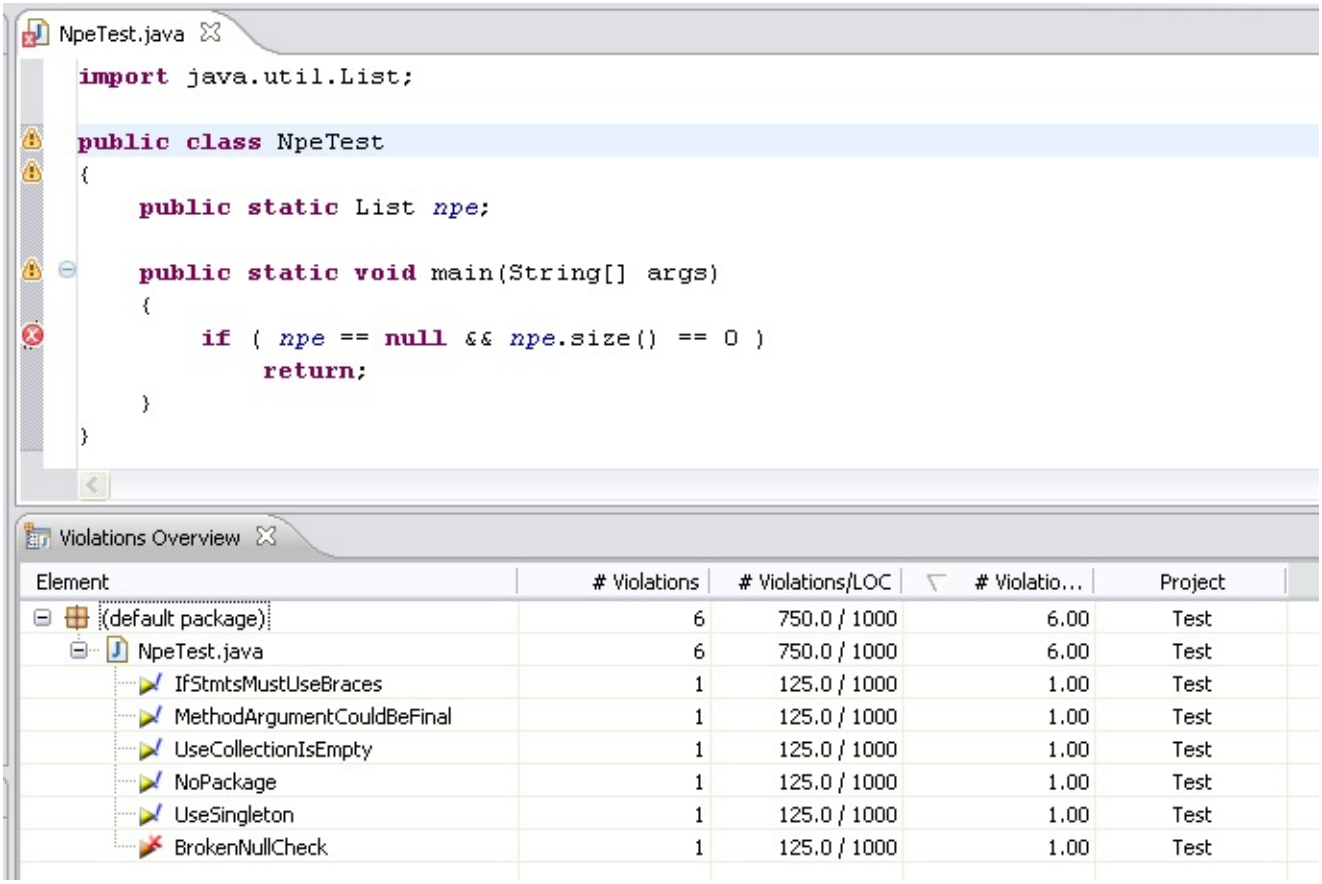


Abbildung 19-4 *PMD-Meldungen, z. B. mögliche* `NullPointerException`

Das Tool FindBugs

FindBugs ist, wie PMD, ein kostenlos verfügbares Hilfsprogramm, das eine statische Analyse durchführt. Im Gegensatz zu PMD untersucht FindBugs den generierten Bytecode und nicht den Sourcecode nach Schwachstellen. Ein Eclipse-Plugin lässt sich unter `http://findbugs.cs.umd.edu/eclipse` beziehen. FindBugs erkennt eine Menge von Fallstricken, etwa missverständliche API-Methoden oder falsche Annahmen bei der Auswertung boolescher Bedingungen.

Integration in den Build-Prozess

Ebenso einfach wie die anderen Tools lässt sich auch FindBugs wie folgt in den Gradle-Build-Lauf einbinden:

```
apply plugin: 'findbugs'
```

Befunde mit FindBugs

Abbildung 19-5 zeigt das Analyseergebnis von FindBugs für den gleichen Sourcecode-Abschnitt wie zuvor bei PMD.

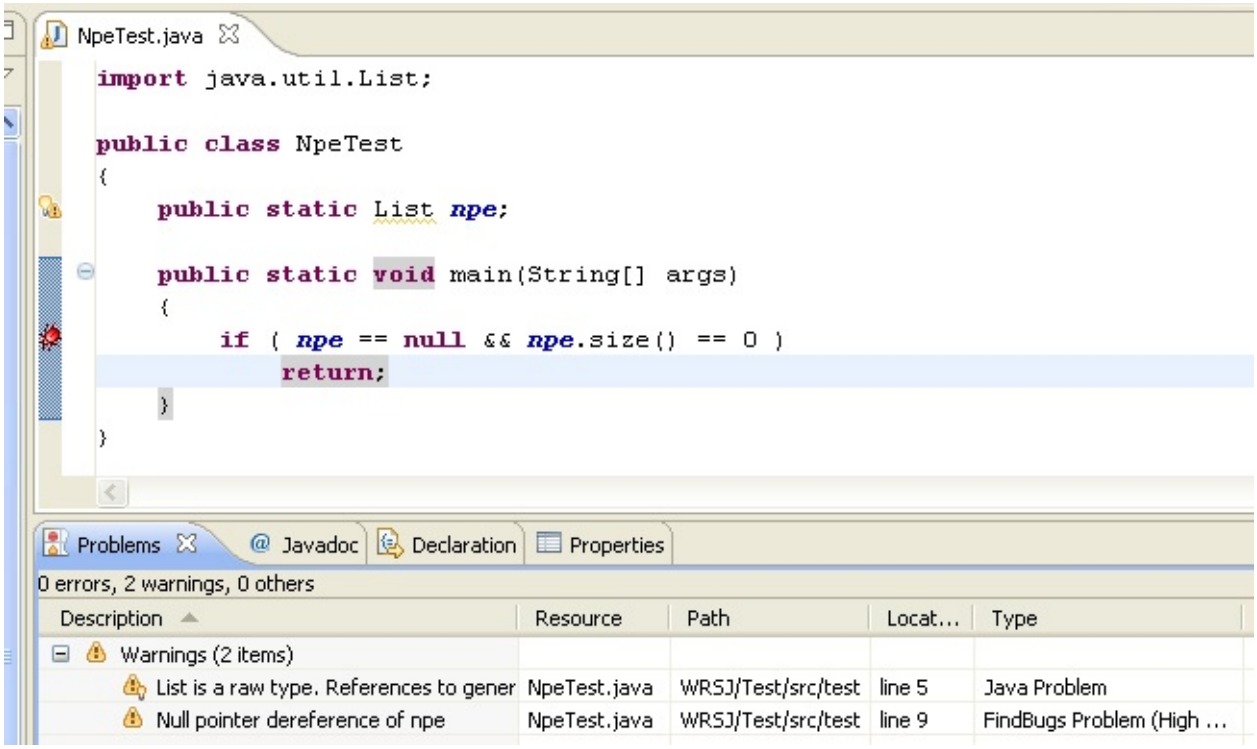


Abbildung 19-5 *FindBugs-Meldung für den gleichen Sourcecode*

Vergleich: PMD vs. FindBugs

PMD besitzt einen umfangreicheren Regelsatz als FindBugs. Dadurch erzeugt PMD mehr Meldungen zu (formalen) Regelverstößen. Da PMD lediglich den Sourcecode analysiert, können einige schwerwiegende Probleme (fehlerhafte Synchronisierung, nicht geschlossene Streams usw.) nicht erkannt werden. Das Tool FindBugs kann Derartiges erkennen, da es eine Analyse des generierten Bytecodes durchführt.

Deswegen ist eine Kombination beider Tools sinnvoll.

Das Tool JDepend

Während zur Analyse verschiedener Design- und Architekturverstöße in den ersten Auflagen dieses Buchs das mittlerweile nicht mehr weiterentwickelte Tool Metrics zum Einsatz kam, wollen wir dafür nun das Tool JDepend betrachten. JDepend ermittelt verschiedene Metriken, wie die Anzahl an Klassen und Interfaces, ein- und ausgehende Verbindungen (Kopplung), die Abstraktheit (Verhältnis abstrakte zu konkreten Klassen) sowie die Stabilität bzw. Instabilität gegeben durch die Kopplung. Weitere Informationen finden Sie online unter `http://clarkware.com/software/JDepend.html` sowie `http://www.onjava.com/pub/a/onjava/2004/01/21/jdepend.html`.

Integration in den Build-Prozess

Die zuvor genannten Metriken lassen sich mit jedem Gradle-Build-Lauf ermitteln. Dazu ergänzt man folgende Zeile im Build-Skript:

```
apply plugin: 'jdepend'
```

Leider erzeugt JDepend seine Reports derzeit nur als XML. Allerdings haben wir auch hier wieder die Möglichkeit, einen Task folgendermaßen selbst zu definieren, der eine Konvertierung in HTML vornimmt:

```
task jdependBuildHtmlReport << {
    if (file("$buildDir/reports/jdepend/main.xml").exists())
    {
        ant.xslt(in: "$buildDir/reports/jdepend/main.xml",
                 style: "config/jdepend/jdepend-frames.xsl",
                  out: "$buildDir/reports/checkstyle/jdepend_main.html")
    }
    if (file("$buildDir/reports/jdepend/test.xml").exists())
    {
        ant.xslt(in: "$buildDir/reports/jdepend/test.xml",
                 style: "config/jdepend/jdepend-frames.xsl",
                  out: "$buildDir/reports/checkstyle/jdepend_test.html")
    }
}
```

Dabei muss man aber noch ein wenig mehr tricksen als bei Checkstyle. Die Datei zur XSL-Transformation findet man unter `https://java.net/projects/slamd/sources/svn/content/trunk/sla` `frames.xsl`. Jedoch muss dort der Verweis auf `redirect` wie folgt korrigiert werden:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0" xmlns:lxslt="http://xml.apache.org/xslt"
    xmlns:redirect="http://xml.apache.org/xalan/redirect"
    extension-element-prefixes="redirect">
```

Befunde mit JDepend

Nach diesen Korrekturen wird dann im Verzeichnis `build/reports/jdepend` eine Menge an Dateien erzeugt, unter anderem eine Datei `index.html`, die einen Überblick bietet und als Ausgangspunkt für weitere Analysen dient (vgl. Abbildung 19-6).

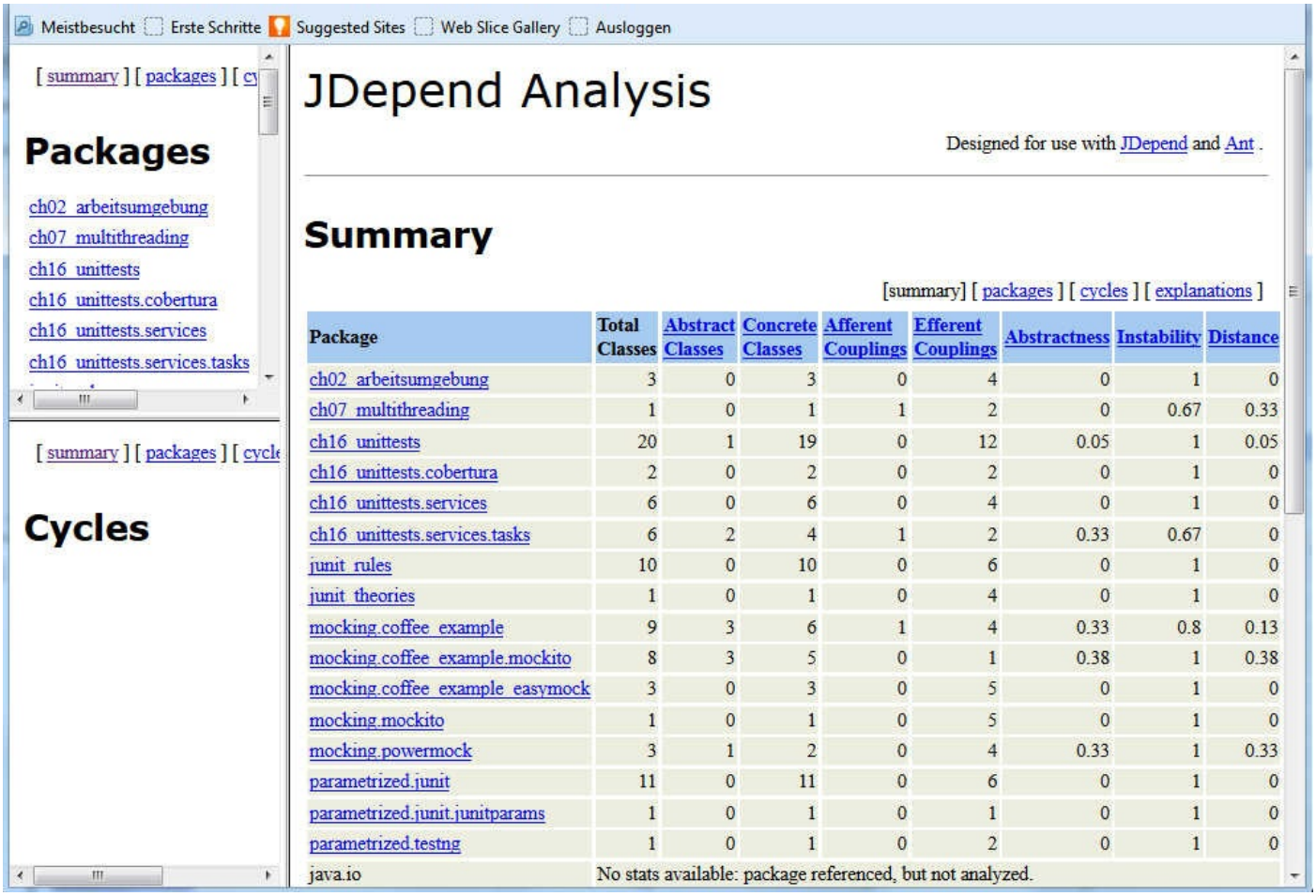


Abbildung 19-6 *JDepend-Report*

Das Tool SonarQube

Die bisher beschriebenen Tools adressierten jeweils einen spezifischen Aspekt der Sourcecodequalität – wobei eine gewisse thematische Streuung zu finden ist. Das Tool Checkstyle prüft schon seit längerer Zeit nicht mehr nur die Einhaltung von Coding Conventions, sondern zudem auch gewisse Designaspekte oder Programmierfehler. Bei Letzteren haben aber vor allem FindBugs und PMD ihre Stärken. Abhängigkeiten und gewisse Architekturverstöße ermittelt man am geeignetsten mit JDepend.

Alle Tools müssen aber separat konfiguriert werden, was glücklicherweise durch Nutzung von Gradle erleichtert wird. Wäre es aber nicht noch angenehmer, wenn sich ein Tool um den gesamten Qualitätssicherungsaspekt kümmern und einen konsistenten Bericht aufbereiten würde? SonarQube, ehemals nur Sonar genannt, ist genau so ein Tool und bietet eine Plattform für die statische Sourcecodeanalyse und integriert dazu verschiedene Tools und deren Analyseergebnisse. Dabei hilft eine komfortable WebOberfläche, die Übersicht zu behalten. Dazu gibt man

`http://localhost:9000` ein. In Abbildung 19-7 sehen wir die Analyseergebnisse der Sourcen zu diesem Buch, wobei auch diejenigen der Bad Smells und anderer Fallstricke mit enthalten sind.

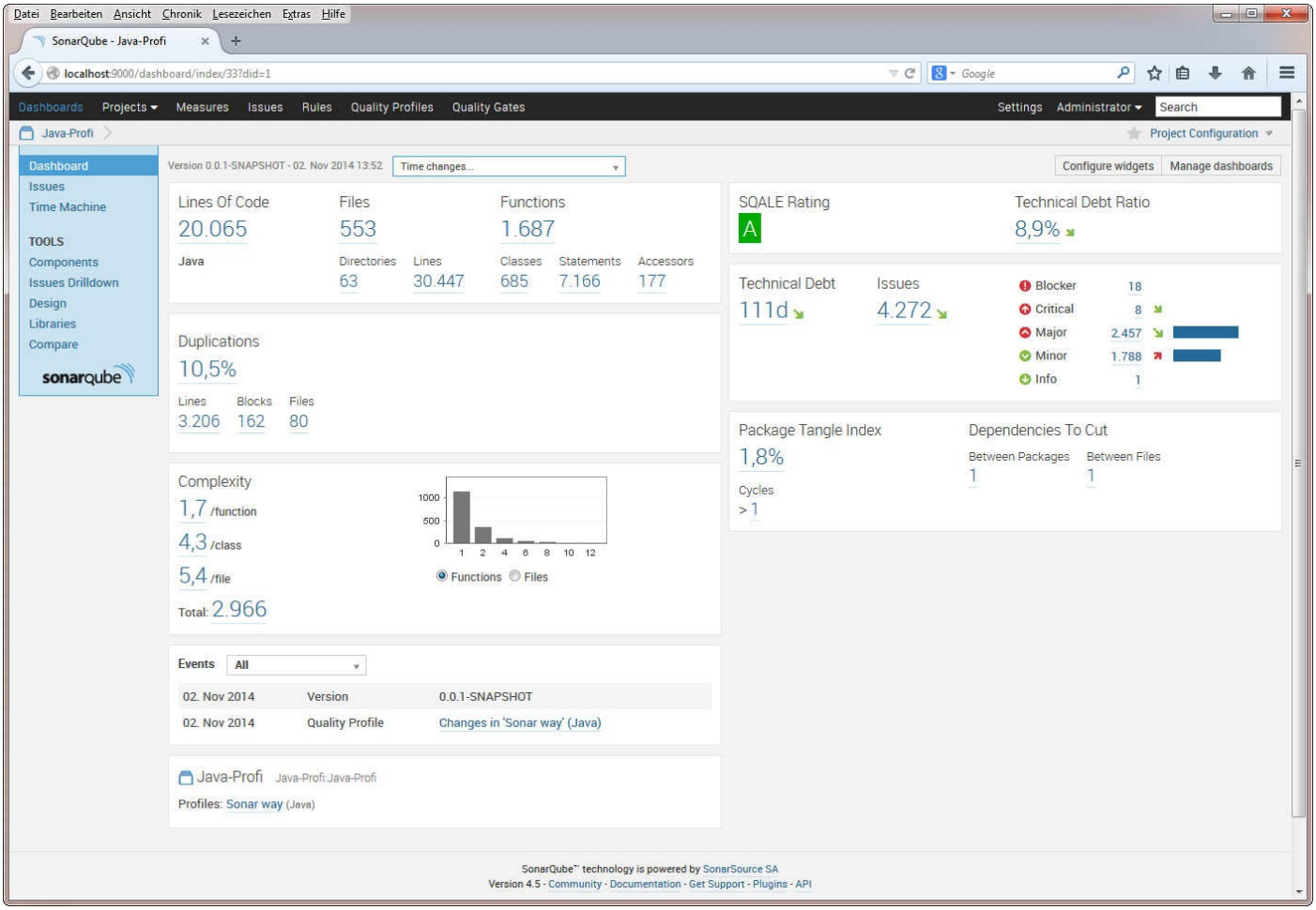


Abbildung 19-7 *SonarQube-Dashboard der Sourcen zu diesem Buch*

Installation und Start

SonarQube besteht aus einer Server- und mehreren Client-Bestandteilen. Eine freie Version kann man unter `http://www.sonarqube.org/` als ZIP beziehen. Dieses kann in ein beliebiges Verzeichnis entpackt werden. SonarQube läuft als Client-Server-Applikation. Der Server-Teil befindet sich in einem Unterordner für das jeweilige Betriebssystem im `bin`-Ordner. Für Windows gibt es dort die Datei `StartSonar.bat`. Mit dieser kann man den Server starten. Danach kann man sich die generierten Reports – wie oben angedeutet – mit einem Webbrowser ansehen.

Wenn wir nun unsere Projekte analysieren wollen, benötigen wir dazu den Sonar-Runner oder ein Plugin für unsere IDE. Wir installieren das im Eclipse Marketplace angebotene SonarQube-Plugin. Dann konfigurieren wir die Verbindung zum SonarQube-Server über das Menü PREFERENCES –> SONARQUBE –> SERVERS. Insbesondere sollte man dort den Benutzernamen und das Passwort setzen. Einen Analyselauf startet man über das Kontextmenü SONARQUBE –> ANALYZE oder aber bequem über das Tastaturkürzel CTRL + ALT + Q.

Historisierung

Bei der Analyse der Qualität interessiert neben dem Ist-Stand vor allem der zeitliche Verlauf, also an welchen Stellen hat sich die Qualität verbessert und wo eventuell sogar verschlechtert und wie ist der Trend.

Dazu kann man ein Projekt einfach mehrmals im Verlaufe eines Monats oder eines Jahres analysieren, z. B. auch idealerweise im Nightly Build. Für die zuvor beschriebenen Tools müsste man dann selbst Statistik führen. SonarQube hilft dabei, Trends zu bilden und Vergleiche mit vorherigen Versionen anzustellen. Dazu stellt SonarQube ein weiteres Dashboard namens Time Machine bereit, wo sich die Daten leicht miteinander vergleichen und insbesondere Trends erkennen lassen.

20 Unit Tests

Dieses Kapitel stellt das Themengebiet Testen vor und geht insbesondere auf Unit Tests ein. Abschnitt 20.1 gibt einen kurzen Überblick über das Testen im Allgemeinen sowie über verschiedene Arten von Tests. Zudem werden deren Auswirkungen auf die Qualität dargestellt. In Abschnitt 20.2 beschäftigen wir uns etwas intensiver mit dem Erstellen von Testfällen. Nach diesem Einstieg zeige ich in Abschnitt 20.3 anhand von zwei Beispielen den Einsatz und die Motivation für Unit Tests aus der Praxis: zum einen beim Entwurf bzw. der Erweiterung einer Softwarekomponente und zum anderen bei der Überarbeitung von vorhandenem, älterem Sourcecode, sogenanntem ***Legacy-Code***. Grundlegende Techniken und die Motivation zum Testen sind damit vorgestellt. Im realen Leben sind aber häufig kompliziertere Gebilde zu testen. Abschnitt 20.4 beschreibt sogenannte Test-Doubles zum Lösen von Abhängigkeiten. Wir lernen Stubs und Mocks als dabei hilfreiche Techniken kennen. Zudem schauen wir uns kurz Möglichkeiten an, private Methoden testbar zu machen. Anschließend zeigt Abschnitt 20.5 anhand eines Beispiels den Einsatz von Unit Tests bei Multithreading. Danach betrachten wir in Abschnitt 20.6 verschiedene Fallstricke beim Formulieren von Unit Tests, sogenannte Test Smells. Ergänzend stelle ich in Abschnitt 20.7 die seit JUnit 4.7 enthalten Rules vor und gehe auch auf sogenannte parametrierte Tests ein. Das sind Tests, die mit variierender Parametrierung ausgeführt werden. Abschließend beleuchtet Abschnitt 20.8 einige nützliche Tools, deren Einsatz das Unit-Testen erleichtern kann.

20.1 Testen im Überblick

Nachfolgend gebe ich eine Einführung in das Thema Testen. Ich beschreibe zunächst kurz verschiedene Arten von Tests, deren beteiligte Akteure und die Auswirkungen der Tests auf die Softwarequalität. Dabei erläutere ich die Begriffe Blackbox- und Whitebox-Tests sowie das Unit-Testen. Im Anschluss widmen wir uns zum Einstieg ganz allgemein der Thematik des Testens und der Qualität.

20.1.1 Was versteht man unter Testen?

Unter ***Testen*** versteht man den Vorgang, das tatsächliche Verhalten des Programms oder eines Teils davon (Ist) mit dem geforderten Verhalten (Soll) zu vergleichen. Demnach entspricht Testen also nicht dem einmaligen Start eines Programms mit ein paar willkürlichen Bedienhandlungen. Es geht vielmehr darum, das Verhalten der Software unter verschiedenen Bedingungen zu prüfen. Dabei muss man fast ein wenig »bösartig« agieren, um versteckte Probleme aufdecken oder Fehlverhalten provozieren zu können. Dazu gehört beispielsweise eine bewusste Fehlbedienung,

etwa die Eingabe ungültiger Werte sowie von Extrem- oder Randwerten. Unter einem ***Fehler***[1] verstehen wir, wenn es beim Testen zu Abweichungen zwischen produzierten Ergebnissen und gewünschten Resultaten kommt, sich das geprüfte Softwaresystem also anders als spezifiziert verhält oder gar abstürzt – weniger formal: wenn die gelieferten Ergebnisse nicht den Bedürfnissen oder Anforderungen der Benutzer entsprechen. ***Testen dient somit der Sicherstellung von Softwarequalität***. Je besser die Anforderungen bekannt und spezifiziert sind, desto einfacher können die Tests definiert werden. Denn dann lassen sich produzierte Ergebnisse leichter mit den gelieferten vergleichen. Damit eine Aussage über die Qualität oder das Testergebnis möglich wird, muss der Vorgang des Testens systematisch erfolgen. Idealerweise werden in einem Testdokument gegebene Rahmenbedingungen, etwa zu nutzende Eingabewerte, auszuführende Aktionen und erwartete Ergebnisse, festgehalten.

Einige Tests (auch Testfälle genannt) lassen sich im Idealfall basierend auf einem Spezifikationsdokument erstellen. Hier spielen dann die frühen Phasen des Entwurfs mit den späteren zusammen. Offensichtlich profitiert man also nicht nur beim Implementieren, sondern auch (später) beim Testen von einer möglichst guten Spezifikation und der gründlichen Arbeit beim Requirements Engineering. Testfälle kann man dann als eine Art »ausführbare Spezifikation« betrachten.

Hinweis: Testen erfordert Kenntnisse der Anforderungen

Um eine qualitative und quantitative Aussage über die Qualität und umgesetzte Funktionalität zu ermöglichen, hilft es, die Erwartungen an das entstehende Softwaresystem so weit zu spezifizieren, dass daraus Testfälle erstellt werden können. Außerdem ist es hilfreich, nicht funktionale Anforderungen, etwa »Antwortzeiten kleiner 2 Sekunden«, als zu erreichende Ziele festzulegen.

Diese einleitenden Gedanken zum Testen möchte ich mit einem schönen Zitat von Edsger W. Dijkstra aus »The Humble Programmer, ACM Turing Lecture, 1972« [20][2] beschließen: »Durch Testen kann man stets nur die Anwesenheit, nie aber die Abwesenheit von Fehlern beweisen.«

20.1.2 Testarten im Überblick

Um ein wenig mehr Klarheit in den weit gefassten Begriff des Testens zu bringen, ist es sinnvoll, die verschiedenen Arten von Tests und die Akteure kurz vorzustellen.

Begrifflichkeiten beim Testen

Für Entwickler sind vor allem ***Tests einzelner Programmbausteine*** in Form von ***Unit Tests*** oder ***Komponententests*** interessant sowie das Prüfen auf ein fehlerfreies Zusammenwirken von Systemkomponenten, ***Integrationstest*** genannt. Nach

erfolgreichen Tests in der Entwicklung wird die Programmfunktionalität durch sogenannte ***Systemtests*** geprüft. Oftmals wird dazu die Software – oder je nach Umfang auch nur ein funktional in sich abgeschlossener Teil – an eine dedizierte Testabteilung mit Testingenieuren übergeben.[3] Der Begriff des ***Applikationstests*** ist etwas weiter gefasst: Hierbei wird das Gesamtsystem unter verschiedenen Aspekten aus Anwendersicht (auch ***Domain*** oder fachliche Ebene genannt) untersucht. Zuletzt erfolgt gewöhnlich ein ***Abnahmetest*** oder ***Akzeptanztest***, der in etwa eine Wiederholung des Applikationstests durch den oder unter Mitwirkung des Kunden ist. Abbildung 20-1 visualisiert die zuvor beschriebene Testhierarchie und zeigt dabei sowohl einige Eigenschaften der jeweiligen Tests als auch die ausführenden Personen.

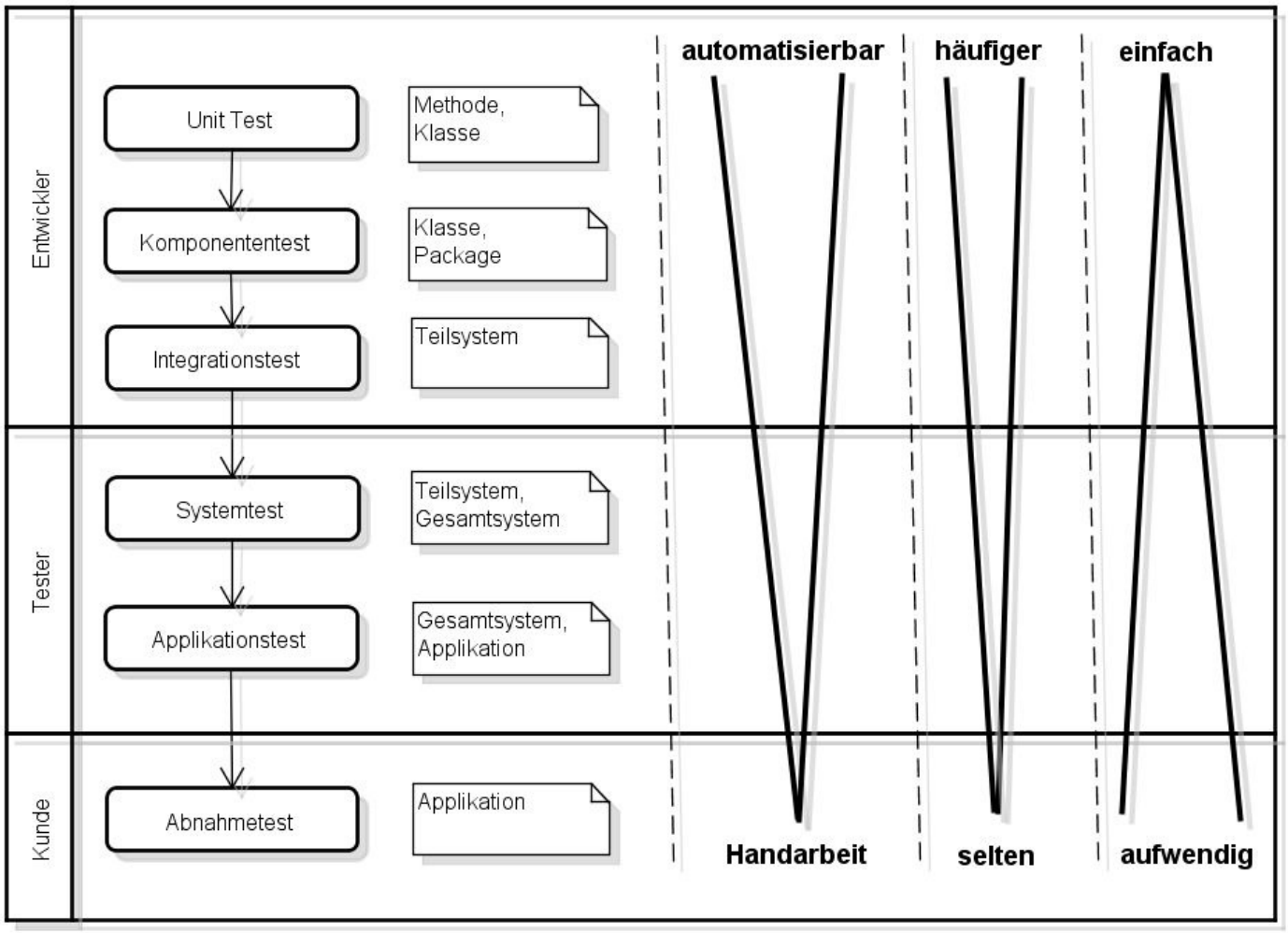


Abbildung 20-1 *Übersicht über die Arten von Tests*

Blackbox- vs. Whitebox-Tests

Beobachten und testen wir die Interaktion mit dem System von außen, so spricht man von einem sogenannten »***Blackbox-Test***« — die »Innereien« bleiben verborgen und man interessiert sich lediglich für gelieferte Resultate, das beobachtbare Verhalten. Dieses Vorgehen kann allerdings Fehler verschleiern: Das gilt etwa, wenn zwei Programmteile beide fehlerhaft sind und sich die Effekte gegenseitig aufheben.[4] Man spricht auch von ***Fehlermaskierung***. Um so etwas möglichst ausschließen und genauere Aussagen über die korrekte Funktion der einzelnen Bestandteile eines

Softwaresystems machen zu können, sind weitere Formen des Testens sinnvoll, die auch das Innenleben der Software berücksichtigen. Man spricht dann von einem sogenannten »***Whitebox-Test***«. Abbildung 20-2 stellt dies schematisch dar.

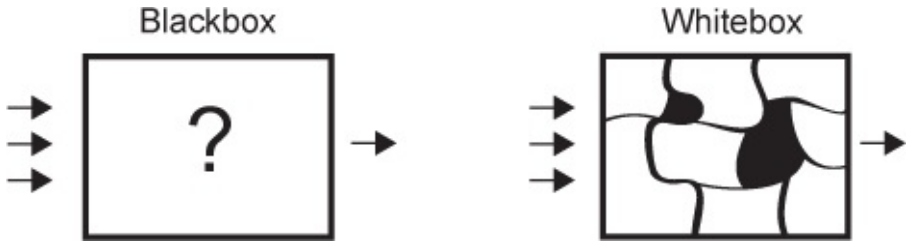


Abbildung 20-2 *Blackbox- und Whitebox-Test*

Bitte bedenken Sie, dass zum Erreichen einer guten Qualität gewöhnlich beide Arten des Testens benötigt werden! Auf Ebene von Applikationstests nutzt man eher das Blackbox-Testen. Manchmal möchte man mehr das Innenleben prüfen. Ergänzend kommt das Whitebox-Testen zum Einsatz, insbesondere bei feingranularen Unit Tests.

Konkretisierung des Begriffs des Unit-Testens

Dem Namen entsprechend, wird beim Unit-Testen eine kleine Einheit (Unit) betrachtet. Ziel dabei ist es, einen einzelnen Baustein des Programms separat und isoliert vom Rest des Gesamtsystems auf korrekte Funktionalität zu prüfen, also möglichst ohne Effekte und Auswirkungen durch das Zusammenspiel mit anderen Komponenten.

Die Testfälle werden als Java-Programme implementiert. Das erlaubt es Entwicklern, ihre Stärken in der Programmierung zu nutzen. Bei moderner Softwareentwicklung sollte das Testen ein elementarer Teil der Entwicklungsarbeit sein, gemäß dem Motto »Code a Little, Test a Little«. So verfährt man auch bei den Programmierparadigmen ***Extreme Programming*** (XP) und ***Test-Driven Development*** (TDD). In jedem Fall ist es das Ziel, beide Tätigkeiten eng miteinander zu verknüpfen und Testfälle in Form von kleinen Unit Tests parallel zum Programmieren entstehen zu lassen. Beim TDD werden erst Unit Tests erstellt, bevor der Applikationscode entwickelt wird.

Zusammenspiel von Unit Tests und Integrationstests

Unit Tests können die Aufwände für Integrationstests senken, da viele, idealerweise alle Komponenten für sich alleine getestet sind. Daher müssen die Integrationstests hauptsächlich noch die Schnittstellen und die Kommunikation sowie die Benutzung gemeinsamer Ressourcen der zu integrierenden Komponenten prüfen. Das sollte man allerdings nicht unterschätzen, denn gerade bei der Interaktion zwischen verschiedenen Komponenten treten des Öfteren noch unerwartet Fehler auf, etwa dadurch, dass zeitliche Randbedingungen oder andere Implementierungsdetails zum Tragen kommen. ***Da die Überprüfung solcher Interaktionsprobleme per Definition***

nicht Bestandteil von Unit Tests ist, müssen zusätzlich immer Integrationstests erfolgen.

Ein Fehlschlagen von Integrationstests kann nicht nur durch das fehlerhafte Zusammenspiel von Komponenten, sondern auch durch verbliebene Implementierungsprobleme in einzelnen Programmteilen verursacht werden: Möglicherweise fallen erst beim Test der Interaktion Probleme auf, die durch Unit Tests nicht abgedeckt werden. In solchen Fällen sollten die Unit Tests um Testfälle für genau diese Probleme erweitert werden. Insgesamt gilt: ***Mit Integrationstests sollte man erst dann beginnen, wenn die jeweiligen Komponenten eine ausreichend gute Qualität besitzen.***

20.1.3 Zuständigkeiten beim Testen

Immer mal wieder hört man die Frage, ob Entwickler ihre Programme selbst testen sollten.[5] Als Gegenargument wird eine mögliche ***Betriebsblindheit*** oder Voreingenommenheit des Erstellers ins Feld geführt. Diese Bedenken sind grundsätzlich richtig, aber nichtsdestotrotz ist es sinnvoll, wenn Entwickler die von ihnen programmierte Funktionalität selbst prüfen, jedoch kommt es dabei auf die richtige Form an: Für Applikationstests halte ich es für sinnvoll, diese von dedizierten Testern durchführen zu lassen (siehe folgenden Meinungskasten »Applikationstests durch dedizierte Tester bzw. Testabteilung«). Zuvor sollten Entwickler jedoch mit Unit Tests sowie Integrationstests ihrer Komponenten eine gute Softwarequalität absichern. Unprofessionell ist dagegen, Testen als lästiges Übel anzusehen und nach dem von Jon Bentley [6] als *falsch* erkannten Motto zu verfahren: »Schreibe deinen Code, […] überlasse die Fehler der Qualitätskontrolle oder Testabteilung.« Vielmehr sollte man als Entwickler das ***Testen als Chance verstehen, qualitativ hochwertigen Sourcecode zu schreiben.*** Das klingt vielleicht zunächst komisch, aber bedenken Sie, dass Sie durch eine gute Qualität Ihrer Implementierungen viel weniger Zeit in die Fehlersuche und anschließende Behebung stecken müssen. Tatsächlich werden Sie durch entwicklungsbegleitendes Testen mehr programmieren können und deutlich seltener über Probleme brüten oder in Stress geraten, wenn Sie einen Fehler finden und fixen sollen. Das gilt allerdings nur, wenn Sie die richtigen und wichtigen Dinge testen, nämlich etwa komplexe Abläufe und Berechnungen, nicht aber Trivialitäten wie Zugriffsmethoden.

Zuständigkeiten bei Applikationstests

Manchmal beruht die Aversion einiger Entwickler gegen das Testen vermutlich darauf, dass Projektleiter von den Entwicklern fordern, umfangreiche Applikationstests, also Tests aus Anwendersicht und auf fachlicher Ebene, durchzuführen. Das wird mitunter als unangenehm und langwierig empfunden, weil zur Ausführung von Applikationstests oftmals spezielle, aufwendige Konfigurationen

erforderlich sind, um gewisse Rahmenbedingungen zu schaffen. Weiterhin sind komplexe Testfälle, die möglicherweise auch einiges an Fachwissen benötigen, abzuprüfen. Es entsteht schnell eine gewisse Abneigung gegen diese Art des Testens, wenn das erforderliche Know-how oder die Zeit zur Einarbeitung fehlt. Das trifft für Entwickler in der Regel zu, da deren Hauptfokus auf dem Entwickeln und dem entwicklungsbegleitenden Unit-Testen liegt.

Meinung: Applikationstests durch dedizierte Tester bzw. Testabteilung

Während agile Vorgehensmodelle propagieren, dass auch System- und Applikationstests vom Entwicklungsteam ausgeführt werden sollten, habe ich dazu eine leicht andere Meinung. Ich persönlich sehe das Ganze eher etwas skeptisch, da jeder Mensch gewisse Stärken und Neigungen hat und somit nicht jeder Entwickler auch ein wirklich guter Tester ist. Im wirklichen Leben ist ein guter Koch ja auch noch lange kein geeigneter Restaurantkritiker.

Meiner Meinung nach sollten Applikationstests bevorzugt von Testern einer Testabteilung[a] durchgeführt werden, die einerseits in der Regel viel Erfahrung beim Konfigurieren und Testen besitzen sowie andererseits die nötige Zeit für eine gründliche Durchführung haben. Auch profitiert man von einem kritischen, unabhängigen Blick.

[a] Wichtig ist eigentlich nur, dass es ein dedizierter Tester ist.

Zuständigkeiten bei Unit Tests

Es ist wünschenswert, Fehler möglichst schnell, noch während der Entwicklung aufzudecken, um die System- und Applikationstests zu erleichtern und kaum mit (einfachen) Fehlern zu behindern. Dabei helfen Unit Tests und Integrationstests. Damit können Flüchtigkeitsfehler oder Fehler in der Funktionalität parallel zur Entwicklung erkannt und meistens mit wenig Aufwand behoben werden. Der Grund ist einfach: Der testende Entwickler besitzt ein gutes Kontextwissen sowie ein tiefes Know-how über seine Implementierung. Zudem kann durch Unit Tests ein Sicherheitsnetz aufgebaut werden, das den Entwickler vor dem erneuten Einbau desselben Fehlers schützt.[6] Man spricht dann von Regression. Durch die Existenz und wiederholte, möglichst häufige Ausführung von Unit Tests profitiert man somit besonders bei Weiterentwicklungen oder Korrekturen.

Problem Betriebsblindheit

Kommen wir nun auf das Thema Betriebsblindheit zurück. Natürlich ist diese gegeben und ein Entwickler wird zuvor bei der Implementierung begangene eigene Denkfehler wahrscheinlich auch beim Schreiben der Unit Tests machen und somit gewisse Fehler eventuell nicht entdecken können. Dagegen können zwei Maßnahmen

helfen: Zum einen kann man dem durch Codereviews des zu testenden Systemteils sowie der zugehörigen Unit Tests entgegenwirken. Zum anderen sollten neben den Unit Tests ergänzende Systemtests von einer anderen Person (oder einer Testabteilung) durchgeführt werden, um die Funktionalität auf einem höheren Abstraktionsniveau und im Zusammenspiel mit anderen Systemkomponenten zu überprüfen.

20.1.4 Testen und Qualität

Wenn wir Software nutzen, dann erfreuen wir uns an guter Qualität. Aber was macht eigentlich Qualität aus? Sicherlich spielen dabei Eigenschaften wie Zuverlässigkeit, Benutzerfreundlichkeit, Performance usw. eine wichtige Rolle, insbesondere für den späteren Anwender. Darüber hinaus kann ein Softwaresystem sich durch klares und verständliches Design, gute Wartbarkeit und Erweiterbarkeit sowie viele, automatisch ausgeführte Tests und damit eine relative große Sicherheit über die korrekte Funktionalität auszeichnen. Letztere Punkte sind gerade für uns als Softwareentwickler von gesteigertem Interesse, weil sie uns das Leben einfacher machen und Probleme und Ärger zu vermeiden helfen. Mit diesem Vorwissen blicken wir auf zwei Arten von Qualität.

Äußere vs. innere Qualität

Die sogenannte ***innere Qualität***, oder auch technische Softwarequalität, hat durch die Programmierparadigmen XP und TDD und die damit verbundenen Unit Tests in den letzten Jahren verstärkt Aufmerksamkeit erfahren. Zuvor wurden Tests häufig nur auf System- oder Applikationsebene und meistens auch erst nach Abschluss der Entwicklung durchgeführt. Diese Tests überprüfen jedoch lediglich das nach außen sichtbare Verhalten, die sogenannte ***äußere Qualität***, die unter anderem die Aspekte korrekte Funktionalität, Benutzbarkeit sowie Performance umfasst.

Normalerweise führt eine gute innere Qualität auch zu einer guten äußeren Qualität. Der Umkehrschluss gilt jedoch nicht: Es gibt Systeme, die mit ungeheurem Aufwand getestet und überarbeitet werden (müssen), bis eine akzeptable äußere Qualität wahrnehmbar ist – intern wird diese Qualität allerdings nicht erreicht. Mit zunehmender Lebenszeit eines solchen Produkts wirkt sich dessen schlechte innere Qualität immer stärker negativ aus. Erweiterungen sind dann nur noch unter großen Anstrengungen und mit enormem Aufwand zu realisieren. Dagegen erlaubt es eine gute innere Qualität, Erweiterungen in der Regel einfach umzusetzen, die zudem kaum zu Folgeänderungen in anderen Programmteilen führen.

Anfangs kostet es sicher einiges an Mehraufwand, Qualitätssicherungsmaßnahmen zu etablieren. Diese Mühe wird aber mittel- bis langfristig belohnt. Der Einsatz wirkt sich extrem positiv auf die innere und äußere Qualität der Software aus. Die gestiegene innere Qualität kann zu deutlichen Kosteneinsparungen führen, da sich

der Aufwand für System- und Applikationstests verringert und zudem weniger Fehler beim Kunden auftreten, wodurch wiederum die Kundenzufriedenheit steigt und es zu weniger Servicefällen kommt. Außerdem lassen sich bei Bedarf Erweiterungen leichter realisieren.

Qualitätsansprüche

Unter Qualität versteht jeder leicht etwas anderes. Für den einen ist eine gute Benutzbarkeit wichtig, ein anderer legt den Fokus auf umfangreiche Funktionalität. Immer jedoch wird man sich ein hohes Maß an Erwartungskonformität und Zuverlässigkeit wünschen. Was meine ich damit? Im richtigen Leben erwarten wir fast immer eine Qualität von nahezu 90 bis 100 %. Einen Kugelschreiber, der nur ab und zu schreibt, werfen wir weg. Eine Leiter, die bis 30 Kilo zugelassen ist, kaufen wir vermutlich gar nicht erst. Oder eine Hängebrücke, an der ein Warnschild hängt, dass die Seile alt und wenig zuverlässig sind, überqueren wir wohl nicht freiwillig, sondern nur in allergrößter Not.

Wieso geben wir uns bei Software oftmals mit nicht immer ausreichender Qualität zufrieden? Was könnten die Ursachen für die Qualitätsprobleme sein?

- Die Komplexität ist in Software häufig recht hoch.
- Die Einzelteile sind zum Teil nicht gut getestet.
- Die Informatik hat (leider) noch nicht den Ingenieursgrad wie die Autoindustrie oder der Maschinenbau erreicht.

Einfluss der Qualität der Einzelteile

Die Industrie besitzt Normen für z. B. Schrauben, Muttern und Gewinde. Weil es eben Normen sind, variieren diese nicht von Hersteller zu Hersteller, sondern man kann sich darauf verlassen, dass eine durch die Norm geforderte Qualität sowie Interoperabilität gegeben ist. Weiterhin hat man in der Automobilindustrie beispielsweise Folgendes herausgefunden: Je höher die Qualität der Einzelteile, desto höher ist auch die Qualität des Gesamtprodukts. Um dies zu verstehen, betrachten wir ein einfaches lineares System mit drei Bausteinen (vgl. Abbildung 20-3), die jeweils eine Qualität von 90 % aufweisen.

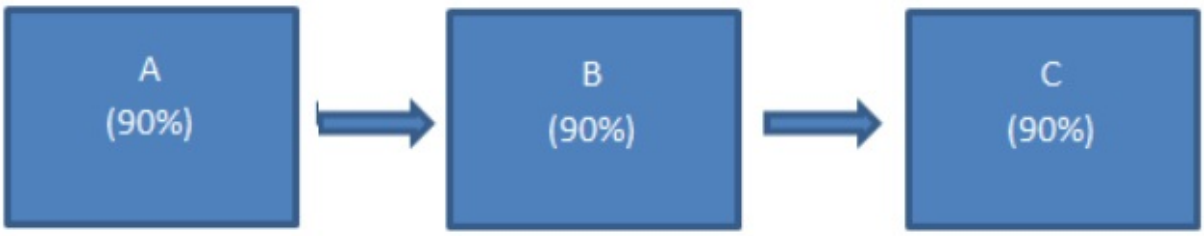


Abbildung 20-3 *Drei voneinander abhängige Systeme mit jeweils 90 % Qualität*

Schätzen Sie mal, wie hoch die Gesamtqualität ist? Auch 90 %? Etwas weniger? Laut der Systemtheorie ergibt sich diese als das Produkt der Einzelqualitäten als:

$$0.9 * 0.9 * 0.9 = 0.73 => 73\ \%$$

Wenn wir das auf Software übertragen, werden folgende Schwierigkeiten deutlich:

1. Wer hat schon einmal Systeme mit nur 3 Klassen oder Bestandteilen gebaut?
2. Mehr noch: Normale Anwendungen bestehen nicht nur aus einer Vielzahl an Klassen und Objekten, sondern diese besitzen insbesondere auch komplizierte, teils verzwickte, nicht lineare Abhängigkeiten untereinander.

Eigenschaften von Unit Tests

Wir haben gerade erkannt, dass die Qualität der Einzelbausteine großen Einfluss auf die Qualität des Gesamtsystems besitzt. Mit Unit Tests betreiben wir also Qualitätssicherung auf Kleinteilebene.

Betrachten wir einige Eigenschaften von Unit Tests, die direkt oder indirekt positive Auswirkungen auf die innere Qualität haben:

- **Klares Ergebnis** – Die Durchführung der Tests liefert ein eindeutiges Ergebnis: bestanden (Grün) oder Fehler (Rot).
- **Messbar** – Die Anzahl der Testfälle sowie die Testabdeckung lässt sich leicht auswerten. Führt man Statistik, kann man ungewünschte Trends erkennen und ihnen durch passende Maßnahmen entgegenwirken.
- **Implementierungsnah und bessere Fokussierung** – Zwischen der Implementierung und den Tests vergeht wenig Zeit. Wir erhalten schnell Feedback und meistens lassen sich Fehler auch leicht korrigieren, weil man sich besser an Details erinnert und es wenig Mühe kostet, sich in den Sourcecode einzudenken. Zudem ist der verursachende Sourcecode eng umrissen. Bei Entwicklungen mit nachfolgenden Funktionstests ist die Fehlersuche um einiges aufwendiger.
- **Know-how-Gewinn** – Beim Schreiben von Unit Tests muss man sich automatisch intensiver mit dem Sourcecode beschäftigen. Es baut sich dadurch kontinuierlich ein besseres Verständnis auf. Änderungen fallen anschließend leichter und lassen sich zudem mit vorhandenen Unit Tests sofort überprüfen. Das führt meistens zu durchdachteren Implementierungen, die dann wiederum weniger fehleranfällig sind und weniger Test- und Wartungsaufwand verursachen.
- **Wiederholbar** – Nach Änderungen können Tests beliebig oft wiederholt werden. Das erhöht die Sicherheit[7], keine Defekte eingefügt zu haben.

Von gut formulierten Unit Tests profitiert man am meisten, wenn man diese regelmäßig ausführt. Diese Form des Testens wird ***Regressionstest*** genannt. Allerdings wäre es aufwendig, wenn man dazu nach jeder kleineren Änderung alle Tests von Hand ausführen müsste. Diesen Prozess automatisiert z. B. das Tool Infinitest (vgl. Abschnitt 20.8.3) innerhalb der IDE. Ebenfalls sollten die Tests

natürlich auch als Teil des automatischen Build-Laufs ausgeführt werden, wie ich es bereits in Abschnitt 2.7 motiviert habe.

Auswirkungen von Unit Tests auf das API-Design

Beim Schreiben von Unit Tests muss man sich auch mit Entwurfsentscheidungen beschäftigen, etwa mit Kohäsion, Kopplung und dem Design des APIs. Durch das Implementieren von Testfällen nutzt man das API eigner Klassen. Dadurch fällt die Beurteilung leichter, ob die angebotenen Schnittstellen sinnvoll und handhabbar sind. ***Unit Tests können also mögliche Schwächen in den von den Tests angesprochenen APIs vor einer Nutzung in anderen Komponenten aufdecken. Durch eine Korrektur erhält man somit gelungenere APIs***. Weiterhin kann man Unit Tests für Entwickler als Dokumentation des erwarteten Programmverhaltens ansehen. Diese Dokumentation ist automatisch immer aktuell, da die Testfälle ansonsten fehlschlagen würden. Das gilt jedoch nur für die mit Tests geprüften Programmteile.

Tipp: Entwicklungsbegleitend oder nach der Implementierung testen?

Man kann sich fragen, warum entwicklungsbegleitendes Testen durch Unit Tests vorteilhafter sein sollte, als Tests erst nach der Implementierung durchzuführen. Tatsächlich entsteht anfangs durch das Schreiben von Tests zusätzlicher Aufwand. Allerdings lassen sich Fehler noch während der Implementierungsphase finden und dadurch meistens relativ leicht beheben. Durch die einfache Wiederholbarkeit und Automatisierbarkeit der Testdurchführung amortisieren sich die Aufwände zum Erstellen der Tests im Vorfeld. Wird das Testen ausschließlich nach Abschluss der Entwicklung durchgeführt, so können Fehler erst spät im Projektzyklus erkannt werden. Es kommt zu größeren Aufwänden während der Integrations-, System- und Applikationstests. Entsprechend teuer ist eine Behebung.

Außerdem gibt es in der Praxis durchaus Situationen, in denen ein Entwickler bereits an einem neuen Projekt arbeitet oder nicht mehr in der Firma beschäftigt ist. Erweiterungen und Korrekturen sind dann aufwendig, da sich ein anderer Entwickler zunächst in den Sourcecode einarbeiten muss. Zudem fehlt die Sicherheit, bei Änderungen keine Fehler zu machen. Wird das Testen jedoch als integraler Bestandteil der Entwicklung angesehen, so hinterlässt ein ausscheidender Entwickler zumindest einen Teil seines Know-hows in Form von Testfällen. Dadurch können andere Entwickler Erweiterungen mit größerer Sicherheit vornehmen.

20.2 Wissenswertes zu Testfällen

Nachdem wir bisher eher einen allgemeinen Blick auf das Testen geworfen haben, wollen wir uns kurz mit Test-Driven Development (TDD) befassen und danach

Testfälle mit dem etablierten Test-Framework JUnit formulieren.[8] Einführende Informationen zu JUnit 4 finden Sie in Abschnitt 2.4.

20.2.1 Test-Driven Development (TDD) im Überblick

Kent Beck stellt in seinem Buch »Test-Driven Development« [5] die testgetriebene Entwicklung vor. Die Kernidee dabei ist es, vor dem Implementieren der Anwendungsfunktionalität mit dem Schreiben von Testfällen zu beginnen. Erst nachdem mindestens ein Test entworfen ist, wird mit der Entwicklung der eigentlichen Programmfunktionalität begonnen. Um TDD in der Praxis einsetzen zu können, ist ein Unit-Test-Framework Voraussetzung, das die automatisierte Ausführung der Testfälle und deren Auswertung ermöglicht. Dieses prüft, ob alle vorhandenen Tests erfolgreich durchlaufen werden (Grün) oder ob dabei ein Fehler auftritt (Rot).

Beim Entwickeln gemäß TDD sollen sich Programmierer auf möglichst kleine Schritte beschränken, um lesbaren, korrekten und gut getesteten Sourcecode zu schreiben. Die Entwicklung erfolgt dazu iterativ immer in folgenden drei einfachen Schritten:

1. **Rot** – Schreibe einen kleinen Test. Dieser darf anfangs fehlschlagen und lässt sich häufig nicht übersetzen, wenn beispielsweise die Implementierung der zu testenden Klasse oder Methode noch fehlt.
2. **Grün** – Sorge dafür, dass der Test so schnell wie möglich erfolgreich bestanden wird. TDD erlaubt dazu auch einige »Programmiersünden«. Dieses Vorgehen halte ich allerdings für gefährlich und übertriebenen Aktionismus. ***Meiner Meinung nach sollte die Maxime besser lauten: Sorge mit der einfachst möglichen, sauberen Lösung für einen funktionierenden Test.***[9]
3. **Refactoring** – Eliminiere duplizierten und unschönen Sourcecode, der im zweiten Schritt eingeführt wurde.

Durch den Test-First-Ansatz sind selbst bei Projektstress immer bereits Tests vorhanden und das Testen wird nicht in der Projekthektik vernachlässigt. Es kann gar nicht erst zu einem Teufelskreis kommen, indem durch erhöhten Stress weniger getestet wird und so noch mehr Stress entsteht, wodurch wiederum weniger Zeit für Tests verbleibt. Droht eine solche Situation, so können beim Vorgehen nach TDD einfach die bereits vorhandenen Tests durchgeführt werden. Im Idealfall bekommt man unmittelbar die Rückmeldung, dass alles noch funktioniert. Ein solches »Sicherheitsnetz« aus Unit Tests sorgt für mehr Ruhe und Gelassenheit, wodurch sich automatisch die Gefahr von stressbedingten Flüchtigkeitsfehlern reduziert.

Testgetriebene Entwicklung bezieht sich allerdings nur auf solche Tests, die Entwickler selbst schreiben und durchführen, d. h. auf Unit Tests. Systemtests durch ein Testteam bzw. weiter gehende Applikationstests sind nicht Bestandteil der eigentlichen TDD-Prozesse, aber auch nicht ausgeschlossen.

Meinung: Vorgehen beim Erstellen von Tests

TDD propagiert, zuerst die Tests und dann die Funktionalität zu schreiben und in kurzen Zyklen zwischen Test und Entwicklung hin- und herzuwechseln.

Mitunter kann man so neue Funktionalität oder Erweiterungen sehr gut in kleinen sicheren Schritten vorantreiben. Ein Beispiel dafür habe ich in Abschnitt 17.1 gezeigt.

Wenn man jedoch das Design schon gut durchdacht hat und das Ziel klar ist sowie die Implementierung nur so »aus den Fingern« fließt, finde ich den Ansatz nicht zielführend, da ich gerne eine Funktionalität in einem Schwung und einer Konzentrationsphase, einem sogenannten ***Flow***, abarbeite und mich die ständigen Kontextwechsel dann stören. Ich bevorzuge stattdessen, eine gewisse Zeit an der eigentlichen Funktionalität bzw. an den Tests am Stück zu arbeiten, wobei die Konzentrationsphasen einmal 15 oder 30 Minuten sein können, selten aber länger als 1 Stunde.

Wichtig ist, dass Sie die Tests in ausreichender Zahl und für die richtigen Dinge schreiben, d. h. insbesondere für die Programmstellen, die schwierig, undurchsichtig und potenziell fehleranfällig sind oder für die bereits Fehler gemeldet wurden. Außerdem sollten Sie iterativ und inkrementell arbeiten, also immer in kleinen überschaubaren Schritten und diese dann durch Tests prüfen, bevor Sie an etwas Neues gehen.

20.2.2 Testfälle mit JUnit 4 definieren

Bekanntermaßen prüft man mit Unit Tests kleine Bausteine, meistens einzelne Klassen, eines Softwaresystems. Ebenso haben wir gelernt, dass man durch das Schreiben von Tests dazu angehalten ist, sich Gedanken über die gewünschte zu implementierende Funktionalität zu machen. Darüber hinaus helfen Unit Tests dabei, Änderungen abzusichern, weil immer geprüft wird, dass die gewünschte Funktionalität auch weiterhin vorhanden ist. Schauen wir uns nun an, wie wir entsprechende Testfälle realisieren.

Erstellen von Testfällen

Die nachfolgend gezeigte Klasse `MyClass` dient lediglich zur Demonstration erster Schritte beim Erstellen von Testfällen und folgt keinem allzu guten OO-Design: Die Methode `calc()` hat eigentlich nichts mit der Klasse gemeinsam und ist eher eine Hilfsmethode, die statisch definiert sein könnte. Insgesamt findet man hier eine schwache Kohäsion – zur Einführung in das Testen soll uns das jedoch nicht stören.

```
public class MyClass
{
    /* private */ static final int BASE = 10_000;
```

```
    private final int value;

    public MyClass(final int offset)
    {
        this.value = BASE + offset;
    }

    public int getValue()
    {
        return value;
    }

    public int calc(final List<Integer> values)
    {
        Objects.requireNonNull(values, "parameter 'values' must not be null");

        int sum = 0;
        for (final int currentValue : values)
        {
            sum += currentValue;
        }
        return sum;
    }
}
```

Für die obige Implementierung können wir uns mit Eclipse eine passende Testklasse erstellen lassen. Dazu wählen wir FILE –> NEW –> JUNIT TEST CASE. Es erscheint ein Dialog. Wenn man dort die beiden Methoden `getValue()` und `calc(List<Integer>)` anwählt, entsteht für jede der Methoden der Klasse `MyClass` eine Testmethode und es wird folgende initiale Implementierung der Testfälle erstellt:

```
import static org.junit.Assert.*;

import org.junit.Test;

public class MyClassTest
{
    @Test
    public void testGetValue()
    {
        fail("Not yet implemented");
    }

    @Test
    public void testCalc()
    {
        fail("Not yet implemented");
    }
}
```

Wie man leicht sieht, unterscheidet sich eine Testklasse von einer normalen Klasse darin, dass es keine `main()`-Methode gibt und die von JUnit auszuführenden Testmethoden mit der Annotation `@Test` gekennzeichnet sind und dort verschiedene Bedingungen über den erwarteten Zustand mithilfe spezifischer Methoden geprüft werden. Hier wird über `fail()` nur signalisiert, dass der Testfall noch zu implementieren ist.

Um mit dem Handling vertraut zu werden, können Sie diesen Unit Test in Eclipse einmal ausführen lassen. Wählen Sie im Kontextmenü RUN AS –> JUNIT TEST oder das Tastaturkürzel ALT+SHIFT+X,T. Die auszuführenden Testmethoden werden vom Framework ermittelt und aufgerufen. Fehler werden nicht durch Rückgabewerte oder Konsolenausgaben angezeigt, sondern durch fehlgeschlagene Asserts, wobei `fail()` eine spezielle Form eines Asserts ist. Abbildung 20-4 zeigt die Ausführung.

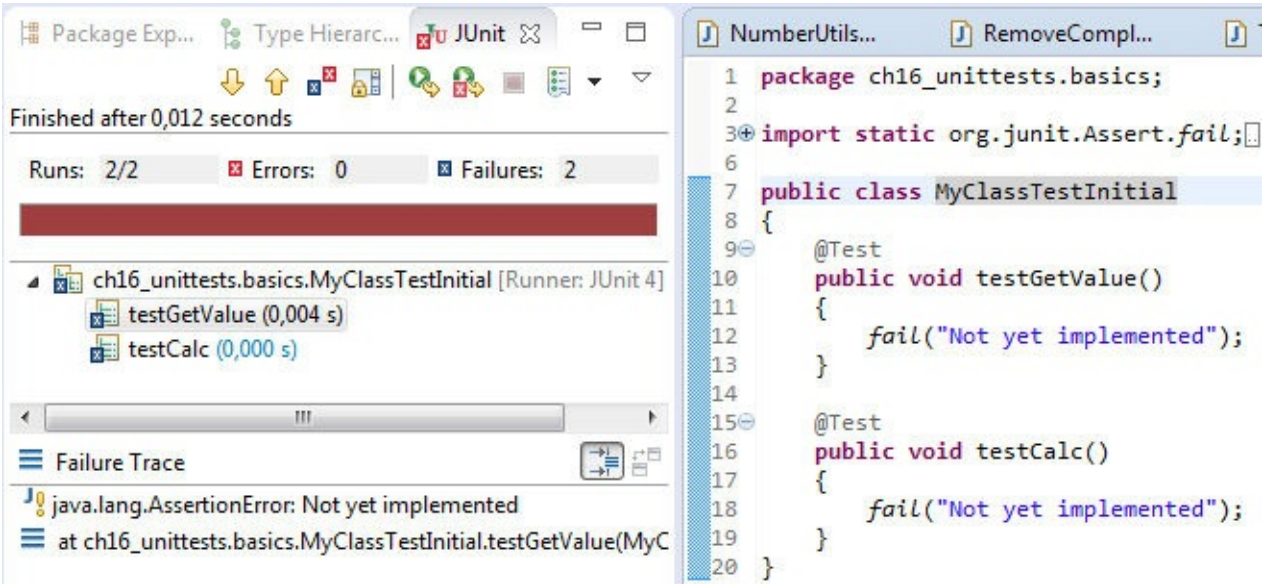


Abbildung 20-4 *Ausführung dieses Unit Test*

Nach ein paar Tipps zum Erstellen von Testfällen machen wir uns an die Implementierung einer sinnvollen Prüfung der Klasse.

Tipps für Klarheit und Verständlichkeit

Um Testfälle wartbar und übersichtlich zu halten, sollte jeder Testfall jeweils nur eine Funktionalität prüfen, wodurch der Testfall automatisch recht kurz und verständlich bleibt. Zudem sollte es idealerweise nur einen Grund geben, warum ein Testfall fehlschlägt. Ebenso wichtig ist es, die Testfälle systematisch zu erstellen und nicht einfach beliebige Methoden zu testen, sondern bevorzugt die »dicken Brocken« oder die potenziell problematischen Teile zu prüfen.

Generell gilt, dass man für alle zu testenden Methoden einer Klasse meistens mehrere Testmethoden implementiert. Diese repräsentieren jeweils einzelne Testfälle. Gewöhnlich erfolgen in jeder Testmethode dazu drei Schritte:

1. Zunächst wird der gewünschte Startzustand, die Ausgangsbasis des Testfalls, auch ***Textfixture*** genannt, hergestellt.
2. Danach folgen dann ein oder selten auch mehrere Aufrufe von Methoden der zu testenden Klasse.
3. Abschließend prüfen wir, ob das erwartete Ergebnis erzielt wurde. Dabei helfen verschiedene Prüfmethoden. In JUnit sind dies die bereits in Abschnitt 2.4 vorgestellten und hier nur kurz aufgelisteten Methoden, die man zur besseren

Lesbarkeit der Testfälle statisch importieren sollte, etwa `assertEquals()`, `assertTrue()` und `assertNotNull()`.

Erweiterung des Beispiels

Wir wollen uns nun daran machen, die initiale Version der Testklasse zu einem wirklichen Unit Test auszubauen und die durch die Klasse `MyClass` implementierte Funktionalität zu prüfen. Wie schon angedeutet, sollte man eher komplexere Methoden testen statt einfache. In unserem Beispiel wollen wir daher die Methode `getValue()` mit nur einem Testfall prüfen und die Methode `calc(List<Integer>)` mithilfe von mehreren Testfällen, wobei sich die Frage stellt, wie und was wollen wir testen? Als erster und einfachster Test wird der »Normalfall« getestet, bei dem eine Liste von Zahlen addiert und anschließend das korrekte Ergebnis geprüft wird. Dann sollten wir uns fragen: Was ist die korrekte Summe, wenn gar keine Zahlen angegeben werden? Für diesen Fall definieren wir als erwartetes Resultat den Wert 0 (zweiter Testfall). Zudem bleibt noch die Möglichkeit, dass an die Methode `calc(List<Integer>)` versehentlich anstelle einer Liste eine `null`-Referenz übergeben wird: Dann erwarten wir eine `NullPointerException`[10]. Auch dieses Verhalten im Fehlerfall wollen wir überprüfen. Dazu dient der dritte Testfall. Gibt es weitere sinnvolle Tests? Wie häufig die Schleife durchlaufen wird, spielt kaum eine Rolle. Allenfalls könnte eine Liste mit nur einem Element interessant sein, auch negative Elemente. Diese Fälle sind aber durch den ersten Testfall adäquat abgedeckt und mit dem Wissen über die Implementierung muss hier keine weitere Prüfung erfolgen. Interessant wäre es sicherlich, einen Überlauf von `int` zu prüfen – welche Fallstricke dort lauern, haben wir bereits in Abschnitt 9.4.1 kennengelernt. Aber hier entscheiden wir uns dafür, dass die drei genannten Testfälle die Methode `calc(List<Integer>)` ausreichend gut testen. Damit ergibt sich in etwa folgende Implementierung der Testklasse:

```
public class MyClassTest
{
    @Test
    public void testGetValue()
    {
        final int base = MyClass.BASE;
        final int offset = 4711;
        final MyClass objectToTest = new MyClass(offset);

        final int result = objectToTest.getValue();

        final int expected = offset + base;
        assertEquals("value should be rebased by " + base, expected, result);
    }

    @Test
    public void testCalcSum_NormalInputs()
    {
        final MyClass objectToTest = createMyClassObjectForTest();
```

```
        final List<Integer> values = Arrays.asList(1, 2, 3, 4, 5, 6);
        final int result = objectToTest.calc(values);

        final int expected = 21;
        assertEquals("sum should be calculated correnty", expected, result);
    }

    @Test
    public void testCalcSum_When_NoValuesGiven()
    {
        final MyClass objectToTest = createMyClassObjectForTest();

        final int result = objectToTest.calc(Collections.emptyList());

        final int expected = 0;
        assertEquals("sum should be zero for empty list", expected, result);
    }

    @Test(expected=NullPointerException.class)
    public void testCalcSum_When_InputIsNull_ThenThrowException()
    {
        final MyClass objectToTest = createMyClassObjectForTest();

        objectToTest.calc(null);
    }

    private MyClass createMyClassObjectForTest()
    {
        final int dummyOffset = 4711;
        return new MyClass(dummyOffset);
    }
}
```

In diesem Beispiel erkennen wir verschiedene Dinge, auf die ich nachfolgend nochmal separat eingehen möchte:

- **Namensgebung** – Die Testfälle besitzen sprechende Methodennamen, die etwas über die Eingabe und die erwarteten Resultate aussagen.
- **AAA-Stil und Ablauf im Testfall** – Die hier gezeigten Testfälle sind immer nach einem ähnlichen Muster implementiert, dass dem sogenannten AAA-Stil (ARRANGE ACT ASSERT) folgt: Zunächst werden Vorbedingungen und Initialisierungen vorgenommen (ARRANGE), danach wird eine Aktion ausgeführt (ACT) und schließlich wird geprüft, dass der erwartete Zustand eingetreten ist (ASSERT).
- **Hilfsmethoden zur Initialisierung und Testfixture** – Oftmals besitzen Testfälle eine immer gleiche Initialisierung. Dafür könnten wir eine Hilfsmethode herausfaktorieren oder eine spezielle mit `@Before` annotierte Set-up-Methode nutzen.

Namensgebung von Testfällen

Wie im vorangegangenen Beispiel bereits angewendet, sollten die Testmethoden bei deren Benennung ebenso wie die Testklassen einem gewissen Schema folgen. Ob

Testmethoden mit dem Kürzel `test` starten sollten oder nicht, kann zu hitzigen Diskussionen führen. Ich habe mich durch JUnit 3 so daran gewöhnt und empfinde es als weiteres Unterscheidungsmerkmal zu Hilfsmethoden in Unit Tests, sodass ich es gerne verwende. Unabhängig davon setzt sich der Methodenname der Testmethode dann aus dem Namen der zu testenden Methode und dem zu prüfenden Sachverhalt sowie dabei geltenden Bedingungen zusammen. Das Ganze wird schon recht lang und hier wird die CamelCase-Notation für Methoden unleserlich. Stattdessen ist es hilfreich, einzelne Namensbestandteile per _ voneinander zu separieren. Roy Osherove beschreibt in seinem Blog[11] eine Variante wie folgt:

```
[UnitOfWork_StateUnderTest_ExpectedBehavior]
```

Man findet auch diese Varianten:

```
MethodName_StateUnderTest_ExpectedBehavior
MethodName_ExpectedBehavior_WhenTheseConditions
```

Besonders wichtig ist, dass der Testkontext gut beschrieben ist, also inklusive erwartetem Verhalten und möglichen Randbedingungen. Zusammen mit meinen Anmerkungen ergeben sich dann in etwa folgende geeignete Methodennamen für Testfälle:

- `calcSum_WithValidInputs_ShouldSumUpAllValues()`
- `calcSum_When_NullInput_Then_ThrowsException()`
- `calcSum_ThrowsException_When_NullInput()`

Gestaltung von Testfällen: GWT- und AAA-Stil

Beim Schreiben unserer Programme haben wir Coding Conventions (vgl. Kapitel 19) kennengelernt, um den Sourcecode besser verständlich und leichter wart- sowie erweiterbar zu halten. Obwohl jeder Testfall in der Regel recht kurz sein sollte, empfiehlt es sich auch beim Implementieren von Testfällen, ein gewisses Standardvorgehen zu befolgen, um lesbare und verständliche Unit Tests zu erstellen.

Dazu hat sich der sogenannte GWT- bzw. AAA-Stil als probates Designmittel etabliert. GWT steht für GIVEN WHEN THEN, AAA steht für ARRANGE ACT ASSERT. Beides meint im Prinzip das Gleiche:

- GIVEN / ARRANGE – Zunächst stellt man die Voraussetzungen her.
- WHEN / ACT – Dann führt man die zu prüfende Aktion des Testfalls aus.
- THEN / ASSERT – Abschließend prüft man, ob die erwarteten Ergebnisse mit den berechneten Werten übereinstimmen.

Betrachten wir ein Beispiel, um die beschriebene Vorgehensweise besser zu verstehen:

```
// GIVEN: An empty list
final List<String> names = new ArrayList<>();

// WHEN: 2 elements are added
names.add("Tim");
```

```
names.add("Mike");

// THEN: List should contain 2 elements
assert("List should contain 2 elements", 2, names.size());
```

Je nach Komplexität des Testfalls kann man neben der einfachen Markierung durch Kommentare wie `// GIVEN` usw. ergänzend auch die Voraussetzung, Aktion und Erwartung aufführen. Eine Markierung mit `// GIVEN`, `// WHEN` und `// THEN` lohnt sich bei einfachen Testfällen nicht sonderlich, weil die Implementierung schon kurz und klar ist. Um sich aber an diese Methodik zu gewöhnen, kann man die Markierung anfangs immer nutzen.

Hilfsmethoden in Unit Tests definieren

Alle Methoden, die nicht mit `@Test` annotiert sind, werden nicht als Testfall ausgeführt. Dadurch kann man in Testklassen auch Hilfsmethoden definieren, die von verschiedenen Testmethoden gemeinsam benötigte Funktionalität implementieren. Somit kann man dann die eigentlichen Testmethoden möglichst kurz und prägnant schreiben. Zur leichteren Trennung von Hilfs- und Testmethoden nutze ich für Testmethoden gerne das Präfix `test` – aber das ist Geschmacksache.

Oftmals benötigt man zur Durchführung der Tests immer wiederkehrende, oft auch aufwendige Initialisierungen, etwa eine konsistente Wertebelegung verschiedener Attribute des zu prüfenden Objekts. Man spricht bei einer Wertebelegung für eine konsistente Testumgebung auch von einer sogenannten ***Testfixture***. Im Beispiel haben wir bereits die Hilfsmethode `createMyClassObjectForTest()` im Einsatz gesehen.

Spezielle Initialisierungen und Aufräumarbeiten

Vor bzw. nach jedem Test werden die mit `@Before` bzw. `@After` annotierten Methoden aufgerufen. Somit kann man Initialisierungen vornehmen bzw. Ressourcen freigeben.

Nachfolgend ist gezeigt, wie sich unser Unit Test durch den Einsatz einer Testfixture verändert – hier auf eine Testmethode beschränkt:

```
public class MyClassTestWithTestFixture
{
    // Testfixture
    private int offset;
    private MyClass objectToTest;

    @Before
    public void initTestFixture()
    {
        offset = 4711;
        objectToTest = new MyClass(offset);
    }

    @Test
```

```
    public void testGetValue()
    {
        final int result = objectToTest.getValue();

        assertEquals("value should be rebased by " + MyClass.BASE,
                     offset + MyClass.BASE, result);
    }
    // …
}
```

Durch den Einsatz einer Testfixture können Initialisierungen, also die ARRANGEAktionen, in Testmethoden entfallen oder aber deutlich kürzer ausfallen.

Sollen Aktionen nur einmalig vor bzw. nach der Durchführung aller Tests ausgeführt werden, so kann man dazu spezielle statische Methoden mit `@BeforeClass` bzw. `@AfterClass` annotieren.

20.2.3 Problem der Komplexität

In diesem Abschnitt wollen wir uns ein grundlegendes Problem beim Schreiben von Tests anschauen, nämlich das Problem der Komplexität bzw. des Umfangs an Eingabedaten. Um die Fehlerfreiheit eines Programms sicherzustellen, müsste man theoretisch alle möglichen Kombinationen von Eingabewerten überprüfen. Der dazu notwendige Aufwand wächst exponenziell. Selbst wenn eine Methode lediglich zwei Eingabewerte vom Typ `int` hätte, wären $2^{32} * 2^{32} = 2^{64}$ Kombinationen auszutesten.[12]

Aus der geführten Argumentation kann man schließen, dass ein Programm niemals (jedenfalls nicht mit vernünftigem und bezahlbarem Aufwand) vollständig getestet werden kann. ***Es gilt daher, diejenigen Testfälle zu bestimmen, die eine gute und möglichst sichere Aussage über die Qualität sowie den Umfang an Funktionalität erlauben.***[13] Dazu muss man sich Strategien überlegen, wie man die riesige Menge an theoretisch zu berücksichtigenden Testkombinationen *drastisch* reduzieren kann. ***Sinnvollerweise konzentriert man sich beim Testen bevorzugt auf die kritischen und komplexen Teile eines Softwaresystems***. Diese sollte man besonders gründlich prüfen und eher triviale Dinge (z. B. Accessor-Methoden) weniger (oder gar nicht) testen. Darüber hinaus kann man auch durch geschickte Wahl von Eingaben möglichst viele gleichartige Wertebelegungen überprüfen, wie wir es im Folgenden sehen werden.

Reduktion des Testaufwands durch Äquivalenzklassentests

Betrachten wir an einem einfachen Beispiel, wie man den Testaufwand auf sinnvolle Weise reduzieren kann, indem man statt einer Vielzahl oder gar aller möglichen Eingaben zum Test lediglich typische Vertreter für Eingabewerte nutzt. Nehmen wir an, eine Methode `calcDiscount(int count)` berechne den Rabatt für eine Bestellung basierend auf der Anzahl der Waren nach folgenden Regeln:

$$Anzahl < 50 \Rightarrow 0\%$$

$$50 <= Anzahl <= 1000 \Rightarrow 4\%$$

$$Anzahl > 1000 \Rightarrow 7\%$$

Schauen wir uns die Implementierung der Methode an, die hier bewusst ein paar Kleinigkeiten falsch realisiert, um auf Probleme aufmerksam machen zu können.

```
// ACHTUNG: Enthält bewusst ein paar kleine Fehler
public int calcDiscount(final int count)
{
    if (count <= 50)
        return 0;
    if (count > 50 && count < 1000)
        return 4;
    if (count > 1000)
        return 7;

    throw new IllegalStateException("programming problem: should never " +
                "reach this line. value " + count + " is not handled!");
}
```

Anhand der drei Gruppierungen können wir drei sogenannte ***Äquivalenzklassen*** bilden. Für diese reicht es, wenn man jeweils nur einen Repräsentanten daraus wählt und diesen prüft. Für die Fälle 0 %, 4 % und 7 % sind in Tabelle 20-1 typische Vertreter als Eingabe angegeben.

Tabelle 20-1 *Testfälle*

Anzahl	Ergebnis
20	0
200	4
2000	7

Optimistisch machen wir uns ans Werk, um die obige Rabattberechnung mithilfe folgender drei Tests abzuprüfen:

```
@Test
public void testCalcDiscount_SmallOrder_NoDiscount()
{
    final int smallAmount = 20;
    assertEquals("no discount", 0, calculator.calcDiscount(smallAmount));
}

@Test
public void testCalcDiscount_MediumOrder_MediumDiscount()
{
```

```
    final int mediumAmount = 200;
    assertEquals("4 % discount", 4, calculator.calcDiscount(mediumAmount));
}

@Test
public void testCalcDiscount_BigOrder_BigDiscount()
{
    final int bigAmount = 2000;
    assertEquals("7 % discount", 7, calculator.calcDiscount(bigAmount));
}
```

Für diese drei Werte wird der Rabatt – wie erwartet – korrekt berechnet. Sind wir damit schon fertig? Nein! Die Erfahrung in der Praxis zeigt, dass ein wenig mehr Testfälle benötigt werden, weil an den Grenzen von Wertebereichen immer wieder Probleme auftreten. Daher prüft man neben den Repräsentanten der Äquivalenzklasse auch noch weitere Repräsentanten, die die Grenzwerte beschreiben. Für natürliche Zahlen ist das recht einfach. Hier bedeutet das, dass auch die direkten Nachbarn unserer Intervallgrenzen, hier die Werte 49, 50, 51 sowie 999, 1000, 1001, zu überprüfen sind, ob diese der richtigen Äquivalenzklasse zugeordnet werden und damit den korrekten Rabatt liefern. Wir ergänzen somit sechs Testmethoden, um diese Randfälle zu prüfen:

```
// ACHTUNG: Hier noch einfache Implementierung mit Magic Numbers

@Test
public void testCalcDiscount_Input_49_Should_Have_NoDiscount()
{
    assertEquals("No discount", 0, calculator.calcDiscount(49));
}

@Test
public void testCalcDiscount_Input_50_Should_Have_MediumDiscount()
{
    assertEquals("Medium discount", 4, calculator.calcDiscount(50));
}

@Test
public void testCalcDiscount_Input_51_Should_Have_MediumDiscount()
{
    assertEquals("Medium discount", 4, calculator.calcDiscount(51));
}

@Test
public void testCalcDiscount_Input_999_Should_Have_MediumDiscount()
{
    assertEquals("Medium discount", 4, calculator.calcDiscount(999));
}

@Test
public void testCalcDiscount_Input_1000_Should_Have_MediumDiscount()
{
    assertEquals("Medium discount", 4, calculator.calcDiscount(1000));
}

@Test
public void testCalcDiscount_Input_1001_Should_Have_BigDiscount()
{
    assertEquals("Big discount", 7, calculator.calcDiscount(1001));
}
```

Beim Betrachten der Testfälle könnten wir uns an den ganzen Magic Numbers stören. Darauf komme ich gleich zurück. Schauen wir zunächst einmal, was passiert, wenn man diese Testfälle ausführt. Dann werden die bewusst integrierten Implementierungsfehler aufgedeckt. Die Werte 50 und 1000 werden nicht korrekt abgebildet. Der Wert 50 wird (versehentlich) noch auf keinen Rabatt abgebildet, was zur Fehlermeldung »`java.lang.AssertionError: Medium discount expected:<4> but was:<0>`« führt. Der Wert 1000 wird durch keinen Fall abgedeckt und wir erhalten eine Exception, mit dem Hinweis: »`java.lang.IllegalStateException: programming problem: should never reach this line. value 1000 is not handled!`«

Basierend auf diesen Fehlermeldungen fällt es nicht schwer, die Implementierung wie folgt zu korrigieren – dabei definieren wir auch gleich passende Konstanten:

```
/* private */ static final int NO_DISCOUNT = 0;
/* private */ static final int MEDIUM_DISCOUNT = 4;
/* private */ static final int BIG_DISCOUNT = 7;

public int calcDiscount(final int count)
{
    if (count < 50)
        return NO_DISCOUNT;
    if (count >= 50 && count <= 1000)
        return MEDIUM_DISCOUNT;
    if (count > 1000)
        return BIG_DISCOUNT;

    throw new IllegalStateException("programming problem: should never " +
                    "reach this line. value " + count + " not handled!");
}
```

Eigentlich sollte das Auslösen einer `IllegalStateException` gar nicht notwendig sein, wenn man alle Pfade vollständig abgedeckt hat. Da hier aber erfahrungsgemäß immer wieder Fehler lauern, ist man mit dem gezeigten Konstrukt auf der sicheren Seite und es lassen sich Flüchtigkeitsfehler leichter aufdecken.

Mithilfe der Konstanten lassen sich auch die Testfälle lesbarer schreiben – hier exemplarisch für einige davon gezeigt:

```
@Test
public void testCalcDiscount_Input_49_Should_Have_NoDiscount()
{
    assertEquals("No discount", NO_DISCOUNT, calculator.calcDiscount(49));
}

// …

@Test
public void testCalcDiscount_Input_1001_Should_Have_BigDiscount()
{
    assertEquals("Big discount", BIG_DISCOUNT, calculator.calcDiscount(1001));
}
```

Als weitere Verbesserung könnte man die Intervallgrenzen auch als Konstanten definieren und für noch mehr Lesbarkeit sorgen, allerdings nimmt dadurch eventuell auch die Verständlichkeit ab, da man immer auf die Definition schauen muss, um den

konkreten Wert zu ermitteln. Je nachdem, ob sich die Prozentwerte oder aber die Grenzen mit größerer Wahrscheinlichkeit ändern, sollte man diese über Konstanten extrahieren oder konfigurierbar machen. In diesem Beispiel ging es aber vornehmlich um das Thema Prüfung von Randfällen, sodass wir den letzten Transformationsschritt nicht mehr vollziehen wollen.

Weitere Prüfungen und Vorbedingungen

Ganz allgemein sollte man zusätzlich noch Über- oder Unterschreitungen von Grenzwerten prüfen, etwa negative Werte oder eine maximale Anzahl, z. B. von Teilnehmern einer Veranstaltung. Dabei handelt es sich um Eingaben, die fachlich keinen Sinn machen, aber zu unsinnigen Ergebnissen führen können. In unserem Beispiel sollte man wohl noch negative Werte für die Anzahl durch eine Vorbedingung ausschließen. Wie in Kapitel 6 vorgestellt, kann man zur Absicherung von Vorbedingungen auch auf die Utility-Klasse `Preconditions` aus Google Guava zurückgreifen:

```
int calcDiscount(final int count)
{
    // Absicherung von Vorbedingungen
    // if (count < 0)
    //     throw new IllegalArgumentException("count should not be negative");
    Preconditions.checkArgument(count >= 0, "count should not be negative");

    if (count < 50)
        return NO_DISCOUNT;
    if (count >= 50 && count <= 1000)
        return MEDIUM_DISCOUNT;
    if (count > 1000)
        return BIG_DISCOUNT;

    throw new IllegalStateException("programming problem: should never " +
                    "reach this line. value " + count + " not handled! ");
}
```

20.3 Motivation für Unit Tests aus der Praxis

In diesem Abschnitt werden wir die positiven Auswirkungen des Einsatzes von Unit Tests anhand zweier Beispiele aus der Praxis konkret nachvollziehen. Dabei wird klar, wieso Unit Tests so wichtig zur Absicherung von Änderungen sind, welche Vorteile sich ergeben können, aber auch welche Fallstricke lauern.

20.3.1 Unit Tests für Weiterentwicklungen

Stellen Sie sich folgende Problemstellung vor: Sie sollen Ausgabegeräte, ähnlich zu LCD-Monitoren, ansteuern, die über ein eigenes Protokoll mit Befehlsnachrichten angesprochen werden, etwa »Text darstellen«, »Rechteck löschen« usw. Zur

Ansteuerung besitzen diese Geräte einen internen Speicher für eingehende Nachrichten, der in sechs gleich große Speicherschlitze unterteilt ist. Die in den Speicherschlitzen abgelegten Befehle werden von einer Kontrolleinheit gelesen und ausgeführt. Die Verwaltung dieses Eingangspuffers ist allerdings nicht sonderlich intelligent realisiert: Werden sukzessive Nachrichten empfangen, so landet jede eingehende Nachricht in einem eigenen Schlitz. Für kürzere Nachrichten verfällt der restliche Speicherplatz eines Schlitzes ungenutzt. Es können somit maximal sechs Nachrichten gespeichert werden, obwohl der Eingangspuffer im Grunde noch Kapazität für deutlich mehr Nachrichten bietet. Das ist in Abbildung 20-5 schematisch dargestellt, wobei die dunkelgrauen Kästchen dem Nutzinhalt der Nachrichten entsprechen und die hellgrauen Balken den restlichen, noch zur Verfügung stehenden Speicherplatz eines Schlitzes repräsentieren.

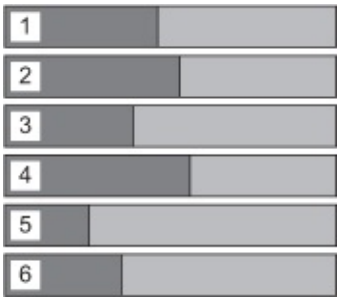


Abbildung 20-5 *Nachrichtenpuffer mit 6 Schlitzen*

Neben der Platzverschwendung gibt es noch ein weiteres Problem: Durch eine schnelle Folge von eingehenden Nachrichten werden zuvor gespeicherte Nachrichten überschrieben. Abschnitt 20.5 greift diese Problematiken wieder auf und stellt Lösungsmöglichkeiten und Unit Tests mit Timing vor.

Schauen wir nachfolgend aber zunächst auf die aus dem Design relevanten Details der Verarbeitungslogik, um diese danach dann mit Unit Tests zu prüfen. Zur besseren Testbarkeit wurde im Design als Abstraktion eines Anzeigegeräts das Interface `IDisplay` eingeführt. Zwei konkrete Ausprägungen beschreiben einerseits die reale Hardwareansteuerung durch die Klasse `LCDDisplay` sowie eine Simulationsklasse, realisiert durch die Klasse `SimluationDisplay`. Die Ansteuerung erfolgt durch eine `Controller` genannte Klasse, die lediglich über das Interface `IDisplay` mit der Anzeige kommuniziert.

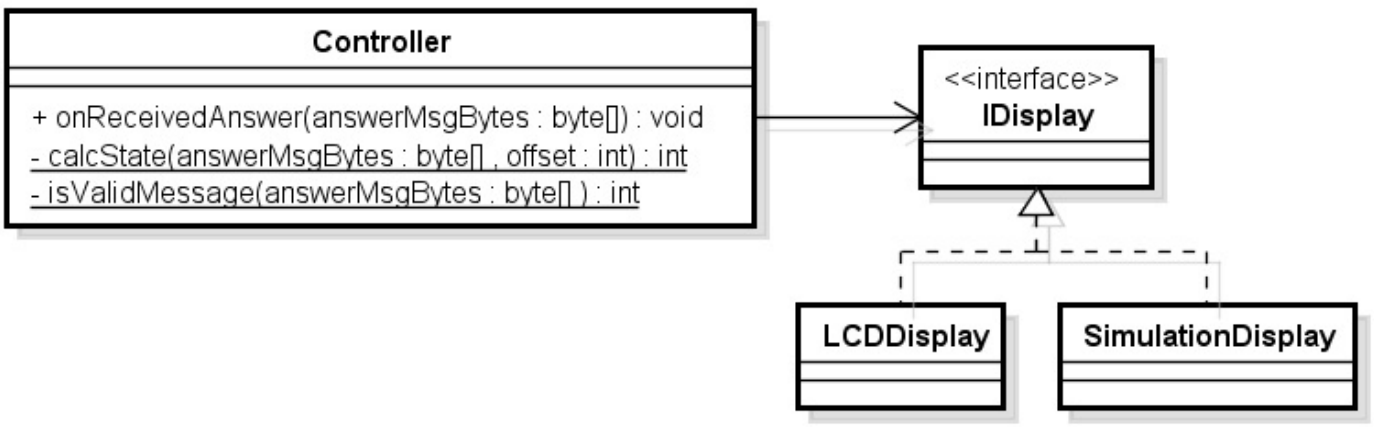


Abbildung 20-6 *Klassendiagramm zur Displayansteuerung*

Auswertung von Statustelegrammen

Die an ein Display gesendeten Befehle werden mit einem Rückgabecode beantwortet, der in einer Nachricht mit mehreren Bytes codiert ist. Eine erste Realisierung der `Controller`-Klasse könnte eine öffentliche Methode `onReceivedAnswer( byte[])` zur Verarbeitung von Antworten wie folgt implementieren:

```
public void onReceivedAnswer(final byte[] answerMsgBytes)
{
    if (isValidMessage(answerMsgBytes))
    {
        final int state = calcState(answerMsgBytes);
        // …
```

In einer Antwortnachricht `answerMsgBytes` finden sich Statusinformationen an einer durch die Konstante `STATUS_OFFSET` definierten Position. Dort gespeicherte Werte werden zur Zustandsberechnung in der statischen, privaten Methode `calcState( byte[])` wie folgt genutzt:

```
private static int calcState(final byte[] msgBytes)
{
    final int stateHigh = (msgBytes[STATUS_OFFSET] - 48);
    final int stateLow = (msgBytes[STATUS_OFFSET + 1] - 48);

    return stateHigh * 16 + stateLow;
}
```

Zum Verständnis der Berechnung muss man wissen, dass die Codierung des Status einer speziellen hexadezimalen Codierung folgt. Das Zeichen '0' besitzt den Wert 48 und hexadezimal den Wert 30. Mit dieser Art von Codierung erhält man lesbare Ausgaben auf der Konsole. Tabelle 20-2 zeigt das exemplarisch für einige Codierungen.

Tabelle 20-2 *Codierung und Rückgabewert*

Codierung	Zeichen auf der Konsole	Hex-Wert	Rückgabewert als `int`
32 30	"20"	20	32
33 3F	"3F"	3F	63
3A 3E	"AE"	AE	174

Das Erstellen von Unit Tests ist für die Methode `calcState( byte[])` mit einigen Problemen verbunden. Zunächst ist die Methode `private` deklariert und somit nicht von den Unit Tests zugreifbar. Als Abhilfe erhöhen wir die Sichtbarkeit auf Package-private, wie dies später in Abschnitt 20.4.4 diskutiert wird. Schwerwiegender

ist hier jedoch, dass zum Test der Zustandsberechnung immer eine Antwortnachricht als `byte[]` mit Inhalt in einem speziellen Format (Framing, Checksumme usw.) erzeugt werden muss, um die Methode `calcState(byte[])` mit gültigen Eingabedaten aufrufen zu können. Um weniger Abhängigkeiten zu besitzen, sorgen wir in einem ersten Umformungsschritt für eine Vereinfachung, damit im Anschluss das Schreiben von Tests leichter fällt. ***Ein solches Vorgehen kann gefährlich sein, ist aber manchmal notwendig, um überhaupt testen zu können***. Gute Tipps, wie man dabei vorgeht, finden Sie im Buch »Working Effectively With Legacy Code« [24] von Michael Feathers.

Schritt 1: Korrekturen der Abhängigkeiten und Lesbarkeit

Wir wollen eine neue Methode erstellen, die nur die tatsächlich benötigten Bytes als Eingabe besitzt. Zudem soll diese Methode unabhängig von zurückgelieferten Nachrichten testbar sein. Nach kurzer Analyse erkennen wir, dass als Eingabe nur zwei Werte benötigt werden, nämlich diejenigen, die an den Positionen `STATUS_OFFSET` und `STATUS_OFFSET + 1` in den Antwortdaten gespeichert sind. Zunächst benennen wir die bisherige Methode in `calcStateOld(byte[])` um und implementieren eine neue Methode `calcState(byte, byte)` mit der Sichtbarkeit Package-private. Es ergibt sich folgende verbesserte Version der Methode:

```
/*private*/ static int calcState(final byte highByte, final byte lowByte)
{
    final int stateHigh = (highByte - 48);
    final int stateLow = (lowByte - 48);

    return stateHigh * 16 + stateLow;
}
```

Die ursprüngliche Methode lässt sich dann wie gezeigt umformulieren (und sobald sie überflüssig geworden ist, leichter entfernen):

```
@Deprecated
private static int calcStateOld(final byte[] msgBytes)
{
    return calcState(msgBytes[STATUS_OFFSET], msgBytes[STATUS_OFFSET + 1])
}
```

Schritt 2: Erstellen erster Unit Tests

Nachdem auf diese Weise eine besser verständliche Implementierung entstanden ist, starten wir mit der Entwicklung von Unit Tests. Dort prüfen wir für verschiedene Eingaben die Berechnung von Statuswerten: Die Eingabewerte '07', '20' und '79' sind (in Anlehnung an Äquivalenzklassen) so gewählt, dass sie sowohl das High- als auch das Low-Byte einzeln und in Kombination abdecken:

```
@Test
public void testCalcState_WithInput_07()
{
```

```
    assertEquals(7, Controller.calcState((byte) '0', (byte) '7'));
}

@Test
public void testCalcState_WithInput_20()
{
    assertEquals(2 * 16 + 0, Controller.calcState((byte) '2', (byte) '0'));
}

@Test
public void testCalcState_WithInput_79()
{
    assertEquals(7 * 16 + 9, Controller.calcState((byte) '7', (byte) '9'));
}
```

Die Tests mit diesen Werten laufen – wie erwartet – erfolgreich. Nun machen wir uns an ein Refactoring, um die Methode besser verständlich zu gestalten, da diese momentan noch auf einem sehr technischen Niveau entwickelt und dadurch nicht gut lesbar ist. Wir ersetzen zunächst die Magic Number `48` durch das `char`-Literal `'0'` und extrahieren dann die Berechnung des Werts in die Hilfsmethode `hexCodedByteValueToInt(byte)`. Die Verständlichkeit und die Lesbarkeit werden dadurch positiv beeinflusst, wie dies folgendes Listing zeigt:

```
/*private*/ static int calcState(final byte highByte, final byte lowByte)
{
    final int stateHigh = hexCodedByteValueToInt(highByte);
    final int stateLow = hexCodedByteValueToInt(lowByte);

    return stateHigh * 16 + stateLow;
}

/*private*/ static int hexCodedByteValueToInt(final byte byteValue)
{
    return (byteValue - '0');
}
```

Schritt 3: Unit Tests um Testfälle für `A` bis `F` erweitern

Durch unsere ersten Tests wissen wir, dass die Methode `calcState(byte, byte)` im Wertebereich der Dezimalzahlen korrekt funktioniert. Es ist aber auch eine Verarbeitung hexadezimaler Eingaben gefordert. Schauen wir auf die hexadezimalen Werte `'A'` bis `'F'`. Für erste Testfälle wählen wir (etwas willkürlich) die zwei Vertreter `'3F'` und `'AE'` als Eingabewerte:

```
@Test
public void testCalcState_WithInput_HexValue_3F()
{
    assertEquals(0x3f, Controller.calcState((byte) '3', (byte) 'F'));
}

@Test
public void testCalcState_WithInput_HexValue_AE()
{
    assertEquals(0xAE, Controller.calcState((byte) 'A', (byte) 'E'));
}
```

Diese beiden neu erstellten Unit Tests schlagen allerdings fehl.

Eine kurze Analyse der fehlgeschlagenen Testfälle ergibt, dass die Berechnung falsche Ergebnisse liefert, weil ein kleiner Denkfehler beim Erstellen der Methode `hexCodedByteValueToInt(byte)` gemacht wurde: Der durchgängige Wertebereich der Hexadezimalzahlen von 0 bis F wurde genauso auf die ASCII-Werte der entsprechenden `char`-Literale '0' bis 'F' übertragen – dort gilt jedoch Folgendes:

- '0' = 48, …, '9' = 57,
- 'A' = 65, …, 'F' = 70.

Damit ergeben sich folgende Berechnungen:

- 'A' - '0' = 65 - 48 = 17 => Wertebereichsfehler: Korrekt ist der Wert 10.
- 'F' - '0' = 70 - 48 = 22 => Wertebereichsfehler: Korrekt ist der Wert 15.

Zur Korrektur wird eine Fallunterscheidung genutzt:

```
/*private*/ static int hexCodedByteValueToInt(final byte byteValue)
{
    if (byteValue >= 'A' && byteValue <= 'F')
    {
        return (10 + (byteValue - 'A'));
    }
    if (byteValue >= '0' && byteValue <= '9')
    {
        return (byteValue - '0');
    }
    throw new IllegalArgumentException("Unexpected byte value: " + byteValue +
            ". Must be in Range '0'-'9' (0x30-0x39) or 'A'-'F' (0x41-
0x46)");
}
```

Als Folge der obigen Korrekturen laufen nun alle Tests. Die Berechnung innerhalb der Methode `hexCodedByteValueToInt(byte)` ist durch die Fallunterscheidung jedoch etwas schlechter lesbar. Da es sich hierbei um eine Package-private Methode handelt, ist dies noch akzeptabel. Für eine öffentliche Methode gelten strengere Maßstäbe: Diese sollte immer gut lesbar sein und kaum technische Details zeigen.

Schritt 4: Unit Tests um Testfälle für die Konvertierung `0` bis `F` erweitern

Da sich die Methode `hexCodedByteValueToInt(byte)` als fehleranfällig herausgestellt hat, ergänzen wir spezielle Tests für diese Methode. Zunächst prüfen wir die korrekte Umwandlung sowohl von Ziffern als auch von gültigen Buchstaben – wobei wir hier eigentlich vier unabhängige Prüfungen in einem Testfall bündeln, weil diese semantisch zusammengehören. Damit sieht man bei einem Fehler möglicherweise nicht sofort, welche Bedingung verletzt wird. Später lernen wir mit der in JUnit 4.7 eingeführten JUnit Rule `ErrorCollector` eine elegantere

Möglichkeit der Prüfung mehrerer Werte in einem Testfall kennen.

```
@Test
public void testHexCodedByteValueToInt_With_SomeNumbersAndHexChars()
{
    assertEquals(0, Controller.hexCodedByteValueToInt((byte) '0'));
    assertEquals(9, Controller.hexCodedByteValueToInt((byte) '9'));
    assertEquals(10, Controller.hexCodedByteValueToInt((byte) 'A'));
    assertEquals(15, Controller.hexCodedByteValueToInt((byte) 'F'));
}
```

Hier erfolgt lediglich eine Prüfung durch Stichproben. Möchte man den gesamten Wertebereich absichern, so bietet es sich an, die Prüfung umzugestalten. Man nutzt dazu zwei Arrays: eines für Eingabewerte und ein weiteres für erwartete Resultate. Dann kann man die Berechnungsergebnisse mit einer Schleife wie folgt prüfen:

```
@Test
public void testHexCodedByteValueToInt_With_AllValidInputs()
{
    final byte[] inputs = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                            'A', 'B', 'C', 'D', 'E', 'F' };
    final int[] expected = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                             10, 11, 12, 13, 14, 15 };

    for (int i = 0; i < inputs.length && i < expected.length; i++)
    {
        assertEquals(expected[i],
Controller.hexCodedByteValueToInt(inputs[i]));
    }
}
```

Auch hier gilt wieder die zuvor geführte Argumentation der leicht ungünstigen mehrfachen Prüfungen semantisch zusammengehörender Werte sowie der Verweis auf eine elegante Umsetzung mit der JUnit Rule `ErrorCollector`.

Sinnvolle Erweiterung

Man sollte neben Standardfällen und -eingaben immer auch Randfälle oder Extremwerte testen. Deswegen ergänzen wir eine Testmethode, die einige ungültige Randwerte prüft. Für alle ungültigen Werte erwarten wir das Auftreten einer `IllegalArgumentException`. Im entsprechenden `catch`-Block reagieren wir daher darauf mit einem `assertTrue(true)`. Das ist zwar eigentlich überflüssig, um den Test zu bestehen, hilft aber, die erwartete Situation klarer auszudrücken. Führt die Umrechnung unerwartet zu keiner Exception, so nutzen wir die Methode `fail()`, um dies als Fehler im Unit Test zu protokollieren. Wir definieren zudem eine Hilfsmethode `checkInvalidValue(byte)`, die diese Details versteckt. Bei den Eingaben beschränken wir uns hier zunächst auf vier wesentliche Randfälle:

```
@Test
public void testHexCodedByteValueToInt_InvalidInputs()
{
    checkInvalidValue((byte) 0x29);
    checkInvalidValue((byte) 0x3a);
    checkInvalidValue((byte) 0x40);
```

```
    checkInvalidValue((byte) 0x47);
}
```

Die Hilfsmethode wird folgendermaßen realisiert:

```
private void checkInvalidValue(final byte value)
{
    try
    {
        Controller.hexCodedByteValueToInt(value);
        fail(Integer.toHexString(value) + " shouln't be valid");
    }
    catch (final IllegalArgumentException ex)
    {
        assertTrue(true);
    }
}
```

Die gesamte Realisierung wirkt etwas umständlich. Tatsächlich ist dies aber für den gezeigten Testcode die einzige Möglichkeit, bei der Prüfung erwartete Exceptions zu verarbeiten und in einem Testfall, also pro Testmethode, auch *mehrere* Randwerte zu prüfen, ohne dazu die erst seit JUnit 4.7 vorhandene JUnit Rule `ErrorCollector` zu nutzen. Später erfahren Sie dazu mehr.

Reaktion auf einen erwarteten Typ von Exception

Soll in einem Testfall auf eine erwartete Exception reagiert werden, so war das eben gezeigte Programmstück mit JUnit 3 der einzig mögliche Weg. Mit JUnit 4 lässt sich dies eleganter formulieren, indem für die Annotation `@Test` im Parameter `expected` der erwartete Typ von Exception spezifiziert wird. Das Ausbleiben dieser Exception führt zu einem Fehlschlagen des Testfalls. Die nachfolgendem Listing gezeigte Testmethode `testInvalidHexCodedByteValueToInt_InvalidInput_ThrowsExceptic` definiert einen Testfall für den Randwert `0x29`. Dafür wird ein Fehlschlagen mit einer `IllegalArgumentException` erwartet:

```
@Test(expected=IllegalArgumentException.class)
public void testHexCodedByteValueToInt_InvalidInput_Throws_Exception()
{
    Controller.hexCodedByteValueToInt((byte) 0x29);
}
```

Mit der gezeigten Realisierung lässt sich jedoch *lediglich* der Test eines *einzigen* Randfalls durchführen. Sind mehrere Randfälle zu prüfen, so kann die zuvor gezeigte Umsetzung mit der Methode `checkInvalidValue(byte)` erfolgen. Alternativ können pro Randfall eigene Testmethoden erstellt werden. Das führt jedoch recht schnell zu sehr vielen Testmethoden.

Schritt 5: Testfälle für Eingabewerte `a` bis `f`

Die Funktionalität ist nun gut mit Unit Tests abgesichert. Ein Testfall fehlt allerdings noch: Für hexadezimale Zahlen sind auch die Kleinbuchstaben `a` bis `f` erlaubt. Nach

dem Erstellen der Tests aus Schritt 4 ist uns schon jetzt klar, dass diese Werte nicht unterstützt werden. Wir schreiben trotzdem einen Testfall, um unsere Vermutung zu untermauern:

```
@Test
public void testCalcStateHexValues_ac()
{
    assertEquals(0xAC, Controller.calcState('a', 'c'));
}
```

Wie erwartet schlägt dieser Testfall fehl.

Schritt 6: Implementierung um Konvertierung `a` bis `f` erweitern

Betrachten wir nun, wie wir diese Anforderung realisieren können. Ein erster spontaner Gedanke zur Verarbeitung von Kleinbuchstaben könnte sein, eine weitere Fallunterscheidung in die Methode `hexCodedByteValueToInt(byte)` aufzunehmen. Das würde allerdings die Lesbarkeit noch weiter verschlechtern. Wir überlegen kurz und erinnern uns dann an die Wrapper-Klassen (vgl. Abschnitt 4.2.2). Die Konvertierung lässt sich über die Klasse `Integer` und deren Methode `parseInt(String, int)` mit der Angabe der Basis 16 durchführen und in der Methode `calcState(byte, byte)` wie folgt nutzen:

```
/*private*/ static int calcState(final byte highByte, final byte lowByte)
{
    final String hexNumber = new String(new byte[] { highByte, lowByte });
    return Integer.parseInt(hexNumber, 16);
}
```

Durch Einsatz dieser Bibliotheksmethode können wir zusätzlich zum Applikationscode auch die Unit Tests vereinfachen. Alle mit Schritt 4 eingeführten Unit Tests sind nun obsolet und können entfallen, da hier lediglich die Methode `hexCodedByteValueToInt(byte)` getestet wird, die aus der Applikation entfernt wurde. Somit spart man über 50 Zeilen Sourcecode und reduziert damit den Testcode um etwa die Hälfte. Anschließend werden alle Tests bestanden. Abbildung 20-7 zeigt dies.

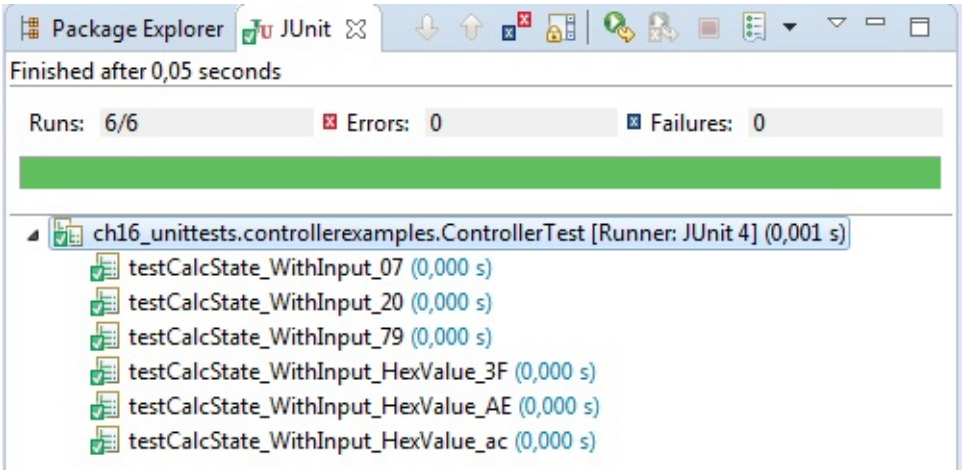


Abbildung 20-7 *Unit Tests nach den Verbesserungen*

Fazit

Wir haben gesehen, wie man Schritt für Schritt Unit Tests entwickeln kann und sich daraus Verbesserungen im Applikationscode ergeben können. Manchmal ist es sogar möglich, wie in Schritt 6, Teile der Implementierung komplett auszutauschen und durch die Unit Tests ein äquivalentes Verhalten absichern zu können. Wie bereits erwähnt, gibt es dabei keine absolute Sicherheit, allerdings eine größtmögliche.

20.3.2 Unit Tests und Legacy-Code

Nachdem wir für Weiterentwicklungen bereits verschiedene positive Effekte durch den Einsatz von Unit Tests kennengelernt haben, betrachten wir nun das Schreiben von Unit Tests für Legacy-Code. Dies gestaltet sich mitunter deutlich anspruchsvoller, als dies für Erweiterungen bzw. Neuentwicklungen der Fall ist. Teilweise ist man selbst nicht Autor des Programms und muss sich zunächst in unbekanntem Sourcecode zurechtfinden. Des Weiteren existieren Beschränkungen, etwa durch unerwartete Abhängigkeiten zwischen Klassen. Auch trifft man auf Implementierungsfehler, Seiteneffekte oder eine Festlegung auf ein veröffentlichtes API, das im Nachhinein nicht geändert werden kann. All dies erschwert es, Unit Tests zu schreiben. Es bietet sich daher an, zunächst ein paar Vorarbeiten in Form von Analysen durchzuführen.

Vorarbeiten: Analyse

Im Folgenden betrachten wir das Schreiben von Unit Tests für eine Auftragsverwaltung, insbesondere eine Utility-Klasse `TaskUtils`. Dieses Praxisbeispiel zeigt, welche Schritte sinnvoll sind und welchen Problemen man begegnen kann.

Bevor wir mit dem Schreiben von Unit Tests beginnen, verschaffen wir uns einen Überblick über den Systemkontext, d. h. die beteiligten Klassen und Interfaces. Dazu erstellen wir ein Klassendiagramm und schauen uns anschließend den zu testenden Sourcecode-Ausschnitt an. Unterschiedliche Aufträge werden durch ein Interface `ITask` abstrahiert. Als Vereinfachung betrachten wir hier lediglich die beiden Klassen `SimpleTask` und `ComplexTask`, die dieses Interface implementieren. Abbildung 20-8 zeigt diese Klassen sowie eine Utility-Klasse `TaskConverter` zur Konvertierung von Auftragsobjekten in eine Stringrepräsentation.

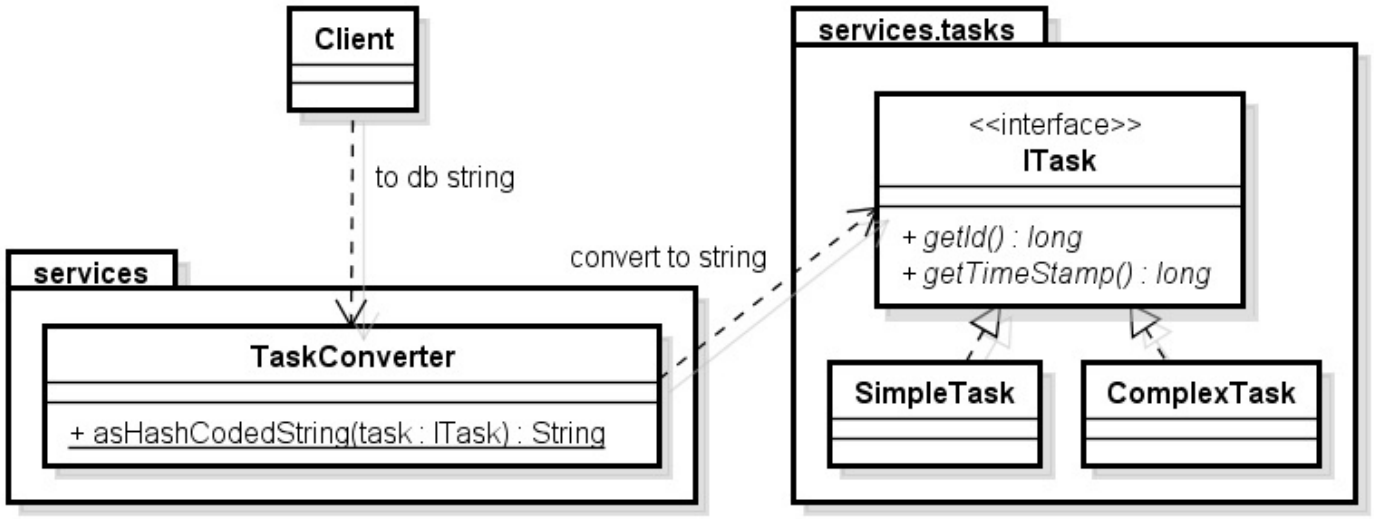


Abbildung 20-8 *Ausgangssituation der Konvertierung von* `ITask`

Zum Zeitpunkt der Applikationserstellung bestand eine Forderung darin, verschiedene Varianten von Tasks in Form von Objekten unterschiedlicher Klassen mit dem Basistyp `ITask` textuell in einer Datenbank speichern zu können. Dazu musste eine geeignete Repräsentation in Form eines Strings gefunden werden. Die Wahl fiel auf eine Notation, die einzelne Werte von Attributen durch '#' voneinander trennt. Werfen wir einen Blick auf die Methode `compareTaskWithTaskData(ITask, String)` der Utility-Klasse `TaskUtilsV1`:

```
public final class TaskUtilsV1
{
    private static final Logger log = Logger.getLogger(TaskUtilsV1.class);

    public static TaskCompareResults compareTaskWithTaskData(final ITask
aTask,
          final String existingTaskData)
    {
        // Parameter-Prüfungen wg. Übersichtlichkeit entfernt

        final String newTaskData = TaskConverter.asHashCodedString(aTask);
        log.debug("comparing '" + existingTaskData +
                  "' with new '" + newTaskData + "'.");

        if (newTaskData.length() > existingTaskData.length())
        {
            log.warn("new task is different from existing (wrong length).");
            return TaskCompareResults.FAILURE_DIFFERENT_TASK_CONTENT;
        }

        // Gleicher Inhalt?
        if (newTaskData.equals(existingTaskData.substring(0,
                               newTaskData.length())))
        {
            log.info("task " + aTask + " already exists.");
            return TaskCompareResults.OK_EXISTING_TASK;
        }

        // Inhalt unterschiedlich!
        log.warn("different content (not equal).");
        return TaskCompareResults.FAILURE_DIFFERENT_TASK_CONTENT;
```

```
    }

    // ...
```

Analyse der Methode

Die obige Methode `compareTaskWithTaskData(ITask, String)` dient dazu, Objekte vom Typ `ITask` mit bereits bestehenden Objekten in Form einer Stringrepräsentation zu vergleichen. Die als `ITask` repräsentierten Aufträge werden mithilfe der Methode `asHashCodedString(ITask)` der Klasse `TaskConverter` in eine '#'-separierte Stringnotation umgewandelt und in der Variablen `newTaskData` als Basis für nachfolgende textuelle Vergleiche gespeichert. Der Ausgang der Vergleiche wird in Form einer `TaskCompareResults`-Aufzählung zurückgeliefert. In Abbildung 20-9 sind die beteiligten Klassen dargestellt.

Implementierungsfehler

Wie bereits erwähnt, hat man in Legacy-Code immer wieder mit unerwarteten, möglicherweise sogar merkwürdigen Designentscheidungen oder auch Implementierungsfehlern zu kämpfen. Betrachten wir einige Unzulänglichkeiten der in diesem Beispiel relevanten Klassen.

Ein Vergleich zweier Aufträge ist nicht über den Aufruf von `equals(Object)` möglich, da aus Unachtsamkeit diese Methode nicht in allen Implementierungen existiert. Stattdessen wird die spezielle Vergleichsmethode `compareTaskWithTask-Data(ITask, String)` aufgerufen, die beliebige Objekte vom Typ `ITask` mit einem Vergleichswert vom Typ `String` vergleicht. Dies wird hier notwendig, da die textuellen Werte direkt aus der Datenbank stammen und eine Rückkonvertierung aus diesen Informationen in Objekte vom Typ `ITask` nicht für alle Auftragsklassen existiert.

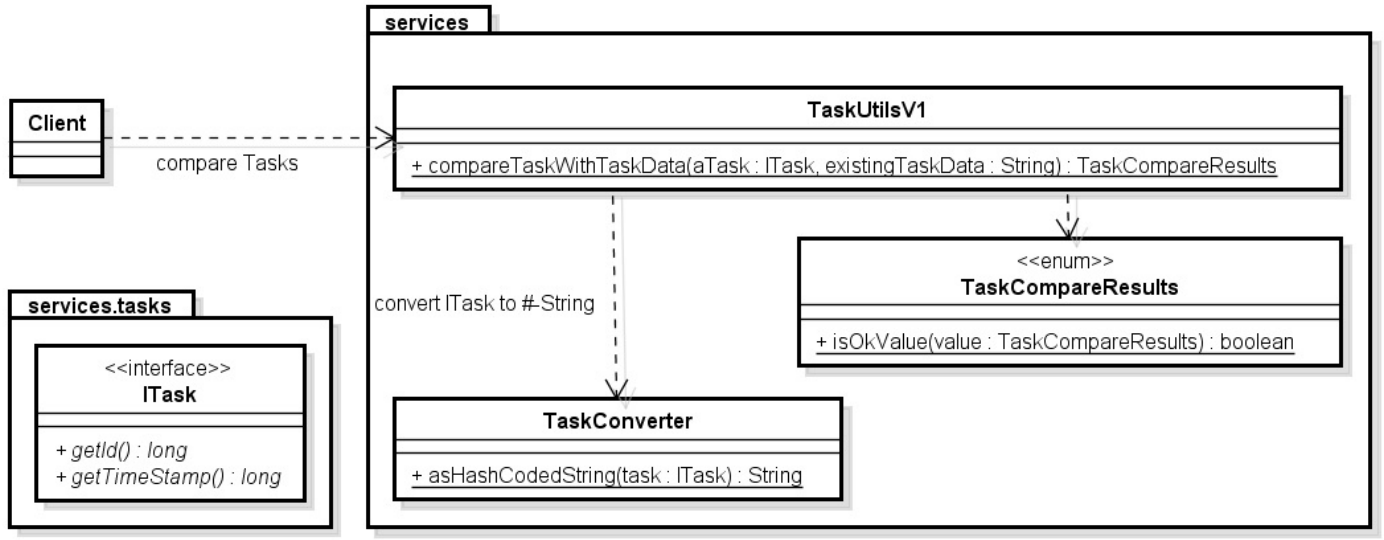


Abbildung 20-9 *Ausgangssituation* `ITask` *und Konvertierung*

Aufgabenstellung

Nehmen wir an, unsere Aufgabe wäre es, diese Methode um Funktionalität zu erweitern. Da die Methode bereits etwas kompliziert wirkt, wollen wir sie zunächst verstehen und gegebenenfalls vereinfachen. Bevor mit der Entwicklung begonnen wird, ist es ratsam, zur Absicherung der Umbauarbeiten einige Unit Tests zu erstellen. Wünschenswert ist es zudem, die Unit Tests so zu gestalten, dass konkrete Daten aus dem Praxiseinsatz als Eingabe verwendet werden können.

Schritt 1: Schreiben von ersten Unit Tests

Exemplarisch zeige ich hier die Implementierung eines Unit Test anhand der folgenden Methode `testCompareTaskWithExampleTask()`, die die Methode `compareTaskWithTaskData(ITask, String)` überprüfen soll. Um dabei mit reproduzierbaren Testdaten und Ausgangssituationen arbeiten zu können, setzt man in der Regel eine spezielle `@Before`-Methode ein, die Initialisierungen vornehmen kann. In diesem Fall nutzen wir eine andere Möglichkeit, da nicht allgemeine Daten aufbereitet werden sollen, sondern lediglich Testdaten in Form eines Objekts vom Typ `ITask` bereitzustellen sind. Die Fabrikmethode `createExampleTask(ITask, String)` erzeugt ein passendes Testexemplar vom Typ `ITask`:

```
@Test
public void testCompareTaskDataWithExampleTask()
{
    final long id = 2L;
    final int line = 50;
    final int course = 26;
    final int route = 996;
    final long timeStamp = 1225112198000L;

    final ITask exampleTask = createExampleTask(id, line, course, route,
          timeStamp);
    final String existingTaskData = "2#50#26#996#1225112198000";

    final TaskCompareResults value = compareTaskWithTaskData(exampleTask,
          existingTaskData);

    assertTrue(TaskCompareResults.isOkValue(value));
}

private static ITask createExampleTask(final long id, final int line,
                                       final int course, final int route,
                                       final long timeStamp)
{
    return new SimpleTask(id, line, course, route, timeStamp);
}
```

Dieser Task wird in eine Folge von '#'-separierten Werten umgewandelt. Einen erwarteten Vergleichswert speichern wir in der Variablen `existingTaskData` und rufen dann die zu testende Methode `compareTaskWithTaskData(ITask, String)` auf.

Während des Schreibens dieses und weiterer Unit Tests erkennen wir schnell, dass Abhängigkeiten von externen Klassen das Unit-Testen erschweren. Abbildung 20-10 zeigt, dass die Testklasse `TaskUtilsV1Test` auf Klassen aus dem Package `services.tasks` verweist und sogar Instanzen dieser Klassen erzeugt.

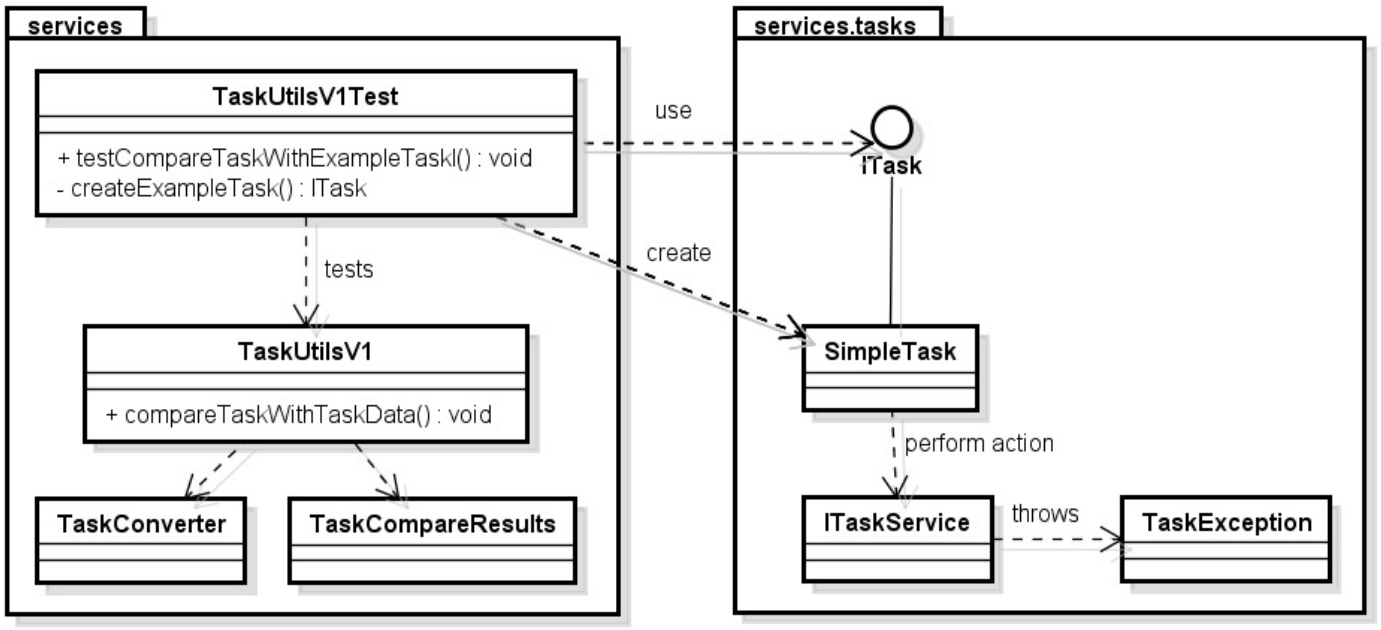


Abbildung 20-10 *Abhängigkeiten in Unit Tests*

Die Abhängigkeiten erschweren es, neue Tests hinzuzufügen. Das ist unpraktisch und führt möglicherweise dazu, dass kaum Unit Tests ergänzt werden, weil das Ganze zu mühevoll ist. ***Daher sollten wir immer dafür sorgen, dass Unit Tests mit wenig Aufwand geschrieben werden können***. Schauen wir mal, was dazu notwendig ist.

Schritt 2: Herauslösen einer durch Unit Tests prüfbaren Methode

Ideal wäre es, auf die Rückkonvertierung zu verzichten, indem Vergleiche von Strings durchgeführt werden. Das löst auch die Abhängigkeit der Testklasse von der Klasse `TaskConverter`. Tatsächlich können wir die Methode `compareTaskWithTaskData(ITask, String)` ohne große Mühe gemäß diesen Forderungen überarbeiten, da sie nach der Umwandlung in Objekte vom Typ `ITask` sowieso nur noch auf Strings arbeitet. Wir extrahieren eine Methode `compareTaskData(String, String)`, die die Vergleichslogik enthält. Die Methode `compareTaskWithTaskData(ITask, String)` delegiert an die neue Methode und wandelt zuvor ein Objekt vom Typ `ITask` in die zugehörige Stringrepräsentation:

```
public final class TaskUtilsV2
{
    private static final Logger log = Logger.getLogger(TaskUtilsV2.class);

    public static TaskCompareResults compareTaskWithTaskData(final ITask
aTask,
            final String existingTaskData)
```

```
    {
        // Parameter-Prüfungen wg. Übersichtlichkeit entfernt

        final String newTaskData = TaskConverter.asHashCodedString(aTask);
        return compareTaskData(newTaskData, existingTaskData);
    }

    public static TaskCompareResults compareTaskData(final String newTaskData,
          final String existingTaskData)
    {
        log.debug("comparing '" + existingTaskData +
                  "' with new '" + newTaskData + "'.");

        if (newTaskData.length() > existingTaskData.length())
        {
            log.warn("new task is different from existing (wrong length).");
            return TaskCompareResults.FAILURE_DIFFERENT_TASK_CONTENT;
        }

        // Gleicher Inhalt?
        if (newTaskData.equals(existingTaskData.substring(0,
                               newTaskData.length())))
        {
            log.info("task " + existingTaskData + " already exists.");
            return TaskCompareResults.OK_EXISTING_TASK;
        }

        // Inhalt unterschiedlich!
        log.warn("different content (not equal).");
        return TaskCompareResults.FAILURE_DIFFERENT_TASK_CONTENT;
    }
    // …
```

Wenn wir in der Testklasse `TaskUtilsV2Test` bevorzugt noch die Methode `compareTaskData(String, String)` und weniger die originale Methode prüfen, so vereinfacht sich dadurch der Testcode erheblich, da die Konvertierung in ein Objekt vom Typ `ITask` und damit ein zusätzlicher, fehleranfälliger Schritt entfallen kann. Folgendes Listing zeigt drei mögliche Unit Tests, die auf Stringvergleichen beruhen. Dabei fällt zudem auf, dass sogar vom eigentlichen Nutzinhalt (den '#'-codierten Werten) abstrahiert werden kann und sich das Ganze mithilfe eines statischen Imports der zu testenden Methode wie folgt schreiben lässt:

```
public class TaskUtilsV2Test
{
    @Test
    public void testCompareTaskDataNewMustNotBeLonger()
    {
        final String newTaskData = "newContent";
        final String existingTaskData = "content";

        final TaskCompareResults value = compareTaskData(newTaskData,
                                                         existingTaskData);
        assertFalse(TaskCompareResults.isOkValue(value));
    }

    @Test
    public void testCompareTaskSameLengthAndSameContent()
    {
        final String newTaskData = "sameContent";
```

```
        final String existingTaskData = "sameContent";

        final TaskCompareResults value = compareTaskData( newTaskData,
                                                          existingTaskData);
        assertTrue( TaskCompareResults. isOkValue( value));
    }

    @Test
    public void testCompareTaskSameLengthAndDifferentContent()
    {
        final String newTaskData = "original!";
        final String existingTaskData = "different";

        final TaskCompareResults value = compareTaskData( newTaskData,
                                                          existingTaskData);
        assertFalse( TaskCompareResults. isOkValue( value));
    }
    // …
```

Sinnvoll ist es – nicht nur beim Unit-Testen –, möglichst wenige Abhängigkeitsbeziehungen zu anderen Klassen zu haben. In diesem Fall konnten durch den Einsatz von Strings die Abhängigkeiten zwischen der Testklasse und den Implementierungen in verschiedenen Packages verringert werden. Das bot sich hier an, stellt aber keine allgemeingültige Empfehlung dar (siehe nachfolgenden Warnhinweis). Durch die Vereinfachung ergibt sich das in Abbildung 20-11 dargestellte UML-Diagramm.

Warnhinweis: Auflösen von Abhängigkeitsbeziehungen

In diesem Fall war diese Transformation und die Wahl von Strings eine natürliche Wahl, da im Endeffekt mit Strings gearbeitet wurde. Es ist jedoch keine allgemeingültige Empfehlung, typsichere Übergabeparameter durch den Typ `String` zu ersetzen, nur um die Abhängigkeiten von anderen Klassen um jeden Preis zu reduzieren.

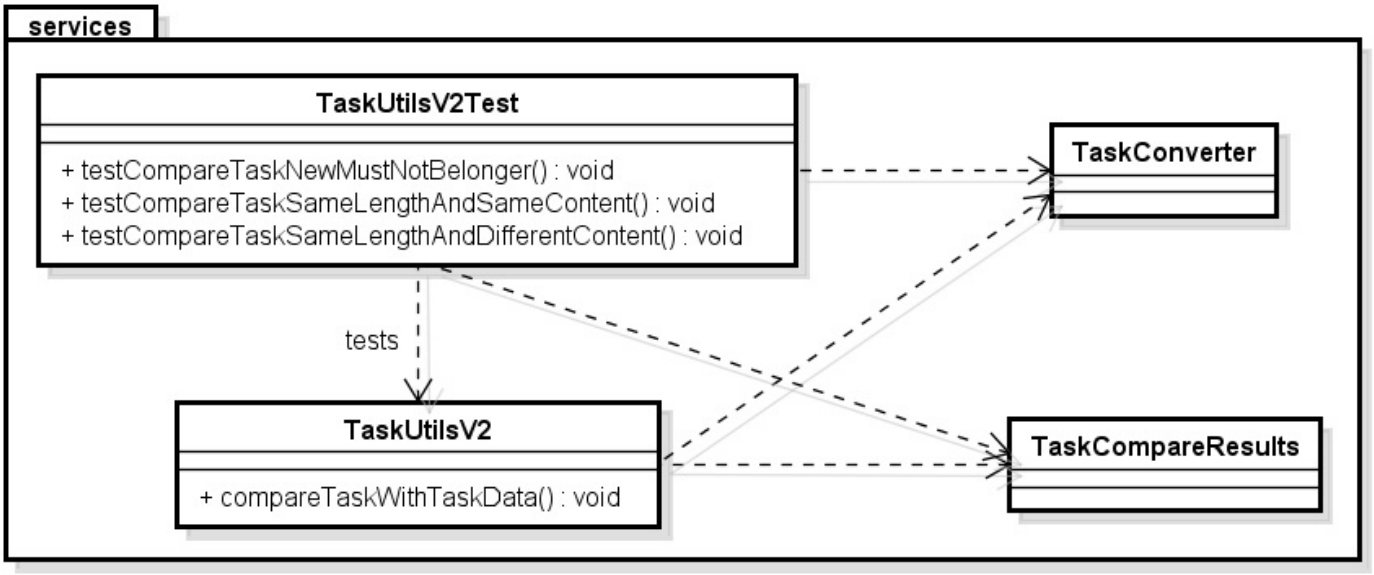


Abbildung 20-11 *Unit Test mit verringerten Abhängigkeiten*

Schritt 3: Anpassungen für bessere Wart- und Lesbarkeit

Die Methode `compareTaskData(String, String)` lässt sich nun (ohne die störenden externen Abhängigkeiten) sehr gut durch Unit Tests prüfen, da die Eingabe lediglich aus zwei Stringrepräsentationen besteht. Allerdings findet man dort noch einen etwas komplexeren Stringvergleich mit Aufrufen von `length()`, `equals(Object)` und `substring(int, int)`. Im Besonderen ist der Aufruf von `substring(int, int)` kompliziert und unleserlich:

```
// Gleicher Inhalt?
if (newTaskData.equals(existingTaskData.substring(0, newTaskData.length())))
```

Er scheint zudem unsinnig, da bei einer Umwandlung gleicher Eingabewerte auch gleiche Stringrepräsentationen geliefert werden sollten. Somit sollte eine Kürzung des einen Strings auf die Länge des anderen überflüssig sein.

Um eine mögliche Erweiterung oder Fehlersuche zu erleichtern, ist es hilfreich, für einfacheren Sourcecode und auch allgemein für mehr Lesbarkeit zu sorgen. Allerdings sollte man dabei vorsichtig sein. Zur Vereinfachung führen wir folgende Änderungen durch: Der Längenvergleich mit dem Operator '>' wird durch den Operator '! =' ersetzt. Dadurch kann der `substring(int, int)` -Aufruf entfallen: Im Falle gleicher Länge ist ein einfacher Aufruf von `equals(Object)` ausreichend:

```
if (newTaskData.length() != existingTaskData.length())
{
    log.warn("new task is different from existing (wrong length).");
    return TaskCompareResults.FAILURE_DIFFERENT_TASK_CONTENT;
}

if (newTaskData.equals(existingTaskData))
{
    log.info("task " + aTask + " already exists.");
    return TaskCompareResults.OK_EXISTING_TASK;
}
```

Zum Zeitpunkt dieser Änderungen waren ein Kollege und ich aber etwas vorschnell, ja gar ein wenig nachlässig oder denkfaul. Da es schon spät war, schrieben wir für diese scheinbar einfachen Änderungen keine weiteren Unit Tests. Außerdem lieferten alle existierenden Unit Tests positive Testresultate und auch ein einfacher Applikationstest war erfolgreich. Es schien so, als ob eine Vereinfachung des Sourcecodes erzielt werden konnte. Allerdings stellte sich bei einem späteren, umfangreicheren Applikationstest heraus, dass durch die Änderungen ein Anwendungsfall falsche Ergebnisse lieferte.

Tipp: Sichere Änderungen möglichst immer durch Unit Test ab

Wenn man Änderungen in Sourcecode durchführt, so sollte man begleitend immer Unit Tests zur Absicherung der Änderungen schreiben, um sich vor (leichten) Fehlern, insbesondere Flüchtigkeits- und Denkfehlern (in späteren Abendstunden), zu schützen.

Schritt 4: Fehlersuche und -korrektur durch Log-Auswertung

Durch den in den Applikationstests beobachteten Fehler wurde klar, dass die vorhandenen Unit Tests nicht alle Fälle abdecken. Wir waren zunächst etwas ratlos. Es musste sich irgendwie ein Fehler durch unsere Änderungen ergeben haben. Um dem Problem auf die Spur zu kommen, wurden die Applikationstests mit detaillierteren Logging-Einstellungen wiederholt. Glücklicherweise lieferte die Methode `compareTaskData(ITask, String)` nach Aktivierung des Debug-Levels genauere Informationen über die verglichenen Werte – vereinfacht nach dem Schema:

```
newTaskData         "<same>"
existingTaskData    "<same>#<back_up_content>"
```

Durch diese Log-Ausgaben wurde uns dann schlagartig klar, wie der Fehler zustande kam: Der Vergleichswert `existingTaskData` ist immer dann länger als die neu erzeugte Stringrepräsentation `newTaskData`, wenn optionale Backup-Informationen vorhanden sind. Dieses Detail hatten wir bei unseren Vereinfachungen nicht bedacht und unvorsichtigerweise Änderungen vorgenommen.

Sowohl der scheinbar unlogische Vergleich mit dem Operator '>' statt mit dem Operator '! =' als auch das komplizierte `equals(Object)`-Konstrukt mit den `substring(int, int)`-Aufrufen hatten also einen Grund gehabt: Die ursprüngliche Realisierung beschneidet den String `existingTaskData` auf die Länge der Eingabe `newTaskData`, und somit ist der komplexe `equals(Object)`-Vergleich erfolgreich. Diese Begründung ist leider nirgendwo erwähnt und ergibt sich auch nicht offensichtlich aus dem Programm. In der durch Schritt 3 veränderten Version der Methode `compareTaskData(ITask, String)` werden die Strings komplett per `equals(Object)` verglichen, und der Vergleich schlägt aus offensichtlichen Gründen fehl.

Tipp: Fehlersuche und Logging

Ein gut konfigurierbares und detailliertes Logging mit Protokollierung von allen Eingangsparametern, Ergebnissen und von Vergleichswerten kann im Nachhinein eine Fehlersuche enorm vereinfachen.

Wir führen eine Rückkorrektur des Programms durch. Allerdings sorgen wir für etwas mehr Lesbarkeit: Wir nutzen eine Hilfsvariable `truncated` sowie erklärende Kommentare, die anderen Entwicklern helfen sollen, die fragwürdigen Programmstellen zu verstehen. Unter anderem soll dies vor dem Fehler bewahren, hier irrtümlich ähnliche, vereinfachende Änderungen durchzuführen.

```
// Wenn die neue Länge größer als die alte ist, sind es unterschiedliche Tasks
if (newTaskData.length() > existingTaskData.length())
{
    log.warn("new task is different from existing (wrong length).");
    return TaskConstants.FAILURE_DIFFERENT_TASK_CONTENT;
}
```

```
// Prüfe auf inhaltliche Gleichheit, aber nur bis zur Länge des neuen Tasks
// Grund: an alten Tasks können Zusatzinformationen hängen, die nicht in den
// Vergleich einbezogen werden dürfen
final String truncated = existingTaskData.substring(0, newTaskData.length());
if (newTaskData.equals(truncated))
{
    log.info("task " + existingTaskData + " already exists.");
    return TaskConstants.OK_EXISTING_TASK;
}
```

Anhand der dargestellten Probleme und des Lösungswegs erkennt man, dass es viel teurer und ärgerlicher war, auf weitere Testfälle zur Absicherung zu verzichten als diese bereitzustellen und dann mit viel mehr Sicherheit die Änderungen durchführen zu können. Möglicherweise hätten auch weitere Tests diesen Spezialfall nicht aufgedeckt, z. B. wenn diese nur die Fälle behandelt hätten, in denen der Inhalt von `newTaskData` länger oder gleich lang wie der von `existingTaskData` ist. Jedoch sind existierende, aber unvollständige Testfälle bereits eine gute Basis für neue Testfälle. Wahrscheinlich hätte man die Unvollständigkeit dann schneller erkannt als ohne Tests und sich gefragt: »Wieso existiert eigentlich kein Test für den Fall, dass der Wert von `existingTaskData` mehr Zeichen enthält als der von `newTaskData`?« Somit hätte man dann leichter den Unterschied in der Semantik zwischen Originalfassung und Änderung entdeckt.

Tipp: Iterative Entwicklung von Tests aus gemeldeten Fehlern

Niemand kann Unit Tests schreiben, die alle möglichen Fehlersituationen berücksichtigen. Daher sollte man Unit Tests iterativ entwickeln. Wenn jeder beobachtete Programmfehler als Vorlage für einen weiteren Unit Test dient, gewinnt man dadurch die Sicherheit, dass dieser Fehler niemals mehr im Programm auftreten wird, ohne durch Tests entdeckt werden zu können.

Zwischenfazit

Zwar wurde durch Unachtsamkeit in Schritt 3 eine Änderung vorgenommen, die zu einem Programmfehler geführt hat. Dies wäre jedoch normalerweise nicht passiert, wenn man von Anfang an intensiver mit Unit Tests gearbeitet und konsequent darauf geachtet hätte, die Änderung mit einem Test und durch sorgfältige Auswahl von Testdaten abzusichern, wie dies im folgenden Schritt nachgeholt wird. ***Es liegt demnach ein Problem der Durchführung und nicht ein prinzipielles Problem von Unit Tests und Testbarkeit von Legacy-Code vor***. Berechtigterweise stellt sich allerdings die Frage: »Wer testet die Tests?« Anhand dieser Frage wird klar, dass ***beim Erstellen von Tests mit ebenso großer Sorgfalt wie beim Entwickeln des Applikationscodes vorgegangen werden muss***.

Im Nachhinein konnte dieses Problem jedoch zum Positiven genutzt werden: Der Sourcecode wurde leicht überarbeitet und zudem mit aussagekräftigen Kommentaren versehen, die ähnliche Fehler sehr unwahrscheinlich machen.

Schritt 5: Komplettierung der Unit Tests

Tests mit künstlichen Eingabedaten wurden bereits formuliert. Allerdings ist durch Unit Tests noch nicht die Situation beschrieben, die in Schritt 3 zu Problemen geführt hat und in Schritt 4 erkannt und im Applikationscode behoben worden ist. Damit ergibt sich ein einzelner, wenn auch wichtiger zu ergänzender Testfall, der auf realen Daten basiert und Backup-Daten enthält:

```
public void testCompareTaskDataWithHistory()
{
    final String newTaskData = "6#46#0#0#50#996#19#12252011";
    // Optionale History soll beim Vergleich nicht betrachtet werden
    final String existingTaskData = "6#46#0#0#50#996#19#12252011#1#490#1";

    final TaskCompareResults value = TaskUtilsV2.compareTaskData(newTaskData,
          existingTaskData);
    assertTrue(TaskCompareResults.isOkValue(value));
}
```

Fazit: Positive Effekte beim Einsatz von Unit Tests

Die bereits beschriebenen positiven Auswirkungen durch den Einsatz von Unit Tests konnten wir anhand der zuvor vorgestellten Beispiele nachvollziehen. Es zeigt sich Folgendes: Unit Tests …

- geben mehr Sicherheit, wenn weitere Änderungen erfolgen müssen.
- lassen sich sehr gut schrittweise erweitern.
- helfen dabei, ein besseres Verständnis für den Applikationscode zu entwickeln.
- ermöglichen es, sinnvolle Kommentare einzufügen und Funktionalität durch Testfälle zu dokumentieren.
- helfen dabei, Abhängigkeiten von anderen Packages und Klassen zu reduzieren.

20.4 Fortgeschrittene Unit-Test-Techniken

In diesem Unterkapitel betrachten wir zunächst mögliche Schwierigkeiten beim Unit-Testen. Als Abhilfe lernen wir Stellvertreterobjekte, auch Test-Doubles genannt, kennen. Vielen sind die Begriffe Stubs und Mocks sicher geläufiger. Beide Begriffe leiten sich aus dem Englischen ab: »stub« = Stumpf, Stummel und »to mock« = nachahmen, vortäuschen. Durch den Einsatz von Stubs und Mocks wird es möglich, andere Komponenten, wie etwa eine Datenbank, für Testfälle zu ersetzen bzw. deren Verhalten zu simulieren. Die folgenden Abschnitte stellen jeweils an einem Beispiel den Einsatz eines Stubs bzw. eines Mocks zum Testen vor. Abschließend wird kurz auf das Thema »Testen privater Methoden« eingegangen.

Schwierigkeiten beim Unit-Testen

Nachdem wir mittlerweile wissen, dass Unit Tests wichtig für die Qualität sind, sollten wir uns aber noch folgende Frage stellen: »Was erschwert es uns, gute Unit Tests zu schreiben?« Wenn man ein wenig nachdenkt, so sind es fast dieselben Dinge, die schon beim Programmieren das Entwickeln erschweren können. Rekapitulieren wir kurz, was in den bisherigen Kapiteln dazu gesagt wurde oder was Sie vielleicht auch schon am eigenen Leib erfahren haben. Bevor man entwickelt, sollte man zunächst einmal wissen, was realisiert werden soll und wie es funktionieren soll. Wenn man dann zu den Anforderungen die passende Software entwickelt, dann ist ein zentraler Punkt die Namensgebung und ein problemangepasstes Design mit klaren Zuständigkeiten. Oftmals sieht die Realität aber anders aus. Zudem bieten Klassen zu viel Funktionalität und besitzen meistens zu viele Abhängigkeiten zu anderen Klassen. Das wiederum führt zu Fragilität: Ändert man an einer Stelle, so zerbricht etwas an anderer Stelle. Kommen wir nun zu Unit Tests zurück. Auch dort helfen uns Informationen zur gewünschten Funktionalität, um adäquate Testfälle erstellen zu können und nicht nur Pseudotests für triviale `get()`-/`set()`-Methoden zu schreiben. Zudem profitieren wir von gelungenen, aussagekräftigen Namen der Testmethoden, um das Wesentliche des Testfalls erkennen zu können (ohne dazu die Implementierung unbedingt anschauen zu müssen). Um jeden Testfall möglichst knackig formulieren zu können, sollten nicht zu viele Abhängigkeiten existieren. Genau dazu nutzt man die im Folgenden vorgestellten Test-Doubles. Der Name ist recht plakativ und ähnelt dem Stunt Double aus Filmen. Genau diese Analogie sollten Sie im Hinterkopf behalten: Ein Test-Double ist ein Stellvertreter für ein Anwendungsobjekt in einem Testfall, um Abhängigkeiten etwa zum Dateisystem oder zu Datenbanken aufzulösen und eine Unit besser testbar zu machen.

20.4.1 Test-Doubles

Wie gerade erwähnt, bezeichnet man Objekte als Test-Doubles, die als Stellvertreter für Applikationsobjekte oder Fremdsysteme zur Erleichterung der Programmierung

von Tests genutzt werden. Dadurch kann man Zustand oder Verhalten prüfen, ohne immer ein ganzes System bereitstellen zu müssen. Das ist hilfreich, weil Unit Tests nur kleine Ausschnitte aus dem Programmverhalten überprüfen sollen. Zudem soll dies isoliert erfolgen, d. h. ohne die Interaktion mit anderen Software- oder Hardwarekomponenten. Oftmals wird die Programmfunktionalität jedoch durch eine Vielzahl miteinander verknüpfter Objekte bereitgestellt. Mitunter existieren darüber hinaus Abhängigkeiten von externen Systemen. Beides erschwert die Testbarkeit. Tatsächlich liegen Abhängigkeiten schon dann vor, wenn etwa Methoden untersucht werden sollen, die mit einer Vielzahl von Objekten interagieren oder die Daten aus einer Datei oder einer Datenbank lesen oder Ergebnisse dort speichern. Durch den Einsatz von Abstraktionen kann man eine Unabhängigkeit von externen Systemen erreichen: Wurde daran nicht im Vorfeld gedacht, so muss entweder durch Refactorings oder andere Umbaumaßnahmen für eine losere Kopplung, z. B. durch Einführen eines Interface, gesorgt werden, um eine bessere Testbarkeit zu erzielen. In den Testfällen nutzt man dann Test-Doubles, die das von der Anwendung benötigte Interface erfüllen. Dabei kommen die nachfolgend kurz vorgestellten Varianten zum Einsatz.[14]

Dummy

Unter einem Dummy versteht man einen Platzhalter, der keine Funktionalität bereitstellt. Obwohl das zunächst wenig hilfreich scheint, erleichtert dies, Abhängigkeiten aufzulösen, wenn man an eine Methode gewisse Parameter eines speziellen Typs übergeben muss. Im einfachsten Fall reicht eine `null`-Referenz, um lediglich die Parameterliste zu erfüllen. Bei `null`-Referenzen setzt das aber voraus, dass auf diese nicht geprüft oder darauf zugegriffen wird, da ansonsten `NullPointerExceptions` drohen. Daher bietet es sich in der Regel an, Null-Objekte als Dummy zu verwenden.

Stub, Spy oder Fake

Deutlich komplexer als Dummies sind ***Stub***, Spy und Fake. Diese möchte ich in diesem Buch nicht groß unterscheiden. Für mich stellt ein Stub, Spy und ein Fake eine meistens rudimentäre, manchmal nur abgespeckte, aber funktionierende Implementierung eines anderen Objekts dar. Dabei kann ein Stub beispielsweise vordefinierte Rückgabewerte bereitstellen, aber auch ein etwas ausgeklügelteres Verhalten besitzen, etwa Daten und Zustand speichern oder Methodenaufrufe protokollieren. Manche würden dann eher von einem Spy sprechen. Ein Fake wäre ein Exemplar eines Test-Doubles mit einer abgewandelten Funktionsweise etwa eine In-Memory-Datenbank (z. B. durch Maps realisiert) statt einer realen Datenbank.

Mock

Schließlich verbleiben ***Mocks***. Diese dienen zum Überprüfen von Verhalten in Form

von erwarteten Methodenaufrufen. Werden im Testfall durch die Abarbeitung der Anwendungsfunktionalität nicht die zuvor spezifizierten Methoden aufgerufen, so führt dies zu einem Fehler.

20.4.2 Testen mit Stubs

Unter ***Stubs*** versteht man Hilfsobjekte, die dabei helfen, Abhängigkeiten zu vermeiden. Dazu stellen sie einen Ersatz für die von der zu testenden Unit angesprochenen Komponenten dar und liefern benötigte Werte und das erwartete Verhalten. Damit man Stubs nutzen kann, müssen die in Tests verwendeten Komponenten über Interfaces oder abstrakte Klassen, jedoch nicht direkt angesprochen werden. In der Praxis ist dies allerdings nicht die Regel, sondern es existieren verschiedene Abhängigkeiten. Im Anschluss wird beschrieben, wie man die benötigten Voraussetzungen schafft, damit man mit Stubs arbeiten kann.

Einführendes Beispiel

Erinnern wir uns an das Beispiel der Ausgabegeräte aus Abschnitt 20.3.1. Dort wurde bereits initial im Design eine Abstraktion für Ausgabegeräte vorgesehen. Somit erreichen wir eine Unabhängigkeit von der realen Hardware. Davon profitieren wir, wenn wir eine Stub-Klasse `SimulationDisplay` implementieren, die das Interface `IDisplay` erfüllt. Statt Ausgaben auf einem physikalischen Gerät vorzunehmen, dient diese Testklasse dazu, eingehende Nachrichten einfach auf der Konsole auszugeben. Eine ausgefeiltere Variante könnte die Nachrichten in einer Liste speichern und Zugriff darauf bereitstellen, um weiter gehende Prüfungen in Tests durchführen zu können.

Die einfache Variante der Klasse wird wie folgt realisiert:

```
public final class SimulationDisplay implements IDisplay
{
    public SimulationDisplay()
    {
    }

    public boolean displayMsg(final DisplayMsg displayMsg)
    {
        System.out.println("SimulationDisplay - got msg '" + displayMsg +
"'");
        return true;
    }
}
```

Ausgangssituation und Vorbereitungen zur Testbarkeit

Nachfolgend betrachten wir als Beispiel eine Utility-Klasse `CSVUtils`, die eine CSVDatei mithilfe einer `FileInputStream`-Instanz und der Methode

`readCSV()` einliest. Abbildung 20-12 zeigt das Klassendiagramm der Ausgangssituation mit den oben genannten Abhängigkeiten auf konkrete Klassen.

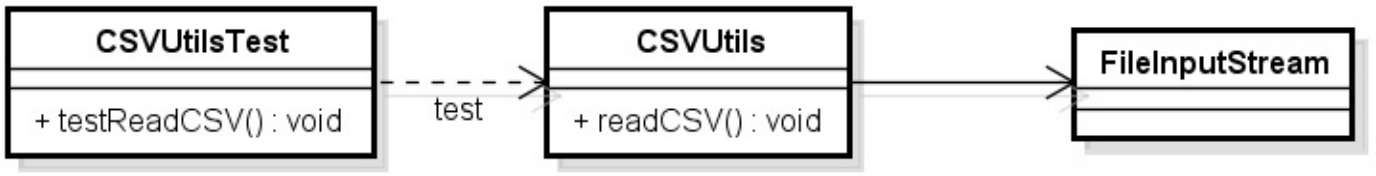


Abbildung 20-12 *Einführen eines Stubs – Ausgangssituation*

Ich stelle nun Schritt für Schritt vor, wie man Abhängigkeiten lösen kann und wie dann durch das Einführen eines Stubs, das Erstellen von Unit Tests erleichtert wird.

Schritt 1: Nutzen von Übergabeparametern

Um die Abhängigkeit zwischen Utility-Klasse und der Streamklasse etwas zu lösen, wird in diesem Schritt, wie in Abbildung 20-13 dargestellt, ein Objekt der Klasse `FileInputStream` an die Methode `readCSV(FileInputStream)` übergeben.

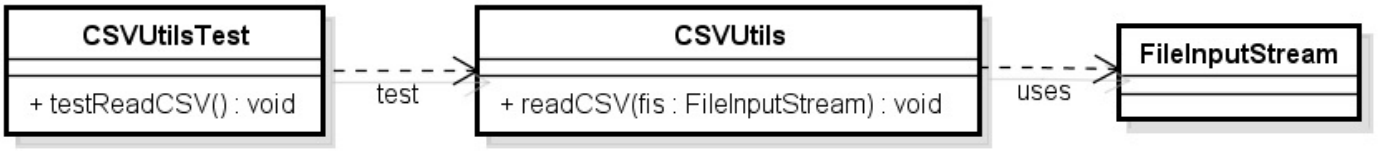


Abbildung 20-13 *Einführen eines Stubs – Schritt 1*

Schritt 2: Loslösung von Abhängigkeiten

Um die Testbarkeit zu verbessern, sollte man statt der konkreten Klasse `FileInputStream` besser die abstrakte Basisklasse `InputStream` nutzen, wie dies Abbildung 20-14 zeigt. Ist eine solche abstrakte Basisklasse oder ein adäquates Interface nicht vorhanden, so sollte man diese Abstraktion erzeugen, um diesen Schritt durchführen zu können.

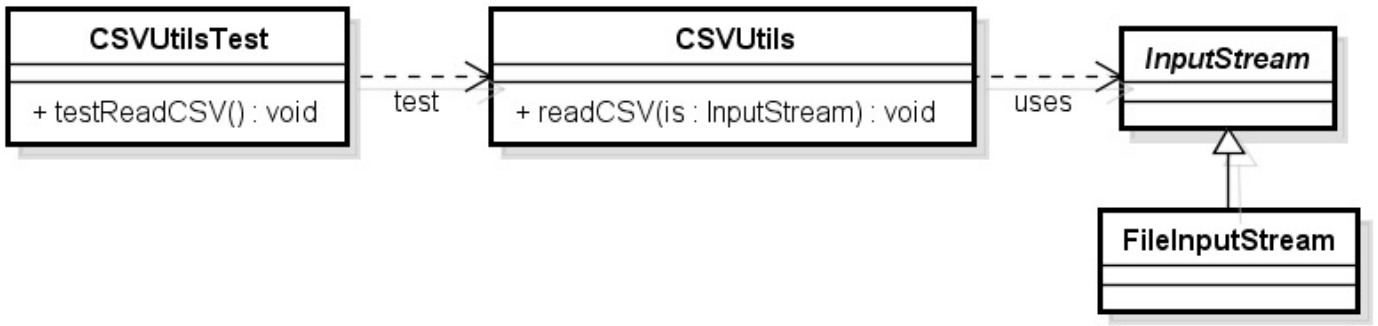


Abbildung 20-14 *Einführen eines Stubs – Schritt 2*

Schritt 3: Erzeugen eines Test-Stubs

Für Unit Tests wollen wir in der Regel nicht mit dem Dateisystem interagieren, sondern Daten aus einer beliebigen Testdatenquelle einlesen. Dazu wird ein Ersatz für einen `FileInputStream` benötigt. In Abbildung 20-15 ist gezeigt, dass dazu

hier die Klasse `InputStreamStub` genutzt wird, die alle Methoden der gemeinsamen Basisklasse `InputStream` geeignet implementiert.

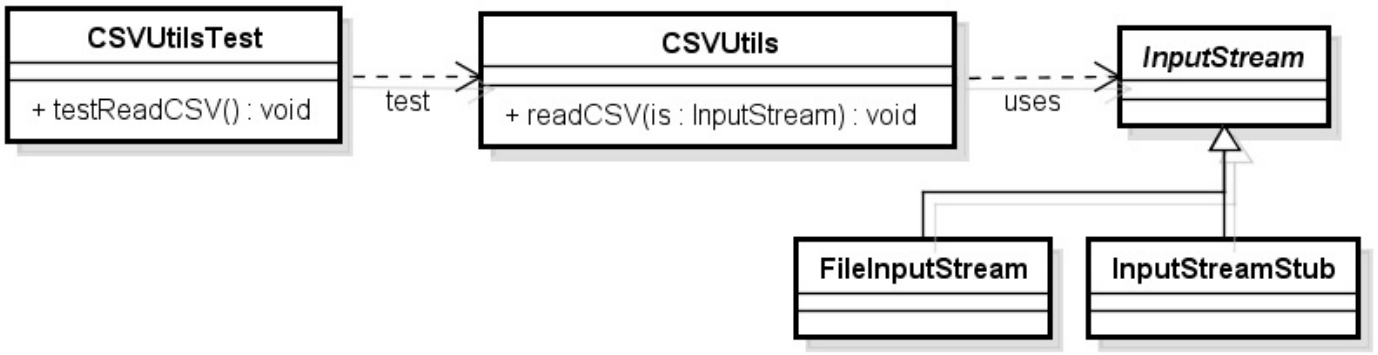


Abbildung 20-15 *Einführen eines Stubs – Schritt 3*

20.4.3 Testen mit Mocks

Genau wie Stubs dienen Mock-Objekte als Ersatz für eine andere Komponente. Im Gegensatz zu Stubs, die tatsächlich Funktionalität bereitstellen, sind Mocks auf das Objektverhalten selbst ausgerichtet. ***Es ist demnach eine komplett andere Denkweise als bei den intuitiv besser erfassbaren Stubs gefordert.***

Betrachten wir das prinzipielle Vorgehen am Beispiel der folgenden Klasse `Members`, die zur Datenspeicherung eine `List<String>` verwendet, die als Parameter im Konstruktor übergeben wird. Um eine Testbarkeit durch Mocks zu erreichen, wird von der konkreten Realisierung der Liste abstrahiert.

```
public final class Members
{
    private final List<String> members;

    Members( final List<String> persons)
    {
        this.members = persons;
    }

    public boolean registerMember( final String member)
    {
        return members.add( member);
    }

    public boolean deregisterMember( final String member)
    {
        return members.remove( member);
    }

    // …
```

Mocks nutzt man zum Testen von Interaktionen. Im Gegensatz zu Stubs betrachtet man aber nicht die Veränderungen im Objektzustand. Beim Mock-Ansatz ist man beispielsweise nicht daran interessiert, ob sich nach einem Aufruf der Methode `registerMember( String)` die Anzahl der gespeicherten Elemente erhöht hat. Wie kann man dann aber überprüfen, ob ein korrektes Verhalten vorliegt? Die

Antwort ist verblüffend, obwohl naheliegend: ***Man ermittelt die für eine Aktion notwendigen Methoden.*** Dazu werden in einem sogenannten Aufzeichnungsmodus die Erwartungen über Methodenaufrufe festgelegt. ***Es wird eine Art Ablauf wie in einem Sequenzdiagramm beschrieben***. Das Vorgehen ist komplizierter als bei Stubs. Deshalb ist es aufwendig und fehlerträchtig, Mocks als Entwickler selbst zu realisieren. Abhilfe leisten spezielle Frameworks, wie etwa EasyMock[15] oder Mockito[16], die beide frei verfügbar sind.

Um die Arbeitsweise von Mocks zu verstehen, betrachten wir ein einfaches Beispiel. Es soll folgender Ablauf geprüft werden: Zunächst werden die beiden Stringobjekte `"Person1"` und `"Person2"` über einen Aufruf der Methode `registerMember(String)` gespeichert. Anschließend wird `"Person1"` über einen Aufruf von `deregisterMember(String)` wieder gelöscht.

Aufgrund des beschriebenen Testfalls ermitteln wir die dafür notwendigen Aufrufe an die Liste, die als Sequenzdiagramm in Abbildung 20-16 dargestellt sind.

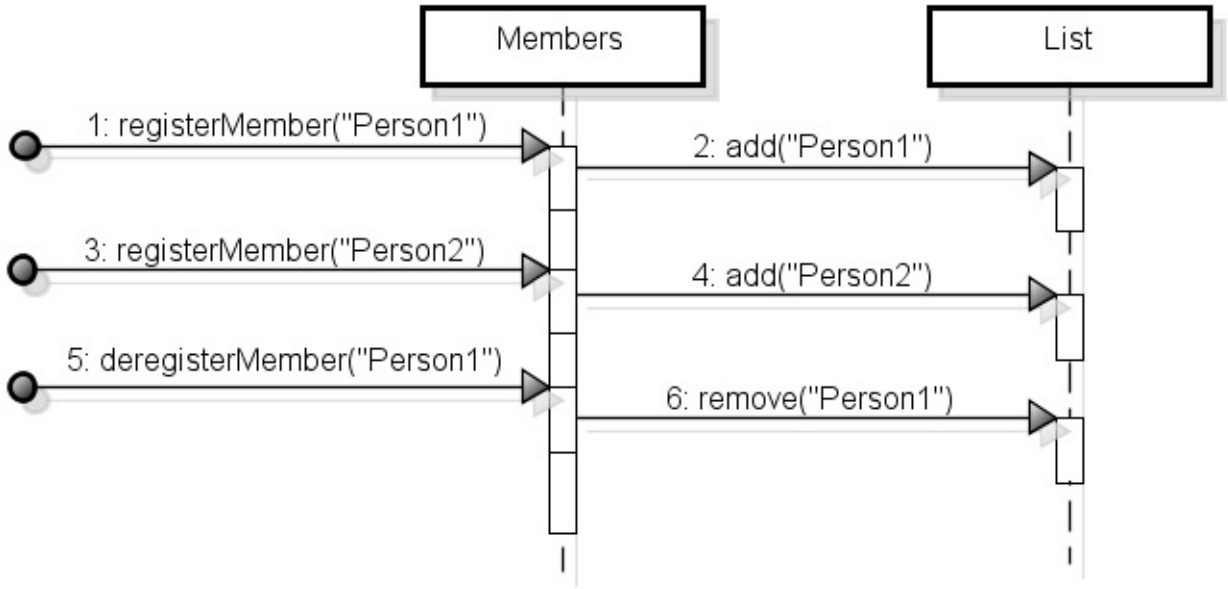


Abbildung 20-16 *Testablauf Mock-Beispiel*

Aufzeichnung von Erwartungen

Als Erwartung kann man aufgrund der Methodenaufrufe an die Klasse `Members` und basierend auf dem Sequenzdiagramm formulieren, dass es zu zwei Aufrufen von `add(String)`, gefolgt von einem Aufruf von `remove(String)` auf der Liste kommt.

Die im Sequenzdiagramm dargestellten Aufrufe an die Liste zur Speicherung von Mitgliedern soll nun durch einen Mock ersetzt werden. Dazu setzen wir zum Test der Klasse `Members` das Framework EasyMock ein, das es erlaubt, über die Methode `createMock(Class<T>)` ein passendes Mock-Objekt zu erzeugen. Anschließend »schiebt« man per Dependency Injection den Mock der zu testenden Klasse unter. Danach kann man den Mock für den Test so präparieren, dass das für diesen Testfall

gewünschte Verhalten aufgezeichnet wird. Dazu ruft man die entsprechende Methode explizit auf (etwa `add()`) und formuliert dann die Erwartung, dass die Methode aufgerufen wird (`expectLastCall()`) und einen Rückgabewert liefert (`andReturn()`). Das Listing zeigt dies für das Hinzufügen und Entfernen von Elementen:

```
public static void main(final String[] args)
{
    // Mock erstellen und nutzen
    final List<String> listMock = createMock(List.class);
    final Members clubMembers = new Members(listMock);

    // Aufzeichnung
    listMock.add("Person1");
    expectLastCall().andReturn(true);
    listMock.add("Person2");
    expectLastCall().andReturn(true);
    listMock.remove("Person1");
    expectLastCall().andReturn(true);

    // …
```

Durchführung von Tests

Zum Testen versetzt man den Mock per Aufruf von `replay()` in einen Abspielmodus. Dann führt man die zu prüfenden Aktionen normal aus, wie dies durch die folgenden Programmzeilen exemplarisch gezeigt ist:

```
   // …

    replay(listMock);

    clubMembers.registerMember("Person1");
    clubMembers.registerMember("Person2");
    // Simuliere hier bewusst einen Fehler!!!
    clubMembers.deregisterMember("Person6");
}
```

Es wird in diesem Fall nicht wie aufgezeichnet "`Person1`", sondern tatsächlich "`Person6`" abgemeldet. Damit verhält sich die Applikation anders als während der Aufzeichnung für den Testfall spezifiziert. Das Mock-Framework erkennt dieses Fehlverhalten und erzeugt folgende Warnmeldung beim Ausführen des Programms:

```
Exception in thread "main" java.lang.AssertionError:
  Unexpected method call remove("Person6"):
    remove("Person1"): expected: 1, actual: 0
```

Fazit

Hier wurde lediglich ein kurzer Einstieg in das komplexe Thema Testen mit Mocks und Stubs gegeben. Weitere Informationen finden Sie im Internet: Ein lesenswerter Artikel zu dem Thema ist unter `http://martinfowler.com/articles/mocksArentStubs.html`

verfügbar. Eine weiterführende Betrachtung von Mocks liefert das Buch »Practical Unit Testing with JUnit and Mockito« [52] von Tomasz Kaczanowski.

Wenn man sich die verschiedenen Arten von Test-Doubles und ihre Einsatzgebiete so anschaut, dann erahnt man, dass man eigentlich für unterschiedliche Arten und Ausrichtungen von Tests auch unterschiedliche Arten von Test-Doubles benötigt:

- Mocks helfen eher bei einem High-Level-Test von Interaktionen.
- Fake, Spy und Stub sind meistens besser zum Prüfen von Low-Level-Funktionalität geeignet.

20.4.4 Unit Tests von privaten Methoden

Manchmal möchte man auch die privaten Bestandteile von Klassen testen, weil diese komplexere Berechnungen enthalten. Allerdings hat man darauf außerhalb der Klassen keinen Zugriff. Also, wie testet man private Methoden? Dazu existieren unter anderem folgende Möglichkeiten:

1. Die einfachste Möglichkeit ist, die Sichtbarkeit einer zu testenden Methode auf Package-private zu ändern. Möchte man die Testbarkeit als Grund dafür explizit klarmachen, so kann man einen Kommentar `/* Test */` o. Ä. oder die Annotation `@VisibleForTesting` aus Google Guava nutzen.
2. Manchmal ist das Design auch noch nicht ausgereift und dann bietet es sich an, ein paar der privaten Methoden in eine eigene Klasse auszulagern und dort mit höherer Sichtbarkeit bereitzustellen.
3. Mithilfe von Reflection (vgl. Abschnitt 8.1) kann man sich Zugriff auf private, zu testende Methoden verschaffen. Allerdings muss dazu der Sichtbarkeitschutzmechanismus mit `setAccessible(true)` außer Kraft gesetzt werden.

Die erste Möglichkeit ist akzeptabel, da die Veränderung der Sichtbarkeit die Kapselung nur minimal beeinflusst, das Testen aber enorm erleichtert. ***Das ist der von mir bevorzugte Weg***. Die zweite Variante ermöglicht es, eher allgemeingültige Hilfsfunktionalität in Utility-Klassen bereitzustellen. Die dritte Möglichkeit hat diverse Nachteile: Zum einen müssen die Namen der aufzurufenden Methoden als Strings in den Sourcecode aufgenommen werden, was die Gefahr von Problemen durch Refactorings enorm erhöht: Werden Methoden umbenannt oder Signaturen verändert, so schlagen als Folge Tests fehl, die zuvor funktionierten, weil die gewünschten Methoden nicht mehr per Reflection gefunden werden. Für öffentliche Methoden eines nahezu unveränderlichen APIs wäre diese Einschränkung weniger problematisch – für private Methoden, an denen häufig Änderungen (Parameterliste, Namensgebung usw.) erfolgen, ist dies jedoch inakzeptabel. Außerdem müssen beim Zugriff per Reflection diverse Exceptions abgefangen werden, was den Testcodes

aufbläht und dessen Lesbarkeit erschwert.

Hinweis: Öffentliche vs. Package-private Methoden testen

Bezüglich der durch Unit Tests zu überprüfenden Methoden gibt es unterschiedliche Auffassungen. Manche Entwickler propagieren den Test der nach außen angebotenen Schnittstelle, also der öffentlichen Methoden. Andere bevorzugen dagegen den Test der kleineren Bausteine (`private`- und `protected`-Methoden). Beides hat seine Berechtigung: Werden ausschließlich öffentliche Methoden geprüft, so erfolgt eher ein Test auf Verhaltensebene der Business-Methoden, der zum Teil in Richtung Integrationstest geht. Interne Probleme oder Berechnungsfehler kann man besser durch Tests von Methoden mit den Sichtbarkeiten `protected`, Package-private und `private` erkennen. Finden jedoch nur solche Tests statt, so wird das Zusammenspiel der Methoden vernachlässigt. Demnach sollte man möglichst beide Arten von Tests durchführen. ***Unabhängig von der Sichtbarkeit sollte man sich auf kompliziertere Programmelemente konzentrieren, da diese eher Fehler enthalten als verständliche und strukturell einfachere Programmteile***. Oft findet sich Komplexität nicht in öffentlichen Methoden, sondern vermehrt in denen der Sichtbarkeit `protected` und eingeschränkter. Allerdings beschreiben öffentliche Methoden das Verhalten der Klasse, das sicher auch getestet werden sollte. Damit sind wir wieder bei der obigen Aussage: ***Teste die wichtigen oder komplexeren Dinge***.

20.5 Unit Tests mit Threads und Timing

Ich greife hier das Beispiel der bereits in Abschnitt 20.3.1 vorgestellten Ausgabegeräte, die ähnlich zu LCD-Monitoren arbeiten und über ein eigenes Protokoll mit Befehlsnachrichten angesteuert werden können, wieder auf. Wir erinnern uns, dass bei einer naiven Ansteuerung jede eingehende Nachricht in einem eigenen Speicherschlitz landet und dadurch bei kürzeren Nachrichten viel vorhandener Platz ungenutzt verfällt. Sendet man nun in schneller Folge einzelne Befehle an ein Ausgabegerät, so kann es zu Problemen bei der Abarbeitung der Befehle kommen. Dies ist immer dann der Fall, wenn Befehle schneller gesendet als bearbeitet werden und daher die Anzahl der Schlitze des Eingabepuffers nicht ausreicht. Als Folge werden gespeicherte Befehle durch neu eintreffende Befehle überschrieben. Eine mögliche Lösung besteht darin, die Ansteuerung geschickter zu implementieren: Man kann in einer Nachricht mehrere Befehle kombinieren und damit weniger Schlitze für die gleiche Anzahl an Befehlen belegen. Dies deutet Abbildung 20-17 an. Mögliche Befehlsnachrichten unterschiedlicher Länge sind dort von 1 bis 6 durchnummeriert.

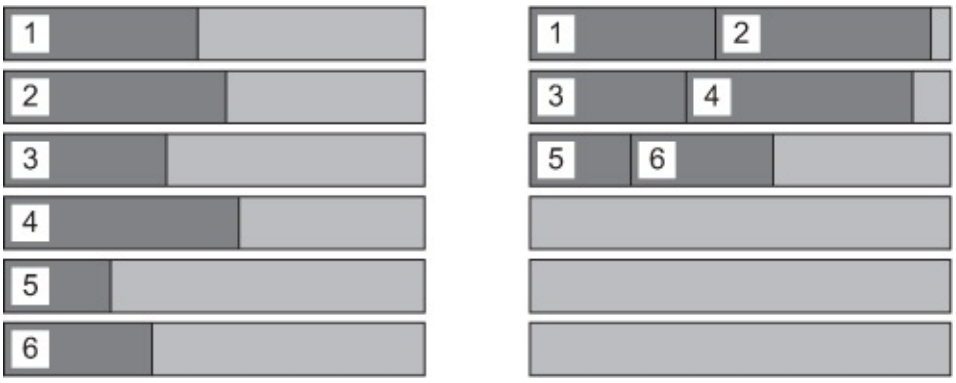


Abbildung 20-17 *»Intelligenter« Nachrichtenpuffer*

Realisierung der Funktionalität

Als Umsetzung wird aufseiten des Befehlssenders eine Aggregationsfunktion für die zu sendenden Befehle durch eine Klasse `MessageConcatenator` realisiert. Dazu werden eintreffende Befehlsnachrichten zunächst gesammelt und eine Gesamtnachricht erst dann erzeugt, wenn nahezu die maximale Speicherkapazität eines Schlitzes im Eingangspuffer erreicht ist. Zur Realisierung dieser Funktionalität klinkt man sich in den `sendMessage(DisplayMsg)`-Aufruf der Klasse `Controller` ein, der zuvor direkt an das Gerät durch Aufruf der Methode `displayMsg(DisplayMsg)` geschrieben hat. Das resultierende Klassendiagramm visualisiert Abbildung 20-18.

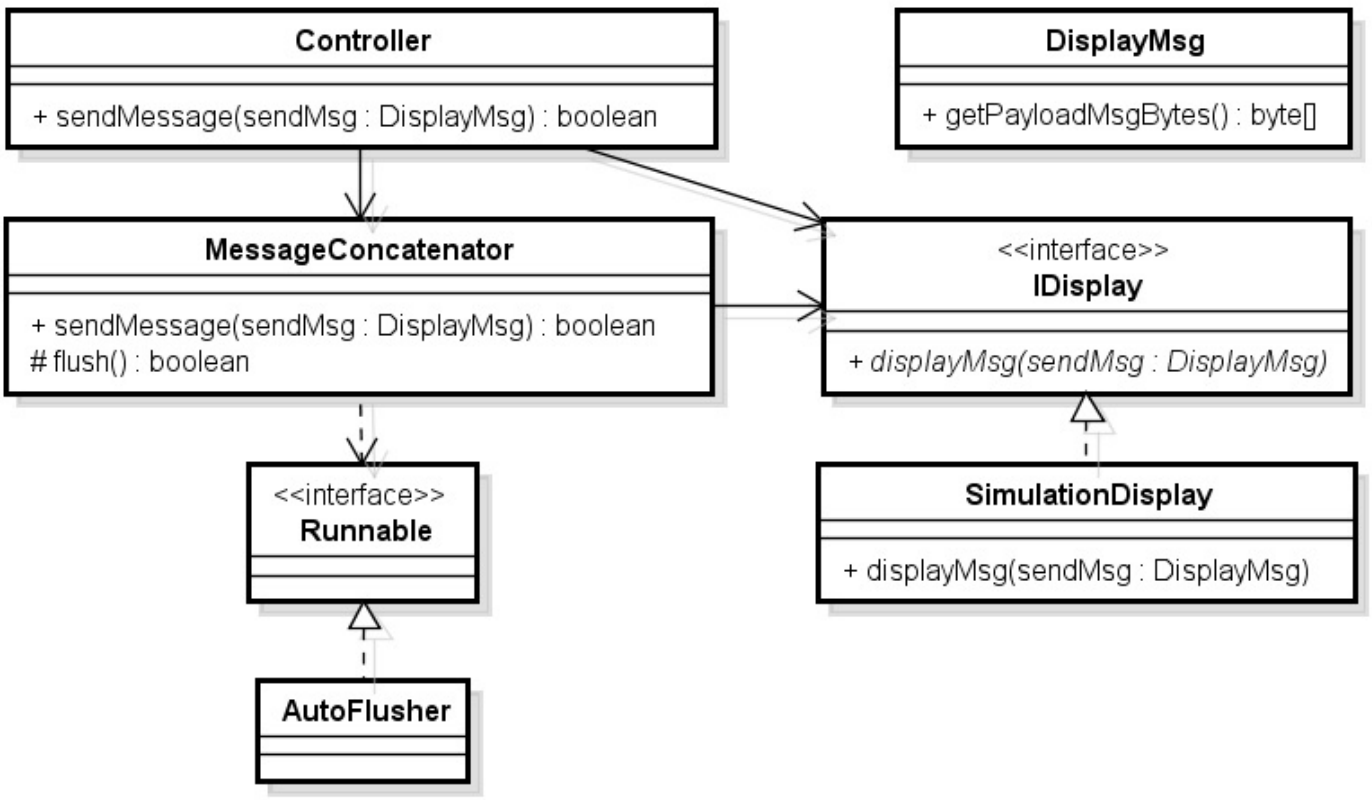


Abbildung 20-18 *Integration der Klasse* `MessageConcatenator`

Konkretisierung des Algorithmus

Die Methode `sendMessage(DisplayMsg)` der Klasse `MessageConcatenator` prüft bei Eintreffen einer Nachricht vom Typ `DisplayMsg`, ob mit deren Nutzlast und den zuvor gesammelten Nachrichten die Länge von 250 Bytes überschritten wird. Solange dies nicht der Fall ist, werden eingehende Nachrichten gesammelt. Wird die Schwelle dagegen erreicht, so kommt es zu einem Versenden der zwischengespeicherten Nachrichtendaten über den Aufruf von `flush()` und die neue Nachricht wird im Puffer gespeichert. Eine interne Methode `flushImpl()` realisiert die Verbindung zum Ausgabemedium. Die gesammelten Nachrichten werden dort als eine Nachricht vom Typ `DisplayMsg` erzeugt und versendet.

Notwendige Verbesserungen

So gut die Idee des Sammelns von Nachrichten zum Optimieren der Verarbeitung und zur Ausnutzung des Speichers auch ist, so gibt es jedoch noch einen Fallstrick. Wird niemals die erforderliche Gesamtlänge durch eingehende Nachrichten erreicht, so werden auch keine Befehlstelegramme versendet und das Display zeigt nichts an. Zur Abhilfe erweitern wir die obige Lösung um einen zeitgesteuerten Anteil, der alle 500 ms automatisch eine Gesamtnachricht erzeugt. Dadurch erreicht man einerseits, dass die Reaktions- bzw. Verarbeitungszeit als gut empfunden wird, und andererseits, dass das Konkatenieren für den Aufrufer transparent bleibt und dieser sich nur um die Ansteuerung mit Befehlen kümmern muss – nicht aber um Details zu deren Verarbeitung. Dies ist ein Beispiel für Information Hiding und Kapselung, wie man es sich von Utility-Klassen und Frameworks wünscht. Würde man darauf nicht achten, so wäre es Aufgabe des Aufrufers, explizit Befehle zum Flushen (Senden) der Gesamtnachrichten abzusetzen. Die günstigsten Zeitpunkte sind durch Aufrufer aber in der Regel schwierig zu bestimmen. Daher realisiert eine Klasse `AutoFlusher` einen Automatismus, der periodisch die Methode `autoFlush()` aus der Klasse `MessageConcatenator` aufruft:

```
private boolean autoflush()
{
    return flushImpl();
}

// access for unit tests
/*private*/ void enableAutoFlush(final boolean autoFlushEnabled)
{
    autoFlusher.enableAutoFlush(autoFlushEnabled);
}
```

Diese Automatik lässt sich über ein Flag deaktivieren, wodurch eine Kompatibilität zu der vorherigen Umsetzung erzielt wird – allerdings muss man dann das Flushen manuell aufrufen. Außerdem wird das Unit-Testen erleichtert, weil zunächst Testfälle ohne Berücksichtigung des Einflusses von Multithreading erstellt werden können.

Vorbereitungen zum Test der Klasse `MessageConcatenator`

Um die Klasse `MessageConcatenator` testen zu können, müssen wir einige kleinere Vorarbeiten erledigen. Die Stub-Klasse `SimulationDisplay` wurde bereits in Abschnitt 20.4.2 vorgestellt. Weiterhin definieren wir einige Konstanten in der Klasse `SampleData`. Diese dienen als Dummy-Nutzinhalt für Nachrichten. Mit dieser definierten Ausgangsbasis an Eingabewerten fällt es leichter, Tests zu formulieren:

```
public final class SampleData
{
    public static final String TEXT_20  = "01234567890123456789";
    public static final String TEXT_40  = TEXT_20 + TEXT_20;
    public static final String TEXT_60  = TEXT_40 + TEXT_20;
    public static final String TEXT_80  = TEXT_40 + TEXT_40;
    public static final String TEXT_100 = TEXT_80 + TEXT_20;
}
```

Definition von Unit Tests für die Klasse `MessageConcatenator`

Zum Test der Funktionalität der Klasse `MessageConcatenator` definieren wir eine Testklasse `MessageConcatenatorTest`. Diese speichert eine Referenz auf die zu testende Klasse sowie das Ausgabemedium in Form des Interface `IDisplay`:

```
public class MessageConcatenatorTest
{
    private MessageConcatenator messageConcatenator;
    private IDisplay            testDisplay;

    @Before
    public void setUp()
    {
        testDisplay = new SimulationDisplay();
        messageConcatenator = new MessageConcatenator(testDisplay);
    }
    // …
```

Wir beginnen mit Tests ohne Aktivierung der 500-ms-Automatik, um die Funktionalität zunächst grundsätzlich zu überprüfen, d. h. die Effekte von Multithreading auszuschließen. Anschließend wird dann diese Komplexität hinzugeschaltet. Ein solches Vorgehen bietet sich grundsätzlich beim Testen an: Man testet anfangs die einfachen Bausteine und danach die komplexeren.

Versenden einer Nachricht: Explizites Flushing

Als ersten Testfall überprüfen wir die Basisfunktionalität des Versendens, das durch den Aufruf von `flush()` ausgelöst wird. Ein Versenden sollte dazu führen, dass ein zuvor gefüllter Puffer anschließend leer ist. Zunächst wird eine Textnachricht durch Aufruf der Methode `createTextMsg(String)` erzeugt. Diese Methode fügt den Nutzdaten einige Zusatzinformationen (Start- und Stoppbytes, Längenangaben und eine Checksumme zur Validierung der übertragenen Daten) hinzu, die 10 Bytes erfordern. Die Speicherung der Daten im Puffer wird anschließend mithilfe von

`getBufferedMsgLength()` ermittelt. Als Ergebnis wird die Länge der Textnachricht + 10 Bytes erwartet. Ein nachfolgender expliziter Aufruf von `flush()` sollte die Daten versenden und den Puffer leeren. Eine Abfrage der Länge muss dann den Wert 0 liefern. Das korrekte Arbeiten von `flush()` kann folgendermaßen als Test formuliert werden:

```
@Test
public void testExplicitFlush()
{
    // Deaktivieren des Auto-Flushings
    messageConcatenator.enableAutoFlush(false);

    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_40));
    assertEquals(40 + 10, messageConcatenator.getBufferedMsgLength());

    // Explizites Flushen, Erwartung: Rest 0
    messageConcatenator.flush();
    assertEquals(0, messageConcatenator.getBufferedMsgLength());
}
```

Hinweis: Mehrmalige Asserts

In den hier gezeigten Testfällen werden Aktionen und `assertEquals()`-Prüfungen miteinander vermischt und auch mehrmals hintereinander ausgeführt. Das scheint ein wenig dem ARRANGE-ACT-ASSERT-Stil zu widersprechen. Tatsächlich empfinde ich die mehrfachen Asserts für die dort benötigte Prüfung eines Ablaufs akzeptabel, weil lediglich ein semantischer Sachverhalt geprüft wird, eben teilweise eine Abfolge von Telegrammen und dem Inhalt des Puffers. Hier findet man zwar einige Zwischen-Asserts, jedoch wird das Wesentliche des Testfalls am Ende geprüft.

Unabhängig von Stilfragen sollte man auf Klarheit und leichte Nachvollziehbarkeit achten. Der Testfall muss kurz, verständlich und auf eine zu prüfende Funktionalität fokussiert bleiben.

Versenden mehrerer Nachrichten: Test des Verhaltens bei Überlauf

Ein weiterer Testfall stellt das Versenden mehrerer Nachrichten und das Zusammenfassen zu einer Gesamtnachricht dar. Dazu werden dem Puffer einige Nachrichten hinzugefügt. Wird die Kapazität eines Schlitzes des Eingabepuffers überschritten, soll die Nachricht automatisch versendet werden. Die zum Überlauf führende Nachricht muss dadurch als einzige im Puffer verbleiben. Diese Funktionalität prüfen wir, indem wir sukzessive Nachrichten hinzufügen und jeweils anschließend die Länge über die Methode `getBufferedMsgLength()` abfragen. Dies kann man wie folgt als Test umsetzen:

```
@Test
public void testOverflow()
{
    // Deaktivieren des Auto-Flushings
```

```
    messageConcatenator.enableAutoFlush(false);

    // Fülle Puffer bis kurz vor Längen-Überschreitung: 90 + 110 + 30 = 230
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_80));
    assertEquals(80 + 10, messageConcatenator.getBufferedMsgLength());
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_100));
    assertEquals(90 + 100 + 10, messageConcatenator.getBufferedMsgLength());
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_20));
    assertEquals(200 + 20 + 10, messageConcatenator.getBufferedMsgLength());

    // Erzwinge Flush durch Längen-Überschreitung: 230 + 70 = 300
    // Daher sollte die folgende Nachricht den Puffer leeren
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_60));
    // Hier sollte noch ein Rest von 70 Bytes existieren
    assertEquals(60 + 10, messageConcatenator.getBufferedMsgLength());
}
```

Auto-Flush

Der erste Testfall zum Multithreading prüft, ob die Auto-Flush-Automatik nach ca. 500 ms ausgeführt wird. Dazu erzeugen wir wieder eine Nachricht und prüfen deren Länge. Anschließend warten wir durch Aufruf einer Hilfsmethode `waitForOneSecond()` eine Sekunde. Nach dieser Zeit sollte der parallel laufende Auto-Flush-Thread die Nachricht versendet haben, was wir über die Bedingung `getBufferedMsgLength() == 0` prüfen:

```
@Test
public void testAutoFlush()
{
    // Aktivieren des Auto-Flushings
    messageConcatenator.enableAutoFlush(true);

    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_80));
    assertEquals(80 + 10, messageConcatenator.getBufferedMsgLength());

    // Prüfe Auto-Flush: Warte 1 Sekunde, dann Erwartung: Rest 0
    waitForOneSecond();
    assertEquals(0, messageConcatenator.getBufferedMsgLength());
}
```

Hinweis: Zeitliche Randbedingungen im Unit Tests

Grundsätzlich kann es problematisch sein, mit Annahmen über fixe Ausführungszeiten in Unit Tests bzw. Integrationstests zu arbeiten, da die Tests dann von äußeren Einflüssen abhängen. Daher ist im vorherigen Beispiel des Tests mit Timing schon bewusst ein recht großzügiger Zeitpuffer vorgesehen, um Effekte hoher Systemlast auf den Ausgang des Tests möglichst auszuschließen.

Auto-Flush mit Überlauf

Im komplexesten, hier vorgestellten Testfall wird das Zusammenspiel von Überlauf und periodischem Auto-Flush überprüft:

```
@Test
```

```
public void testAutoFlushAndOverflow()
{
    // Aktivieren des Auto-Flushings
    messageConcatenator.enableAutoFlush(true);

    // Erzwinge Flush durch Längen-Überschreitung
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_80));
    assertEquals(80 + 10, messageConcatenator.getBufferedMsgLength());
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_100));
    assertEquals(90 + 100 + 10, messageConcatenator.getBufferedMsgLength());

    // Die nächste Nachricht der Länge 90 passt nicht mehr: 90 + 110 + 90 =
290
    messageConcatenator.sendMessage(createTextMsg(SampleData.TEXT_80));
    assertEquals(80 + 10, messageConcatenator.getBufferedMsgLength());

    // Prüfe Auto-Flush: Warte 1 Sekunde, dann Erwartung: Rest 0
    waitForOneSecond();
    assertEquals(0, messageConcatenator.getBufferedMsgLength());
}
```

Fazit

Die hier vorgestellten Tests geben einen Einblick, wie man Abläufe bei Multithreading durch feingranulare Unit Tests in überprüfbare Einheiten zerlegen kann. Mit zunehmender Komplexität der Zusammenarbeit, also mit zunehmender Anzahl von Threads sowie deren Interaktionen untereinander, wird das Ausformulieren von Unit Tests anspruchsvoller bis nahezu unmöglich. Insbesondere können dann unterschiedliche Lastsituationen auf dem Rechner für ein unerwartetes Fehlschlagen eines Testfalls sorgen.

20.6 Test Smells

Gute und verständliche Tests zu schreiben ist gar nicht so einfach, wie es vielleicht zunächst scheinen mag. In diesem Unterkapitel schauen wir deshalb einige Fallstricke an, über die man bei ersten Gehversuchen mit Unit Tests stolpern kann (vgl. Kapitel 16 als Ergänzung für potenziell fehlerhafte Konstrukte im Applikationscode). Wenn Sie diese Fallstricke vermeiden, werden Ihre Tests viel wertvoller zur Beurteilung der Softwarequalität.

Ich möchte Ihnen einige Tipps und Tricks zum Unit-Testen mit auf den Weg geben und dabei mit einigen Aussagen von James Coplien starten, die seinem Artikel mit dem provokanten Titel »Why Most Unit Testing is Waste«[17] [17] entstammen:

- »Design a test with more care than you design the code.«
- »Tests don't improve quality: developers do.«
- »… is one of those rare people who know how to think instead of letting the computer do your thinking for him —- be it in system design or low-level design.«

Dr. Venkat Subramaniam, Autor verschiedener lesenswerter Bücher sowohl zu Groovy als auch zu Java, schreibt in seinem Buch »Programming Groovy 2: Dynamic Productivity for the Java Developer« [85] über Unit Tests, dass diese dem Motto FAIR folgen sollten, was so viel bedeutet wie FAST AUTOMATIC ISOLATED REPEATABLE. Diesem Motto würde ich gern noch die Anforderung Zuverlässigkeit hinzufügen. Kann man sich auf die Aussagen der Tests nicht verlassen, verwirren sie mehr, als dass sie helfen.

Test Smell: Falsche Nutzung von `assertTrue()` und `assertFalse()`

Zur Absicherung von erwarteten Bedingungen dient der Assert-Teil eines Testfalls. Betrachten wir scheinbar ganz einfache Prüfungen von Erwartungen:

```
assertTrue(db.writeCount == 10);
assertTrue(tasks.message.equals("status: 0"));
assertTrue(tasks.totalProcessed == 10);
```

Was ist an diesen Zustandsprüfungen problematisch? Eine ganze Menge!

Hier werden boolesche Bedingungen mithilfe der Methoden `assertTrue()` geprüft. Allerdings erhält man damit lediglich eine Aussage, ob die Bedingung erfüllt ist oder nicht, jedoch keine Aussage darüber, was der erwartete und der beobachtete Wert ist. Man weiß somit nicht, welche der Prüfungen fehlschlug. Das wird noch dadurch verschärft, dass recht viele Prüfungen erfolgen.

Verwendet man statt `assertTrue()` oder `assertFalse()` eine Prüfung mit `assertEquals()`, so erhält man beim Fehlschlagen zumindest Hinweise auf Abweichungen vom erwarteten Wert. Wir korrigieren die Prüfungen:

```
assertEquals(10, db.writeCount);
assertEquals("status: 0", tasks.message);
assertEquals(10, tasks.totalProcessed);
```

Nach diesen Umformungen erhält man bei einem Fehler zumindest die Ausgabe, welcher Wert erwartet und welcher geliefert wurde, in etwa so:

```
java.lang.AssertionError: expected:<10> but was:<12>
```

Das ist zwar eine Verbesserung. Weil hier aber noch die zwei Probleme »Zu viele Asserts im Testfall« und »Asserts ohne Hinweis« vorliegen, ist die Meldung noch nicht ausreichend, weil wir die Aussage kaum einer geprüften Bedingung zuordnen können. Die Abhilfe lernen wir nun bei der Besprechung der nächsten Test Smells kennen.

Test Smell: Zu viele Asserts im Testfall

Mit einem Testfall soll ein möglichst eng umrissenes Verhalten überprüft werden. Im besten Fall lässt sich dies mit nur einem Assert prüfen. Sofern ab und zu ein paar mehr Asserts zur Zustandsprüfung benötigt werden, ist dies auch nicht weiter

tragisch, sofern diese semantisch zusammengehörend (kohärent) sind, also nicht mehrere Aspekte prüfen.

In folgendem Beispiel findet man fünf Asserts, die zumindest zwei Sachverhalte prüfen, hier durch die Objekte `db` und `tasks` repräsentiert.

```
assertEquals(10, db.readCount);
assertEquals(10, db.writeCount);
assertEquals(10, db.commitCount);

assertEquals("status: 0", tasks.message);
assertEquals(10, tasks.totalProcessed);
```

Wenn ein Testfall mehrere Asserts enthält, kann es mehrere Gründe geben kann, warum der Test nicht bestanden wird. Damit fällt es schwer, einen Verstoß direkt einem Assert zuzuordnen und somit auch die Ursache zu ermitteln.

Für das Beispiel bietet es sich an, die Prüfungen in zwei Testfälle aufzuteilen, einen, der sich um das `db`-Objekt kümmert, und einen anderen, der das `task`-Objekt prüft.

In der gezeigten Kombination mit mehreren Prüfungen ist es ohne Angabe eines Hinweistextes recht schwer, die Fehlerursache einzugrenzen, zumindest wenn sehr ähnliche Prüfungen und erwartete Werte genutzt werden. Schauen wir uns nun also die Möglichkeit an, Hinweistexte bereitzustellen.

Test Smell: Asserts ohne Hinweis

Wenn es in einem Testfall mehr als ein Assert gibt, so ist es generell hilfreich, für die einzelnen Asserts einen Hinweistext bereitzustellen, das gilt insbesondere für `assertTrue()` bzw. `assertFalse()`.

Schauen wir auf folgendes Assert mit dem Vergleich zweier Laufzeiten:

```
assertTrue(runtime2 > runtime1);
```

Dieser Testfall schlug des Öfteren fehl. Die Gründe dafür waren durch diese Formulierung kaum zu erkennen. Ich habe mir das Ganze angeschaut und folgende Korrektur vorgenommen:

```
assertTrue("runtime2 " + runtime2 + " ms should be > " + runtime1 + " ms",
           runtime2 > runtime1);
```

Damit ließ sich das Problem durch die erzeugte Fehlermeldung

```
java.lang.AssertionError: runtime2 0 ms should be > 0 ms
```

eingrenzen und konnte so überhaupt erst adressiert werden. Neben Asserts ohne Hinweis ist es eben keine gute Idee, fixe Laufzeitannahmen in Unit Tests zu machen. Das ist eigentlich auch ein eigener Test Smell, der hier jedoch nicht dargestellt wird, sondern im Rahmen der Darstellung von »Test Smell: Unit-Tests, um variierende Laufzeiten zu prüfen« kurz behandelt wird.

Test Smell: Einsatz von `toString()` in `assertEquals()`

Eine Problematik beim Testen besteht in der sicheren Wiederholbarkeit. Diese ist dann nicht gegeben, wenn nicht immer eindeutig reproduzierbare Werte oder eine Vielzahl von einzelnen Werten in Kombination miteinander verglichen werden. So etwas sieht man beispielsweise im Zusammenspiel mit der Methode `toString()`. Lasse Koskela spricht in seinem Buch »Effective Unit Testing: A Guide for Java Developers« [58] dann von ***Hyperassertions***.

Nachfolgend sehen wir eine Prüfung einer Datenbankkonfiguration. Dieser Test basiert auf einer Stringrepräsentation, hier stark gekürzt:

```
assertEquals("mongodb.writeConcern.timeout=10000," +
             "mongodb.writeConcern.writes=1, " +
             "mongodb.port=27017, " +
             "mongodb.password=ksdjsa2455aAYdsj," +
             "mongodb.user=ABCD", db.toString())
```

Wie schon in Kapitel 16 über Bad Smells erwähnt, sind Vergleiche, basierend auf den von `toString()` produzierten Ausgaben, fragil. Außerdem sollte es jederzeit möglich sein, eine so generierte Stringrepräsentation informativer oder lesbarer zu gestalten, etwa könnte man Stringwerte in Hochkommata einschließen o. Ä. Darüber hinaus würde jede kleinste interne Änderung an der Datenbankkonfiguration (z. B. der Portnummer oder Timeouts) zu einem Fehlschlag des Testfalls führen. Derartige Details sollten sicher keinen Einfluss auf den Ausgang von Unit Tests haben. Kurz: Jede minimale Rekonfiguration bricht den Test. Das ist schlecht.

Würde man statt des gesamten Inhalts nur einzelne relevante Bestandteile des Strings mithilfe von `contains()` abfragen, so würde die Prüfung weniger fragil sein und damit das Ganze etwas besser machen – dieses Vorgehen entspricht dann allerdings eher einem Trostpflaster. Deutlich sinnvoller ist es, lediglich die wichtigen Informationen mithilfe von Zugriffsmethoden zu ermitteln und zu überprüfen.

Als Grundregel gilt: ***Prüfe in Unit Tests lediglich diejenigen Werte, die wirklich relevant sind und benötigt werden, um das Bestehen zu garantieren.*** Je mehr Zustand drumherum geprüft wird, der nicht (oder kaum) mit dem eigentlichen Test verbunden ist, desto größer ist die Fragilität des Tests. Damit beginnt ein Wartungsalbtraum für Tests: Diese müssen dann ständig angepasst werden, weil sie sonst bei vielen eigentlich unbedeutenden Programmänderungen fehlschlagen.

Test Smell: Unit-Tests, um variierende Laufzeiten zu prüfen

Manchmal sollen die Laufzeiten von Programmteilen gegen ein erwartetes Laufzeitverhalten geprüft werden. Leider erfolgt dies zum Teil durch den Einsatz von Unit Tests, obwohl dies eigentlich kein probates Mittel darstellt, insbesondere dann nicht, wenn man keine Varianzen in den Laufzeiten berücksichtigt. Schauen wir uns ein Beispiel eines problematischen Sourcecodes an:

```
final int count = 1000;
```

```
// Laufzeitmessung für Algorithmus 1
long runtime1 = 0

for (int i = 0; i < count; i++)
{
     final long start = System.currentTimeMillis();
     performCalculation_V1();
     final long end = System.currentTimeMillis() - start;
     runtime1 += end;
}

// Verbesserte Laufzeitmessung für Algorithmus 2
final long start2 = System.nanoTime(); // Verbesserung 1: nanoTme()
for(int i = 0; i < count; i++)
{
     performCalculation_V2();
}
final long end2 = System.nanoTime();      // Verbesserung 2: Gesamtlaufzeit

// Verbesserung 3: Berechnung von Messung getrennt
final long runtime2 = TimeUnit.NANOSECONDS.toMillis(end2 - start2);

// Fragile Annahme über die Laufzeiten
assertTrue("calc V2 runtime should be > calc V1 runtime", runtime2 >
runtime1);
```

In diesem (bis auf Kleinigkeiten) so in der Praxis gefundenen Sourcecode wurden gleich mehrere Fehler gemacht. So etwas ist nicht ungewöhnlich, denn oftmals findet man eine Fehlerhäufung gerade in den aus anderen Gründen schon besonders fragilen Teilen (in diesem Fall im Sourcecode des Tests). Das Ganze bestärkt die Aussage von James Coplien, dass einige Tests, zumindest in der Art, wie sie erstellt wurden, überflüssig sind bzw. Tests mit ebensolcher Sorgfalt wie der eigentliche Sourcecode erstellt werden müssen. Noch unerfahrene Entwickler sollten sich einen erfahrenen Kollegen suchen, der mit ihnen ein Codereview durchführt. Machen wir dieses Codereview einmal virtuell: Bei der ersten Variante der Laufzeitberechnung ist schlecht, dass die Zeitmessung mithilfe von `currentTimeMillis()` erfolgt. Die zurückgelieferten Werte sind recht ungenau. Zudem ist es ungeschickt, die einzelnen Laufzeiten aufzuaddieren, statt eine Gesamtlaufzeit zu berechnen. So verstärken sich Fehler bzw. die Abweichungen. Dadurch wird meistens eine deutlich geringe oder größere Laufzeit als die tatsächliche gemessen, sodass die Gesamtlaufzeit kürzer oder länger scheint, als sie ist. Auch der Variablenname `end` ist irreführend, weil hier die Laufzeit einer Einzelberechnung ermittelt wird und nicht der Endzeitpunkt. Wie man es besser machen kann, habe ich im Listing für den zweiten Messlauf gezeigt.

Die fragile Prüfung der Laufzeit habe ich um einen Hinweistext ergänzt und ansonsten original übernommen. Diese ist hochgradig von äußeren Einflüssen abhängig und somit sicher nicht zuverlässig wiederholbar. Durch die jeweils 1000 Durchläufe – wohl zur Steigerung der Messgenauigkeit eingesetzt – verstößt dieser Test höchstwahrscheinlich auch gegen die Forderung, schnell ausführbar zu sein. Insgesamt sollte man vorsichtig sein, wenn man in Unit Tests mit Annahmen zum Timing arbeitet, da sich Effekte von der momentanen Systemlast möglicherweise

stark auf den Testablauf auswirken können und zu fragilen Tests führen: Von Lauf zu Lauf oder bei variierender Systemlast bekommt man unterschiedliche Testresultate. Das ist zu vermeiden, weil die Tests stark an Aussagekraft verlieren.

20.7 JUnit Rules und parametrierte Tests

Im vorangegangenen Unterkapitel haben wir mögliche Probleme beim Formulieren von Unit Tests kennengelernt. Auch sind wir beim Erstellen der Tests für die bisherigen Beispiele über die eine oder andere Schwierigkeit gestolpert.

Als Abhilfe lernen wir im Anschluss sowohl JUnit Rules als auch sogenannte Parametrized Tests kennen. JUnit Rules bieten verschiedene Hilfestellungen beim Aufstellen von Testbehauptungen. Parametrierte Tests erleichtern das Testen von mehreren Parametersätzen, wie wir es zuvor schon selbst programmiert haben.

20.7.1 JUnit Rules im Überblick

Die nachfolgenden Abschnitte beschreiben mit JUnit Rules einige Erweiterungen zu JUnit, die in Version 4.7 hinzugefügt wurden.

Die Rule `TestName`

Ich beginne die Darstellung der JUnit Rules mit der unspektakulärsten und einfachsten, nämlich der JUnit Rule `TestName`, die wenig überraschend den Namen des momentan ausgeführten Testfalls, also der ausgeführten Methode, zurückliefert. In der Praxis nutze ich diese Funktionalität nicht, hier dient sie nur zum Einstieg. Dass die Rule wie gewünscht arbeitet, davon kann man sich mithilfe des folgenden Unit Test überzeugen:

```
import  org.junit.rules.TestName;

public class  NameRuleTest
{
    @Rule
    public  TestName name =  new  TestName();

    @Test
    public void  testMethod()
    {
        assertEquals( "testMethod", name.getMethodName());
    }
}
```

Die Rule `Timeout`

Per Definition sollten unsere Testfälle schnell abgearbeitet werden, um die

Unterbrechung zwischen Codieren und Erhalt des Testresultats möglichst kurz zu gestalten.

Zum Teil möchte man die Ausführungsdauer von Testfällen begrenzen. Man kann zwar jedem Testfall einzeln in der Annotation `@Test` über den Parameter `timeout` explizit einen Wert für die maximale Ausführungsdauer setzen, jedoch möchte man dies oftmals globaler machen. Auch dafür gibt es eine JUnit Rule, nämlich `Timeout`. Dadurch werden alle Testfälle spätestens beim Überschreiten der angegebenen Timeout-Zeit abgebrochen und als Fehlschlag gewertet. Das kann man wie folgt einsetzen:

```
import org.junit.rules.Timeout;

public class TimoutRuleTest
{
    @Rule
    public Timeout timeout = new Timeout(500);

    @Test
    public void longRunningAction() throws InterruptedException
    {
        for (int i=0; i < 20; i++)
        {
            TimeUnit.SECONDS.sleep(1);
        }
    }

    @Test
    public void loopForever() throws InterruptedException
    {
        for (;;)
        {
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

Die beiden Tests in Form der Methoden `longRunningAction()` und `loopForever()` würden normalerweise 20 Sekunden bzw. unendlich laufen. Sie werden nach der vorgegebenen Timeout-Zeit von 500 Millisekunden mit folgender Meldung abgebrochen:

```
java.lang.Exception: test timed out after 500 milliseconds
```

Etwas schade ist, dass man den Parameter für den Timeout immer nur in Millisekunden angeben kann. Schöner wäre sicher der Einsatz der Klasse `TimeUnit` gewesen. Ebenso zu kritisieren ist der Typ der geworfenen Exception, nämlich `Exception`.

Tatsächlich wurden beide Negativpunkte mit der neuesten Version 4.12 (im November 2014 noch im Beta-Stadium) von JUnit adressiert. Man kann nun `TimeUnit` zur Angabe des Timeouts verwenden und erhält bei einem Verstoß eine `TestFailedOnTimeoutException`.

Die Rule `ExpectedException`

Manchmal sollen Testfälle das Auftreten von Exceptions während der Abarbeitung prüfen und ein Ausbleiben würde einen Fehler darstellen. Ein ganz simples Beispiel ist ein bewusster Zugriff auf ein nicht existentes Element eines Arrays. Eine `ArrayIndexOutOfBoundsException` sollte die Folge sein. Um erwartete Exceptions im Testfall so zu behandeln, dass diese einen Testerfolg und keinen Fehlschlag darstellen, besitzt man verschiedene Alternativen.

Vor JUnit 4 gab es nur folgendes Konstrukt, das einen `try-catch`-Block und einen Aufruf von `fail()` wie folgt einsetzt:

```
try
{
   actionsThrowingAnException();
   fail();                  // Sollte hier nicht hinkommen
}
catch (final ExpectedException e)
{
   assertTrue(true);        // Erwarteter Fall
}
```

Mit JUnit 4 wurde es dann möglich, eine bei der Testausführung erwartete Exception in der Annotation `@Test` als Parameter `expected` anzugeben:

```
@Test(expected = ExpectedException.class)
```

Beide Varianten haben ihre Stärken und Schwächen. Insbesondere sind sie entweder leicht unleserlich (`try-catch`) oder leicht zu übersehen und zudem auf einen Typ von Exception beschränkt (`@Test`). Details beschreibt der folgende Praxistipp »Beschränkungen der annotationsbasierten Exception-Variante«.

Wie geht es also besser? Ab JUnit 4.7 gibt es die JUnit Rule `ExpectedException`, die sich mit der Verarbeitung von Exceptions in Unit Tests beschäftigt:

```
import org.junit.rules.ExpectedException;

public class ExpectedExceptionTest
{
    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void noExceptionExpected()
    {
        // …
    }

    @Test
    public void illgealStateExceptionWithMessageExpected()
    {
        thrown.expect(IllegalStateException.class);
        thrown.expectMessage("XYZ is not initialized");

        throw new IllegalStateException("XYZ is not initialized");
    }
```

```
}
```

Im Listing sehen wir, dass standardmäßig keine Exception erwartet wird: `ExpectedException.none()`. Das scheint für die meisten Testfälle eine sinnvolle Annahme und Vorbelegung. Erwartet ein Testfall eine Exception, so ist dies explizit zu spezifizieren. Während die dafür genutzten und zuvor gezeigten Ansätze ihre Schwächen besaßen, ist die Realisierung mit der JUnit Rule `ExpectedException` relativ gut lesbar und nachvollziehbar.

Tipp: Beschränkungen der annotationsbasierten Exception-Variante

Nutzt man die Annotation `@Test(expected = ExpectedException.class)`, birgt dies neben den bereits genannten Nachteilen noch weitere Schwachpunkte: Es wird in der gesamten Methode das Auftreten der angegebenen Exception als Erfolg gewertet, nicht nur in einem kleinen genau abgegrenzten Bereich. Das kann problematisch sein, weil es zu falschen Resultaten (sogenannten ***False Positives***) führen kann. Nehmen wir an, die zu prüfende Programmstelle soll auf eine `IllegalArgumentException` prüfen. Wird bereits im Arrange-Teil ein solcher Typ von Exception ausgelöst, so wird der Testfall beendet und positiv gewertet, obwohl die eigentlich Logik und Prüfung niemals durchlaufen wurde.

Nebenbei existiert die Schwierigkeit, auf die Nachrichtentexte in der ausgelösten Exception zuzugreifen. Eine Motivation, warum das von Interesse sein kann, haben wir im einleitenden Beispiel zu Refactorings in Abschnitt 17.1 kennengelernt. Kurz rekapituliert: Die Texte einer ausgelösten Exception erlauben Rückschlüsse über die Güte der enthaltenen Fehlernachricht. Fehlt diese Nachricht, so war der Programmierer entweder (zu) faul oder es handelt sich um eine vom System generierte Exception, etwa eine `NullPointerException`. Als Anwendungsentwickler sollte man immer die Chance nutzen, eine ausgelöste Exception mit möglichst hilfreichen Informationen zur Fehlerursache und zum Kontext zu versehen, etwa »Can not open properties file '<path>/XYZ.properties'. file does not exist.«

Die Rule `TemporaryFolder`

Mitunter benötigt man in Unit Tests auch Zugriffe auf Dateien. Praktischerweise erlaubt JUnit die Ausführung von Unit Tests unter Bereitstellung eines jeweils eigenen temporären Verzeichnisses für einen Testfall, wobei abschließend automatisch Aufräumarbeiten erfolgen. Dadurch bleibt auch die Forderung nach Unabhängigkeit der Testfälle untereinander bestehen. Führt man lediglich kleine Aktionen aus, so erfüllt man sicher auch das Kriterium FAST. Zudem lässt sich der Test oftmals klarer formulieren, als wenn man versucht, alle Abhängigkeiten mit Mocks zu beschreiben.

Legen wir nun ein Unterverzeichnis sowie eine Datei in einem temporären Verzeichnis an und prüfen dessen Inhalt sowie die Existenz der Datei:

```
import org.junit.rules.TemporaryFolder;

public class TemporaryFolderTest
{
   @Rule
   public TemporaryFolder folder = new TemporaryFolder();

   @Test
   public void testUsingTempFolder() throws IOException
   {
      final File subFolder = folder.newFolder("subfolder");
      final File createdFile = folder.newFile("abc.txt");

      // Ermittle den Inhalt des Temp-Folders, normales File-API
      final String[] dirContents = folder.getRoot().list();

      // Als Menge, damit die Reihenfolge keinen Einfluss hat
      final List<String> expectedNames = Arrays.asList("subfolder",
"abc.txt");
      final Set<String> expectedFiles = new TreeSet<>(expectedNames));
      assertEquals(expectedFiles, new TreeSet<>(Arrays.asList(dirContents)));

      // Prüfe, ob File existiert
      assertTrue(createdFile.exists());
   }

   @Test
   public void testUsingTempFolder_Again() throws IOException
   {
      final String[] currentContents = folder.getRoot().list();
      assertEquals(0, currentContents.length);
   }
}
```

Beim Programmieren dieses Unit Test merkt man, dass es etwas unhandlich ist, die Erwartung in Form eines `String[]` mit den tatsächlich durch `list()` gelieferten Dateibzw. Verzeichnisnamen abzugleichen. Damit die Reihenfolge keine Rolle spielt, wird hier zunächst in eine Liste und danach in ein `TreeSet<String>` gewandelt. Das bläht den Sourcecode auf und macht die Intention des Tests weniger klar. Greifen wir etwas vor und vereinfachen die Prüfungen mit dem Tool Hamcrest wie folgt:

```
import static org.hamcrest.collection.IsArrayContainingInAnyOrder.
      arrayContainingInAnyOrder;
import static org.hamcrest.collection.IsIterableContainingInAnyOrder.
      containsInAnyOrder;

// …

final String[] currentContents = folder.getRoot().list();

// Hamcrest-Variante 1 mit Arrays
final String[] expectedAsArray = { "subfolder", "abc.txt" };
assertThat(expectedAsArray, arrayContainingInAnyOrder(currentContents));

// Hamcrest-Variante 2 mit Iterable
```

```
final List<String> expectedAsList = Arrays.asList("subfolder", "abc.txt");
assertThat(expectedAsList, containsInAnyOrder(currentContents));
```

Die JUnit Rule `TemporaryFolder` kann hilfreich sein, wenn kleinere Interaktionen mit dem Dateisystem zu testen sind. Insbesondere befreit sie von den normalerweise benötigten manuellen Aufräumarbeiten, nämlich dem Löschen von temporär erzeugten Dateien und Verzeichnissen. Einige »Hardliner« würden dann nicht mehr Unit Test dazu sagen. Tatsächlich sind Dateiaktionen in Unit Tests eher zu vermeiden, es gibt jedoch Ausnahmen.

Kombinierbarkeit von JUnit Rules

Praktischerweise kann man JUnit Rules auch miteinander kombinieren, wobei diese sich (eigentlich) nicht auf eine Abarbeitungsreihenfolge verlassen dürfen. Im nachfolgenden Beispiel ist die Reihenfolge jedoch durch die Implementierung vorgegeben:

```
public class TimeoutAndExpectedExceptionTest
{
    @Rule
    public Timeout timeout = new Timeout(500);

    @Rule
    public ExpectedException thrown = ExpectedException.none();

    @Test
    public void longRunningActionThrowTimeoutException() throws
          InterruptedException
    {
        thrown.expect(Exception.class);
        thrown.expectMessage("test timed out after 500 milliseconds");


        for (int i=0; i < 20; i++)
        {
            TimeUnit.SECONDS.sleep(1);
        }
    }
}
```

Tipp: Fragilität der Kombinierbarkeit

Bitte beachten Sie, dass es bei der Kombination von JUnit Rules leicht einmal zu Problemen kommt, wenn diese nicht wirklich voneinander unabhängige Aktionen ausführen. Wenn wir im obigen Listing die Definition der beiden JUnit Rules wie folgt vertauschen, wird eine Exception ausgelöst und nicht abgefangen.

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Rule
public Timeout timeout = new Timeout(500);
```

Das ist natürlich recht fragil.

20.7.2 Parametrierte Tests

Teilweise muss man ein Vielzahl an Wertebelegungen testen. Für jede davon eine eigene Testmethode zu erstellen, bläht die Testklasse nur unnötig auf. Im Beispiel der Auswertung von gültigen hexadezimalen Zahlen haben wir gesehen, wie man mehrere Werte z. B. mithilfe eines Arrays erwarteter Werte in einer Schleife prüfen kann:

```
@Test
public void testHexCodedByteValueToInt_With_AllValidInputs()
{
    final byte[] inputs = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                            'A', 'B', 'C', 'D', 'E', 'F' };
    final int[] expected = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                             10, 11, 12, 13, 14, 15 };

    for (int i = 0; i < inputs.length && i < expected.length; i++)
    {
        assertEquals(expected[i],
Controller.hexCodedByteValueToInt(inputs[i]));
    }
}
```

Diese Variante besitzt jedoch den Nachteil, dass die Testmethode beim ersten Fehler abgebrochen wird. Enthält die Funktionalität mehrere Fehler, so muss man sich sukzessive durch immer weitere Fehler kämpfen, bis der Testfall (endlich) bestanden wird.

JUnit Parametrized Tests

Seit JUnit 4 kann man einen sogenannten parametrierten Test nutzen, der es erlaubt, einen Testfall mit unterschiedlichen Wertebelegungen immer wieder auszuführen. Dazu muss man Folgendes machen:

1. Die Testklassen mit `@RunWith(Parameterized.class)` annotieren.
2. Einen Konstruktor erstellen, der die Testdaten entgegennimmt. Als Eingabewerte benötigt dieser Konstruktor jeweils Eingabewert und erwartetes Ergebnis.
3. Attribute für jeden der Übergabeparameter des Konstruktors bereitstellen.
4. Eine spezielle Datenlieferanten-Methode erstellen. Diese muss als `public` und `static` definiert sein und als Rückgabe ein `Iterable<Object[]>` besitzen.
5. Eine Testmethode, die dann mit den Werten »gefüttert« wird, erstellen.

Anhand dieser Liste sieht man schon, dass der Einsatz von parametrierten Tests zulasten der Einfachheit und teilweise der Verständlichkeit geht, wenn man den Test wie beschrieben implementiert:

```
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
```

```
@RunWith( Parameterized. class)
public class ParametrizedTestExample
{
    private byte input;
    private int expected;

    // Jeder Methodenparameter und auch der erwartete Wert werden als
    // Parameter an den Konstruktor übergeben
    public ParametrizedTestExample( final char inputNumber,
                                    final int expectedResult)
    {
        this. input = ( byte) inputNumber;
        this. expected = expectedResult;
    }

    @Parameterized. Parameters
    public static Iterable<Object[]> hexInputs()
    {
       return Arrays. asList( new Object[][]
       {
         { '0', 0 }, { '1', 1 }, { '2', 2 }, { '3', 3 }, { '4', 4},
         { '5', 5 }, { '6', 6 }, { '7', 7 }, { '8', 8 }, { '9', 9},
         { 'A', 10}, { 'B', 11}, { 'C', 12}, { 'D', 13}, { 'E', 14}, { 'F', 15}
       });
    }

    @Test
    public void testHexCodedByteValueToInt_With_AllValidInputs()
    {
        assertEquals( expected, Controller. hexCodedByteValueToInt( input));
    }
}
```

Bei diesem Vorgehen ist problematisch, dass die Testfälle von JUnit während der Ausführung einfach durchnummeriert werden. Somit ist bei einem Fehlschlagen kein Rückschluss auf die tatsächliche Wertebelegung möglich. Weil man diese Schwachstelle erkannt hat, kann man seit JUnit 4.11 Informationen zu den einzelnen Parametrierungen ausgeben, wenn man die Annotation etwas erweitert, um fehlschlagende Tests leichter identifizieren zu können:

```
@Parameterized. Parameters( name= "{ index}: hexInput ({0}) => value {1}")
```

Dabei bedeuten die Platzhalter Folgendes: `{ index}` entspricht dem Index in den Testdaten, und `{ 0}`, `{ 1}`, `{ 2}` usw. referenzieren die entsprechenden Datenelemente.

Zwischenfazit

Zwar sind parametrierte Tests in JUnit grundsätzlich eine gute Idee, jedoch sind sie eher unbefriedigend umgesetzt, weil dadurch die Komplexität der Testklasse erhöht wird und sich Tests nicht sonderlich elegant formulieren lassen. Schlimmer noch: Man kann in einer solchen Testklasse keine normalen Testmethoden ergänzen, die nicht mit der Parametrierung ausgeführt werden sollen. Darüber hinaus ist es nicht möglich, für verschiedene Methoden auch unterschiedliche Parametrierungen zu verwenden.

Glücklicherweise gibt es die Erweiterung JUnit Params,[18] mit der diese Schwachstellen adressiert und zum Teil behoben werden. Ich möchte hier aber eine einfache und elegante Variante zeigen, die rein auf JUnit basiert.

Intelligente parametrierte Test mit JUnit Rule `ErrorCollector`

Zur Realisierung parametrierter Test zeige ich eine einfache Variante, die die bisher noch nicht vorgestellte, aber meines Erachtens äußerst nützliche JUnit Rule `ErrorCollector` verwendet.

```
import org.junit.rules.ErrorCollector;

public class ParametrizedTest_WithErrorCollector
{
    @Rule
    public ErrorCollector errors = new ErrorCollector();

    @Test
    public void testHexCodedByteValueToInt_With_AllValidInputs()
    {
        final byte[] inputs = { '0', '1', '2', '3', '4', '5', '6', '7', '8',
'9',
                                'A', 'B', 'C', 'D', 'E', 'F' };
        final int[] expected = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
                                 10, 11, 122, 13, 144, 15 };

        for (int i = 0; i < inputs.length && i < expected.length; i++)
        {
           try
           {
              assertEquals(expected[i],
                           Controller.hexCodedByteValueToInt(inputs[i]));
           }
           catch (final Throwable e)
           {
              errors.addError(e);
           }
        }
    }
}
```

Im Listing habe ich bewusst zwei Fehler in die Ergebniswerte integriert, um aufzeigen zu können, wie klar sich Testfälle gestalten lassen und wie hilfreich die obige Lösung ist, um Fehler erkennbar zu machen.

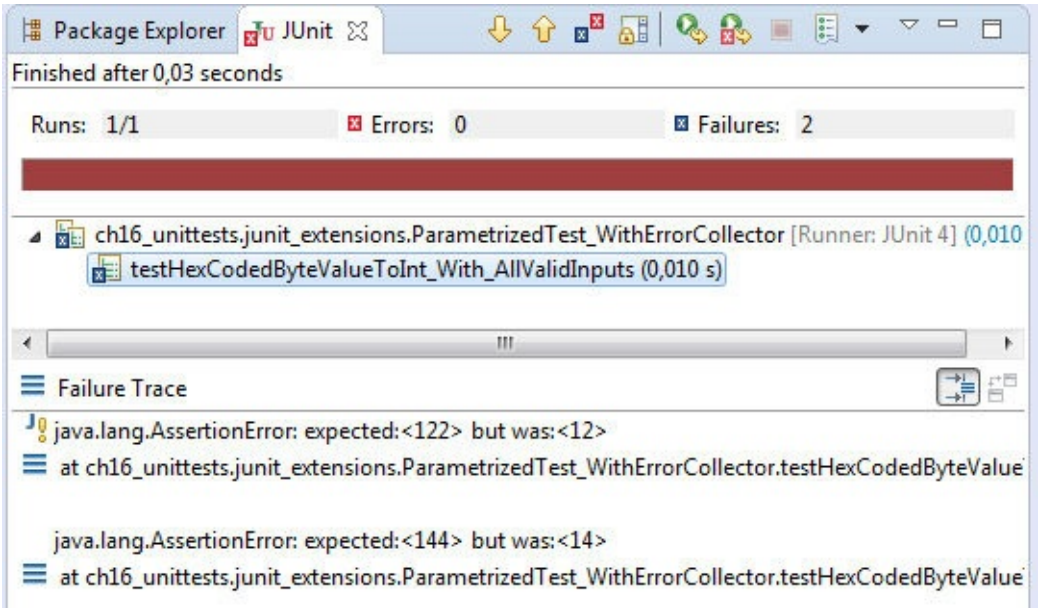


Abbildung 20-19 *»Intelligenter« parametrierter Test mit* `ErrorCollector`

20.8 Nützliche Tools für Unit Tests

In diesem Abschnitt werden mit Hamcrest, MoreUnit, Infinitest und Cobertura sowie EclEmma nützliche Tools zum Unit-Testen vorgestellt. Hamcrest hilft beim Formulieren von Testbedingungen. MoreUnit erleichtert das Erstellen und Ausführen von Unit Tests. Infinitest sorgt für die Durchführung von Unit Tests als Regressionstests. Das Tool Cobertura ermittelt die sogenannte Testabdeckung (Prozentanteil der durch Tests geprüften Programmteile) und zeigt vor allem auch die Programmteile, die nicht (oder zumindest nicht ausreichend) von Unit Tests geprüft werden. Cobertura arbeitet derzeit lediglich bis JDK 7. Zum Prüfen von aktuellem, mit JDK 8 erstelltem Sourcecode bietet sich der Einsatz des Tools EclEmma[19] an, das abschließend in diesem Unterkapitel beschrieben wird.

20.8.1 Hamcrest

Hamcrest ist ein Tool, dessen Einsatz die Lesbarkeit und die Verständlichkeit von Unit Tests erhöhen kann.[20] Hamcrest ist im Grunde genommen ein Framework, das es erlaubt, Vergleichsoperationen, sogenannte ***Matcher***, zu schreiben. Diese lassen sich beliebig kombinieren. Außerdem kann man einige Bedingungen nahezu umgangssprachlich deklarativ formulieren. Für Unit Tests ist dies nützlich, um die Lesbarkeit zu erhöhen. Hamcrest fügt sich dabei nahtlos in verschiedene Test-Frameworks ein, etwa JUnit oder TestNG. Die ohne Hamcrest erstellten Tests bleiben weiterhin gültig, wodurch ein schrittweiser Wechsel erleichtert wird und man die neue Schreibweise sukzessive einführen kann.

Um alle nachfolgend beschriebenen Funktionalitäten von Hamcrest nutzen zu können, müssen wir folgende Abhängigkeit in unsere Gradle-Build-Datei

`build.gradle` hinzufügen:

```
compile 'org.hamcrest:hamcrest-all:1.3'
```

Hamcrest-Integration in JUnit

Weil der Einsatz von Hamcrest die Lesbarkeit deutlich erhöht, wurde es als erste und einzige externe Bibliothek in JUnit 4 integriert – allerdings nur einige Teile davon. Das sind insbesondere die Methode `assertThat()` zum Aufstellen einer Testbehauptung sowie verschiedene Methoden zum Formulieren einzelner Bedingungen.

Schauen wir, wie sich etwa folgende mit JUnit formulierte Testbehauptung vereinfachen lässt:

```
assertEquals(1234, calculatedPrice());
```

Ganz offensichtlich ist dies recht technisch. Nutzen wir dagegen Hamcrest ist das Ganze besser lesbar:

```
assertThat(calculatedPrice(), is(1234));
```

Wie schon beschrieben, bietet sich oftmals die Angabe eines Hinweistextes zur besseren Identifikation der Fehlersituation an, insbesondere wenn entweder mehrere Asserts oder die Varianten `assertTrue()` bzw. `assertFalse()` genutzt werden, da letztere keine verständliche Meldung erzeugen. Zwar wird die produzierte Fehlermeldung durch den Hinweistext informativer, dafür reduziert sich die Lesbarkeit des Sourcecodes aber ein wenig.

```
assertEquals("price", 1234, calculatedPrice());
assertThat("price", calculatedPrice(), is(1234));
```

Hamcrest in Aktion: Der erste Test

Um Hamcrest kennenzulernen, schreiben wir einen kleinen JUnit-Test. Statt der bekannten `assertEquals()`-Methode zum Aufstellen einer Testbehauptung nutzen wir hierbei die Methode `assertThat()`. Zum Formulieren von Bedingungen wird ein sogenannter Matcher benötigt.

In folgendem Beispiel soll mit `assertThat()` geprüft werden, ob der Parameter `mike` mit dem Wert von `otherPerson` übereinstimmt. Dazu wird der Matcher `equalTo()` eingesetzt. Aus Gründen der besseren Lesbarkeit werden die Hamcrest-Matcher und Hilfsmethoden statisch importiert:

```
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.core.IsEqual.equalTo;

import org.junit.Test;

public class SimplePersonTest
{
    @Test
    public void testEquals()
```

```
    {
        final Person mike = new Person("Mike", 38, "Aachen");
        final Person otherPerson = new Person("Mike", 38, "Aachen");

        assertThat(mike, equalTo(otherPerson));
    }
}
```

Die Testbehauptung besteht hier aus zwei Teilen, die durch zwei Parameter repräsentiert werden. Der erste Parameter gibt das zu überprüfende `Person`-Objekt `mike` an. Durch den zweiten Parameter wird der Vergleich mithilfe eines Matchers oder einer Kombination von Matchern definiert. Dadurch wird der Vergleich festgelegt: Mit `equalTo(T)` erfolgt der Vergleich über die `equals(Object)`-Methode des im ersten Parameter angegebenen Objekts. Der Test wird erfolgreich durchlaufen, sofern die Klasse `Person` die `equals(Object)`-Methode korrekt implementiert.

Lesbarkeit weiter erhöhen

Ein `is`-Matcher trägt zur Lesbarkeit bei, besitzt aber keine eigene Funktionalität, sondern ruft nur den `equalTo`-Matcher auf. Folgende Zusicherungen drücken alle das Gleiche aus, wodurch man je nach persönlichem Geschmack die für einen selbst am besten lesbare Variante wählen kann.

```
assertThat(mike, equalTo(otherPerson));
assertThat(mike, is(equalTo(otherPerson)));
assertThat(mike, is(myPerson));
```

Prüfen mehrerer Zusicherungen

Wie auch in JUnit, kann man jede Zusicherung mit einem Hinweistext versehen. Nachfolgend werden statt Objekten nur einzelne Attribute zum Aufstellen einer Behauptung genutzt:

```
assertThat("age", mike.getAge(), equalTo(38));
assertThat("city", mike.getCity(), equalTo("Zürich"));
```

Generierte Fehlermeldungen

Die große Stärke von Hamcrest sind die generierten Fehlermeldungen. Diese werden erst beim Fehlschlagen von Bedingungen sichtbar. Schauen wir uns dies an einem Beispiel an. Wir wollen sicherstellen, dass die Variable `age` einen Wert größer 30 enthält und eine ungerade Zahl darstellt, was wir über den Einsatz des Modulo-Operators '`%`' lösen. Beginnen wir mit einem Test in JUnit, und zwar mit einem simplen `assertTrue()`:

```
assertTrue(mike.getAge() > 30 && mike.getAge() % 2 != 0);
```

Man erhält eine wenig aussagekräftige Fehlermeldung, etwa wie folgt:

```
junit.framework.AssertionFailedError
```

Wir nutzen Hamcrest und schreiben den Test folgendermaßen:

```
assertThat(mike.getAge(),
           allOf(greaterThan(30), not(evenNumber())));
```

Nun ist die generierte Fehlermeldung deutlich besser lesbar und macht die Ursache klar:

```
java.lang.AssertionError:
Expected: (a value greater than <30> and not even number)
but: not even number was <38>
```

Zwar kann man Assertions in JUnit eine Fehlerbeschreibung mitgeben, etwa wie folgt:

```
assertTrue("age > 30 and odd", mike.age > 30 && mike.age % 2 != 0);
```

Trotzdem ist die generierte Fehlermeldung

```
java.lang.AssertionError: age > 30 and odd
```

schlechter verständlich als die automatisch generierte von Hamcrest, weil Letztere mehr Informationen transportiert. Darüber hinaus hat man weniger Aufwand bei Anpassungen der Logik, da Hamcrest die Texte dynamisch generiert. Bei JUnit muss man immer achtgeben, dass der Hinweistext mit dem geprüften Verhalten konsistent ist. Allerdings gibt es doch noch eine Sache … der obige Matcher `evenNumber` ist kein Standard-Matcher von Hamcrest, sondern ein selbst erstellter.

Sehen wir uns zunächst an, welche Standard-Matcher in Hamcrest existieren, bevor wir dann auf die Implementierung des eigenen `IsEvenNumber`-Matchers eingehen – insbesondere, wie einfach das geht.

Übersicht über die wichtigsten Matcher

Hamcrest bietet diverse Matcher zur Beschreibung von Bedingungen, wodurch sich Unit Tests besser lesbar gestalten lassen. Dazu sollten die Matcher mit statischen Imports aus dem Package `org.hamcrest` eingebunden werden. Leider gibt es in der Dokumentation keine Übersicht, welcher Matcher wo zu finden ist. Damit man Hamcrest effektiv einsetzen kann, sind diese Informationen aber wichtig. Daher erfolgt hier eine Auflistung der wichtigsten Matcher mit den Packages, aus denen sie stammen.

Object Matcher

Referenz- und inhaltliche Gleichheit prüft man mit den Matchern `sameInstance` bzw. `equalTo`. Nützlich ist auch der typprüfende Matcher `instanceOf`. Zum Prüfen auf `null` dienen die Matcher `notNullValue` und `nullValue`.

```
import static org.hamcrest.core.IsSame.sameInstance;
import static org.hamcrest.core.IsEqual.equalTo;
import static org.hamcrest.core.IsInstanceOf.instanceOf;
import static org.hamcrest.core.IsNull.notNullValue;
```

```
import static org.hamcrest.core.IsNull.nullValue;
```

Logische Matcher

Logische Operationen werden durch die Matcher `allOf`, `anyOf` und `not` realisiert. Diese entsprechen den logischen Operatoren '`&&`', '`||` " bzw. '`!` ' in Java.

```
import static org.hamcrest.core.AllOf.allOf;
import static org.hamcrest.core.AnyOf.anyOf;
import static org.hamcrest.core.IsNot.not;
```

String Matcher (nicht alle in JUnit)

Für den Vergleich von Strings existieren diverse Matcher mit sprechenden Namen. `equalToIgnoringCase` testet die case-insensitive Gleichheit von Strings. Mit `equalToIgnoringWhiteSpace` kann man die Gleichheit testen, ohne auf Leerzeichen zu achten. Die Matcher `containsString`, `endsWith`, `startsWith` erlauben den Test auf Übereinstimmung mit Teilstrings.

```
import static org.hamcrest.text.IsEqualIgnoringCase.equalToIgnoringCase;
import static org.hamcrest.text.IsEqualIgnoringWhiteSpace.
      equalToIgnoringWhiteSpace;
import static org.hamcrest.core.StringContains.containsString;
import static org.hamcrest.core.StringEndsWith.endsWith;
import static org.hamcrest.core.StringStartsWith.startsWith;
```

Number Matcher (nicht in JUnit)

In Abschnitt 4.1.2 haben wir Probleme beim Vergleich von Gleitkommazahlen kennengelernt. Der Matcher `closeTo` prüft, ob Gleitkommazahlen innerhalb eines angegebenen Toleranzbereichs liegen. Die Matcher `greaterThan`, `greaterThanOrEqualTo`, `lessThan`, `lessThanOrEqualTo` prüfen entsprechend ihrem Namen auf größer, größer gleich, kleiner und kleiner gleich.

```
import static org.hamcrest.number.IsCloseTo.closeTo;
import static org.hamcrest.number.OrderingComparison.greaterThan;
import static org.hamcrest.number.OrderingComparison.greaterThanOrEqualTo;
import static org.hamcrest.number.OrderingComparison.lessThan;
import static org.hamcrest.number.OrderingComparison.lessThanOrEqualTo;
```

Collections Matcher (nicht alle in JUnit)

Auch im Bereich von Collections existieren eine Menge an hilfreichen Matchern, unter anderem folgende:

```
import static org.hamcrest.core.IsCollectionContaining.hasItem;
import static org.hamcrest.core.IsCollectionContaining.hasItems;
import static org.hamcrest.collection.IsMapContaining.hasEntry;
import static org.hamcrest.collection.IsMapContaining.hasKey;
import static org.hamcrest.collection.IsMapContaining.hasValue;
```

Lesbare Suchen in Collections

Zur Suche nach einen bestimmten Element in einer Collection hat man mit JUnit in etwa Folgendes geschrieben:

```
boolean found = false;
for (final Person currentPerson: customers)
{
    if (currentPerson.equals(desiredPerson))
    {
        found = true;
        break;
    }
}
assertTrue(found);
```

In Hamcrest lässt sich dies deutlich besser lesbar mit `hasItem()` schreiben:

```
assertThat(customers, hasItem(desiredPerson));
```

Einbinden von Matchern

Das Schwierigste beim Einsatz der Matcher von Hamcrest besteht teilweise darin, das entsprechende Package zu finden, in dem der gewünschte Matcher definiert ist. Das kann ich Ihnen leider nicht abnehmen, aber durch die obige Liste ein wenig erleichtern. Wenn man die in JUnit integrierten Matcher nutzt, so muss man die Packages genau kennen. Sofern man Hamcrest separat einbindet, dann stehen weitere Matcher zur Verfügung. Praktischerweise kann man diese über eine Utility-Klasse `Matchers` ansprechen, die alle Matcher bündelt und so deren Einsatz erleichtert. Nachfolgend ist gezeigt, wie man den `equalTo`-Matcher einbinden kann:

```
import static org.hamcrest.Matchers.equalTo;
import static org.hamcrest.core.IsEqual.equalTo;
```

Definition eigener Matcher

Hamcrest lässt sich leicht um eigene Realisierungen von Matchern erweitern. Um das nachzuvollziehen, werden wir den Matcher `IsEvenNumber` wie folgt selbst definieren:

```
import org.hamcrest.Description;
import org.hamcrest.Factory;
import org.hamcrest.Matcher;
import org.hamcrest.TypeSafeMatcher;

public class IsEvenNumber extends TypeSafeMatcher<Integer>
{
    @Override
    public boolean matchesSafely(final Integer number)
    {
        return number.intValue() % 2 == 0;
    }

    public void describeTo(final Description description)
    {
```

```
        description.appendText("even number");
    }

    @Factory
    public static Matcher<Integer> evenNumber()
    {
        return new IsEvenNumber();
    }
}
```

Wir nutzen dazu die Basisklasse `TypeSafeMatcher<T>` und Folgendes:

1. Eine Fabrikmethode `evenNumber()`, die eigene Instanzen eines Matchers erzeugt.
2. Eine Methode `matchesSafely(T)`, die das Matching implementiert.
3. Eine Beschreibungsmethode `describeTo(Description)`, die eine aussagekräftige Fehlermeldung aufbereitet.

20.8.2 MoreUnit

JUnit ist gut in Eclipse integriert und Testfälle lassen sich über Buttons und Kontextmenü ausführen und sogar debuggen. Allerdings gibt es doch noch Raum für Verbesserungen bei der Integration und beim Schreiben von Unit Tests. Beispielsweise sind die Tastaturkürzel zum Ausführen von Unit Tests recht kryptisch und unhandlich, etwa ALT+SHIFT+X,T. Diesem und weiteren Problemen nimmt sich das Eclipse-Plugin MoreUnit an. Es steht frei im Eclipse Marketplace zur (einfachen) Installation bereit und bietet folgende Features:[21]

- MoreUnit bietet Tastaturkürzel zum Ausführen (CTRL+R) und zum Hin-und-her-Wechseln zwischen Implementierung und Unit Test zu einer Klasse (CTRL+J). Existiert kein Test, so wird mit CTRL+J ein Dialog zum Anlegen angeboten.
- Sofern eine Klasse Abhängigkeiten besitzt, kann man sich dafür automatisch Mock-Objekte (z. B. für Mockito, EasyMock, jMock 2) erstellen lassen.
- Man erhält eine Icon-Dekoration, sodass man direkt im Package Explorer durch einen kleinen grünen Punkt sieht, ob ein Test zu einer Klasse existiert.
- Beim Refaktorisieren werden Klassen und korrespondierende Testklassen automatisch synchron zueinander verschoben oder umbenannt.

20.8.3 Infinitest

Infinitest ist ein Tool zur Unterstützung von kontinuierlichen Testläufen von Unit Tests.[22] Infinitest adressiert eine Unzulänglichkeit von normalen Unit Tests: Diese Tests müssen nach Änderungen im Sourcecode in der Regel von Hand ausgeführt

werden. Schnell wird dies in der Hektik des Alltags vergessen oder ein relevanter Test übersehen, und so werden durch Änderungen neu eingeführte Fehler eventuell nicht aufgedeckt. Durch Infinitest findet man Fehler in der Regel schneller und sicherer. Es erlaubt Unit Tests automatisch als Regressionstests direkt in der IDE auszuführen.[23] Die auszuführenden Tests werden intelligent ausgewählt, d. h., es werden nur die Tests durchgeführt, die von den Änderungen im Sourcecode betroffen sind. Bei auftretenden Fehlern erfolgt eine automatische Rückmeldung in der Statuszeile.

Das direkte Ausführen von Unit Tests noch innerhalb der IDE beim Entwickeln ist aus zweierlei Hinsicht erstrebenswert:

1. Man erhält eine sofortige Rückmeldung, ob man mit einer Änderung etwas kaputt gemacht hat. Das ist insbesondere bei TDD hilfreich.
2. Durch die instantane Rückmeldung erleichtert man nochmals die Arbeit mit Continuous Integration, da Fehler nicht erst durch den Build-Lauf und das Ausführen von Unit Tests auf dem Build-Server gefunden werden.

20.8.4 Cobertura

In diesem Kapitel habe ich bereits motiviert, dass eine durch Unit Tests abgesicherte Implementierung für qualitativ bessere Software sorgen kann. ***Die erzielbare Qualität hängt jedoch maßgeblich von der Güte und der Anzahl der Testfälle ab***. Erinnern Sie sich an den Satz von James Coplien: »Tests don´t improve quality: developers do.« Gute Tests erhöhen allerdings nur die Qualität der durch diese tatsächlich überprüften, abgedeckten Programmteile. Man spricht in diesem Zusammenhang auch von ***Testabdeckung***. Diese durch Sourcecode-Analyse von Hand zu bestimmen, ist nahezu unmöglich. Mithilfe von Tools kann man die Testabdeckung klassen- und methodengenau ermitteln und so die Programmteile aufzeigen, die vollständig, nur teilweise oder auch gar nicht von Tests überprüft werden.

Das Tool Cobertura hilft bei der Bestimmung der Testabdeckung.[24] Zur Aufbereitung der Informationen klinkt sich Cobertura automatisch und dynamisch in den erzeugten Bytecode des zu untersuchenden Programms ein, wodurch vom Entwickler selbst keine Änderungen am Programm vorzunehmen sind. Die ermittelten Informationen pro Klasse, pro Package und für das gesamte Projekt werden durch Cobertura übersichtlich als Reports in HTML aufbereitet. Werfen wir dazu einen Blick auf Abbildung 20-20.

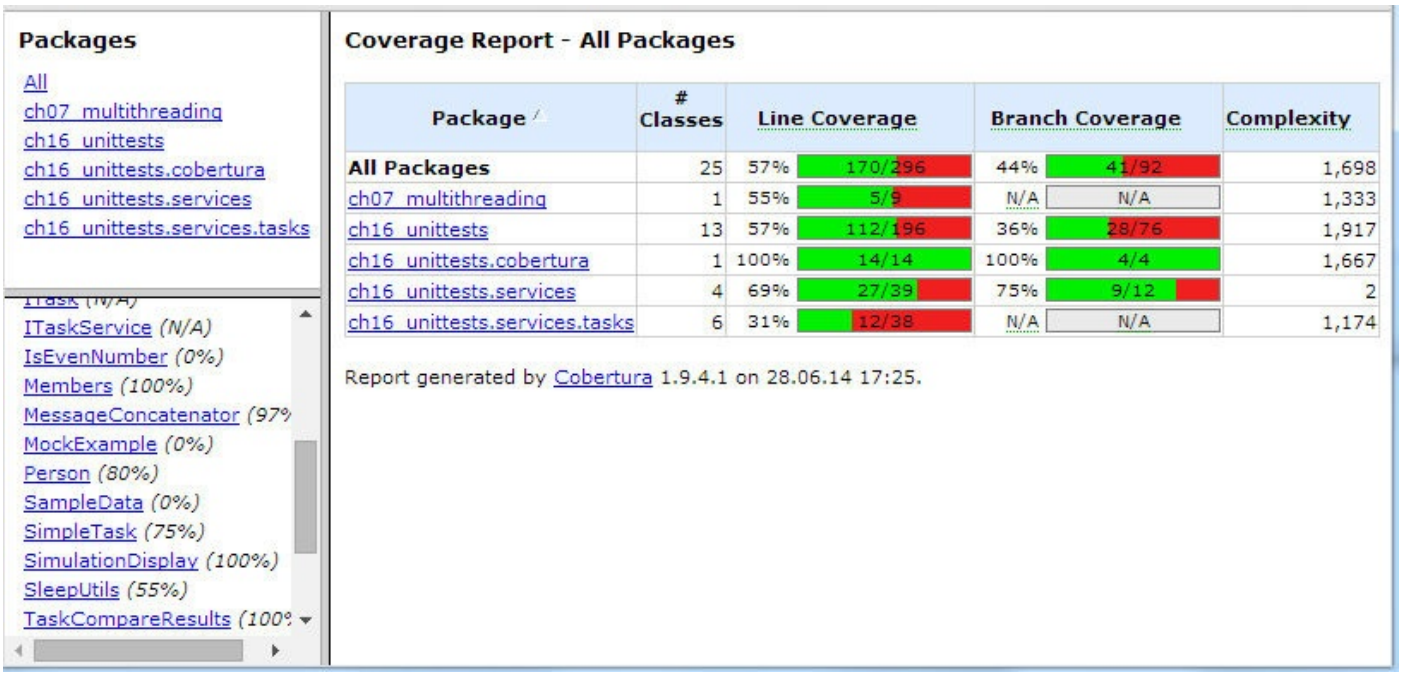


Abbildung 20-20 *Von Cobertura generierter Report*

Bewertung der Messergebnisse und Ermittlung fehlender Testfälle

Basierend auf den erzeugten Reports kann man (weitgehend) ungetestete Teile der Software identifizieren und dafür bei erkanntem Bedarf neue Tests erstellen. Ziel ist es, weitere Fehler durch Test aufzudecken und die Qualität zu erhöhen, nicht aber nur willkürlich die Testabdeckung zu erhöhen. Abgesehen von der gemessenen Testabdeckung ist es viel wichtiger, sinnvolle Testfälle zu erstellen. Was ist damit gemeint? Wenn man Tests nur entwickelt, um die Testabdeckung zu erhöhen, wird man oftmals für weniger kritische Teile Test erstellen. Zudem werden dann eventuell einige für ein zuverlässiges Verhalten der Klasse essenzielle Tests nicht erstellt, weil diese (kaum) zur Testabdeckung beitragen. Eine hohe Testabdeckung ist also kein Selbstzweck!

Mit diesem Hinweis im Hinterkopf kann uns die Testabdeckung aber eine gute Hilfestellung sein, um diejenigen Programmstellen zu erkennen, die weitere Tests bedürfen. Werden dies ergänzt, so steigt auch die Testabdeckung. Durch weitere relevante Unit Tests verbessert sich auch die innere Qualität: Dies ist dadurch begründet, dass ungetesteter Sourcecode normalerweise eine Menge versteckter Fehler enthält. Aber je weniger Testfälle existieren, desto weniger können von diesen »schlummernden Zeitbomben« gefunden werden. Das Erstellen von Tests sollte man nicht unüberlegt fortführen, sondern nur so lange, wie sich messbar die Qualität verbessert.

Mit gewissem Aufwand lassen sich recht gut etwa 50 – 70 % Testabdeckung erreichen. Gerade für komplexere Berechnungen sind höhere Werte wünschenswert. Eine Testabdeckung von 100 % ist allerdings nur selten zu erzielen, da es Programmzeilen, Methoden oder im Extremfall sogar ganze Klassen gibt, die sich

schwierig testen lassen. Ein Beispiel dafür sind unter anderem `catch`-Blöcke, die nicht angesprochen werden, oder Methoden, die leer implementiert werden, nur um ein Interface zu erfüllen. Abbildung 20-21 zeigt die `enum`-Aufzählung `TaskCompareResults`, die eine statische Prüfmethode `isOkValue(TaskCompareResults)` anbietet. Für diese sieht man, dass nicht alle Kombinationen der Oder-Verknüpfung durch Tests geprüft werden und diese Zeile dadurch nicht vollständig getestet worden ist. Daraus ergeben sich die dargestellten Werte für »Line Coverage« und »Branch Coverage«.

Abbildung 20-21 *Coverage-Report für eine* `enum`*-Aufzählung*

Auswirkungen auf die Qualität

Wie eingangs erwähnt, wird die erzielbare Qualität der Software maßgeblich von der ***Güte*** der Testfälle und im geringeren Maße von deren *Anzahl* bestimmt. Wie fast immer gilt »Qualität vor Quantität«. Mit Tools wie Cobertura bestimmt man aber lediglich die Testabdeckung, die normalerweise durch die Anzahl der Testfälle steigt. Man erhält jedoch keine Aussage über die Güte jedes einzelnen Tests. Werden nur »harmlose« Testmethoden geschrieben, um eine möglichst hohe Testabdeckung zu erreichen, ist dies nicht hilfreich, sondern manchmal sogar kontraproduktiv. Entscheidend ist, dass die Testmethoden Fehlerfälle provozieren und somit helfen, Fehler aufzudecken. ***Eine geringere Testabdeckung mit guten Tests für komplexe und fehleranfällige Teile des Programms ist einer höheren Testabdeckung vorzuziehen, wenn diese nur dadurch erzielt wird, dass triviale*** **`get-`** ***und*** **`set-`** ***Methoden geprüft werden***.

Ein letzter Punkt ist noch zu beachten: Eine hohe Testabdeckung besagt nicht,

dass die Unit Tests *erfolgreich* durchlaufen werden, sondern lediglich, dass die Zeilen von einem Unit Test geprüft werden. Das Testergebnis kann jedoch negativ ausfallen. Betrachten wir dies für folgende Methode `coverage100ButNPE()`, die auf Anregungen aus einem interessanten Artikel[25] basiert.

```
public class Coverage
{
    public String coverage100ButNPE(final boolean condition)
    {
        String value = null;
        if (condition)
        {
            value = String.valueOf(condition);
        }
        return value.trim();
    }
    // …
```

Für diese Methode kann man mit zwei Testfällen für 100 % Abdeckung sowohl für die »Line Coverage« als auch für die »Branch Coverage« sorgen:

```
public class CoverageTest
{
    final Coverage coverage = new Coverage(4711, "4711");

    @Test
    public void testCoverage100ButNPE1()
    {
        coverage.coverage100ButNPE(true);
    }

    @Test
    public void testCoverage100ButNPE2()
    {
        // Löst eine NullPointerException aus
        coverage.coverage100ButNPE(false);
    }
}
```

Wie wenig Aussagekraft 100 %-Abdeckung besitzt, erkennen wir, wenn wir den Test ausführen. Für den Übergabewert `false` wird eine `NullPointerException` ausgelöst. Das Programm enthält also trotz einer vermeintlich perfekten Testabdeckung von 100 % dennoch einen eklatanten Fehler. Als Fazit könnte man feststellen, dass eine Testabdeckung von 25 % durch Tests geschäftskritischer Funktionalität viel mehr Wert sein kann als 70 %, wenn dieser Wert nur durch Tests einfacher Methoden erreicht wird.

Ausführen von Cobertura

Um die Testabdeckung mit Cobertura bestimmen zu können, binden wir es in unseren Build-Prozess ein. Zum Ausführen von Cobertura muss zunächst das Plugin samt einer kleinen Konfiguration in die Build-Datei `build.gradle` aufgenommen werden:

```
apply plugin: 'cobertura'
```

```
cobertura
{
    format = 'html'
    includes = ['**/*.java']
    excludes = ['**/*Test.class', '**/*Test.java']
}

// Integration als externes Plugin
buildscript
{
    repositories
    {
        mavenCentral()
    }
    dependencies
    {
        classpath 'com.eriwen:gradle-cobertura-plugin:1.1.1'
    }
}
```

Da es sich bei Cobertura um kein Standard-Plugin von Gradle handelt, sind weitere Angaben in der Build-Datei notwendig, die auf das Plugin verweisen und in der Sektion `buildscript` angegeben werden. Zur Berechnung der Testabdeckung muss man lediglich einen Build ausführen. Dadurch werden die Klassen instrumentiert und die Unit Tests ausgeführt. Das Ergebnis wird dann ähnlich zu der zu Beginn gezeigten Ausgabe im Unterordner `build/reports/cobertura` in der zuvor definierten Form als HTML oder XML abgelegt.

Tipp: Wissenswertes zur Ausführung von Cobertura

Fehlschlagende Tests Wenn die Tests nicht bestanden werden, wie dies bewusst hier für die Klasse `CoverageTest` der Fall ist, so bricht der Build nach dem Test ab. Um dennoch einen Cobertura-Report zu erzeugen, muss man das Kommando `gradle testCoberturaReport` manuell ausführen, nachdem die Tests z. B. mit `gradle clean test` ausgeführt wurden.

Abhilfe bei VerifyError Beim Ausführen von Cobertura kommt es teilweise zu merkwürdigen Fehlern folgender Art: `java.lang.VerifyError: Expecting a stackmap frame at branch target 70 in method` .... Als Abhilfe ergänzt man dann im Build-Skript folgende Zeilen:

```
test
{
    jvmArgs '-XX:-UseSplitVerifier'
}
```

20.8.5 EclEmma

Ähnlich wie Cobertura ist EclEmma ein freies Tool zur Bestimmung der Testabdeckung. Der Vorteil von EclEmma gegenüber Cobertura ist, dass die IDE nicht verlassen werden muss und die Testabdeckung beim Ausführen von Unit Tests

berechnet wird. Einzige Voraussetzung ist die Installation des EclEmma-Plugins über den Eclipse Marketplace.[26] Danach bietet EclEmma folgende Eigenschaften:

- Es besitzt eine gute und nahtlose Integration in Eclipse.
- Die initialen Konfigurationsarbeiten sind kinderleicht: Es muss nichts am Projekt verändert oder konfiguriert werden.
- Die Prüfung ist per Kontextmenü einfach auszuführen – analog zu Test mit JUnit.
- Es werden diverse Metriken ermittelt und übersichtlich in einem speziellen View dargestellt. Darüber hinaus werden die nicht abgedeckten Zeilen im Sourcecode-Editor farblich markiert.

Insgesamt ist EclEmma eine sinnvolle Ergänzung zu Cobertura. Ersteres hilft bei der alltäglichen Arbeit direkt in der IDE. Cobertura kann dagegen im Rahmen des Build-Prozesses und auch von Continuous Integration eingesetzt werden. Die dabei produzierten HTML-Reports kann man dann historisieren und so Trends ermitteln. Mit EclEmma erhält man direkt Rückmeldung und sieht schon während der Entwicklung, ob man auf dem richtigen Weg ist, seine Testabdeckung zu verbessern, und insbesondere auch, wo Bedarf für weitere Tests besteht.

20.9 Schlussgedanken

Zum Abschluss dieses Kapitels möchte ich Ihnen ein paar persönliche Gedanken zum Thema Testen, Erstellen von Designs und Unit Tests sowie zum Vorgehen beim Entwurf von Softwaresystemen darlegen. Lassen Sie sich doch von dem folgenden Meinungskasten inspirieren. Zu guter Letzt wünsche ich Ihnen viel Spaß beim Entwickeln und Unit-Testen und weniger Stress durch die neu gewonnene Sicherheit und das Vertrauen in den eigenen Sourcecode.

Meinung: »Think-First-Then-Code-And-Test«

Der Weg zu testbarer Software erfordert sauberes Design und eine hochwertige Implementierung, was eine strukturierte Arbeitsweise voraussetzt. Für den konkreten Ablauf beim Entwurf von Software gibt es diverse, sehr kontroverse Vorgehensweisen. Ich halte von Extremen nichts, denn die Wirklichkeit ist nie schwarz oder weiß. Vielmehr möchte ich nachfolgend beschreiben, welches Vorgehen sich für mich beim Entwurf in den letzten zwanzig Jahren bewährt hat, um stabile, wartbare und (nach wenigen Testläufen) nahezu fehlerfreie Software zu erstellen. Meiner Erfahrung nach ist weder der Weg des Big Upfront Design (BUD) noch derjenige des Test-Driven Development (TDD) zielführend, sondern eine inkrementelle iterative Kombination aus dem Besten beider Ansätze. Warum?

Mögliche Beschränkungen beider Extreme

Beim BUD wird oftmals nur zu Beginn des Projekts mit dem Kunden gesprochen, um seine Anforderungen zu ermitteln. Zudem wird viel zu häufig eine Interaktion mit dem Nutzer oder Kunden vernachlässigt und wenig flexibel ein einmal eingeschlagener Weg weiterverfolgt. Sehr schnell entwickelt man dadurch an den wirklichen Anforderungen vorbei – damit dies nicht geschieht, sollte man mit Kunden Zwischenstände begutachten. Kommen wir kurz zum TDD: Nicht dass ich missverstanden werde, ich befürworte das Testen und insbesondere auch das Testen der eigenen Programme durch Entwickler mit Unit Tests. ***Allerdings halte ich es für einen Irrglauben, ein gutes Design lediglich basierend auf Tests evolutionär entwickeln zu können***. Sie kennen es sicher auch: Manchmal verrennt man sich beim Programmieren der Lösung eines Problems: Das Gewirr von Methodenaufrufen und Klassen fängt an, mehr Verwirrung zu stiften, als einem lieb ist. Man beginnt, die Kontrolle zu verlieren, weil man zu tief in die Details abgetaucht ist und man so das eigentliche Ziel momentan nicht im Blick hat. Dann hilft es, sich zurückzulehnen, einen Kaffee zu trinken und einen Schritt zurück zu machen, um auf eine andere Ebene der Problemlösung zu gelangen. Schaut man aus einem anderen Blickwinkel, so ergibt sich teilweise eine viel elegantere Lösung. Um aber dorthin zu gelangen, benötigt man – meiner Erfahrung nach – immer ein klares Ziel und eine Vision, was die Software können soll, damit man erfolgreich arbeiten kann. Wenn man ein Grobdesign besitzt, dann kann man das Feintuning sehr gut basierend auf Tests durchführen. Vergleichen Sie es mit dem Schreiben eines Buchs: Wenn man keine grobe Struktur hat, wird man niemals einfach Sätze oder Worte aneinanderreihen können und einfach so lange umordnen, bis irgendetwas Sinnvolles entsteht. Es fehlt einfach der Masterplan. Ohne diesen geht meiner Meinung nach nichts. Wenn einige Verfechter von TDD wirklich ehrlich wären, würden sie Ihnen nicht nur triviale Beispiele zeigen, sondern komplexere Gebilde mit vielen Abhängigkeiten. So etwas kann man eben nicht kleinteilig entwickeln. Vielmehr benötigt man zum Schreiben von Tests ziemlich genaue Anforderungen. Das Schlimmste, was passieren kann, ist, dass man sich alle Testfälle selbst ausdenkt, ohne die Anforderungen zu kennen. Dann entwickelt man nämlich genauso an den Kundenbedürfnissen vorbei wie beim BUD ohne weitere Interaktionen und Rückmeldungen vom Kunden.

Wie schreibt man also qualitativ hochwertige Software (und Tests)?

Ich möchte mit einem Zitat von Antoine de Saint-Exupéry beginnen:

> Wenn du ein Schiff bauen willst, so fange nicht damit an, Holz zu sammeln, Planken zu schneiden und die Arbeit einzuteilen, sondern erwecke in den Menschen die Sehnsucht nach dem weiten endlosen Meer.

Nach diesem gedanklichen Ausflug zum Erreichen großer Ziele gebe ich Ihnen nun ein paar Tipps mit auf den Weg, mit denen ich in einigen Projekten bereits gute

Erfahrungen gemacht habe. Ich nenne es den »Think-First-Then-Code-And-Test«-Ansatz. Wichtig ist mir, immer den gesunden Menschenverstand einzusetzen und in etwa folgende Schritte zu befolgen:

1. Versuche das zu lösende Problem möglichst gut zu verstehen. Hierbei können etwa Anforderungsdokumente, Pflichtenhefte und insbesondere auch persönliche Gespräche mit den Kunden helfen.
2. Entwickle ein erstes Design und prüfe Anforderungen, z. B. indem ein GUIPrototyp entweder auf Papier oder mithilfe von Mock-up-Tools entsteht. Das hilft, mögliche Schwachpunkte oder fehlerhafte Annahmen aufzudecken.
3. Kläre Fragen mit geeigneten Ansprechpartnern oder mithilfe von Dokumenten.
4. Zerteile das System in kleinere, überschau- und beherrschbare Einzelbausteine und überlege die jeweiligen Verantwortlichkeiten.
5. Nun erst beginnt die Entwicklung (auch die der Tests). Man kann mit einem ersten prototypischen Entwurf oder dem Erstellen von Testfällen anfangen. Zwischenzeitlich kann man auf Stub-Klassen oder Mocks zurückgreifen, wenn die konkrete Funktionalität noch nicht vollständig entwickelt ist.
6. Wenn man diese Basisbausteine erstellt hat, so sorgt man mithilfe geeigneter Unit Tests dafür, dass diese im Idealfall fehlerfrei sind. Das ist einfacher möglich als bei komplexen Programmbausteinen, da die Basisbausteine relativ wenig Eingabeparameter haben und nur einfache Dinge ausführen. Eine Spezifikation und Überprüfung fällt dann relativ leicht. Dazu haben wir bereits Unit Tests, Äquivalenzklassen, Mocking und parametrierte Tests als gutes Handwerkszeug kennengelernt.
7. Füge die kleineren Komponenten zu größeren Einheiten zusammen. Entwickle die Interaktionen und teste diese. Da man hier auf einer eher logischen Ebene arbeitet, fällt auch hier das Erstellen von Testfällen nicht mehr so schwer. Hier verschwimmt die Grenze zwischen Unit Test, Integrationstest und Systemtest.
8. Stelle einen solchen (Zwischen-)Stand dem Kunden vor und hole Feedback ein. Nutze dieses zum Feintuning für die nächsten Arbeiten.

Die obigen Schritte werden wiederholt, bis das gewünschte Resultat erzielt wurde. Danach kann man sich über einen zufriedenen Kunden freuen, der die Software gerne benutzt, da sie auf seine Bedürfnisse zugeschnitten ist.

20.10 Weiterführende Literatur

Dieses Kapitel hat einen Einblick in das Thema Unit-Testen gegeben. Weiterführende Informationen finden Sie in den folgenden Büchern:

- »**Unit-Tests mit JUnit**« von Andrew Hunt und David Thomas [49]

  Ein empfehlenswertes Buch, das Unit-Testen motiviert und vom Einstieg bis zur

Gestaltung komplexerer Testfälle mit guten Tipps und Ratschlägen weiterhilft.

- »**Unit Tests mit Java**« von Johannes Link [60]

  Dieses Buch beschreibt den Test-First-Ansatz beginnend mit den Grundlagen von Unit Tests. Es geht auch auf Stub- und Mock-Objekte ein und behandelt zudem das Testen von Datenbankabfragen sowie den Test nebenläufiger Programme.

- »**Next Generation Java Testing: TestNG and Advanced Concepts**« von Cédric Beust und Hani Suleiman [7]

  In diesem Buch wird das Thema Testen detailliert behandelt. Es beginnt mit JUnit und TestNG und geht auch auf komplexere Testszenarien mit Applikationsservern, Datenbanken usw. ein.

- »**Test Driven**« von Lasse Koskela [57]

  Dieses Buch stellt eine thematische Fortsetzung der zuvor genannten Bücher dar und beschreibt das Unit-Testen im Rahmen der testgetriebenen Entwicklung.

- »**Testgetriebene Entwicklung mit JUnit & FIT**« von Frank Westphal [90]

  Frank Westphal beginnt mit einer Beschreibung der Grundlagen von JUnit 3 und 4 und geht auf die Integration von Unit Tests in den Build-Prozess ein. Außerdem liefert er Tipps zum Erstellen von Testfällen sowie zum Testen mit Stubs und Mocks.

- »**Practical Unit Testing with JUnit and Mockito**« von Tomasz Kaczanowski [52]

  Dieses Buch von Tomasz Kaczanowski führt zunächst schrittweise und gründlich in die Thematik Unit-Testen ein. Anschließend geht er ausführlicher auf TDD, aber auch auf Beschränkungen davon ein. Der Hauptteil widmet sich dann Test-Doubles, im Speziellen Mocks und Mockito als Framework.

- »**Effective Unit Testing**« von Lasse Koskela [58]

  Lasse Koskela zeigt in seinem Buch, wie man lesbare, wartbare und vertrauenswürdige Tests schreibt. Abgerundet wird das Ganze mit einigen, zum Teil kontroversen Gedanken zum Design von Tests.

21 Codereviews

In diesem Kapitel werden sogenannte Codereview-Meetings, kurz Codereviews, vorgestellt. Zunächst beschreibe ich in Abschnitt 21.1, was man darunter versteht. Die Veranstaltung solcher Meetings ist nicht immer ganz unproblematisch. Daher nenne ich in Abschnitt 21.2 einige Probleme, die sich bei der Durchführung ergeben können, und zeige, welche Lösungsmöglichkeiten existieren. In Abschnitt 21.3 motiviere ich, warum Codereviews einen wichtigen Teil der Qualitätssicherung darstellen. Danach gehe ich in Abschnitt 21.4 auf einige Tools zur Unterstützung von Codereviews ein. Abschließend wird in Abschnitt 21.5 eine Checkliste vorgestellt, die als Basis für die Durchführung eigener Codereviews dienen kann.

21.1 Definition

Unter ***Codereviews*** versteht man Meetings von mehreren Entwicklern, in denen Sourcecode-Abschnitte mit dem Ziel betrachtet werden, mögliche Softwaredefekte aufzuspüren. Nach dem Motto »Viele Augen sehen mehr als die eigenen zwei« nutzt man das Potenzial und Wissen anderer Personen, um Probleme im vorliegenden Sourcecode aufzudecken. Die Teilnahme einiger projektfremder Entwickler ist durchaus wünschenswert; häufig kann man von ihrem Wissen profitieren. Vielleicht werden bislang nicht bedachte Themen wie Synchronisation beim Einsatz von Multithreading oder die Nutzung anderer Datenstrukturen angesprochen.

Durchführung

Für ein Codereview in seiner einfachsten Form reicht es bereits, dass beim Pair Programming der jeweils zuschauende Entwickler dem anderen kritisch beim Tippen über die Schulter schaut und Kommentare abgibt. Das Codereview findet dann aber nur nebenbei statt. Alternativ kann man auch ganz gezielt mit einer Gruppe von zwei oder drei Entwicklern einige Klassen oder Methoden direkt am Rechner durchgehen. Häufig ist es für eine konzentrierte Analyse sinnvoller, die zu untersuchenden Teile des Sourcecodes auszudrucken und sich in einem ruhigen Besprechungsraum zusammenzusetzen. Diese Form des Codereviews kann man auf eine größere Anzahl von Entwicklern ausweiten – wobei ich durchaus Meetings mit bis zu 20 beteiligten Personen erlebt habe. Sinnvoll und produktiver sind allerdings die zuvor beschriebenen Meetings in kleinerer Runde mit zwei bis fünf Teilnehmern. Bei einer größeren Teilnehmerzahl nimmt in der Regel die Konzentration ab und soziale Aspekte treten mehr in den Vordergrund: Aus einem zwanglosen und informellen Gespräch wird ein eher schwergewichtiger Vorgang. Dieser Effekt wird verstärkt, wenn das gesamte Vorgehen formaler wird, etwa mit diversen zu erstellenden Protokollen und speziellen Formularen. Dies ist meiner Meinung nach eher hinderlich.

Vorteile

Um Softwaredefekte aufzudecken und die Qualität des Sourcecodes auf einem hohen und einheitlichen Niveau zu halten, bieten sich regelmäßig stattfindende Codereviews an. Ergänzend zu den täglichen (Mini-)Reviews beim Pair Programming können separat stattfindende Codereviews Themengebiete gründlicher beleuchten und mehr in Richtung Konzeptreview gehen. In jedem Fall sollte der begutachtete Sourcecode zusätzlich zu dem Programmlisting auch konzeptuell vorgestellt werden. Verweise auf eingesetzte Entwurfsmuster und ein Systemschaubild in Form von UML-Diagrammen können viele erklärende Worte ersetzen und gleichzeitig als Diskussionsbasis dienen. So entsteht im Laufe der Zeit ein gemeinsames Verständnis von Softwarequalität, wodurch man Problemen vorbeugen kann. Wichtig ist es aber auch, die erarbeiteten Ergebnisse zeitnah als Fehlerbehebungen in den Sourcecode einfließen zu lassen und nicht nur den präventiven Effekt zu nutzen.

Tipp: Zeitpunkt der Codereviews

Codereviews können entweder als sogenanntes ***Pre-Commit-Review*** oder als sogenanntes ***Post-Commit-Review*** durchgeführt werden. Bei ersterem erfolgt die Sourcecode-Überprüfung vor dem Einspielen in ein Repository, beim zweiten nachträglich. In der Praxis finden größere Codereview-Meetings allerdings viel seltener als Commits statt. Häufig sieht man daher eine Mischform: Vor einem Commit kann optional ein spontanes Codereview in kleinerem Rahmen und Umfang stattfinden, um gemäß dem Vieraugenprinzip Flüchtigkeitsfehler zu vermeiden. Regelmäßige Meetings folgen dagegen in der Regel nicht speziellen Commit-Zyklen, sondern dienen als »Nachsorge«-Untersuchung, einerseits, um Defekte aufzudecken, und andererseits, um einen Coding-Standard sicherzustellen.

Psychologische Aspekte und Probleme

Wie bereits angedeutet, besitzen Codereviews auch eine soziale und psychologische Komponente. Der Sourcecode eines Entwicklers wird untersucht. Die Schwierigkeit besteht darin, notwendige Kritik zu äußern, ohne dies vorwurfsvoll oder belehrend zu tun. Ansonsten kann sich der reviewte Entwickler schnell in eine Verteidigungsposition gedrängt fühlen. Auch die Anzahl der Teilnehmer an einem solchen Meeting kann dies auslösen: Je mehr Leute teilnehmen, desto offizieller wird der Charakter des Reviews und es entsteht schnell eine Konfrontationssituation vom betroffenen Entwickler zu den Kritikern. Die positiven Effekte des Codereviews kommen dann nicht zum Tragen und ein solches Meeting schadet eher, denn es wird nichts mehr gelernt. Zudem besteht die Gefahr, dass bei mehreren ähnlich ablaufenden Meetings sich eine Kontra-Codereview-Stimmung entwickelt und niemand mehr freiwillig seinen Sourcecode vorstellen möchte. Es ist Aufgabe des Veranstalters, als Moderator zu fungieren und dafür zu sorgen, dass Reviews in einer angenehmen Atmosphäre ablaufen. Der folgende Abschnitt geht auf mögliche

Probleme und deren Lösung bei der Durchführung von Codereviews ein.

21.2 Probleme und Tipps zur Durchführung

Codereviews zu veranstalten ist oft eine undankbare Aufgabe, denn Qualitätssicherung (QS) muss häufig anderen Dingen weichen, insbesondere dann, wenn es im Projektzeitplan eng wird. Das liegt auch daran, dass Projektleiter nicht immer leicht von den Vorteilen von QS-Maßnahmen zu überzeugen sind, vor allem dann nicht, wenn dies den Projektplan zunächst etwas verzögert. Diese Einstellung ist jedoch eher kontraproduktiv: QS-Maßnahmen wie Unit Tests und Codereviews können in hektischen Projektphasen besonders nützlich sein, da sie das Verständnis für realisierte Konzepte vertiefen und für weniger Aufregung bei Änderungen trotz knapper Zeit sorgen können. Wie bereits in Kapitel 20 erwähnt, führt zudem eine gute Testabdeckung durch Unit Tests zu mehr Vertrauen und Sicherheit, da mögliche Probleme in Erweiterungen schnell sichtbar werden. Meint man, auf QS-Maßnahmen verzichten zu können, so erhöht sich in der Regel der nachträgliche Testaufwand für Integrations- und Applikationstests beträchtlich. Auch die Fehlerrate beim Einsatz der Applikation und die daraus resultierenden Wartungsaufwände steigen. Ähnlich zu einem Darlehen steht anfangs nur vermeintlich mehr Zeit für Implementierungsaufgaben zur Verfügung. Tatsächlich wird aber nur kurzfristig Zeit eingespart, die später in Form von umfangreicheren Tests »zurückgezahlt« werden muss. Bei nicht vollständig umgesetzter Funktionalität spricht man auch von technischen Schulden oder ***technical debt***.

Es gibt aber auch andere Gründe für eine Ablehnung von Codereviews: Von einigen Kollegen werden nahezu alle Meetings als lästiges Übel angesehen – da macht dann auch das Codereview keine Ausnahme. Und so sprechen gegen die Teilnahme immer einige gute und weniger gute Gründe, wie etwa Termindruck oder generelles Desinteresse. Selbst bei einer anfänglichen Befürwortung kann die Motivation zur Teilnahme an Codereviews im Laufe der Zeit abnehmen. Zudem ist es häufig so, dass genau die Leute am meisten von Codereviews profitieren würden, die sich dagegen aussprechen.

Werfen wir kurz einen Blick auf mögliche Ursachen der genannten Probleme und erarbeiten einige Vorschläge, wie man diesen gegensteuern kann.

1. **Sourcecode-Qualität**

   Problem: Die vorgestellten Programmteile haben häufig keine besondere Qualität und man diskutiert immer wieder bereits hinlänglich bekannte Fehler (z. B. keine Abfrage auf `null`, zu lange Methoden, fehlende Dokumentation).

   Abhilfe: Lästige Routinearbeiten – wie das Prüfen auf Einhaltung von Coding Conventions – sollten im Vorfeld durch Tools, etwa die bereits in Abschnitt 19.4 vorgestellten Checkstyle, FindBugs oder PMD, durchgeführt werden. Damit entfällt der rein formale und syntaktische Aspekt der Sourcecode-Qualität und die

Teilnehmer können sich mit Fragen zu Design und Konzepten beschäftigen, ohne sich in Nebensächlichkeiten zu verlieren.

2. **Vorgestellter Sourcecode-Umfang**

   Problem: Die vorgestellten Abschnitte sind meistens deutlich zu lang (fünf und mehr DIN-A4-Seiten sind keine Seltenheit).

   Abhilfe: Der Umfang sollte auf einige 100 Zeilen oder ausgedruckt auf wenige, d. h., etwa eine bis drei DIN-A4-Seiten beschränkt werden. Ansonsten können die vorgestellten Programmstellen und möglicherweise darin enthaltene Probleme nicht gründlich analysiert werden. Dann ist der Gegenwert im Verhältnis zum Zeitaufwand zu gering.

3. **Vorgestellte Themen**

   Problem: Der Lerneffekt bleibt aus, da zu selten spannende Konzepte vorgestellt und stattdessen immer wieder lange »Sourcecode-Wüsten« durchforstet werden.

   Abhilfe: Es sollte vorwiegend thematisch ausgewählter, interessanter Sourcecode untersucht werden, der zudem den zuvor genannten Umfang nicht überschreitet. Aber nicht nur der Sourcecode an sich, sondern auch die realisierten Konzepte, also nicht nur das »Wie«, sondern auch das »Warum so und nicht anders« sind von Interesse. Konzept-, Design- und Architekturreviews mit einer Vorstellung von Entwurfsmustern sowie von Tipps und Tricks können die Meetings bereichern. Hier ist man als Veranstalter gefragt, die richtige Mischung zu finden.

4. **Veranstaltungsdauer**

   Problem: Das Meeting zieht sich über mehr als eine Stunde hin; im Extremfall über mehr als zwei Stunden.

   Abhilfe: Bei Codereview-Meetings geht es um konzentriertes Analysieren. Mehr als eine Stunde ist das kaum möglich. Daher sollte die Meetingdauer auf maximal eine Stunde beschränkt werden. Können in dieser Zeit nicht alle Probleme diskutiert werden, so sollten sie in einem Folgemeeting wieder aufgegriffen werden. Eine Zeitbegrenzung bietet außerdem den Vorteil, dass selbst bei Projektstress eine Teilnahme möglich und auch bei schwierigeren Themen noch ausreichend Konzentration vorhanden ist.

5. **Gegenwert**

   Problem: Es kommt vor, dass mit großer Mannschaft und viel Aufwand Codereviews durchgeführt werden, ohne jedoch im Nachhinein die gefundenen Defekte zu beheben. Werden Resultate aber nicht verwendet, stellt sich die Frage nach dem Nutzen des Meetings. Es kommt schnell das Gefühl auf, dass geäußerte Kritik keine Konsequenzen auf die Sourcecode-Qualität hat. Dies erzeugt unter Umständen das Gefühl, zu viel Zeit für zu wenig Gegenwert aufzubringen. Der Eindruck verstärkt sich vor allem in »heißen« Projektphasen.

   Abhilfe: Codereviews dienen der aktiven Verbesserung von Softwarequalität.

Daher sollten festgestellte Defekte möglichst schnell aus dem untersuchten Sourcecode entfernt werden. Falls es die Änderungen und Anmerkungen sinnvoll erscheinen lassen, kann der überarbeitete Sourcecode später einem erneuten Review unterzogen werden. Manchmal findet man bei einem solchen iterativen Vorgehen noch einige weitere Verbesserungsmöglichkeiten.

6. **Teilnahme**

   Problem: Die Erfahrung zeigt, dass bei einer freiwilligen Teilnahme immer wieder anderen Arbeiten mehr Priorität eingeräumt wird.

   Abhilfe: Codereviews sollten, wie andere Meetings auch, Pflichttermine sein. Einerseits verhindert man dadurch Ausreden von Entwicklern, die eine Teilnahme vermeiden wollen. Andererseits erschwert dies Projektleitern, einzelne Entwickler von der Teilnahme abzuhalten.

21.3 Vorteile von Codereviews

Nachdem die Durchführung von Codereviews inklusive einiger möglicher Probleme vorgestellt wurde, zeigt dieser Abschnitt die positiven Auswirkungen auf die Softwarequalität anhand von Beispielen aus der Praxis. Basierend auf den Erkenntnissen des vorangegangenen Abschnitts sorgen wir durch den Einsatz von Sourcecode-Analysetools bereits dafür, dass ein Mindeststandard bezüglich der Sourcecode-Qualität erreicht wird, bevor wir mit den Codereviews beginnen. Formale Aspekte wie Namenskonventionen, Methodenlänge usw. lassen sich mit Tools gut überprüfen, semantische Aspekte und die Korrektheit der eingesetzten Algorithmen sind dagegen durch Tools nur schwer oder gar nicht überprüfbar. Durch Codereviews ist beides jedoch leicht möglich. Betrachten wird dazu einige Beispiele, um dies zu verdeutlichen.

Beispiel 1

Beginnen wir mit der Klasse `DepartureLineTimeComparator`, die Objekte vom Typ `Departure` nach Linie und Uhrzeit vergleicht und wie folgt definiert ist:

```
public final
class  DepartureLineTimeComparator  implements  Comparator<Departure>
{
    private final  Comparator<Departure> lineComparator =  new
LineComparator();
    private final  Comparator<Departure> timeComparator =  new
TimeComparator();

    public int  compare( final  Departure departure1,  final Departure
departure2)
    {
        int  result = lineComparator.compare( departure1, departure2);
        if  (result ==  0)
        {
             timeComparator.compare( departure1, departure2);
```

```
        }
        return  result;
    }
}
```

Diese Klasse ist kurz und übersichtlich, enthält aber dennoch einen versteckten Fehler. Es ist intuitiv klar, dass eine Hintereinanderausführung der beiden `Comparator<Departure>`-Instanzen durchgeführt werden soll. Ein aufmerksamer Blick zeigt schnell den Fehler: Beim Aufruf von `compare(Departure, Departure)` der `TimeComparator`-Instanz fehlt eine Zuweisung an die Variable `result` bzw. alternativ eine `return`-Anweisung, wodurch lediglich die Linieninformationen verglichen wird. Bei einer Prüfung in einem Codereview wäre dieser Fehler höchstwahrscheinlich nicht übersehen worden. Eine Korrektur fällt nicht schwer:

```
public int compare (final Departure departure1, final Departure departure2)
{
    int  result = lineComparator.compare(departure1, departure2);
    if  (result == 0)
    {
        // KORREKTUR //
        result = timeComparator.compare(departure1, departure2);
    }
    return result;
}
```

Beispiel 2

Ein anderes Beispiel ist folgende Zuweisung:

```
boolean onlyTimeChanged = hasSameLine() && hasSameEndPos() && hasSameEndPos();
```

Bei einem Codereview wäre bestimmt aufgefallen, dass man nicht zweimal die Endposition durch Aufruf der Methode `hasSameEndPos()` vergleichen möchte, sondern eigentlich die Start- und die Endposition, wodurch sich folgende Korrektur ergibt:

```
boolean onlyTimeChanged = hasSameLine() && hasSameStartPos() &&
hasSameEndPos();
```

Beispiel 3

Neben solchen Flüchtigkeitsfehlern kann man mit Codereviews auch Fehler im Algorithmus aufdecken, wie in diesem Beispiel:

```
private static volatile boolean shouldTerminate = false;

private static void work()
{
    final Telegram telegram = waitForTelegram();
    if (telegram.getType().equals(TELEGRAM_TYPE_XYZ))
    {
        telegram.performTelegramHandling();
    }
    if (telegram.getType().equals(TELEGRAM_TYPE_QUIT))
```

```
    {
        terminate(); // Seiteneffekt, setzt shouldTerminate-Flag
    }

    if (!shouldTerminate())
    {
        work();
    }
}
```

Das Beispiel war in der Realität noch deutlich komplizierter, sodass ich einige Zeit gebraucht habe, die Intention zu erkennen: Es wird hier lediglich eine `while`-Schleife durch Rekursion nachgebaut. Allerdings ist die Anzahl der Schleifendurchläufe durch die Größe des Stacks bestimmt. Nach einiger Zeit kommt es zu einem `java.lang.StackOverflowError`. Bis dahin kann es aber durchaus längere Zeit dauern, da immer nur eine Methode auf den Stack gelegt wird. Auf meinem Rechner können etwa 12.000 Aufrufe der Methode erfolgen. Wenn man von Methodenlaufzeiten von einigen Sekunden ausgeht, so läuft das Programm durchaus einige Stunden scheinbar fehlerfrei, bis es dann auf einmal abstürzt.

Eine Korrektur fällt einfach, besitzt keine Laufzeitbeschränkungen mehr und ist intuitiv verständlich:

```
private static volatile boolean shouldTerminate = false;

private static void workCorrected()
{
    while (!shouldTerminate())
    {
        final Telegram telegram = waitForTelegram();
        if (telegram.getType().equals(TELEGRAM_TYPE_XYZ))
        {
            Telegram.performTelegramHandling();
        }
        if (telegram.getType().equals(TELEGRAM_TYPE_QUIT))
        {
            terminate();    // Seiteneffekt, setzt shouldTerminate-Flag
        }
    }
}
```

Fazit

Kleine Fehler dieser Art »verstecken« sich leider häufiger im Sourcecode. Entweder werden sie beim Testen erkannt oder sie machen später beim Einsatz des Programms Probleme. ***Codereviews sind hervorragend dazu geeignet, solche »schlummernden Zeitbomben« aufzudecken und zu beheben, bevor sie sich als Fehler (beim Kunden) manifestieren.***

Obwohl Codereviews viel zur Qualitätssicherung beitragen können, so ist es jedoch aufgrund des Umfangs an Sourcecode häufig nicht möglich, jede Zeile zu reviewen. Damit werden logischerweise auch nicht alle »schlummernden Zeitbomben« entdeckt. Allerdings sollte es das Ziel sein, möglichst viel Sourcecode

zu reviewen. Eine gute Abdeckung erreicht man beispielsweise dadurch, dass kein Commit ohne ein vorheriges Codereview stattfinden darf. Selbst wenn man diese Regel nicht strikt umsetzt, so sind Codereviews trotzdem sehr wertvoll, um ein gemeinsames Qualitätsverständnis zu erzielen. Dadurch wird die Wahrscheinlichkeit für Fehler in neu entwickeltem Sourcecode reduziert. Pair Programming kann zudem helfen, aufgedeckte Mängel als Problem zu akzeptieren und anschließend eine Verbesserung in den Sourcecode zu integrieren.

21.4 Codereview-Tools

Obwohl es unbestritten ist, dass Codereviews die Qualität deutlich verbessern können, werden sie zum Teil als lästig empfunden – häufig aufgrund der zuvor in Abschnitt 21.2 diskutierten Probleme bei der Durchführung. Zudem war die Unterstützung durch Tools bis vor einigen Jahren kaum existent. Die Situation hat sich aber verbessert. Nun sind entwicklungsbegleitend »leichtgewichtige« Codereviews möglich, die spezielle Anmerkungen zum Sourcecode erlauben, statt aufwendige Meetings durchzuführen.

Durch den Einsatz von Tools werden Terminkonflikte vermieden, die eine Teilnahme an einem Meeting zu einer festgelegten Zeit verhindern könnten. Anmerkungen können in Ruhe formuliert und später genau einer Zeile im Sourcecode zugeordnet werden. Die zugrunde liegende Idee ist nicht prinzipiell neu, sondern mehr das konkrete Vorgehen. Die einfachste Form ist es, spezielle Kommentare in den Sourcecode einzufügen. Bei dieser Vorgehensweise sind die Anmerkungen allerdings Bestandteil des Sourcecodes und müssen abgeglichen werden.

Oft wird eine zusätzliche Abstimmung und Kommunikation benötigt, etwa über E-Mails. Es ist jedoch manchmal schwierig, die passenden Sourcecode-Fragmente von E-Mail und realem Sourcecode abzugleichen, etwa wenn aufgrund von Änderungen unterschiedliche Versionsstände existieren. Durch eine Toolunterstützung vermeidet man eine Vermischung von »normalen«, die Funktionalität dokumentierenden Kommentaren und Codereview-Kommentaren, die aufgedeckte Softwaredefekte markieren und beschreiben. Durch die Trennung wird die Auswertung der Codereview-Kommentare erleichtert.

Im Folgenden möchte ich kurz auf einige Tools eingehen, die die genannten Punkte adressieren und den Codereview-Prozess erleichtern sollen. Mithilfe dieser Tools kann man Codereviews komfortabler als mit den herkömmlichen Ansätzen mit Meetings und E-Mails durchführen.

Das Tool Code Collaborator

Code Collaborator ist ein kommerzielles Tool (auch als Evaluationsversion verfügbar) und kann unter `http://smartbear.com/products/software-`

development/Codereview/ bezogen werden. Weitere Informationen sowie einige interessante Tipps und Tricks zum Codereview finden Sie unter http://smartbear.com/.

Das Tool Crucible

Crucible ist ein weiteres kommerzielles Tool zur Unterstützung bei der Durchführung von Codereviews. Auch hier gibt es eine freie Evaluationsversion. Detaillierte Informationen finden sich unter http://www.atlassian.com/software/crucible/.

Das Tool Jupiter

Abschließend möchte ich Ihnen noch das frei erhältliche Tool namens Jupiter vorstellen. Es ist als Eclipse-Plugin verfügbar. Die Installation gestaltet sich einfach. Es muss lediglich eine JAR-Datei in das Verzeichnis plugins von Eclipse gelegt werden. Diese JAR-Datei kann unter http://code.google.com/p/jupiter-eclipse-plugin/ heruntergeladen werden.

Codereviews können dann arbeitsbegleitend beim Programmieren stattfinden, indem Anmerkungen zu gefundenen Defekten über einen Dialog aufgenommen werden, wie dies Abbildung 21-1 für das in Abschnitt 21.3 vorgestellte Beispiel der Komparatoren zeigt.

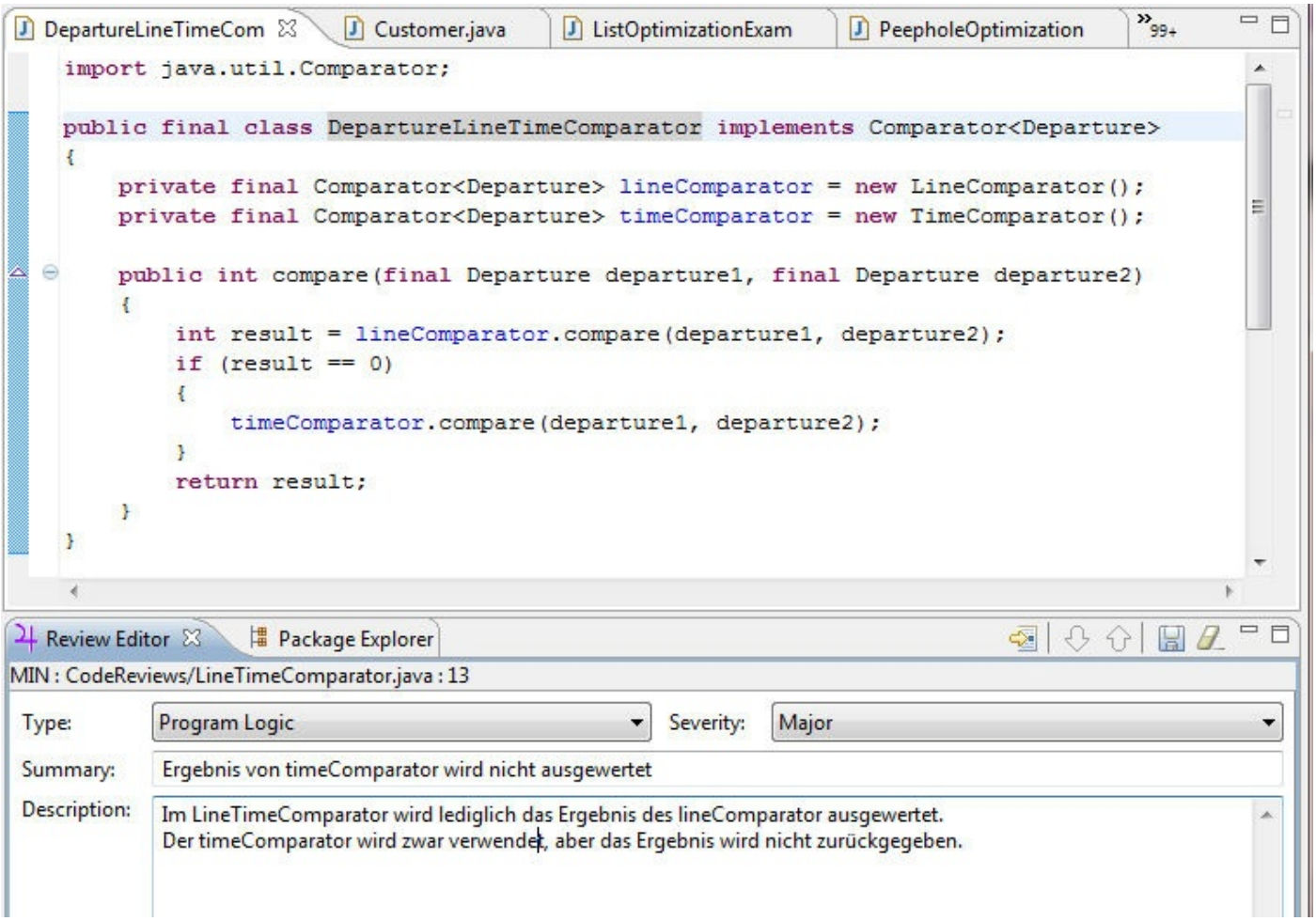


Abbildung 21-1 *Eingabe von Anmerkungen mit dem Jupiter-Plugin*

In einer speziellen Task-Liste werden die gefundenen Defekte dargestellt und man kann schnell zum jeweiligen Defekt im Sourcecode navigieren.

Fazit

Dieses Kapitel hat einen einführenden Überblick in das Themengebiet Codereview gegeben. Ich ermutige Sie ausdrücklich, Codereviews durchzuführen. Beginnen Sie zum Einstieg mit einem »netten« Kollegen als Reviewer und nutzen Sie dabei die im folgenden Abschnitt eingeführte Checkliste als Hilfestellung.

21.5 Codereview-Checkliste

Bei der Durchführung eigener Codereviews kann folgende Checkliste helfen, einige wichtige Aspekte beim Analysieren von Sourcecode im Blick zu behalten. Es bietet sich an, diese Liste zu kopieren und vor einem Codereview an alle Teilnehmer zu verteilen.

1. **Variablen und Namensgebung**

   Sind Namen sinnvoll gewählt und konsistent? Werden Namensrichtlinien beachtet? Vermeide den Gebrauch einer Variablen für verschiedene Zwecke!

2. **Variableninitialisierung**

   Sind alle Variablen möglichst lokal definiert und korrekt initialisiert? Sind Konstanten `final` definiert?

3. **Typsicherheit**

   Vermeide die Verwendung von Raw Types, benutze bevorzugt Generics.

4. **Magic Numbers**

   Sind Konstanten anstelle von nichtssagenden Magic Numbers verwendet? Sind zusammengehörende Konstanten nicht einzeln, sondern besser als `enum` definiert?

5. **Struktur**

   Ist der Sourcecode optisch klar strukturiert und nachvollziehbar?

6. **Kommentare**

   Sind Kommentare verständlich, ausreichend und erforderlich? Veraltete und überflüssige Kommentare sollten entfernt werden!

7. **Redundanzen**

   Ist kein Sourcecode-Fragment doppelt, überflüssig oder unbenutzt?

8. **Korrektheit und Tests**

Arbeitet das Programm bereits augenscheinlich so wie gewünscht? Gibt es Unit Tests? Sind diese sinnvoll und erzielen eine hohe Testabdeckung (z. B. über 80%, mindestens aber 65%)?

9. **Seiteneffekte**

   Existieren Seiteneffekte? Entferne diese wenn möglich, ansonsten dokumentiere sie ausreichend!

10. **Interface-Design**

    Ist das Interface einer Klasse nach außen möglichst klein? Andere Methoden sollten bevorzugt `private` oder `protected` definiert sein.

11. **Fehlerbehandlung**

    Werden Rückgabewerte (korrekt) ausgewertet und Fehlersituationen behandelt? Werden Exceptions, sofern sinnvoll möglich, behandelt?

12. **Casting**

    Werden Casts nur in Ausnahmefällen verwendet? Gibt es stattdessen eine objektorientierte Lösungsmöglichkeit mit gemeinsamer Basisklasse?

13. **Achtung » == « != » equals() «**

    Werden == und `equals()` korrekt eingesetzt?

22 Optimierungen

In diesem Kapitel werden Strategien und Techniken zum Optimieren von Programmen vorgestellt. Unter Optimierung wird meistens verstanden, dass sich die Ausführungsgeschwindigkeit eines Programms verbessert oder sich dessen Speicherbedarf reduziert.

Abschnitt 22.1 gibt einen Überblick über den Themenkomplex der Performance-Analyse. Unter anderem werden verschiedene, die Performance beeinflussende Faktoren sowie Techniken und Tools zur Analyse vorgestellt. Diese helfen dabei, mögliche Ursachen erkennen und kritische Stellen finden zu können. In Abschnitt 22.2 werden wir den Einfluss des Einsatzes unterschiedlicher Datenstrukturen auf die Laufzeit und den Speicherverbrauch diskutieren. Neben den gewählten Algorithmen und Datenstrukturen kann der zur Laufzeit benötigte Speicher die Ausführungsgeschwindigkeit beeinflussen. Ganz besonders macht sich dies bemerkbar, wenn einer JVM wenig Speicher zur Verfügung steht und es dadurch vermehrt zu Garbage Collections kommt. Als eine mögliche Abhilfe wird die Technik Lazy Initialization zur Vermeidung einer speicherund zeitaufwendigen Konstruktion komplexer Objekte in Abschnitt 22.3 vorgestellt.

Negative Auswirkungen auf die Programmlaufzeit können durch unterschiedliche Abhilfemaßnahmen gemildert oder behoben werden. Dabei hängt der Erfolg einer Optimierung entscheidend von der problemangepassten Ebene ab: Bevor wir diese detailliert betrachten, stellt Abschnitt 22.4 konkret an einem Beispiel den Ablauf und verschiedene Techniken beim Optimieren eines Programmteils vor. Dadurch lassen sich die in den nachfolgenden Abschnitten beschriebenen Techniken leichter verstehen und einordnen. Eine Verbesserungen der Ein- und Ausgabeverarbeitung (I/O) wird in Abschnitt 22.5 besprochen. Abschnitt 22.6 diskutiert einige Auswirkungen des zur Verfügung stehenden Speichers und stellt diesbezüglich einige Optimierungstechniken vor. Abschließend betrachtet Abschnitt 22.7 Techniken, die zu einer schnelleren Ausführung beitragen, indem CPU-intensive Berechnungen optimiert werden.

Generell sollten Optimierungen nicht allzu häufig notwendig sein. Wenn allerdings ein Performance-Problem vorliegt, so ist dessen Beseitigung elementar wichtig, beispielsweise um eine flüssige Abarbeitung in einem GUI oder einen guten Datendurchsatz in einer komplexen Berechnung zu erzielen. Manche Performance-Probleme lassen sich während der Entwicklung jedoch nicht oder nur schwierig erkennen, da sie sich erst beim Kunden durch ein viel höheres Datenvolumen oder andere Hardwarevoraussetzungen als bei entwicklungsbegleitenden Tests manifestieren. Wichtig ist daher auch, die Zielumgebung, vor allem deren Beschränkungen und Datenraten, zu bedenken und gegebenenfalls zu simulieren.

22.1 Grundlagen

Optimierungen können auf verschiedenen Ebenen vorgenommen werden, die ich später im Detail vorstelle. Wichtig ist in diesem Zusammenhang die folgende Faustregel: ***Je höher die Abstraktionsebene ist, auf der wir Änderungen vornehmen, desto größer sind die erzielbaren Performance-Gewinne***. Die Wahl geeigneter Kommunikationsformen mit externen Systemen sowie der Einsatz passender Algorithmen und Datenstrukturen für ein zu lösendes Problem bewirken häufig viel mehr als feingranulare Änderungen auf der Ebene des Sourcecodes. Letztere machen ein Programm sogar teilweise lediglich schlechter lesbar und damit auch schlechter wartbar. Sinnvoll ist es daher, zunächst immer nahe an der zu realisierenden Problemstellung und möglichst verständlich zu programmieren, bevor man bei Bedarf mit Optimierungen beginnt. Befolgt man diese Vorgehensweise nicht, werden dadurch in der Regel das Design und der Sourcecode komplizierter. Dies erschwert spätere Refactorings. Außerdem leiden Testbarkeit und Verständlichkeit. ***Deshalb sollte man immer im Hinterkopf behalten, dass mögliche Einbußen an Lesbarkeit und Erweiterbarkeit (für die Zukunft) schwerwiegender sein können als kurzfristig erzielbare Performance-Gewinne.*** Dieses Kapitel stellt daher bevorzugt solche Techniken vor, die zwar gute Performance-Gewinne ermöglichen, jedoch die Lesbarkeit und Verständlichkeit erhalten oder nur unwesentlich verringern. Abstriche sollte man nur dann in Kauf nehmen, wenn tatsächlich schwerwiegende Performance-Probleme durch Messungen nachgewiesen wurden.

Donald E. Knuth schrieb schon 1974 (meistens reduziert auf das zur Hervorhebung in Fettschrift angegebene Zitat): »Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. **We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil**« [56]. Frei übersetzt: »Programmierer verschwenden enorme Zeit damit, über die Laufzeit unkritischer Bestandteile ihrer Programme nachzudenken oder sich darüber zu sorgen. Diese Versuche der Performance-Steigerung wirken sich jedoch sehr negativ auf das Debugging und die Wartbarkeit aus. **In 97% aller Fälle sollten wir alle kleineren Verbesserungen nicht betrachten: Vorzeitige Optimierung ist die Quelle allen Übels**.«

Info: Performance-Mythen über Java

Die meisten Performance-Probleme sind heutzutage nicht – wie zum Teil fälschlicherweise behauptet – durch den Einsatz von Java verursacht, sondern stellen Programmierprobleme dar, die Programme auch in anderen Programmiersprachen, etwa C++ oder C#, langsam machen.

Tatsächlich bietet Java sogar den Vorteil, dass zusätzlich zu Optimierungen während der Kompilierung in Bytecode einige weitere Optimierungen zur Laufzeit

durch den Hotspot-Optimierer durchgeführt werden. Abschnitt 22.1.3 geht auf diese Optimierungen genauer ein.

Liegt ein Performance-Problem vor, so sollte man zunächst umfangreiche Analysen und Messungen vornehmen, um die verursachenden Stellen zu identifizieren. Führt man stattdessen unüberlegt an diversen Stellen im Programm Änderungen durch, die lediglich auf Vermutungen beruhen, kann dies zu weiteren Problemen führen. Der Grund dafür ist so einleuchtend wie einfach: Wenn wir mit unseren Vermutungen falsch liegen, so vergeuden wir wertvolle Zeit damit, die falschen Programmteile zu ändern und zu optimieren. Wahrscheinlich machen wir die Programmteile zusätzlich noch unleserlich und bauen vielleicht sogar ungewollt Fehler ein, ohne einen Vorteil bezüglich der Performance zu erreichen. ***Optimierungen sind daher mit Vorsicht zu genießen, und keinesfalls sollte man nur basierend auf Vermutungen optimieren***. Jon Bentley betont: »Das wichtigste Prinzip beim Code-Tuning ist: Tun Sie es selten« [6].

Tipp: Einflüsse von Nebenläufigkeit und Parallelverarbeitung

Durch die Integration mehrerer Prozessorkerne in CPUs kommt heutzutage eine weitere Dimension ins Spiel: Das Aufteilen von Programmen auf mehrere Threads, ohne dass sich diese gegenseitig zu sehr ausbremsen. Dies kann schnell geschehen, wenn Programme intensiv von Synchronisation Gebrauch machen. Wenn (jedoch) massive Parallelität und Nebenläufigkeit benötigt werden, kann dieser Effekt durch den Einsatz der Concurrent Collections (vgl. Abschnitt 7.6) gemildert werden. Zudem kann mit JDK 7 der Einsatz von Fork-Join-Frameworks bei der Parallelisierung von Aufgaben helfen (vgl. Abschnitt 7.6.3). Auch mit Java 8 sind diverse Erweiterungen zur Parallelverarbeitung in das JDK integriert worden. Diese werden in Kapitel 15 vorgestellt.

Die tatsächlichen Auswirkungen dieser Techniken lassen sich nur schwierig formalisieren und übersteigen den Rahmen dieses Buchs. Ich verweise daher für Details auf die Bücher »Concurrent Programming in Java« von Doug Lea [59] und »Java Concurrency in Practice« von Brian Goetz et al. [31].

22.1.1 Optimierungsebenen und Einflussfaktoren

Wie bereits eingangs erwähnt, existieren verschiedene Einflussfaktoren, die sich auf die Performance auswirken können.

Hängt die Ausführungsgeschwindigkeit maßgeblich von den zu berechnenden Aufgaben ab, dann bezeichnet man ein Programm als ***CPU-bound***. Mit anderen Worten bedeutet dies: Verwendet man einen schnelleren Prozessor, so steigt die Ausführungsgeschwindigkeit proportional zur schnelleren CPU. Für ***Memory-bound***-Programme gilt, dass Zugriffe auf den Hauptspeicher den kritischen Pfad darstellen und starke Auswirkungen auf die Ausführungsgeschwindigkeit haben. Zwei Faktoren

spielen dabei eine Rolle: Einerseits wirkt sich ein zu klein dimensionierter Speicher negativ auf die Ausführungsgeschwindigkeit aus. Andererseits können auch häufige, eventuell vermeidbare Speicherzugriffe die Performance verschlechtern. Im einfachsten Fall kann man einer JVM mehr Speicher zur Verfügung stellen, um die Ausführungsgeschwindigkeit zu erhöhen. Unter ***I/O-bound*** versteht man, dass die Ausführungsgeschwindigkeit eines Programms maßgeblich durch Zugriffe auf externe Systeme (Festplatte, Datenbank usw.) bestimmt wird. Manchmal können enorme Geschwindigkeitsgewinne erreicht werden, indem man Daten gepuffert verarbeitet. Die zuvor vorgestellte Klassifizierung ergibt sich aus den unterschiedlichen Zugriffszeiten auf die entsprechenden Komponenten: Der Zugriff auf Register eines Prozessors ist deutlich schneller als der Zugriff auf nachgelagerte Prozessor-Caches. Beide sind jedoch sehr schnell. Hauptspeicherzugriffe sind im Gegensatz dazu bereits etwas langsamer. Nochmals um Größenordnungen langsamer sind Datei- und Netzwerkzugriffe.

War früher noch der Prozessor, später dann der Hauptspeicherzugriff ein Flaschenhals bezüglich der Performance, so gilt dies mittlerweile nur noch selten, vielmehr sind Performance-Probleme in der Regel eine Kombination aus Memory-bound und I/O-bound. Das liegt daran, dass heutzutage Programme oftmals mit anderen Komponenten, etwa einer Datenbank, oder mit anderen Programmen über Remote Calls interagieren. Ziel von Optimierungen muss es daher sein, zunächst die Kommunikation mit externen Systemen genauer zu betrachten, weil dort in der Regel das größte Optimierungspotenzial steckt. Nachdem auf dieser Ebene alle Möglichkeiten ausgeschöpft sind, kann man versuchen, Speicherzugriffe sowie eingesetzte Algorithmen und Datenstrukturen zu optimieren. Abgesehen von der Optimierung der genutzten Algorithmen sollte man von CPU-bound-Optimierungen auf der Ebene einzelner Anweisungen möglichst Abstand nehmen, weil diese meistens zwei Nachteile mit sich bringen: Zum einen muss man schon Experte sein und genau wissen, was man tut, um überhaupt Vorteile erzielen zu können und das Ganze besser zu machen als der Hotspot-Optimierer der JVM. Dieser optimiert unter anderem Speicherzugriffe und ordnet Anweisungen um oder fasst diese zusammen. Zum anderen ist derart per Hand optimierter Sourcecode oftmals unleserlich – nach einiger Zeit auch für einen selbst!

22.1.2 Optimierungstechniken

Obwohl die genannten Ebenen sehr unterschiedlich sind, basieren die Strategien und Techniken zur Optimierung immer wieder auf ähnlichen Ideen:

- **Wahl passender Strategien** – Die Wahl passender Strategien zum Zugriff oder zur Berechnung kann massive Auswirkungen auf die Performance haben. Erfolgt ein Zugriff auf Dateien beispielsweise Byte für Byte, ist das der Aufgabe nicht angemessen, da es nicht an die Funktionsweise der Hardware angepasst ist. Festplatten sind darauf optimiert, größere Datenmengen an einem Stück zu

verarbeiten. Eine Pufferung sowie ein blockweises Lesen und Schreiben kann gegenüber mehrfachen Einzelzugriffen daher eine enorme Geschwindigkeitssteigerung bewirken, ohne jedoch konzeptionelle Änderungen zu bedingen. Ähnliche positive Effekte beobachtet man beim Zugriff auf Datenstrukturen: Ein indizierter Zugriff auf ein Array oder eine `ArrayList<E>` ist bei sehr umfangreichen Datenmengen deutlich schneller als bei einer `LinkedList<E>`. Der Einfluss verschiedener Datenstrukturen wird ausführlich in Abschnitt 22.2 besprochen. Ein weiteres Beispiel ist das Zusammenfassen von Aktionen: Eine zu erledigende Arbeit wird nicht in vielen kleinen, wiederholten Schritten, sondern als eine größere Aktion ausgeführt, etwa indem man mehrere Methodenaufrufe zu einem zusammenfasst.

- **Caching und Pooling** – Das Zwischenspeichern von Ergebnissen und das Wiederverwenden bereits erzeugter Objekte oder errechneter Ergebnisse fällt in die Kategorie von Caching und Pooling. Wie bereits erwähnt, gibt es eine Hierarchie bezüglich der Zugriffszeiten auf verschiedene Datenspeicher oder Medien. Caches können auf verschiedenen Ebenen eingesetzt werden. Ziel dabei ist es, Zugriffe auf das jeweils teurere Medium zu reduzieren, indem ein möglichst passender Satz an benötigten Daten im Cache gehalten wird. Dabei macht man sich die sogenannte ***Lokalität*** zunutze. Darunter versteht man, dass man gewisse ***Muster beim Datenzugriff*** findet. Oftmals werden Daten während einer Verarbeitung innerhalb kürzerer Zeit mehrfach gelesen und zudem auch benachbarte Daten (z. B. Daten auf der Festplatte, Artikel einer Einkaufsliste oder ähnliche Top-Seller). Aufgrund dieser Beobachtungen lohnen sich Caches häufig in tieferen Schichten, beispielsweise bei Zugriffen auf Datenbanken oder das Dateisystem. Innerhalb einer Applikation profitiert man von Caches jedoch nur, wenn die Zugriffszeit auf Daten einen signifikanten Einfluss auf die Performance besitzt.
- **Vermeidung unnötiger Aktionen** – Je weniger unnütze Arbeit verrichtet werden muss, desto besser. Dies gilt beispielsweise für Zugriffe auf das Dateisystem, etwa bei Log-Ausgaben. Weiterhin versteht man darunter Strategien zur Vermeidung der Konstruktion optionaler Programmteile oder im Speziellen von Teilkomponenten schwergewichtiger Objekte. Eine sehr nützliche Technik ist die Lazy Initialization, die in Abschnitt 22.3 genauer betrachtet wird.

Tabelle 22-1 zeigt weitere Beispiele für die zuvor genannten Techniken auf den drei Optimierungsebenen.

Tabelle 22-1 *Optimierungstechniken und Beispiele*

Technik	Ebene		
	I/O	**Memory**	**CPU**
Wahl passender Strategien	Gepufferte Zugriffe, Anpassungen der Serialisierung	Algorithmen und Datenstrukturen	Algorithmen und Datenstrukturen, Peephole-Optimierungen
Caching und Pooling	Daten-Caches, Data Transfer Objects	Daten-Caches, Objekt-Pools	Registerzugriffe, Vorausberechnung und Zwischenspeicherung von Werten
Vermeidung unnötiger Aktionen	Log-Level-Prüfung, Lazy Initialization	Lazy Initialization	Vermeidung von Auto-Boxing/-Unboxing

Die zuvor erwähnten Beispiele machen deutlich, dass die Abgrenzung der Techniken nicht immer messerscharf ist: Beispielsweise dient ein Cache bzw. ein Objekt-Pool dazu, unnötige Aktionen (Speicherzugriffe) zu vermeiden, und ist damit thematisch nahe der Technik »Vermeidung unnötiger Aktionen« angesiedelt. Zudem erkennen wir, dass die Technik Lazy Initialization sowie die Wahl passender Algorithmen und Datenstrukturen[1] auf unterschiedlichen Ebenen angewendet werden können.

22.1.3 CPU-bound-Optimierungsebenen am Beispiel

Performance-Optimierung wird von vielen häufig immer noch als eine Verbesserung auf der Ebene einfacher Anweisungen im Sourcecode und maximal des eingesetzten Algorithmus betrachtet. Zuvor habe ich bereits angedeutet, dass diese Art der Optimierung heutzutage nur in seltenen Fällen eine entscheidende Verbesserung bezüglich der Ausführungsgeschwindigkeit eines Programms bewirken wird. Um dies etwas zu präzisieren und ein Gefühl für mögliche Umsetzungen und Auswirkungen zu vermitteln, möchte ich daher im Folgenden auf einige Optimierungsmöglichkeiten auf den Ebenen Design, Sourcecode und Bytecode bzw. Kompilierung eingehen.

- **Design** – Auf der Ebene des Designs gilt es, die eingesetzten Algorithmen und Datenstrukturen optimal auf die Anforderungen anzupassen. Ein sauber strukturiertes Design, das bevorzugt mit Interfaces statt konkreten Klassen arbeitet, kann dabei helfen, spätere Korrekturen leichter vornehmen zu können, als dies bei direkten Abhängigkeiten von konkreten Klassen der Fall wäre.

Manchmal reicht es bereits, einen Teil eines Algorithmus in eine Methode auszulagern und dadurch austauschbar zu machen (vgl. Abschnitt 18.3.4 zum STRATEGIE-Muster). Betrachten wir dies für folgendes Interface zur Summenberechnung und eine naive Realisierung der dort deklarierten `calcSumOfN(int)` -Methode, die die Summation der Zahlen von 0 bis n mit einer Schleife löst:

```
public interface SumCalculator
{
    public long calcSumOfN(final int n);
}

public class NaiveSum implements SumCalculator
{
    public long calcSumOfN(final int n)
    {
        long sum = 0;
        for ( int i = 0; i <= n; i++)
        {
            sum += i;
        }
        return sum;
    }
}
```

Weil hier gegen das Interface `SumCalculator` programmiert wird, kann die derzeit genutzte Implementierung bei Performance-Problemen im Nachhinein ausgetauscht werden, ohne Änderungen in aufrufenden Klienten erforderlich zu machen. Man kann beispielsweise eine Klasse `OptimizedSum` wie folgt implementieren:

```
public class OptimizedSum implements SumCalculator
{
    public long calcSumOfN (final int n)
    {
        return n * (n + 1) / 2;
    }
}
```

Bei dieser Realisierung der Methode `calcSumOfN(int)` nutzt man, dass folgende mathematische Gleichung gilt:

$$\sum_{i=0}^{n} i = \frac{n * (n+1)}{2}$$

Ähnliche Ideen haben Steve Wilson und Jeff Kesselman in ihrem Buch »Java Platform Performance« [91] entwickelt, allerdings ohne Bezug zum STRATEGIE-Muster, dafür aber mit einer trickreicheren Summierungsmethode.

- **Sourcecode** – In seltenen Fällen können Optimierungen auf Sourcecode-Ebene durch Nutzung anderer Anweisungen die Ausführungsgeschwindigkeit wirklich erhöhen. Für viele Konstrukte gilt dies jedoch nicht oder nur extrem eingeschränkt. Betrachten wir dies für die Optimierung, eine Multiplikation durch einen Bit-Shift zu realisieren. Immer wieder mal liest man, dass Bit-Shifts für die Multiplikation deutlich performanter sein sollen. Ein Bit-Shift um eine

Position nach links entspricht einer Multiplikation mit dem Faktor zwei. Im folgenden Beispiel wird eine Multiplikation mit der Zahl sechs durch zwei Bit-Shifts (einmal um 1 Bit und einmal um 2 Bit gefolgt von einer Addition) realisiert:

```
final long ONE_BILLION = 1_000_000_000L;
for (long i = 0; i < ONE_BILLION; i++)
{
   value = (i << 2) + (i << 1); // Multiplikation durch Bit-Shifts
}
```

Tatsächlich benötigt die derart optimierte Variante bei 1 Milliarde (!) Durchläufen auf meinem Notebook – Quad-Core Intel i7 mit 2,6 GHz mit JDK 7, identisch auch für alle folgenden Performance-Messungen – lediglich ca. 900 Millisekunden und ist damit genauso schnell wie folgende lesbare normale Multiplikation:

```
for (long i = 0; i < ONE_BILLION; i++)
{
    value = i * 6;
}
```

Gerade in älterer Literatur werden Optimierungstechniken auf Sourcecode-Ebene in ihrer Wirksamkeit auf die Gesamtperformance stark überbewertet. Dies hat sich leider bei einigen Entwicklern als Mythos im Gedächtnis verankert. Wie die ermittelten Zahlen zeigen, muss man diese Art der Optimierung sehr kritisch betrachten: Erst bei extrem Performance-kritischen Applikationen und nach Ausschöpfen aller anderen Möglichkeiten kann in seltenen Fällen ein Performance-Gewinn erzielt werden. Ansonsten gilt: ***Vorsicht vor solchen Optimierungen!***

- **Bytecode bzw. Kompilierung** – Bei der Kompilierung des Java-Sourcecodes in korrespondierenden Bytecode werden ***vom Compiler automatisch*** verschiedene kleinere Optimierungen ausgeführt. Zur Laufzeit erfolgt eine weitere Optimierung durch den sogenannten Hotspot-Optimierer (vgl. folgenden Praxistipp »Arbeitsweise und Einfluss des Hotspot-Compilers«), der unter anderem die in der folgenden Aufzählung beschriebenen Techniken Inlining, Loop-Unrolling und Peephole-Optimierung vornimmt. Diese Techniken tragen dazu bei, unnötige Berechnungen zu vermeiden und Programmanweisungen gut auf die ausführende Hardware abzustimmen, beispielsweise durch die Nutzung von Caches und Registern.
  - **Inlining** – Kurze Methoden können vom Compiler direkt in den Methodenrumpf des Aufrufers eingefügt werden, statt angesprungen zu werden. Die Methode `processValuesInlined(int, int)` zeigt dies für die Integration des Methodenrumpfs der Methode `multiply(int, int)`:

    ```
    private static int processValues(final int x, final int y)
    {
        return multiply(x, y);
    }
    ```

```
private static int multiply(final int x, final int y)
{
    return x * y;
}
```

Mit Inlining wird daraus Bytecode, der folgendem Sourcecode entspricht:

```
// Integration der Methode multiply()
private static int processValuesInlined(final int x, final int y)
{
    return x * y;
}
```

- **Loop-Unrolling** – Kurze Schleifen werden ähnlich zur Inlining-Technik »ausgerollt«, d. h., die Anweisungen des Schleifenrumpfs werden mehrfach hintereinander kopiert. Betrachten wir folgende Schleife:

```
private static int conventionalLoop()
{
    int x = 0;
    for (int i = 1; i < 4; i++)
    {
        x += i * i;
    }
    return x;
}
```

Durch den Compiler optimiert wird daraus eine Folge von Anweisungen, die als Terme nur noch Konstanten enthalten. Zur Verdeutlichung der Technik sind hier die Berechnungen noch einzeln aufgeführt. In der nachfolgend beschriebenen Peephole-Optimierung werden diese weiter zusammengefasst.

```
private static int unrolledLoop()
{
    int x = 0;
    x += 1 * 1;
    x += 2 * 2;
    x += 3 * 3;
    return x;
}
```

- **Peephole-Optimierung** – Es werden beispielsweise Speicher- und Registerzugriffe optimiert und Berechnungen zur Kompilierzeit durchgeführt. Im nachfolgenden Beispiel wird ein Variable `x` mit dem Wert `0` initialisiert und danach eine Berechnung ausgeführt:

```
private static void assigment()
{
    int x = 0;
    x = 4 * 25;
    // …
```

Zur Kompilierzeit wird das Ergebnis der Multiplikation der Zahlenliterale berechnet und dieser Wert, hier `100`, direkt an die Variable zugewiesen:

```
private static void assigmentOptimized()
{
    int x = 100;  // Peephole: Zuweisung und Multiplikation
```

```
// ...
```

Tipp: Arbeitsweise und Einfluss des Hotspot-Compilers

Beim Start eines Java-Programms wird sämtlicher Bytecode interpretiert, d. h., Anweisung für Anweisung wird von einem Interpreter ausgeführt. Besonders häufig benutzte Programmteile, sogenannte »Hotspots«, werden von einem in die JVM integrierten Optimierer erkannt und kompiliert. Dieser sogenannte Hotspot-Compiler erzeugt den Maschinencode erst zur Laufzeit parallel zur Abarbeitung des eigentlichen Programms. Man spricht daher von einem ***Just-in-Time-Compiler*** (JIT). Dieser Übersetzungsvorgang wirkt sich geringfügig verzögernd auf die eigentliche Programmlaufzeit aus. Die Auswirkungen sind jedoch minimal, da die Kompilierung nur für einige ausgewählte Methoden ausgeführt wird, die zuvor erkannten Hotspots.

Die JVM kann in zwei Optimierungsmodi laufen. Im Server-Modus (`-server`) werden die in der obigen Aufzählung genannten Optimierungen besonders stark durchgeführt. Die Kompilierung ist dadurch etwas langsamer, aber besser optimiert als im Client-Modus (`-client`), der weniger optimiert, um schnell zu kompilieren und somit weniger Einfluss auf die Anwendungslaufzeit zu haben.

Diskussion

Je höher die Abstraktionsebene ist, auf der wir Änderungen vornehmen, desto größer sind in der Regel die zu erzielenden Performance-Gewinne. Daher bietet es sich an, zunächst das Design zu verbessern: Manchmal gibt es für ein zu lösendes Problem besser passende Algorithmen oder Datenstrukturen als die momentan gewählten. Beginnt man mit Optimierungen zunächst auf Designebene, lassen sich als Folge nahezu immer die zuvor vorgestellten Optimierungen auf Sourcecode-Ebene vermeiden. Nur wenn auf der Ebene des Designs tatsächlich keine weiteren Verbesserungen mehr zu erzielen sind und weiterhin Optimierungsbedarf besteht, kann man Optimierungen auf Sourcecode-Ebene in Betracht ziehen. Somit wird nur in wirklich notwendigen Fällen die Lesbarkeit des Sourcecodes durch Optimierungen beeinträchtigt. Nichtsdestotrotz sieht man leider gerade auf der Ebene des Sourcecodes immer wieder vorschnell durchgeführte Optimierungen, die Lesbarkeit gegen vermutete Performance-Verbesserungen eintauschen, indem sie die Transformationen des Compilers auf Bytecode-Ebene in den Sourcecode aufnehmen. Dies sollte man vermeiden, da die Hotspot-Optimierung sehr effizient erfolgt und zudem automatisch geschieht – ohne Einfluss auf die Lesbarkeit.

Vor Beginn von Optimierungen ist es wichtig, dass der Programmteil korrekt funktioniert und ausreichend getestet ist. Eine Absicherung durch Unit Tests (vgl. Kapitel 20) bietet sich an, um nachfolgende Änderungen auf Korrektheit und Kompatibilität zur bisherigen Umsetzung prüfen zu können. Außerdem werden durch ein solches Vorgehen die Schritte Optimierung und Programmentwicklung sauber voneinander getrennt. Änderungen erfolgen dann entweder zur Realisierung

der Funktionalität oder zur Optimierung. Man vermeidet so, beide Tätigkeiten miteinander zu vermischen und am Ende weder einen korrekten Algorithmus noch eine sinnvolle Optimierung realisiert zu haben.

22.1.4 Messungen – Erkennen kritischer Bereiche

Es ist es nahezu unmöglich, Performance-kritische Bereiche lediglich durch visuelle Sourcecode-Analyse zu erkennen. Dies gilt umso mehr, je komplizierter und umfangreicher Applikationen werden, etwa bei verteilten Applikationen oder beim Einsatz von Applikationsservern.

Für kleinere Applikationen kann man im einfachsten Fall über sogenannte Stoppuhr- bzw. Vorher-nachher-Messungen bereits Aussagen treffen. Dies setzt allerdings voraus, dass man bereits eine ungefähre Ahnung hat, welche Abschnitte des Sourcecodes Performance-Probleme verursachen könnten. In der Regel gilt: ***Alles, was nicht innerhalb von Schleifen oder anderweitig wiederholt ausgeführt wird, ist höchstwahrscheinlich für die Performance unkritisch.***

Warum es elementar wichtig ist, nicht auf Verdacht, sondern lediglich durch Messungen begründet zu optimieren, wird durch die im folgenden Abschnitt vorgestellte »80-zu-20-Regel« motiviert.

Die »80-zu-20-Regel«

Die sogenannte »80-zu-20-Regel« (auch Pareto-Prinzip genannt) lässt sich interessanterweise auf viele Situationen anwenden und wird in den unterschiedlichsten Bereichen der Softwareentwicklung verwendet. Eine Auslegung der Regel besagt, dass man die ersten 80% des Sourcecodes in 20% der Projektzeit schreibt. Für die restlichen 20% des Sourcecodes benötigt man allerdings die verbleibenden 80% der Zeit[2]. ***Konkret bedeutet dies, dass schnell prototypische Umsetzungen mit einem relativ großen Funktionsumfang entstehen können, dass es aber bis zu einem fertigen Produkt noch ein sehr langer Weg ist.***

An dieser Stelle ist jedoch ein weiteres Auftreten der Regel noch interessanter: Für die Ausführungszeit gilt häufig, dass 80% des Sourcecodes nur 20% der Rechenleistung fordern und dass 20% für 80% Last verantwortlich sind. Im Normalfall kann man also davon ausgehen, dass nur einige wenige kritische Performance-Bremsen im Programm existieren. Die meisten davon lassen sich entsprechend der erstgenannten Auslegung der 80-zu-20-Regel schnell beheben: Man erzielt 80% Verbesserung mit 20% Zeiteinsatz. Glücklicherweise sind aber diese ersten 80% meistens vollkommen ausreichend, um eine ansprechende Performance zu erzielen. Eine umfassendere Optimierung, um das letzte Quäntchen Performance (die letzten 20%) herauszukitzeln, ist dann in der Regel aber mühselig und unangemessen zeitaufwendig. Die wesentliche Erkenntnis ist jedoch: ***Optimiert man, ohne vorher zu messen und kritische Stellen erkannt zu haben, so liegt die***

Wahrscheinlichkeit bei ca. 80%, dass man eine Stelle bearbeitet, die keinen signifikanten Einfluss auf die Performance besitzt. Man kann also nahezu endlos an unwichtigen Stellen feilen – ohne jedoch nennenswerte Verbesserungen zu erzielen.

Messungen des Zeitverhaltens

Für Performance-Abschätzungen werden in den folgenden Abschnitten immer wieder Stoppuhr-Messungen durchgeführt. Man kann dazu die Methode `System.currentTimeMillis()` nutzen. Für exaktere Messungen und Analysen ist diese Methode aufgrund ihrer Ungenauigkeit[3] jedoch nur bedingt geeignet. Wird eine größere Genauigkeit benötigt, so sollte für diese Fälle besser die Methode `System.nanoTime()` genutzt werden, die immer den genauesten verfügbaren Zeitgeber des Betriebssystems wählt.

Zur Unterstützung unserer Zeitmessungen entwickeln wir eine Utility-Klasse `PerformanceUtils`, die die Methode `System.nanoTime()` einsetzt. Eine innere Klasse `TimingEntry` speichert einen Namen und gemessene Zeiten. Die Zeiten werden über die Hilfsmethoden `startMeasure(String)` und `stopMeasure(String)` der äußeren Klasse ermittelt.

Das folgende Listing zeigt die Realisierung der Utility-Klasse `PerformanceUtils`, die die genannten Zeitmessungen ermöglicht:

```
public final class PerformanceUtils
{
    private static final Logger log =
Logger.getLogger(PerformanceUtils.class);

    private static final Map<String, TimingEntry> timingMap = new HashMap<>();

    public static void startMeasure(final String name)
    {
        timingMap.put(name, new TimingEntry(System.nanoTime()));
    }

    public static void stopMeasure(final String name)
    {
        final TimingEntry timingEntry = getTimingEntry(name);
        timingEntry.setStopTime(System.nanoTime());
    }

    public static void printTimingResult(final String name)
    {
        final TimingEntry timingEntry = getTimingEntry(name);
        printTimingResult(name, timingEntry.startTime, timingEntry.stopTime);
    }

    public static void printTimingResult(final String info,
                                         final long begin, final long end)
    {
        log.info(info + " took " + TimeUnit.NANOSECONDS.toMillis(end - begin)
+
                 " ms");
```

```
    }

    public static void printTimingResultWithAverage(final String name,
                                                    final long count)
    {
        printTimingResult(name);

        final TimingEntry timingEntry = getTimingEntry(name);
        final double avg = TimeUnit.NANOSECONDS.toMillis(timingEntry.stopTime
-
                           timingEntry.startTime) / (double) count;
        log.info(String.format(info + " avg %f ms", avg));
    }

    private static TimingEntry getTimingEntry(final String name)
    {
        final TimingEntry timingEntry = timingMap.get(name);
        if (timingEntry==null)
            throw new IllegalArgumentException("No data for '" + name + "'");

        return timingEntry;
    }

    private static final class TimingEntry
    {
        private final long    startTime;
        private long          stopTime;

        public TimingEntry (final long startTime)
        {
            this.startTime = startTime;
        }

        public void setStopTime(final long stopTime)
        {
            this.stopTime = stopTime;
        }

        @Override
        public int hashCode()
        {
           return Objects.hash(this.startTime, this.startTime);
        }

        @Override
        public boolean equals(final Object obj)
        {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;

            final TimingEntry other = (TimingEntry) obj;
            return (startTime == other.startTime && stopTime ==
other.stopTime);
        }
    }
}
```

Bei der Ausgabe der Messergebnisse erfolgt eine Umrechnung auf eine Genauigkeit in Millisekunden, da dies in der Regel für eine Einschätzung der Auswirkungen verschiedener Designentscheidungen ausreichend ist.

Mit einfachen Vorher-nachher-Messungen lassen sich nur relativ triviale Abläufe untersuchen. Müssen komplexere Aktionen analysiert werden oder ist das Verhalten über mehrere Durchläufe oder Stunden interessant, so helfen Profiling-Tools, kritische Stellen aufzuspüren. Die Stoppuhr-Messung hat trotzdem ihre Daseinsberechtigung, weil sie, im Gegensatz zu einer Messung mit einem Profiler, keinen Einfluss auf die Ausführungsdauer des gemessenen Programmabschnitts hat.

Messungen des Speicherbedarfs

Die obigen Aussagen zur Ausführungszeit gelten analog für den von einer Applikation verbrauchten Speicher, der ohne Profiling-Tools nur rudimentär mit den Methoden `freeMemory()` und `totalMemory()` der Klasse `java.lang.Runtime` ermittelt werden kann. Allerdings ist durch die automatisch, zu beliebigen Zeitpunkten stattfindende Speicherbereinigung durch Garbage Collections zur genaueren Messung ein vorheriger (gegebenenfalls mehrfacher) Aufruf von `System.gc()` ratsam. Wie bereits in Abschnitt 8.5 erwähnt, erfolgt dadurch nicht zwangsläufig eine Garbage Collection. In vielen Fällen wird diese jedoch durchgeführt, und es kommt zu einer verbesserten Genauigkeit der Speichermessung.

Der einer Applikation zur Verfügung stehende Hauptspeicher kann großen Einfluss auf die Performance haben. Um eine erste Einschätzung des Speicherbedarfs zu bekommen und einige Effekte zu analysieren, kann man sich vergleichbar zur obigen Klasse `PerformanceUtils` eine Klasse `MemoryInfo` schreiben, die erste, aber rudimentäre Auswertungen erlaubt. Gerade bei der Untersuchung des Speicherverbrauchs besitzen Profiling-Tools besondere Vorzüge, da diese die Belegung des Speichers analysieren können und somit Angaben zu den genutzten Typen und der Anzahl ihrer Instanzen ermöglichen. Dadurch lässt sich z. B. die Anzahl von `Integer`-Instanzen ermitteln.

Profiling-Tools

Um Performance-Probleme in eigenen Programmen aufdecken zu können, helfen Profiling-Tools. Teilweise sind die im JDK bereitgestellten Tools bereits ausreichend. Sowohl `JConsole` als auch `VisualVM` bieten ein GUI, das den Einstieg in die Performance-Optimierung erleichtert. `VisualVM` wurde mit JDK 6 Update 7 eingeführt und bündelt einige bisher einzeln vorhandene Performance-Auswertungsprogramme, wie `jmap` (Speicherübersicht), `jps` (Übersicht über alle laufenden JVMs eines Rechners), `jstack` (Info über alle Threads eines Programms mit Stacktrace), `jstat` (Anzeige von Performance-Informationen), zu einer integrierten Performance-Messsuite, die zudem über verschiedene Plugins erweitert werden kann. Mit `VisualVM` lässt sich das Verhalten von Anwendungen detailliert

auswerten und grafisch visualisieren. In Abbildung 22-1 ist eine Profiling-Session gezeigt.

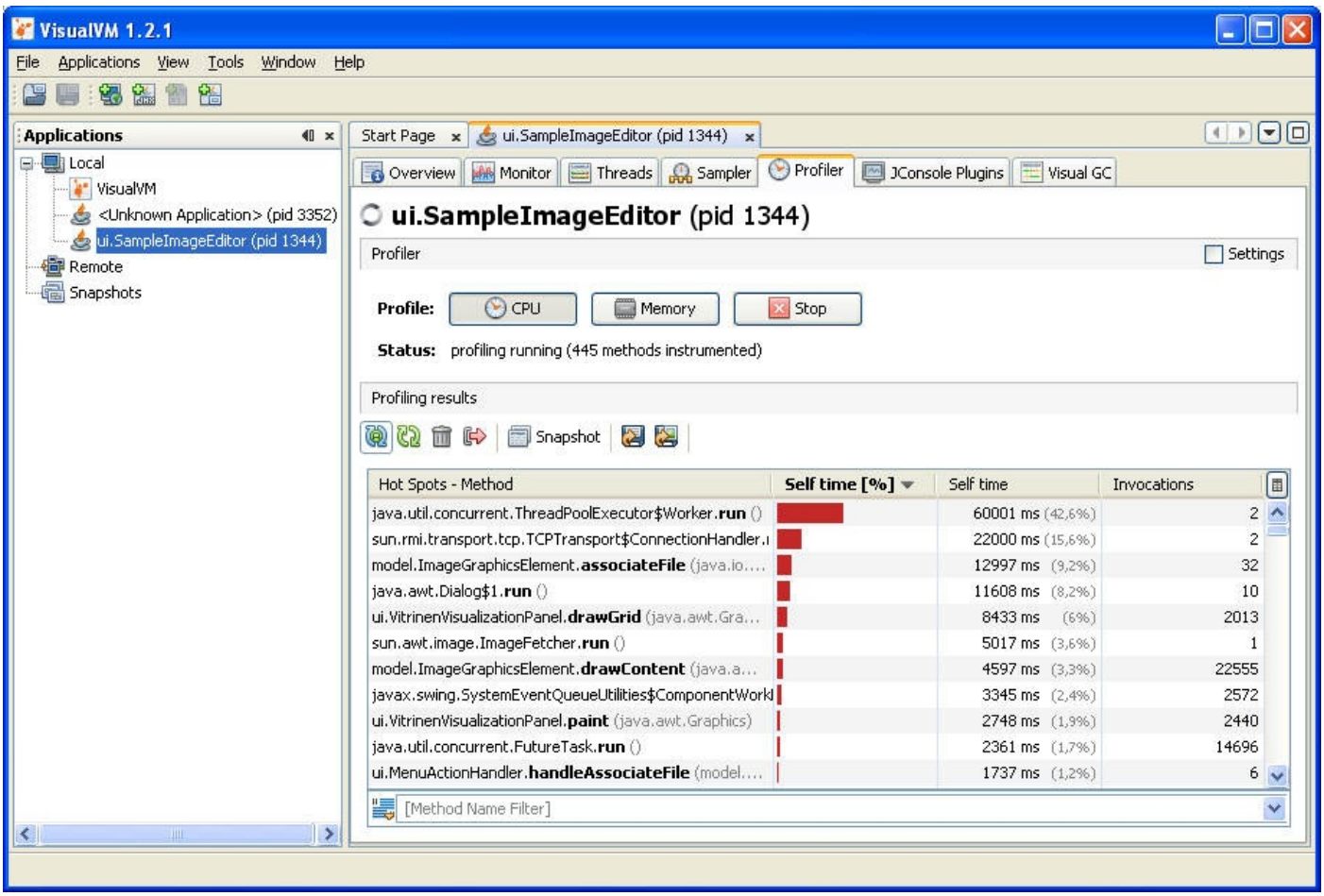


Abbildung 22-1 *VisualVM beim CPU-Profiling*

Das ehemalige Stand-alone-Tool `VisualGC` existiert als Plugin für `VisualVM`. Damit lassen sich Garbage-Collection-Vorgänge gut beobachten und auswerten. Dies ist in Abbildung 22-2 exemplarisch gezeigt. Man sieht auf einen Blick den Füllgrad der Speicherbereiche Eden, Survivor, Old und Perm sowie die »Wanderung« der Objekte durch diese Bereiche. Außerdem kann man die Häufigkeit und Dauer von Garbage Collections ablesen. Grundlagen zum Thema Garbage Collection sind in Abschnitt 8.5 beschrieben.

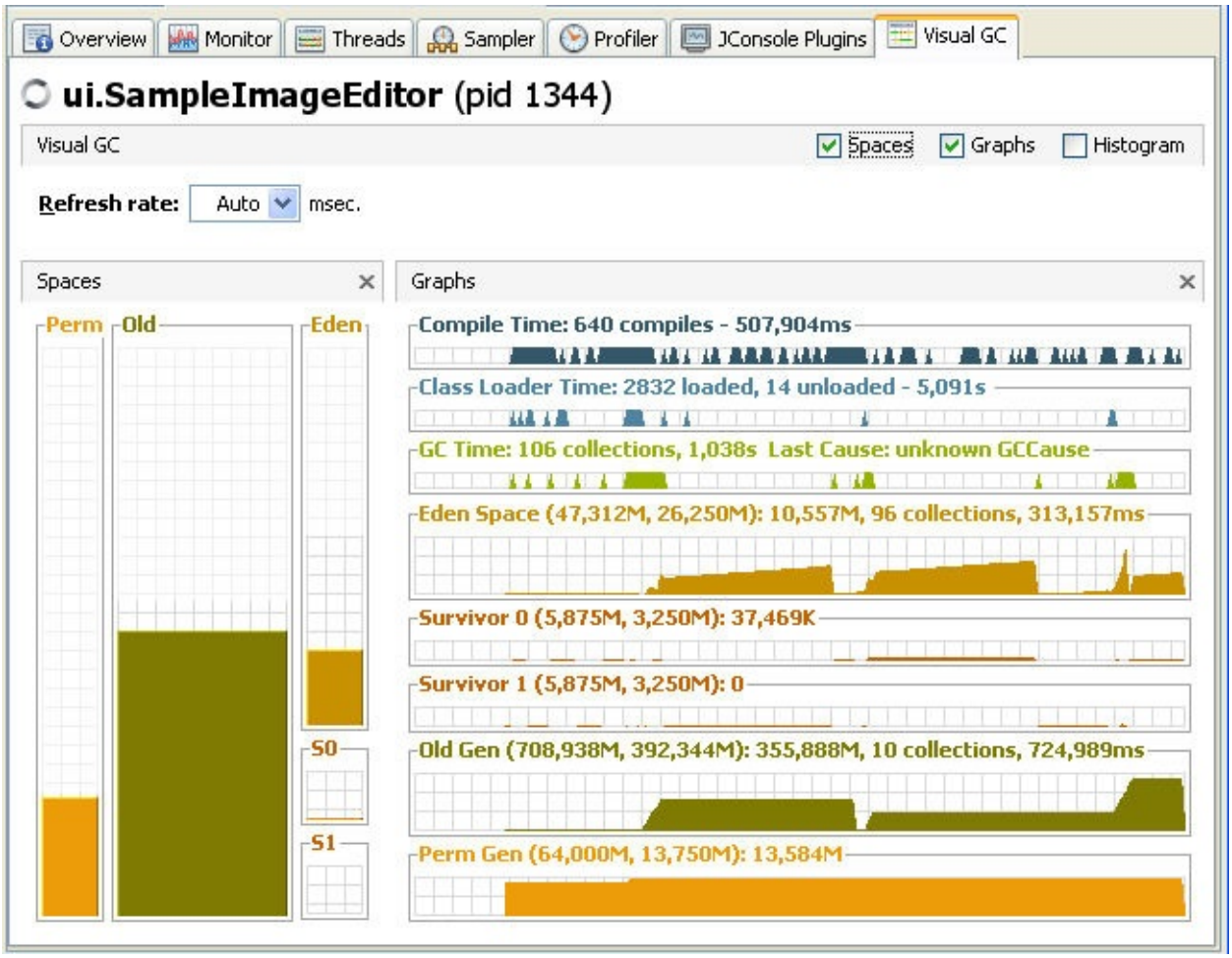


Abbildung 22-2 *VisualVM mit VisualGC-Plugin*

Kommerzielle Alternativen

Als Alternative zu den bereits recht hilfreichen, informativen frei verfügbaren Tools des JDKs existieren einige kommerzielle Programme, die noch ein wenig mehr Funktionalität bieten, um etwa einen Applikationsserver leichter beobachten zu können. Exemplarisch seien hier folgende Tools empfohlen, die als Evaluationsversionen aus dem Internet heruntergeladen werden können:

- **JProfiler** – `http://www.ej-technologies.com/`
- **JProbe** – `http://www.quest.com/jprobe/`

Validierung von Änderungen

Nachdem die kritischen Stellen in einem untersuchten Programm gefunden, analysiert und modifiziert wurden, sollte man zum einen alle Tests ausführen und zum anderen erneut Profiling-Messungen durchführen. Durch den Einsatz von Tests wird sichergestellt, dass durch die Optimierungen keine Fehler eingebaut wurden. Die Performance-Messungen werden wiederholt, um zu beurteilen und im Idealfall zu bestätigen, dass es tatsächlich zu den erwarteten Verbesserungen gekommen ist. Ist dies nicht der Fall, so verwerfen wir normalerweise die Änderungen. In einigen wenigen Fällen können Optimierungen aber auch für mehr Klarheit durch

strukturelle Verbesserungen oder eine bessere Namensgebung gesorgt haben. Derartige Änderungen sollten natürlich beibehalten werden.

Außerdem kann man beginnen, spezielle Performance-Tests basierend auf bereits existierenden Unit Tests mithilfe des Tools `JUnitPerf` zu erstellen. Weitere Informationen dazu finden Sie online unter `http://clarkware.com/software/JUnitPerf.html`.

Skalierbarkeit

Es ist wichtig, Performance-Tests mit einem umfangreichen, praxisrelevanten Satz an Testdaten mit annähernd realen Datenvolumina nachzustellen, weil einige Performance-Probleme erst bei sehr großen Datenmengen auftreten. Man spricht von schlechter ***Skalierbarkeit***, wenn mit zunehmender Datenmenge die Ausführungsdauer signifikant, mitunter sogar überproportional, zunimmt. Einige Designentscheidungen können zu einer schlechten Skalierbarkeit führen. Dies betrifft vor allem die eingesetzten Algorithmen und Datenstrukturen. In der einführenden Diskussion haben wir ein Beispiel dafür mit der Summierung von Zahlen kennengelernt.

Auch Aspekte der Parallelisierung von Aufgaben spielen eine Rolle: Ob man voneinander unabhängige Anfragen (Aufrufe) sequenziell oder aber parallel durch mehrere Threads bearbeitet, kann sich massiv auf die Performance und das Antwortzeitverhalten auswirken. Bei einer großen Zahl solcher Anfragen kommt es dann bei sequenzieller Abarbeitung möglicherweise zu miserablen Antwortzeiten und bei großer Parallelität bestenfalls zu einem geringen Einfluss der Anzahl von Anfragen auf die Antwortzeit.

Beispiel

Betrachten wir den Einfluss von Datenstrukturen exemplarisch an einer Software zur Kundenverwaltung. Beim Entwurf eines solchen Systems muss entschieden werden, in welcher Datenstruktur die Kundendaten gespeichert werden sollen. Geschieht dies in Form einer Liste, so ist zwar ein indizierter Zugriff schnell. In der Regel wird man auf Kunden aber nicht indiziert, sondern bevorzugt über deren Namen oder andere Auswahlkriterien zugreifen wollen. Eine lineare Suche über alle Datensätze ist für einige Hundert Einträge sicherlich kein Problem. Allerdings skaliert die Lösung schlecht. Je mehr Einträge vorhanden sind, desto länger dauert die Suche. Deren Ausführungszeit wächst in diesem Fall proportional zur Anzahl der gespeicherten Daten. Nutzt man eine `TreeMap<K,V>`, so liegen die gespeicherten Elemente sortiert vor, und eine Suche ist wesentlich performanter möglich. Die Ausführungsdauer wächst logarithmisch zum Datenvolumen.

Wie man bereits an dieser kurzen Diskussion sieht, ist eine formalere Betrachtung und Einordnung der Komplexität wünschenswert. Dazu wird im folgenden Abschnitt die O-Notation vorgestellt.

22.1.5 Abschätzungen mit der O-Notation

Zur Abschätzung und Beschreibung der Komplexität von Algorithmen und damit zur Einordnung ihres Zeitverhaltens wäre es unpraktisch, immer Messungen vornehmen zu müssen. Außerdem spiegeln Messungen lediglich das Laufzeitverhalten unter gewissen Randbedingungen der Hardware (Prozessortakt, Speicher usw.) wider.

Um Folgen von Designentscheidungen unabhängig von solchen Details und auf einer abstrakteren Ebene einordnen zu können, verwendet man in der Informatik die sogenannte ***O-Notation***, die die obere Schranke für die Komplexität eines Algorithmus angibt. Man möchte folgende Frage beantworten können: »Wie verhält sich ein Programm, wenn statt 1.000 Eingabewerten beispielsweise 10.000 oder 100.000 Eingabewerte verarbeitet werden?« Zur Beantwortung dieser Frage müssen die einzelnen Schritte eines Algorithmus betrachtet und klassifiziert werden. Ziel ist es, die Berechnung der Komplexität zu formalisieren, um Auswirkungen von Veränderungen an der Anzahl der Eingabedaten auf die Programmlaufzeit abschätzen zu können.

Betrachten wir folgende `while`-Schleife als einführendes Beispiel:

```
int i = 0;                          // O(1)
while (i < n)                       // O(n)
{
    createPersonInDb(i);            // O(1)
    i++;                            // O(1)
}
```

Jede einzelne Anweisung wird mit der Komplexität $O(1)$ bewertet. Die Schleife selbst erhält aufgrund der n Ausführungen des Schleifenrumpfs die Komplexität $O(n)$ zugeordnet.[4] Rechnet man diese Werte zusammen, dann sind die Kosten für die Abarbeitung des Programms folglich: $O(1) + O(n) * (O(1) + O(1)) = O(1) + O(n) * 2$. Für eine Abschätzung der Komplexität spielen konstante Summanden und Faktoren keine Rolle. Nur die höchste Potenz von n ist von Interesse. Somit kommt man für das abgebildete Programmstück auf eine Komplexität von $O(n)$. Diese Vereinfachung ist zulässig, da für größere Werte von n der Einfluss von Faktoren und kleineren Komplexitätsklassen unbedeutend ist. Für das Verständnis der Betrachtungen in den folgenden Abschnitten sollte diese informelle Definition ausreichend sein.

Nachfolgend möchte ich noch zwei die O-Notation charakterisierende Sätze von Robert Sedgewick aus seinem Standardwerk »Algorithmen« [79] zitieren: »[…] ist die O-Schreibweise ein nützliches Hilfsmittel, um obere Schranken für die Laufzeit anzugeben, die von den Einzelheiten der Eingabedaten und der Implementation unabhängig sind.« Und weiter heißt es dort: »Die O-Schreibweise erweist sich als äußerst nützlich, indem sie Analytikern hilft, Algorithmen nach ihrer Leistungsfähigkeit zu klassifizieren, und indem sie Entwickler von Algorithmen bei der Suche nach den »besten« Algorithmen unterstützt.«

Komplexitätsklassen

Um das Laufzeitverhalten verschiedener Algorithmen miteinander vergleichen zu können, reichen in der Regel sieben unterschiedliche Komplexitätsklassen aus. Folgende Aufzählung nennt die jeweilige Komplexitätsklasse und einige Beispiele dazu:

- $O(1)$ – Die konstante Komplexität liefert eine von der Anzahl der Eingabedaten n unabhängige Laufzeit. Diese Komplexität repräsentiert häufig ***eine Anweisung*** oder eine einfache Berechnung, die aus einigen Berechnungsschritten besteht.
- $O(log(n))$ – Bei der logarithmischen Komplexität kommt es zu einer Verdopplung der Laufzeit, wenn die Eingabedatenmenge n quadriert wird. Ein bekanntes Beispiel für diese Komplexität ist die ***Binärsuche***.
- $O(n)$ – Bei der linearen Komplexität wächst die Laufzeit proportional zur Anzahl der Elemente n. Dies ist bei einfachen Schleifen und Iterationen der Fall, etwa bei einer ***Suche in einem Array*** oder einer Liste.
- $O(n * log(n))$ – Diese Komplexität ist eine Kombination aus linearem und logarithmischem Wachstum. Einige der schnellsten ***Sortieralgorithmen*** (z. B. Mergesort) besitzen diese Komplexität.
- $O(n^2)$ – Die quadratische Komplexität führt bei einer Verdopplung der Menge der Eingabedaten n bereits zu einer Vervierfachung der Laufzeit. Bei einer Verzehnfachung der Eingabedaten kommt es schon zu einem Verhundertfachen der Laufzeit. In der Praxis findet man diese Komplexität bei ***zwei ineinander verschachtelten*** `for`- *oder* `while`-***Schleifen***. Einfache Sortieralgorithmen haben normalerweise diese Komplexität.
- $O(n^3)$ – Bei der kubischen Komplexität kommt es bei einer Verdopplung von n bereits zu einer Verachtfachung der Laufzeit. Die naive ***Multiplikation von Matrizen*** ist ein Beispiel für diese Komplexitätsklasse.
- $O(2^n)$ – Die exponentielle Komplexität führt bei einer Verdopplung von n zu einer Quadrierung der Laufzeit. Das klingt zunächst wenig. Bei einer Verzehnfachung steigt die Laufzeit aber um den Faktor 20 Milliarden! Die exponentielle Komplexität tritt häufig bei ***Optimierungsproblemen*** auf, etwa dem sogenannten Problem des Handlungsreisenden (***Traveling-Salesman-Problem***), bei dem es darum geht, den kürzesten Weg zwischen verschiedenen Städten zu ermitteln und dabei alle Städte zu besuchen. Um dem Problem der exorbitanten Laufzeit Herr zu werden, verwendet man Heuristiken, die zwar nicht die optimale Lösung, sondern nur eine Annäherung daran ermitteln, dafür aber eine viel geringere Komplexität und deutlich kürzere Laufzeit besitzen.

Tabelle 22-2 zeigt eindrucksvoll, welche Auswirkungen die genannten Komplexitätsklassen für verschiedene Mengen von Eingabedaten n besitzen:[5]

Tabelle 22-2 *Auswirkungen verschiedener Zeitkomplexitäten*

n	$O(log(n))$	$O(n)$	$O(n * log(n))$	$O(n^2)$	O
10	1	10	10	100	1
100	2	100	200	10.000	1.000
1.000	3	1.000	3.000	1.000.000	1.000.000
10.000	4	10.000	40.000	100.000.000	1.000.000.000
100.000	5	100.000	500.000	10.000.000.000	1.000.000.000.000
1.000.000	6	1.000.000	6.000.000	1.000.000.000.000	1.000.000.000.000.000

Anhand der dargestellten Werte bekommt man ein Gefühl für die Auswirkungen unterschiedlicher Komplexitäten: Bis etwa $O(n * log(n))$ sind die Komplexitätsklassen günstig. Optimal und wünschenswert, wenn auch für viele Algorithmen nicht erreichbar, sind die Komplexitäten $O(1)$ und $O(log(n))$. Bereits $O(n^2)$ ist für größere Eingabemengen in der Regel nicht mehr günstig, kann aber bei einfachen Berechnungen und kleineren Werten für n problemlos eingesetzt werden.

Hinweis: Einfluss der Eingabedaten

Einige Algorithmen verhalten sich abhängig von den Eingabedaten unterschiedlich. Für Quicksort ergibt sich im durchschnittlichen Fall eine Komplexität von $n * log(n)$, die aber im Extremfall auf n^2 ansteigen kann. Da die O-Notation den »Worst Case« beschreibt, wird Quicksort eine Komplexität von $O(n^2)$ zugeordnet.

Auswirkungen der Komplexität auf die Programmlaufzeit

Mögen die durch eine spezielle O-Komplexität errechneten Zahlen für eine Menge von Eingabewerten n manchmal auch abschreckend sein, so sagen diese jedoch nichts über die konkrete Ausführungszeit aus, sondern nur über deren Wachstum bei einer Vergrößerung der Eingabemenge. Wie bereits anhand des einführenden Beispiels deutlich wird, macht die O-Notation keine Aussage über die Dauer einzelner Berechnungsschritte: Das Inkrement `i++` und der Datenbankzugriff `createPersonInDb(i)` wurden jeweils mit $O(1)$ bewertet, obwohl der Datenbankzugriff auf die Ausführungszeit bezogen um mehrere Größenordnungen teurer als das Inkrement ist.

Für »normale« Anweisungen ohne Zugriffe auf externe Systeme, wie Dateisystem, Netzwerk oder Datenbank, also Additionen, Zuweisungen usw., ist der Einfluss von n

bei heutigen Computern nahezu unbedeutend, wie dies bereits im Beispiel für Multiplikationen und Bit-Shifts bei 1 Milliarde Durchläufen gezeigt wurde. Die Auswirkungen auf die tatsächliche Laufzeit sind für kleine n (< 1000) bei den Komplexitäten $O(n)$ bzw. $O(n^2)$ und sogar manchmal noch $O(n^3)$ heutzutage häufig kaum relevant. Dadurch können mitunter auch mehrfach ineinander verschachtelte Schleifen mit der Komplexität $O(n^2)$ oder $O(n^3)$ absolut gesehen viel schneller ausgeführt werden als einige Datenbankabfragen über ein Netzwerk mit der Komplexität $O(n)$. Ähnliches gilt für eine Suche in einem Array ($O(n)$) und einen Zugriff auf ein Element einer hashbasierten Datenstruktur ($O(1)$). Für kleine n kann die Berechnung der Hashwerte länger dauern als eine lineare Suche. Je größer allerdings n wird, desto mehr wirkt sich die schlechtere Komplexitätsklasse auf die tatsächliche Laufzeit aus.

22.2 Einsatz geeigneter Datenstrukturen

Wie bereits in der Einleitung angedeutet, lassen sich viele Performance-Probleme auf Fehler im Algorithmus oder in der Benutzung ungeeigneter Datenstrukturen zurückführen. Auch hier zitiere ich gerne wieder Jon Bentley [6]: »Eine richtige Sicht der Daten strukturiert tatsächlich die Programme. […] Immer für einfachen Code zu sorgen, ist normalerweise der Schlüssel zur Korrektheit!« Im Folgenden werden einige Implikationen auf die Performance durch die Wahl der Datenstrukturen Arrays, Listen, Sets und Maps erläutert und mithilfe der eingeführten O-Notation bewertet. Dabei betrachten wir einige gebräuchliche Operationen wie Einfügen oder Zugriff auf Elemente.

Aus dem Nähkästchen: Wahl von Datenstrukturen

Ich habe einmal einen Programmteil überarbeitet, der Jobs und Gruppen von Jobs verwalten sollte: Eigentlich ein klarer Fall für den Einsatz von Baumstrukturen und der klassische Ansatz wäre das KOMPOSITUM-Muster (vgl. Abschnitt 18.2.4). Der damalige Entwickler hatte wenig Erfahrung und kannte Baumstrukturen nur unzureichend (allerdings war dies Ende der 90er in C++ auch keine gebräuchliche, sofort verfügbare Datenstruktur).

Um Jobs und Gruppen von Jobs zu verwalten, wurde versucht, den Job-Baum in einer Liste zu speichern. So etwas kann man machen, allerdings erfordert dies eine präzise Vorstellung, wie man Baumoperationen auf Listenoperationen abbildet. Selbstverständlich sind die Operationen »Einfügen« und »Löschen« von Jobs nicht einfach zu realisieren: In einer Liste müssen diverse Elemente verschoben und anschließend einige Referenzanpassungen vorgenommen werden. Bei Verwendung eines Baums als Datenstruktur sind nur einfache Operationen auf Baumknoten auszuführen, die Komplexität wird durch die Datenstruktur gekapselt.

Der Einsatz eines Baums als Datenstruktur half dabei, den zuvor extrem komplizierten Sourcecode innerhalb von wenigen Tagen komplett umzustellen und

zu vereinfachen. Sowohl Komplexität und Umfang wurden deutlich reduziert. Zusätzlich kam es zu dem positiven Nebeneffekt, dass alle durch den ursprünglichen Ansatz verursachten Fehler gleich mit behoben wurden. Neben den positiven Auswirkungen auf Wartbarkeit und Erweiterbarkeit war die neue Realisierung deutlich performanter, vor allem für die Anwendungsfälle, in denen die Tiefe und Breite des Baums aufgrund einer komplexen Job-Gruppen-Struktur zunahm.

22.2.1 Einfluss von Arrays und Listen

In vielen Programmen müssen Daten in Arrays bzw. Listen gespeichert oder verarbeitet werden. Am Beispiel der Datenstrukturen `ArrayList<E>`, `LinkedList<E>`, `Vector<E>` und Array weise ich auf mögliche Folgen für die Performance hin.

Die Klassen `ArrayList` und `Vector`

Erinnern wir uns an den Aufbau und die Wirkungsweise der Datenstrukturen `ArrayList<E>` und `Vector<E>`, die in Kapitel 5 beschrieben sind. Beide speichern ihre Daten in einem Array und stellen somit eine komfortable Hülle um ein Array dar, die eine Schnittstelle mit Containerfunktionalität anbietet.

Zugriff auf ein Element an Position *i*

Der Zugriff auf beliebige Elemente wird durch einen indizierten Array-Zugriff an Position *i* realisiert und dessen Ausführungszeit ist unabhängig von der Anzahl der gespeicherten Elemente ($O(1)$).

Neues Element an Position *i* hinzufügen

Wenn an einer beliebigen Position *i* ein Element eingefügt wird, müssen alle Elemente mit einem Index >= *i* nach hinten kopiert werden, um Platz für das neue Element zu schaffen ($O(n)$). Reicht die Kapazität nicht mehr aus, so muss zusätzlich eine Erweiterung der Datenstruktur erfolgen. Es wird ein neues, um 50% vergrößertes Array erzeugt[6], und alle Elemente aus dem alten Array werden in dieses neue kopiert ($O(n)$).

Im günstigsten Fall wird ein neues Element an der letzten Position der Datenstruktur hinzugefügt. Wenn genügend Platz vorhanden ist, muss die Referenz im Array lediglich an der gewünschten Position abgelegt werden ($O(1)$). Ist jedoch kein Platz für das neue Element mehr vorhanden, so muss dafür Platz geschaffen werden ($O(n)$).

Element an Position *i* entfernen

Wenn an einer beliebigen Position *i* ein Element gelöscht wird, müssen alle dahinter liegenden Elemente mit einem Index > *i* um eine Position nach vorne kopiert werden. Dadurch ergibt sich eine Komplexität von *O* (*n*).

Die Klasse `LinkedList`

Die Klasse `LinkedList<E>` arbeitet intern mit untereinander verbundenen Knoten zur Speicherung von Elementen.[7] Es finden beim Einfügen und Löschen keine Verschiebeund Kopieraktionen statt, da jeweils nur Referenzen zwischen den Knoten angepasst werden müssen. Schauen wir, wie sich dies im Gegensatz zur Speicherung in einer `ArrayList<E>` bzw. einem `Vector<E>` auswirkt.

Zugriff auf ein Element an Position *i*

Um auf ein Element an einer Position *i* zuzugreifen, muss vor dem Zugriff (*O*(1)) zunächst bis zu dieser gewünschten Position über alle Vorgängerknoten iteriert werden (*O*(*n*)).

Neues Element an Position *i* hinzufügen

Der Einfügevorgang besteht lediglich aus dem Erzeugen eines neuen Knotens und aus einigen Referenzanpassungen und besitzt daher die Komplexität *O*(1). Allerdings führt die vorher durchzuführende Suche nach der Einfügeposition insgesamt zur Komplexität *O*(*n*).

Element an Position *i* entfernen

Der Löschvorgang besteht lediglich aus Referenzanpassungen und besitzt die Komplexität *O*(1). Auch hier führt die vorher durchzuführende Suche nach der Löschposition insgesamt zur Komplexität *O*(*n*).

Betrachtung von Vor- und Nachteilen

In den vorherigen Abschnitten wurden bereits einige Auswirkungen verschiedenster Operationen besprochen. Es ergibt sich folgende Vermutung: Die Anzahl der Daten, die Dynamik in der Zusammensetzung und die Position von Änderungen innerhalb der Datenstruktur besitzen großen Einfluss auf die Performance. Die Effekte sollen sich für verschiedene Implementierung des Interface `List<E>` deutlich unterscheiden. Rekapitulieren wir die wichtigsten Punkte: Für die `ArrayList<E>` bzw. den `Vector<E>` vermuten wir Performance-Probleme durch Einfüge- und Löschoperationen, die zu einem Verschieben von einigen Elementen führen – im ungünstigsten Fall sogar von allen. Beide Operationen sind auch für die `LinkedList<E>` durch die Suche nach der korrekten Position für Einfüge- oder Löschoperationen ausgehend vom Start- bzw. Endknoten aufwendig. Beides scheint

für größere Datenmengen nicht optimal zu sein.

Wir folgen den Empfehlungen aus Abschnitt 22.1 und führen deshalb Messungen durch, um die obigen Vermutungen zu bestätigen oder zu widerlegen. Für die Klassen `ArrayList<E>`, `LinkedList<E>` und `Vector<E>` erfolgen jeweils Messungen der Operationen Einfügen, Löschen und Iterieren für 10.000, 100.000 und 1.000.000 Elemente. Für die `ArrayList<E>` ermitteln wir zusätzlich die Laufzeiten mit initialer Kapazität von 100.000 Elementen. Das korrespondierende Programm ist stark gekürzt in folgendem Listing gezeigt:

```
public final class ListPerformanceExample
{
     private static final SimplePerson OBJ_TO_ADD = new SimplePerson( "Test",
42);

     private static final long[] maxElements = { 10_000, 100_000, 1_000_000 };

     public static void main( final String[] args)
     {
          performListTests( "ArrayList", new ArrayList<SimplePerson>());
          performListTests( "ArrayList IK", new ArrayList<SimplePerson>
(100_000));
          performListTests( "Vector", new Vector<SimplePerson>());
          performListTests( "LinkedList", new LinkedList<SimplePerson>());
     }


     private static void performListTests( final String dsName,
                                           final List<SimplePerson> list)
     {
          for ( final long max : maxElements)
          {
               System.out.println( "Element count " + max);

               list.clear();
               PerformanceUtils.startMeasure( dsName + " add front");
               addPersonFront( list, max);
               PerformanceUtils.stopMeasure( dsName + " add front");
               PerformanceUtils.printTimingResult( dsName + " add front");

               PerformanceUtils.startMeasure( dsName + " remove front");
               removePersonFront( list, max);
               PerformanceUtils.stopMeasure( dsName + " remove front");
               PerformanceUtils.printTimingResult( dsName + " remove front");

               // …
```

Zur Erinnerung nochmal: Die Messergebnisse sind für JDK 7 auf einem Quad-Core mit 2,6 GHz entstanden und in Tabelle 22-3 dargestellt. Zum Vergleich können Sie das Programm LISTPERFORMANCEEXAMPLE ausführen.

Tabelle 22-3 *Performance-Messungen für Listen*

Typ	Elemente	add [ms]			remove [ms]			[ms]
		Anfang	Mitte	Ende	Anfang	Mitte	Ende	for
ArrayList	10.000	32	3	0	31	3	0	2
ArrayList	100.000	731	326	3	729	333	5	6
ArrayList	1.000.000	95.455	46.594	2	95.946	46.914	1	0
ArrayList IK	10.000	6	3	0	5	2	0	0
ArrayList IK	100.000	736	328	0	721	345	0	0
ArrayList IK	1.000.000	95.799	467.627	2	95.290	46.909	1	1
Vector	10.000	32	3	0	32	3	0	1
Vector	100.000	733	343	2	758	335	1	0
Vector	1.000.000	95.816	47.301	7	97.376	46.973	9	8
LinkedList	10.000	3	43	1	3	43	3	2
LinkedList	100.000	2	4.527	0	0	4.550	0	0
LinkedList	1.000.000	11	682.676	4	3	681.926	7	2

Bevor ich die Ergebnisse inhaltlich bewerte, möchte ich noch etwas zur Genauigkeit der Angaben sagen: Da die Zeitmessung auf einem System immer gewissen Schwankungen unterliegt und es sich um auf Millisekunden gerundete Angaben handelt, sind Zahlenwerte von 0 bis 20 ms durchaus als nahezu gleich anzusehen. Ein Wert von etwa 50.000 ms bis 100.000 ms entspricht ungefähr 1 bis 2 Minuten, 700.000 ms sind demnach etwa 11 Minuten. Noch vor wenigen Jahren waren die Rechner und die JVM deutlich langsamer und ich habe Laufzeiten von bis zu einer Stunde gemessen.

Entgegen den zuvor geäußerten Erwartungen wirken sich Vergrößerungsschritte der Klasse `ArrayList<E>` nicht signifikant aus. Dies zeigen die Zahlen, die für die `ArrayList<E>` ohne eine Festlegung einer initialen Kapazität und für eine solche

mit der Anfangskapazität von 100.000 Elementen, die `ArrayList IK`, ermittelt wurden. Auch die in anderer Literatur als »teuer« bezeichneten synchronisierten Methoden der Klasse `Vector<E>` haben bei *nicht* nebenläufigen Zugriffen auf die Datenstruktur keinen merklichen Einfluss – das ändert sich erst dann, wenn eine Vielzahl an Threads um die Locks konkurriert. Zudem kann man erkennen, dass ein Iterieren mit einer `for`-Schleife über alle Elemente relativ schnell und günstig ist. Problematisch sind, wie erwartet, Einfüge- bzw. Löschoperationen. Allerdings sind die gemessenen Unterschiede bis etwa 10.000 gespeicherte Elemente recht gering. Darunter, etwa für 1.000 Einträge, sind die ermittelten Zeiten unter der Messbarkeitsgrenze und werden daher hier nicht aufgelistet. Nur bei größeren Datenmengen machen sich die zuvor diskutierten Effekte sehr deutlich bemerkbar. ***Die gemessenen Werte zeigen, dass die Einfüge- bzw. Löschposition entscheidenden Einfluss auf die Performance besitzt***. Für Array-basierte Listen ist das Einfügen und Löschen im vorderen Bereich der Liste am teuersten und nimmt Richtung Ende der Datenstruktur signifikant ab, da viel weniger Elemente bewegt werden müssen. Für die `LinkedList<E>` sind Einfügen und Löschen am Ende und Anfang der Datenstruktur nahezu unabhängig von der Menge der gespeicherten Werte. Dafür machen sich Positionen im Bereich der Mitte der Datenstruktur sehr viel stärker bemerkbar, als man es intuitiv vermuten würde. Das Einfügen von 1 Million Daten an der Mittenposition benötigt auf meinem Laptop über 1 Stunde. Array-basierte Listen sind etwa um den Faktor 17 schneller – selbst in deren ungünstigstem Fall sind sie noch etwa um den Faktor 9 schneller.

Lediglich bei einer großen Anzahl an Elementen und einer hohen Dynamik in der Zusammensetzung am Beginn der Liste kann die `LinkedList<E>` aufgrund ihres internen Aufbaus besser als eine `ArrayList<E>` geeignet sein. Dies gilt allerdings nur dann, wenn im Programmverlauf wenig indizierte Zugriffe erfolgen.

Insgesamt betrachtet skalieren die Array-basierten Listen besser. Die `LinkedList<E>` zeigt verheerende Performance-Einbrüche bei großen Datenmengen (ab einigen 100.000 Elementen). Das gilt vor allem bei Aktionen nahe der Listenmitte. Für derartige Zugriffe sind die `ArrayList<E>` und der `Vector<E>` deutlich besser geeignet. Wie wir später anhand eines Beispiels sehen werden, gilt dies im Besonderen für Tabellenmodelle. Der Einsatz einer `LinkedList<E>` ist dort extrem kontraproduktiv.

Tipp: Konfigurierbarkeit durch Einsatz des Interface `List<E>`

Programmiert man konsequent gegen das Interface `List<E>`, so kann man die Wahl einer konkreten Listenimplementierung konfigurierbar machen und diese mithilfe einer FABRIKMETHODE (vgl. Abschnitt 18.1.2) erzeugen. Dieses Vorgehen bietet sich in der Praxis immer dann an, wenn man die Dynamik und das Datenvolumen schlecht im Voraus schätzen kann. Durch nachfolgende Profiling-Messungen lässt sich die für den Anwendungsfall geeignete Realisierung ermitteln.

22.2.2 Optimierungen für `Set` und `Map`

Nachdem wir verschiedene Listen auf ihre Performance untersucht haben, werfen wir nun einen Blick auf die Realisierungen der Interfaces `Set<E>` und `Map<K, V>`. Diese sind bereits für viele Anwendungsfälle ausreichend optimiert. Durch die passende Wahl einer für einen konkreten Anwendungsfall geeigneten Realisierung sind jedoch weitere Performance-Gewinne möglich. Die folgenden Abschnitte beschreiben daher einige Performance-Implikationen und die Laufzeiten einiger wichtiger Zugriffsmethoden (Einfügen, Löschen und Zugriff auf Elemente) für konkrete Typen von Sets und Maps.

`HashSet` und `HashMap`

Die Realisierung der Klasse `HashSet<E>` verwendet intern die Klasse `HashMap<K, V>`. Daher können hier – bezogen auf die Auswirkungen auf die Performance – beide Datenstrukturen gemeinsam betrachtet werden. Die Implementierung der Klasse `HashMap<K, V>` ist extrem performant. Die Abfrage der Anzahl gespeicherter Einträge (`size()`) benötigt eine Laufzeit von $O(1)$. Auch die gebräuchlichsten Operationen `put()`, `get()`, `remove()` und `containsKey()` haben alle in der Regel eine Laufzeit von $O(1)$. Nur bei einer extrem ungünstig gewählten Hashfunktion (z. B. Implementierung der `hashCode()`-Methode, die einen konstanten Wert liefert) kann das natürlich nicht mehr gelten. Weitere Details finden Sie in Abschnitt 5.1.7.

Einfluss von `equals()`, `compareTo()` und `hashCode()`

Neben der Wahl einer geeigneten Datenstruktur kann eine weitere Verbesserung der Performance indirekt dadurch erreicht werden, dass man die steuernden Methoden optimiert. Wie bereits in Abschnitt 5.1.9 diskutiert, sind dies für Sets und Maps die Methoden `equals()`, `hashCode()` und `compareTo()` bzw. `compare()`. Die Methoden `equals()` und `compareTo()` besitzen in der Regel nur geringen Einfluss auf die Performance. Für beide Methoden besteht ein Optimierungspotenzial darin, zum einen lediglich die tatsächlich für den Vergleich relevanten Attribute zu betrachten und zum anderen Auto-Boxing und Auto-Unboxing möglichst zu vermeiden. Abschnitt 22.7.3 geht auf die Auswirkungen dieser Optimierung genauer ein. Auf keinen Fall sollte man derartige Optimierungen auf dieser CPU-basierten Ebene beginnen, ohne vorher alle anderen Optimierungsmöglichkeiten ausgeschöpft zu haben.

Die Methode `hashCode()` kann im Gegensatz zu den zuvor besprochenen Methoden größeren Einfluss auf die tatsächliche Performance besitzen. Bei der Implementierung sollten eine möglichst gleichmäßige Verteilung bzw. eine gute Streuung der berechneten Hashwerte und damit eine ausgewogene Verteilung der Objekte auf die Buckets angestrebt werden. Ansonsten nehmen die Kollisionen zu

und im Extremfall verhält sich ein `HashSet<E>` bzw. eine `HashMap<K, V>` nahezu wie eine lineare Liste mit vorangestellter Hashberechnung. Damit werden die Zugriffszeiten deutlich schlechter als bei einer ausgewogenen Verteilung der Objekte auf alle Buckets.

Neben der Implementierung der Methode `hashCode()` sind für hashbasierte Container bei Performance-Optimierungen zudem folgende weitere Dinge zu beachten:

1. **Kapazität und Füllgrad** – Um Kollisionen möglichst zu vermeiden und eine gute Performance beim Zugriff zu erzielen, sollte der Füllgrad erfahrungsgemäß maximal im Bereich von ca. 75% liegen. Wird der Füllgrad größer gewählt, so wird zwar der Speicher besser genutzt, allerdings kommt es dann auch häufiger zu Kollisionen, wodurch sich die Kosten beim Zugriff auf Elemente erhöhen.
2. **Vermeidung von Größenanpassungen und Rehashing** – Die initiale Kapazität sollte auf der geplanten Nutzlast basieren: Die voraussichtliche Anzahl von Elementen wird dazu mit dem Faktor 1,3 multipliziert. Dadurch bleiben zwar etwa ein Viertel aller Buckets unbelegt, aber man vermeidet so weitgehendst automatische Größenanpassungen, die wiederum dazu führen, dass die gespeicherten Elemente neu auf die Buckets verteilt werden müssen. Bekanntermaßen kostet ein solcher Umsortiervorgang zunächst einmal Rechenzeit, sorgt aber dafür, dass spätere Zugriffe wieder performant sind.

`ConcurrentHashMap`

Die Klasse `ConcurrentHashMap<K, V>` realisiert eine auf hohe Parallelität ausgelegte hashbasierte Map, deren Methodenkontrakte sich nicht, wie der Name vermuten lässt, an denen der Klasse `HashMap<K, V>`, sondern an denen der Klasse `Hashtable<K, V>` orientieren. Dadurch ist diese Map ein Ersatz für eine `Hashtable<K, V>`. Soll die `ConcurrentHashMap<K, V>` als Ersatz der Klasse `HashMap<K, V>` dienen, so ist dies in der Regel problemlos möglich. Es dürfen jedoch aufgrund der gewählten Umsetzung keine `null`-Werte für Schlüssel und Wert gespeichert werden.

Die zuvor gemachten Performance-Betrachtungen für Methoden der Klasse `HashMap<K, V>` gelten gleichermaßen für die Klasse `ConcurrentHashMap<K, V>`. Allerdings sind die konkreten Ausführungszeiten der Methoden einer `ConcurrentHashMap<K, V>` etwas höher, da intern zusätzlich einige Aktionen notwendig sind, um Thread-Sicherheit zu erreichen. Nur in einem Punkt gibt es einen signifikanten Unterschied: Die Abfrage der Größe über die Methode `size()` ist bei der `ConcurrentHashMap<K, V>` nicht in konstanter Zeit ($O(1)$) möglich, sondern erfordert signifikant mehr Rechenzeit, da alle Elemente zur Ermittlung der Größe traversiert werden müssen ($O(n)$). Aufgrund der genannten Arbeitsweise sollte die `ConcurrentHashMap<K, V>` mit Bedacht und gezielt dann verwendet werden, wenn tatsächlich Bedarf an Nebenläufigkeit mit häufigen, konkurrierenden Zugriffe

besteht. Bei seltenen konkurrierenden Zugriffen kann die `ConcurrentHashMap<K, V>` ihre Vorteile nicht adäquat ausspielen.

Würde man eine `ConcurrentHashMap<K, V>` anstelle einer `HashMap<K, V>` für alle Anwendungsfälle nutzen, die eine hashbasierte Speicherung ohne konkurrierende Zugriffe benötigen, so suggerierte man damit zudem, dass ein Programmstück auf Parallelität ausgelegt ist. Ist dies nicht der Fall, ist die Wahl dieser Datenstruktur irreführend, da sie nicht nahe am zu lösenden Problem ausgerichtet ist.

`TreeSet` und `TreeMap`

Da die Klasse `TreeSet<E>` unter Verwendung der Klasse `TreeMap<K, V>` implementiert ist, können wieder beide Datenstrukturen gemeinsam betrachtet werden.

Die in vielen Anwendungsfällen eingesetzten Methoden `containsKey()`, `get()`, `put()` und `remove()` besitzen aufgrund der Verwendung eines balancierten Baums als Datenstruktur alle eine garantierte Laufzeit von $O(log(n))$. Eine Größenabfrage über `size()` besitzt die Komplexität $O(1)$. Beide Realisierungen sind demnach sehr performant. Da die Daten in einer sortierten Reihenfolge in diesem Baum abgelegt werden (müssen), sind die Ausführungsgeschwindigkeiten für `put()` und `remove()` etwas geringer als diejenigen einer `HashMap<K, V>`.

`ConcurrentSkipListSet` und `ConcurrentSkipListMap`

Die Klasse `ConcurrentSkipListSet<E>` ist eine Thread-sichere Realisierung des Interface `SortedSet<E>` und verwendet zur Implementierung ihrer Funktionalität eine spezielle Map, die `ConcurrentSkipListMap<K, V>`. Für beide Klassen können im Gegensatz zu den nicht Thread-sicheren Realisierungen `TreeSet<E>` und `TreeMap<K, V>` sowohl Einfüge- als auch Löschoperationen gefahrlos durch mehrere Threads ausgeführt werden.

Die zuvor gemachten Performance-Betrachtungen für die Methoden der Klasse `TreeMap<K, V>` gelten hier gleichermaßen. Analog zur `ConcurrentHashMap<K, V>` werden die Methoden einer `ConcurrentSkipListMap<K, V>` durch den Mehraufwand zum Erreichen von Thread-Sicherheit etwas langsamer als die einer `TreeMap<K, V>` abgearbeitet. Weiterhin gilt hier wieder, dass die Abfrage der Größe über `size()` nicht in konstanter Zeit ($O(1)$) möglich ist, sondern deutlich mehr Rechenzeit erfordert ($O(n)$). Demzufolge sollten beide Datenstrukturen nur dann verwendet werden, wenn tatsächlich Bedarf an (hoher) Parallelität besteht.

22.2.3 Design eines Zugriffsinterface

Neben dem Einfluss einiger Algorithmen des Collections-Frameworks kann auch die Gestaltung des Zugriffs auf eine Datenstruktur Performance-relevant sein.

Aufgabenkontext und Anforderungsdefinition

Nehmen wir an, eine Klasse `AccessInterfaceExample` soll `Person`-Objekte in einer beliebigen Containerklasse verwalten, etwa einer `ArrayList<Person>`:

```
public AccessInterfaceExample
{
    private final List<Person> persons = new ArrayList<>();

    // …
```

Nehmen wir weiter an, ein Anwendungsfall bestünde darin, sämtliche Wohnorte von Personen zu ermitteln und diese sortiert bereitzustellen. Es ist demnach nur ein kleiner Teil eines `Person`-Objekts, der Wohnort, für Klienten von Interesse.

Beim Entwurf eines Zugriffsinterface erlauben Klassen allerdings häufig unnötigerweise durch ein zu breites Interface (viele öffentliche Methoden) zu viele Änderungen am Objektzustand. Darüber hinaus werden oft unbedacht Referenzen herausgereicht oder konkrete Datentypen verwendet, wodurch Datenkapselung und lose Kopplung leiden. Betrachten wir nun einige mögliche Realisierungen eines Zugriffsinterface und diskutieren kurz deren Vor- und Nachteile.

Realisierung eines direkten Zugriffs

Eine erste Idee könnte sein, eine Referenz auf die speichernde Datenstruktur herauszureichen, d. h. eine Referenz auf die `ArrayList<Person>`:

```
public List<Person> getPersons()
{
    return persons;
}
```

Diese Art der Realisierung ist häufig ungünstig: Klienten können die Zusammensetzung der Containerklassen modifizieren. Probleme bereitet dies im Besonderen, wenn es sich bei den zurückgelieferten Referenzen um solche handelt, die auf interne Daten verweisen, wie dies hier der Fall ist. Der Objektzustand kann dann auf unerwartete Weise durch Klienten geändert werden. Dies wurde bereits in Abschnitt 5.3.2 erläutert und hat dort zum Einsatz der `unmodifiable`-Wrapper geführt.

Neben der fehlenden Kapselung kommt erschwerend hinzu, dass die Logik zum Zugriff auf die Datenstruktur (hier die Ermittlung des Wohnorts) von jedem Klienten erneut programmiert werden muss. Es bietet sich an, diese Funktionalität in einer Utility-Klasse oder einer Hilfsmethode zu kapseln. Das wird im Folgenden gezeigt.

Realisierung einer spezialisierten Zugriffsmethode

Zur Umsetzung der Anforderungen muss keine Referenz auf eine `List<Person>` zurückgegeben werden. Es wird lediglich eine sortierte Liste von Städten benötigt. Eine naheliegende Realisierungsidee besteht darin, ein `TreeSet<String>` durch eine Iteration über alle gespeicherten Kunden und den Aufruf von `getCity()` zu ermitteln:

```
public Set<String> getCitySet()
{
    final Set<String> resultSet = new TreeSet<>();

    for (final Person person : persons)
    {
        resultSet.add(person.getCity());
    }

    return resultSet;
}
```

Durch die Realisierung der Methode `getCitySet()` sind die Anforderungen sehr gut erfüllt. Auch die Datenkapselung ist gelungen: Es wird kein direkter Zugriff auf die interne Datenstruktur `persons` geboten. Stattdessen wird ein `TreeSet<String>` neu erzeugt und mit allen vorkommenden Städtenamen gefüllt. Die Aufbereitung der Ergebnisse in der Methode `getCitySet()` ist vorteilhaft: Die Kopplung ist gelöst und die Ergebnisse können unabhängig von `Person`-Objekten verarbeitet werden. Bei Bedarf kann sogar die interne Form der Speicherung geändert werden, ohne Auswirkungen auf Klienten zu verursachen.

Wie bereits erwähnt, ist es problematisch, Klienten Möglichkeiten zur Veränderung der Zusammensetzung von zurückgelieferten Containerklassen zu ermöglichen. Häufigste Ursache ist das Herausreichen von Referenzen auf interne Datenstrukturen. Das geschieht hier nicht, sondern es wäre lediglich eine Änderung auf der temporären Ergebnismenge möglich. Das kann man unterbinden, wenn man diese folgendermaßen mit einem `unmodifiable`-Wrapper ummantelt:

```
public final Set<String> getUnmodifiableCitySet()
{
    return Collections.unmodifiableSet(getCitySet());
}
```

Realisierung mithilfe eines Iterators

Im vorherigen Beispiel wurde von `Person`-Objekten vollständig abstrahiert und nur auf einer Menge von Strings gearbeitet. Dadurch wird bereits eine gute Datenkapselung erreicht. Möchte man noch weiter von der eingesetzten Datenstruktur abstrahieren, könnte man auf die Idee kommen, Klienten lediglich einen Iterator auf die Ergebnisdatenstruktur zu liefern. Diese Idee wird durch folgende Methode `getCityIterator()` umgesetzt, in der zunächst über die Methode `getCitySet()` eine Ergebnismenge mit den Namen von Städten erstellt

und anschließend ein `Iterator<String>` zurückgeliefert wird:

```
public final Iterator<String> getCityIterator()
{
    return getCitySet().iterator();
}
```

Mit dieser Realisierung werden die beiden Ziele gute Datenkapselung und lose Kopplung erreicht. Diese Lösung scheint wirklich richtig gut zu sein! Oder doch nicht?

API-Design `Collection` vs. `Iterator`

Die Rückgabe einer Ergebnismenge als `TreeSet<String>` ermöglicht Klienten, von ihnen benötigte Containerfunktionalität direkt zu nutzen. Sie können beispielsweise durch einen Aufruf von `size()` feststellen, wie viele Elemente die Ergebnismenge enthält, oder mit `contains(Object)` abfragen, ob ein spezielles Element gespeichert ist. Die Realisierung dieser Containerfunktionalität erfordert mit einem Iterator jeweils einen manuellen Durchlauf über alle Elemente. Das kann aufwendig werden, wenn derartige Funktionalität häufig benötigt wird.

Die Einschränkung einer Rückgabe auf das Interface `Iterator<String>` ist dann unpraktisch, weil dadurch die Auswertung von Ergebnissen mithilfe der Algorithmen und Methoden der Containerklassen des Collections-Frameworks erschwert wird. Allerdings bietet die Lösung mit Iterator eine bessere Kapselung und löst die Kopplung. Je nach Anwendungsfall ist also die eine oder andere Lösung zu bevorzugen.

Zuvor habe ich motiviert, warum es bei der Rückgabe kleiner Datenmengen häufig praktischer ist, das `Collection<E>`-Interface zu nutzen, statt lediglich Zugriff auf ein `Iterator<E>`-Objekt anzubieten. Für sehr große Ergebnismengen kann es sinnvoll sein, anstelle des Typs `Collection<E>` ein `Iterator<E>`-Objekt zurückzugeben. Außerdem ist ein Iterator vorzuziehen, wenn die Ergebnismenge zwischen verschiedenen Rechnern übertragen werden muss. Die Idee dahinter ist, statt einmalig eine große Ergebnismenge bereitzustellen, diese schritt- oder blockweise anzufordern. Benötigen Berechnungen lediglich Zugriff auf aufeinanderfolgende Elemente, so kann durch einen Iterator ein sukzessives Durchlaufen der Daten gewährleistet werden, ohne viel Speicher zu verbrauchen. Diese Technik kommt beispielsweise bei der Suche im Internet und der Präsentation von Ergebnissen zum Einsatz. Häufig reicht bereits der erste Schwung an möglichen Treffern aus, der relativ schnell bereitgestellt werden kann. Müsste stattdessen zunächst die Ergebnismenge vollständig an einen Klienten übertragen werden, so hätte dieser eventuell recht lange zu warten. Außerdem müsste er die gelieferte Datenmenge zwischenspeichern und verarbeiten können.

Info: Einfluss des Algorithmus und der Ausführungsschicht

Leider sieht man immer wieder unnötige, aber aufwendige Berechnungen.

Beispielsweise werden mithilfe von SQL-Abfragen viele Datenbankeinträge ausgewählt und die ermittelten Ergebnisdaten als Liste von DTOs (vgl. Abschnitt 3.4.5) an einen Aufrufer übertragen. Wird aber lediglich die Größe der Liste abgefragt und ist der Inhalt der Datenobjekte nicht von Interesse, so war die aufwendige Konstruktion der DTOs sowie das Übertragen der Daten überflüssig. Als Folge müssen zudem alle temporär konstruierten Objekte anschließend vom Garbage Collector weggeräumt werden.

Sinnvollerweise ermittelt man über eine `SELECT COUNT(*)`-Abfrage einfach die Anzahl der Datensätze als `int`-Wert. Dies kommt der Performance zugute, da zum einen die Aktionen in der Datenbank selbst erfolgen (also nahe am zu lösenden Problem) und zum anderen viel weniger Speicher zum Transport und weniger Zeit zum Aufbereiten der Ergebnisse benötigt wird.

Die für beide Varianten eingesetzten Algorithmen sind nahezu gleich. Allerdings sind die ausführenden Stellen unterschiedlich. Einmal wird das Zählen der Einträge auf Applikationsebene gelöst, im zweiten Fall direkt in der Datenbank. An diesem Beispiel erkennt man sehr schön, dass die Wahl des richtigen Abstraktionsniveaus und die Wahl der passenden ausführenden Systemkomponente großen Einfluss auf die Performance haben können.

22.3 Lazy Initialization

Die sogenannte ***Lazy Initialization*** ist eine Optimierungstechnik, deren Ziel es ist, so wenig aufwendige Initialisierungen und Aktionen wie möglich durchzuführen. Daher werden nur tatsächlich benötigte oder nur wenig Aufwand verursachende Programmteile beim Programmstart initialisiert. Konkret heißt das: Berechnungen und Initialisierungen komplexer Objekte werden so lange verzögert, bis von anderen Programmkomponenten darauf zugegriffen wird, wodurch man Arbeitsschritte vermeidet, deren Ergebnisse eventuell niemals benötigt werden.

22.3.1 Lazy Initialization am Beispiel

Wir wollen die Optimierung durch Lazy Initialization anhand eines vereinfachten Beispiels aus der Praxis nachvollziehen. Betrachten wir dazu eine Applikation, die Kunden und deren Adressen (Privat-, Firmen- und Lieferadresse) verwaltet.[8] Zur Unterscheidung verschiedener Adressinformationen ist eine `enum`-Aufzählung `AddressType` mit den Werten `HOME`, `WORK`, `DELIVERY` definiert. Kunden werden durch die Klasse `Customer` modelliert. Diese aggregiert diverse andere Klassen wie folgt:

```
public class Customer
{
    private final long customerId;
    private AddressInfo homeAddress, workAddress, deliveryAddress;
```

```
    private CustomerInfo customerInfo;
    private ContactInfo contactInfo;

    public Customer(final long newCustomerId)
    {
        customerId = newCustomerId;

        homeAddress = new AddressInfo(customerId, AddressType.HOME);
        workAddress = new AddressInfo(customerId, AddressType.WORK);
        deliveryAddress = new AddressInfo(customerId, AddressType.DELIVERY);

        customerInfo = new CustomerInfo();
        contactInfo = new ContactInfo();
    }

    // …
```

Im Konstruktor der Klasse `Customer` werden verschiedene aggregierte `AddressInfo`-Objekte erzeugt. Für diese existieren folgende Zugriffsmethoden, die einen direkten Zugriff auf interne Referenzen auf `AddressInfo`-Objekte bieten:

```
public AddressInfo getHomeAddress()
{
    return homeAddress;
}

public AddressInfo getWorkAddress()
{
    return workAddress;
}

public AddressInfo getDeliveryAddress()
{
    return deliveryAddress;
}
```

Dadurch ist es möglich, den Objektzustand unkontrolliert von außen zu verändern. Das ist normalerweise unerwünscht und zu vermeiden. Da die am Ende dieses Abschnitts dargestellte Lösung dieses Problem auf elegante Art und Weise behebt, gehe ich im Folgenden auf eine andere Abhilfe nicht weiter ein.

Nehmen wir an, die obigen Methoden würden von einer Applikation verwendet, um einige Informationen in der folgenden Methode auszugeben:

```
public static void printHomeAddress(final int customerId)
{
    final Customer customer = new Customer(customerId);
    final AddressInfo address = customer.getHomeAddress();

    System.out.println("Street" + address.getStreet());
    System.out.println("City" + address.getCity());

    //…
}
```

Betrachtet man das Listing für sich allein, scheint zunächst kein Problem zu existieren. Dieser erste Entwurf stellt sich allerdings beim Testen und anschließendem Debugging als Engpass heraus: Man erwartet, dass die Methode `printHomeAddress(int)` schnell abgearbeitet würde, doch die Ausführung

dauert immer einige Sekunden.

Analyse

Bevor man größere Umbaumaßnahmen beginnt, sollte man durch Sourcecode-Analyse und Profiling-Messungen eine konkrete Vorstellung der Performance-Bremsen haben.

Für unser Beispiel erhält man durch Profiling einen Hinweis, dass ziemlich viel Zeit im Konstruktor der aggregierten Klasse `AddressInfo` verbracht wird. Schaut man die Implementierung an, so erkennt man, dass dort Informationen mit Datenbankabfragen im Konstruktor ermittelt werden:

```
AddressInfo(final long customerId, final AddressType addressType)
{
    this.customerId = customerId;
    this.addressType = addressType;

    initAddress(DBAccess.selectAddressByAddressType(customerId, addressType));
}
```

Ein solches Vorgehen kann zum einen sehr laufzeitintensiv sein und zum anderen zu Problemen durch nicht freigegebene Ressourcen beim Auftreten von Exceptions führen. Abschnitt 16.3.11 stellt dies als BAD SMELL: RESOURCE LEAKS DURCH EXCEPTIONS IM KONSTRUKTOR vor.

Bezogen auf die Performance erkennen wir folgendes Problem: Das komplexe `Customer`-Objekt wird immer vollständig mit all seinen aggregierten Objekten erzeugt. Allerdings benötigt die Applikation in dieser Ausbaustufe nur einen ganz geringen Teil eines `Customer`-Objekts, hier lediglich die über `getHomeAddress()` ermittelte Anschrift. Alle anderen aggregierten Teilkomponenten werden zwar erzeugt, aber niemals verwendet. Handelte es sich nur um kleine, einfache Objekte, wäre dies nicht allzu kritisch. Bei schwergewichtigen Objekten mit Datenbankzugriffen wie in diesem Fall wirkt sich dies allerdings negativ auf die Performance und den Speicherverbrauch aus. Wie man derart verursachte Performance-Probleme lösen kann, beschreibt der folgende Abschnitt.

Einführen von Lazy Initialization

Bei der Konstruktion schwergewichtiger Objekte sind durch den gezielten Einsatz von Lazy Initialization – in diesem Fall zur Ermittlung der Adressinformationen – gute Performance-Gewinne zu erzielen. Dies gilt vor allem, wenn dadurch Netzwerkzugriffe oder Datenbankabfragen innerhalb von häufig durchlaufenen Schleifen vermieden werden können. Teilweise scheint als Folge die Handbremse im Programm gelöst worden zu sein. Der Lazy Initialization liegen dazu folgende Gedanken zugrunde:

1. Verzögere Berechnungen und Initialisierungen wenn möglich: Konstruiere nur

sofort benötigte Teile eines Objekts, aber nicht optionale, aggregierte Komponenten.

2. Initialisiere Referenzen auf optionale, aggregierte Komponenten mit `null` oder durch Einsatz des NULL-OBJEKT-Musters (vgl. Abschnitt 18.3.2).

3. Erzeuge Komponenten erst dann, wenn sie tatsächlich benötigt werden. Setze gegebenenfalls ein Stellvertreterobjekt gemäß dem PROXY-Muster (vgl. Abschnitt 18.3.6) ein, um mit dem ehemals schwergewichtigen Objekt bereits ansatzweise arbeiten zu können.

Betrachten wir, wie sich die Anwendung dieser Maßnahmenliste auf den Sourcecode des Beispiels auswirkt. Beginnen wir mit dem Konstruktor der Klasse `Customer`. Hier werden gemäß den Punkten 1 und 2 alle `AddressInfo`-Referenzen mit `null` initialisiert:

```
Customer(final int newCustomerId)
{
    customerId = newCustomerId;

    homeAddress = null;
    workAddress = null;
    deliveryAddress = null;
    customerInfo = new CustomerInfo();
    contactInfo = new ContactInfo();
}
```

Da die Klassen `CustomerInfo` und `ContactInfo` keine Datenbankabfragen durchführen und leichtgewichtig sind, wird für diese auf den Einsatz von Lazy Initialization verzichtet. Auch hier gilt: ***So wenig wie möglich, so viel wie nötig***. Der Grund ist einfach: Durch Lazy Initialization führt man Komplexität ein und erhöht somit die Wahrscheinlichkeit für Fehler. Wird die Initialisierung von aggregierten Objekten verzögert, so müssen dann gemäß Punkt 3 die jeweiligen Zugriffsmethoden prüfen, ob bereits eine Initialisierung erfolgt ist und gegebenenfalls ein benötigtes Objekt erzeugen:

```
// Achtung: Nicht Thread-sicher!
public AddressInfo getHomeAddressLazy()
{
    if (homeAddress == null)
    {
        homeAddress = new AddressInfo(this.customerId, AddressType.HOME);
    }
    return homeAddress;
}
```

Diese Methode ist funktional in Ordnung, weist jedoch noch Probleme bezüglich korrekter Initialisierung und Thread-Sicherheit auf. Darauf geht der folgende Abschnitt ein.

22.3.2 Konsequenzen des Einsatzes der Lazy Initialization

Der Einsatz von Lazy Initialization führt zu mehr Komplexität im Sourcecode: Als Folge muss an diversen Stellen geprüft werden, ob benötigte Komponenten bereits initialisiert sind oder nicht. Dies erhöht die Gefahr für Probleme, wenn Multithreading eingesetzt wird. Unter ungünstigen Bedingungen – und diese treten laut Murphy's Law mit ziemlicher Sicherheit auf – kommt es zu Synchronisationsproblemen bei der Initialisierung bzw. beim Zugriff auf Attribute. Dies ist dann der Fall, wenn mehrere Threads fast gleichzeitig zugreifen, möglicherweise veraltete Werte sehen und somit mehrfach eine Initialisierung durchführen. Das erinnert an die Probleme von `getInstance()`-Methoden beim SINGLETON-Muster (vgl. Abschnitt 18.1.4).

Synchronisationsprobleme

Als erste Abhilfe kommt das Schlüsselwort `synchronized` zum Einsatz:

```
public synchronized AddressInfo getHomeAddressLazyCorrected()
{
    if (homeAddress == null)
    {
        homeAddress = new AddressInfo(this.customerId, AddressType.HOME);
    }
    return homeAddress;
}
```

Werden alle Zugriffe auf Adressen derart geschützt, so verhindert man eine Parallelausführung der Zugriffsmethoden. Über jeweils eigene Synchronisationsobjekte kann bei Bedarf für etwas mehr Nebenläufigkeit gesorgt werden:

```
private static final Object homeAddressLock = new Object();
private static final Object workAddressLock = new Object();
```

Die Zugriffsmethoden werden dann jeweils über den speziellen Lock des korrespondierenden Synchronisationsobjekts geschützt. Abschnitt 7.2 beschreibt dies im Detail. Bei Bedarf nach mehr Parallelität und einer unterschiedlichen Sperre für Lese- bzw. Schreibzugriffe sollte man den Einsatz von `ReadWriteLocks` in Betracht ziehen – jedoch nur, wenn wenige Schreibzugriffe erfolgen sollen, weil ansonsten längere Verzögerungen zu beobachten sind.

Initialisierungsprobleme

Greift man auf lazy-initialisierte Attribute nicht über die dafür vorgesehenen Zugriffsmethoden zu, so kann es leicht zu `NullPointerExceptions` durch eine fehlende Initialisierung der Objekte kommen. Dies lässt sich zwar durch konsequentes Einführen von `null`-Prüfungen und durch Erzeugen benötigter Komponenten lösen, allerdings bläht sich der Sourcecode dadurch enorm auf. Zum einen dupliziert man die Prüfungen und zum anderen wird schnell eine wichtige Programmstelle übersehen.

Eine fehlertolerantere Lösung lässt sich durch den Einsatz des NULL-OBJEKTMusters (vgl. Abschnitt 18.3.2) erzielen. Wir implementieren dazu eine Klasse `EmptyAddressInfo` wie folgt:

```
public class EmptyAddressInfo extends AddressInfo
{
    public static final long NO_ID = -1;

    public EmptyAddressInfo()
    {
        super( NO_ID,  AddressType. UNKNOWN);
    }
    // …
```

Nach kurzer Überlegung erkennt man allerdings, dass es unelegant ist, von der Klasse `AddressInfo` zu erben. Zum einen erzeugt dies eine ungewünschte Abhängigkeit von der Basisklasse. Zum anderen werden im Konstruktor der Basisklasse bereits Datenbankverbindungen aufgebaut, die für das Null-Objekt nicht benötigt werden. Damit sich die Klasse `EmptyAddressInfo` gemäß ihrer Basisklasse `AddressInfo` verhält, muss man zudem eine künstliche »ungültige« Id (`NO_ID`) sowie einen speziellen Wert `UNKNOWN` für den Aufzählungstyp `AddressType` definieren. Diese Lösung ist damit unnatürlich, kompliziert und wirkt sich dadurch negativ auf die Verständlichkeit aus. Es ist daher sinnvoll, ein gemeinsames Interface `AddressInfoIF` herauszufaktorieren und die Realisierung wie in Abbildung 22-3 gezeigt vorzunehmen.

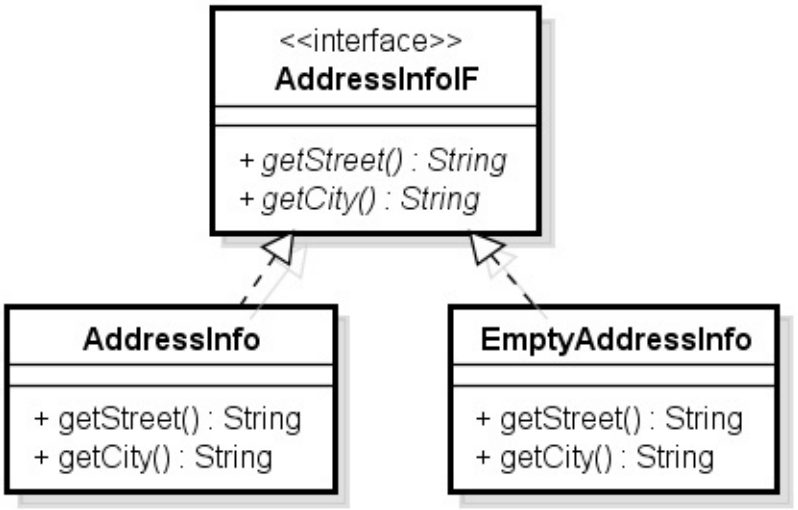


Abbildung 22-3 `AddressInfoIF`*-Interface als Basis für Klasse und Null-Objekt*

Durch Einsatz des neu eingeführten Null-Objekts wird die Applikation toleranter gegenüber Zugriffen auf vorher uninitialisierte Variablen:

```
private static final AddressInfoIF EMPTY_ADDRESS = new EmptyAddressInfo();

Customer( final int newCustomerId)
{
    customerId = newCustomerId;

    homeAddress = EMPTY_ADDRESS;
    workAddress = EMPTY_ADDRESS;
    deliveryAddress = EMPTY_ADDRESS;
    // …
```

Diese Lösung ist aber problematisch, da sie die Initialisierungsprobleme lediglich verlagert und Anwendungsfehler verschleiert: Es kommt im Fehlerfall nun nicht mehr zu Programmabstürzen, sondern zu merkwürdigem, fehlerhaftem Programmverhalten. Wir wollen eine sichere und elegante Lösung erstellen, dabei den Zugriff einfach gestalten und die Details der Erzeugung und Initialisierung verstecken. Die Grundlage dafür bildet das gerade herausfaktorierte Interface `AddressInfoIF`.

22.3.3 Lazy Initialization mithilfe des PROXY-Musters

Mithilfe des in Abschnitt 18.3.6 vorgestellten PROXY-Musters kann eine vorteilhafte Erweiterung in Form einer Abstraktionsschicht um die Lazy Initialization gebaut werden. Wenn die Initialisierung zentral im Proxy erfolgt, lassen sich Anwendungsfehler vermeiden. Das resultierende Klassendiagramm ist in Abbildung 22-4 dargestellt.

Der Proxy `LazyAddressInfo` führt die Lazy Initialization für die Applikation vollständig transparent durch. Dazu wird in jeder Realisierung einer im Interface `AddressInfoIF` deklarierten Zugriffsmethode zunächst die Methode `ensureAddressInfoExistence()` aufgerufen. Diese sorgt dafür, dass beim ersten Zugriff ein entsprechendes `AddressInfo`-Objekt erzeugt wird. Zur Vermeidung von Multithreading-Problemen ist diese Methode synchronisiert. Erfolgt niemals ein Methodenaufruf an das `AddressInfoIF`, so wird demzufolge auch kein `AddressInfo`-Objekt erzeugt. Eine mögliche Umsetzung sieht wie folgt aus:

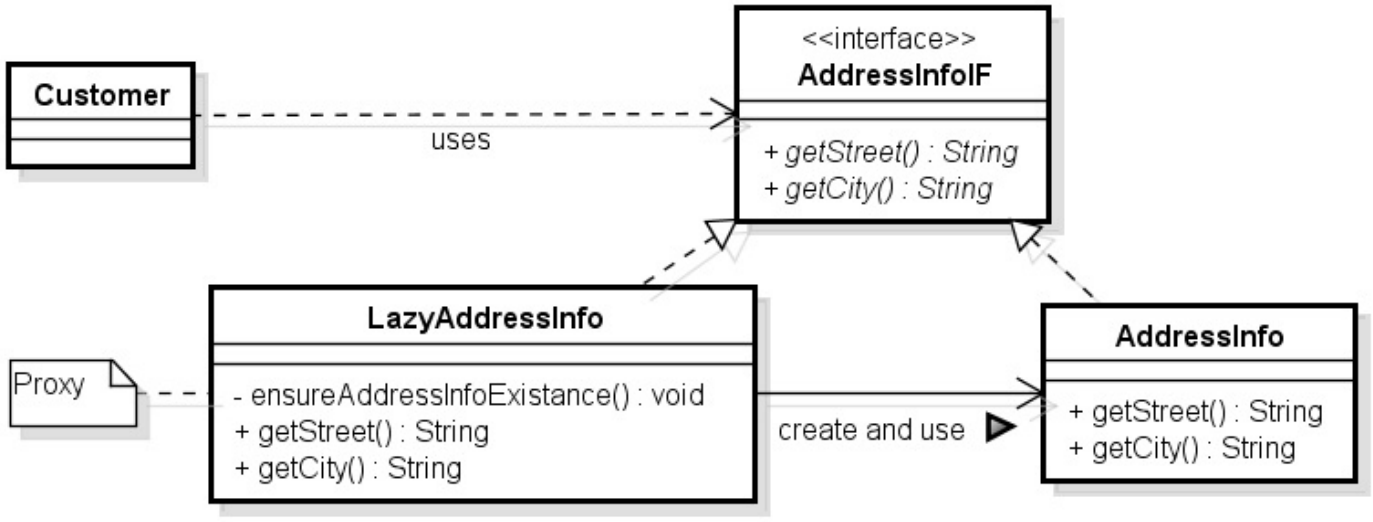


Abbildung 22-4 *Lazy Initialization mit der Klasse* `LazyAddressInfo`

```
public class LazyAddressInfo implements AddressInfoIF
{
    private  AddressInfoIF    realAddressInfo;

    private final long        customerId;
    private final AddressType addressType;
```

```
    LazyAddressInfo(final long customerId, final AddressType addressType)
    {
        this.customerId = customerId;
        this.addressType = addressType;
    }

    // …

    private synchronized void ensureAddressInfoExistence()
    {
        if (realAddressInfo == null)
        {
            realAddressInfo = new AddressInfo(customerId, addressType);
        }
    }

    public String getStreet()
    {
        ensureAddressInfoExistence();
        return realAddressInfo.getStreet();
    }

    public String getCity()
    {
        ensureAddressInfoExistence();
        return realAddressInfo.getCity();
    }

    // …
```

Um den Proxy zu verwenden, sind in diesem Beispiel lediglich Anpassungen im Konstruktor durchzuführen:

```
Customer(final int newCustomerId)
{
    customerId = newCustomerId;

    homeAddress = new LazyAddressInfo(customerId, AddressType.HOME);
    workAddress = new LazyAddressInfo(customerId, AddressType.WORK);
    deliveryAddress = new LazyAddressInfo(customerId, AddressType.DELIVERY);
    // …
    customerInfo = new CustomerInfo();
    contactInfo = new ContactInfo();
}

public AddressInfoIF getHomeAddress()
{
    return homeAddress;
}
```

Wenn alle Aufrufstellen ein Proxy-Objekt vom Typ `LazyAddressInfo` verwenden, anstatt auf das eigentliche Objekt zuzugreifen, führt erst ein Zugriff auf die Daten etwa über `getStreet()` zu einer Konstruktion eines `AddressInfo`-Objekts, was aber für den Aufrufer transparent bleibt. Auch müssen die Zugriffsmethoden keine Initialisierungen durchführen und können vereinfacht werden. Durch den Einsatz des Proxys können sämtliche `null`-Prüfungen und zugehörige Objektkonstruktionen aus dem Applikationscode entfernt werden. Dadurch sind alle den Sourcecode aufblähenden und fehleranfälligen Stellen überflüssig, die als Folge der ersten, naiven

Realisierung der Lazy Initialization entstanden sind. Als weiterer positiver Nebeneffekt wurde durch das Einführen des Interface `AddressInfoIF` die Kopplung zur implementierenden Klasse gelöst. Da in diesem Fall im Interface sogar nur Leseoperationen auf unveränderliche Daten angeboten werden, sind die Adressinformationen zudem nach außen unveränderlich. Wie die Methode `getHomeAddress()` exemplarisch zeigt, ist nicht einmal eine Synchronisierung in der `Customer`-Klasse nötig. Auch darum kümmert sich der Proxy.

Empfehlungen

Nachdem wir unter Betrachtung möglicher Fallstricke und Lösungen die Lazy Initialization schrittweise realisiert haben, leiten wir folgende Hinweise ab:

1. Verwende Lazy Initialization immer mit Bedacht und Vorsicht und möglichst in Kombination mit dem PROXY-Muster (vgl. Abschnitt 18.3.6). Dessen Einsatz reduziert die Gefahr von uninitialisierten Attributen sowie von Synchronisationsproblemen bei Multithreading und erhöht die Lesbarkeit des Sourcecodes.
2. Verwende Lazy Initialization lediglich für spezielle, aufwendige Initialisierungen. Prüfe dazu, ob große Objekte tatsächlich immer bereits alle Daten im Voraus halten müssen oder ob Teile nicht erst bei Bedarf angelegt werden können.
3. Vermeide den Einsatz von Lazy Initialization für alle Komponenten, die mit ziemlicher Sicherheit verwendet werden oder deren Erzeugung nicht sehr teuer ist.

Fazit

Bis hierher haben wir lediglich die Vereinfachungen in der Applikation selbst durch Einführen eines Proxys betrachtet. Abschließend wollen wir kurz die Auswirkungen auf die Performance der Methode `printHomeAddress(int)` analysieren: Zur Ausgabe der Adresse eines Kunden muss immer noch eine Datenbankabfrage erfolgen und die `homeAddress` ermittelt werden.

Allerdings werden als Folge der Lazy Initialization für die Attribute vom Typ `AddressInfo` bei der Konstruktion eines `Customer`-Objekts keine `AddressInfo`-Objekte mehr erzeugt, sondern dies geschieht erst bei Bedarf während des Zugriffs über `get()`-Methoden. Da in der eingangs als laufzeitproblematisch vorgestellten Methode `printHomeAddress(int)` die beiden Attribute `workAddress` und `deliveryAddress` nicht benötigt werden, spart man mit dieser Technik folglich das Auslesen jener Werte aus der Datenbank. Der Performance-Gewinn liegt demnach in zwei eingesparten Datenbankzugriffen. Wie bereits angedeutet und im folgenden Abschnitt weiter vertieft, besitzen Zugriffe auf externe Systeme häufig deutlich negative Auswirkungen auf die Performance. Überflüssige Aufrufe in diese Systeme zu vermeiden, kann somit die Performance spürbar verbessern.

22.4 Optimierungen am Beispiel

Anhand eines Optimierungsbeispiels möchte ich den Einsatz des in Abschnitt 22.1.4 erwähnten Tools `VisualVM` motivieren, um mögliche Performance-Probleme aufzuspüren. Zudem bekommen Sie ein Gefühl für Lösungsansätze und Umsetzungen von Optimierungen. Dieses Beispiel basiert auf einer von mir vor langer Zeit (JDK 1.2) tatsächlich durchgeführten Optimierung einer Tabellenkomponente. Diese enthielt neben der textuellen Darstellung von Einträgen verschiedene Grafiken, die zur Anzeige aus Dateien nachgeladen wurden.

Problemkontext

Damals war das Zeichnen und Scrollen mit bis etwa 100 Tabelleneinträgen ausreichend schnell. Als jedoch Tests mit wenigen Tausend Einträgen erfolgten, wurde die Benutzeroberfläche schrecklich langsam und man konnte nur noch stockend durch die Liste navigieren. Heutzutage sind sowohl die Rechner als auch die JVM wesentlich schneller geworden. Damit wir überhaupt ein sinnvolles Performance-Tuning durchführen können, müssen wir die Randbedingungen enorm verschärfen.

Betrachten wir eine Beispielapplikation, die eine Tabelle mit 50.000 Einträgen von `Person`-Objekten darstellt. Deren Speicherung erfolgt in einem Tabellenmodell `PersonTableModel` in einer `LinkedList<Person>`. Ein spezieller Renderer `SimpleImageTableCellRenderer` lädt die anzuzeigenden Grafiken bei jedem Zeichenvorgang aus dem Dateisystem. Folgendes Listing zeigt den relevanten Auszug aus der Implementierung:

```
public final class ListOptimizationExample extends JFrame
{
   private final PersonTableModel personTableModel;
   private final JTable           personTable;

   public ListOptimizationExample(final int personCount,
                                  final ListType listType,
                                  final TableCellRenderer cellRenderer)
   {
       setTitle("ListOptimizationExample Persons: " + personCount + " / Using:
"
        + listType + " / Renderer: " +
cellRenderer.getClass().getSimpleName());

       personTableModel = new PersonTableModel(listType, personCount);
       personTable = new JTable(personTableModel);
       personTable.setRowHeight(64);
       personTable.setDefaultRenderer(ImageIcon.class, cellRenderer);

       getContentPane().add(new JScrollPane(personTable));

       setDefaultCloseOperation(DISPOSE_ON_CLOSE);
   }

   public static void main(final String[] args)
```

```
    {
        final JFrame demoframe = new ListOptimizationExample(50_000,
                        ListType.LinkedList, new
SimpleImageTableCellRenderer());

        demoframe.setSize(700, 500);
        demoframe.setVisible(true);
    }
    // …
```

Listing 22.1 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLE'**

Führt man das Programm LISTOPTIMIZATIONEXAMPLE aus, so zeigt sich, dass das Scrolling etwas träge ist. Auch beim Vergrößern des Fensters kommt es zu Nachzieheffekten. Die Applikation sieht in etwa wie in Abbildung 22-5 aus.

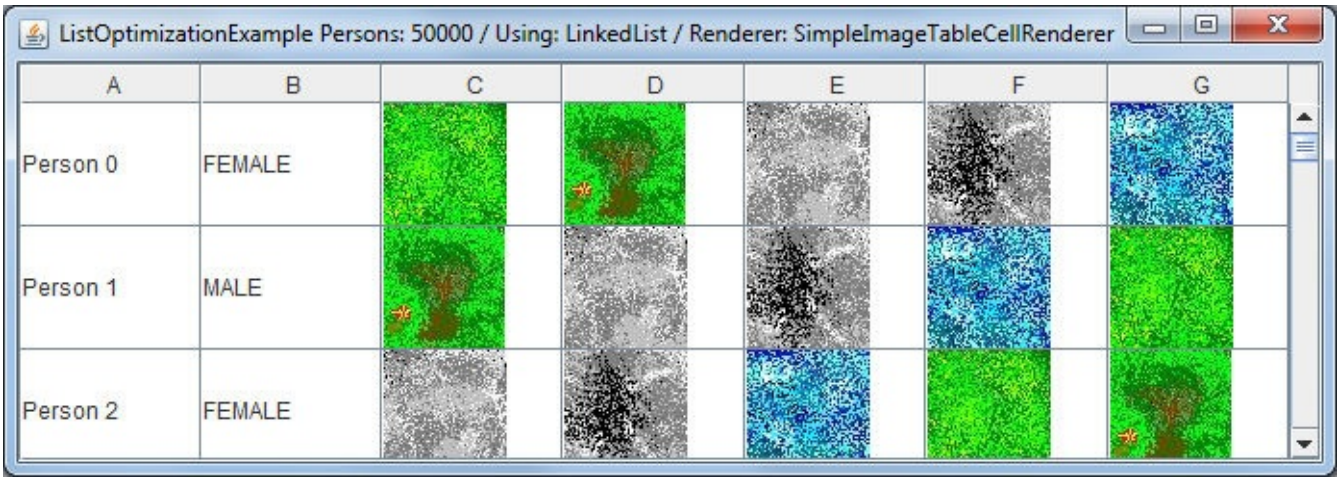


Abbildung 22-5 *Optimierungsbeispiel: Tabelle mit 50.000 Einträgen*

Analyse

Zuerst wollen wir durch Messungen diejenigen Programmbereiche ermitteln, die sich negativ auf die Performance auswirken. Dazu starten wir das Tool `VisualVM` und die obige Applikation. Die Profiling-Messungen zeigen zwei Hotspots: Zum einen ist das Laden der Grafiken über `loadTileImage(int)` teuer und zum anderen sehen wir, dass relativ viel Zeit für die Zugriffe auf das Datenmodell über `getValueAt(int, int)` verbraucht wird. Abbildung 22-6 zeigt die zugehörige Profiling-Messung.

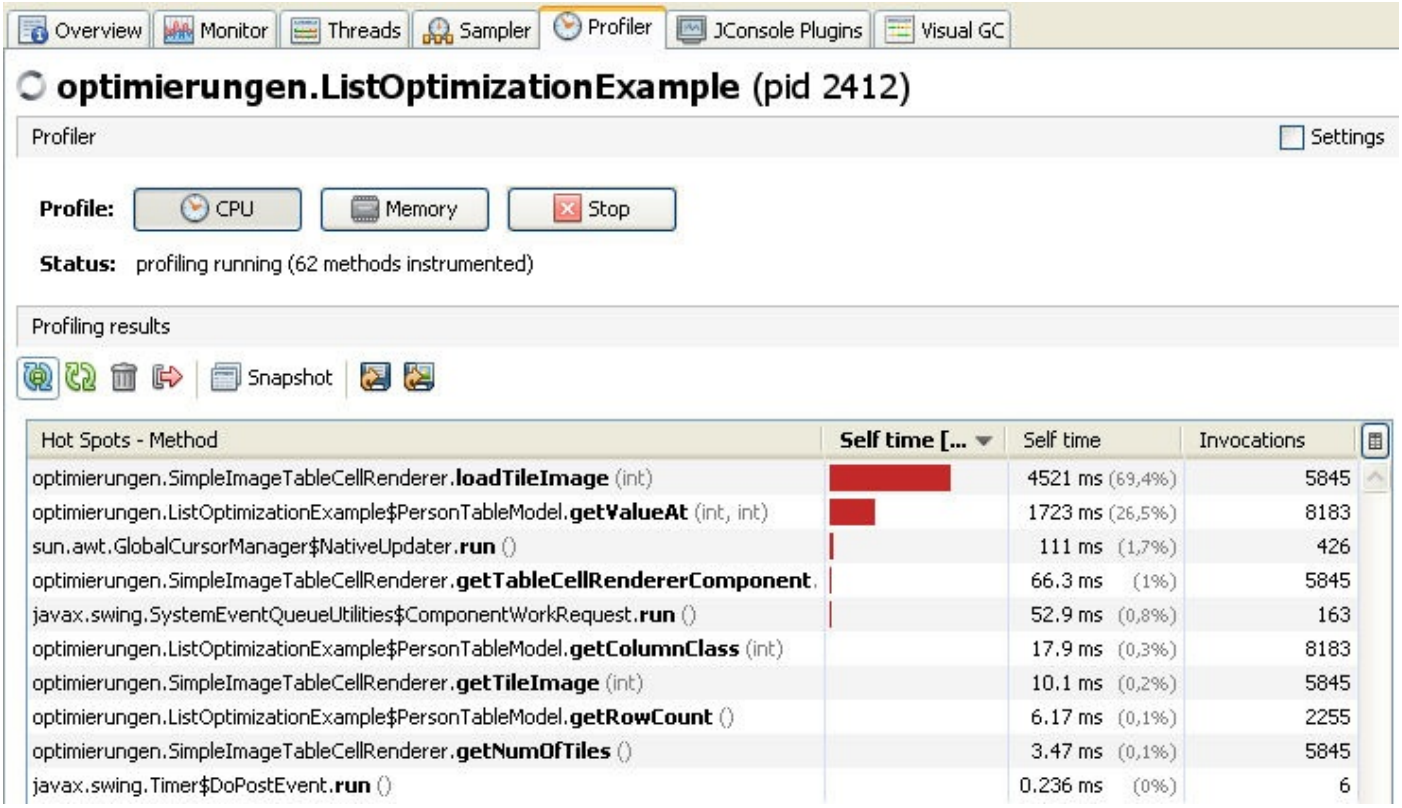


Abbildung 22-6 *Profiling einer* `LinkedList` *mit 50.000 Einträgen*

Zudem erkennen wir beim Betrachten der CPU-Auslastung immer wieder Spitzen bis ca. 50%. Abbildung 22-7 stellt dies dar.

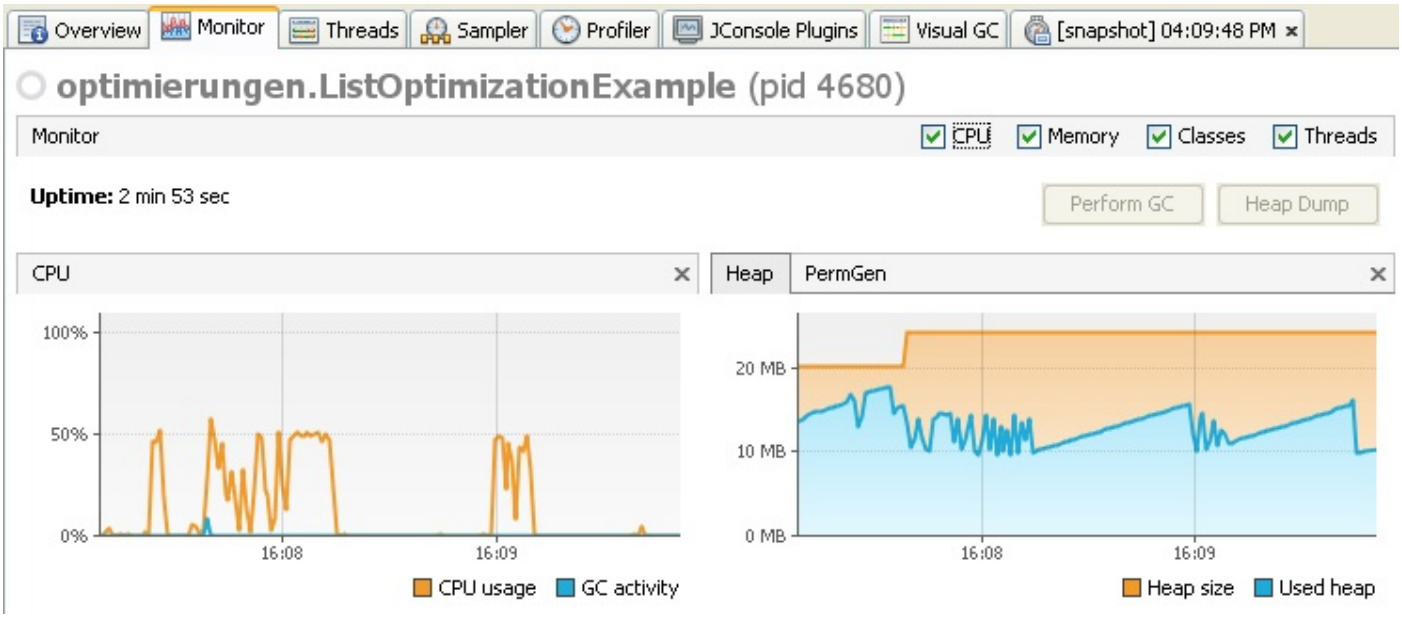


Abbildung 22-7 *Profiling einer* `LinkedList`*: Das CPU-Profil*

Zu den Belastungsspitzen kommt es beim Scrollen durch die Liste. Das ist erschreckend und lässt befürchten, dass bei der Speicherung von wesentlich mehr Datensätzen die Performance weiter einbricht, die Applikation also schlecht skaliert.

I/O-bound-Optimierungen

Durch die Profiling-Messungen erkennen wir als vorrangigen Hotspot die Methode `loadTileImage(int)` der Klasse `SimpleImageTableCellRenderer`. Wir

beginnen dort mit der Analyse und betrachten die Methode genauer:

```
public ImageIcon loadTileImage(final int i)
{
   final int imageNo = (i % getNumOfTiles());
   final File imageFile = new File(PATH_TO_IMAGES, tileFileNames[imageNo]);
   try
   {
      return new ImageIcon(ImageIO.read(imageFile));  // Dateisystemzugriff
   }
   catch (final IOException e)
   {
      return EMPTY_IMAGE; // FALLBACK: EMPTY IMAGE
   }
}
```

Neben der eigentlichen Methode sollten wir auch die Aufrufhierarchie kritisch untersuchen: `getTableCellRendererComponent()`) `getTileImage()`) `loadTileImage()`. Schauen wir auf die Implementierung der Klasse `SimpleImageTableCellRenderer`:

```
public class SimpleImageTableCellRenderer extends DefaultTableCellRenderer
{
    private final String   PATH_TO_IMAGES = "../config/tiles/";
    private final String[] tileFileNames  = { "tile_gras_1.jpg",
                           "tile_gras_2.jpg", "tile_rock_1.jpg",
                           "tile_rock_2.jpg", "tile_water.jpg" };

    private final ImageIcon EMPTY_IMAGE = new ImageIcon(new BufferedImage(1,
1,
                                                BufferedImage.TYPE_INT_RGB));

    public SimpleImageTableCellRenderer()
    {
        setHorizontalTextPosition(JLabel.RIGHT);
        setHorizontalAlignment(JLabel.LEFT);
    }

    public Component getTableCellRendererComponent(JTable table, Object value,
          boolean isSelected, boolean hasFocus, int row, int column)
    {
         super.getTableCellRendererComponent(table, null, isSelected, false,
                                             row, column);
         if (column >= 2)
         {
              // Ermitteln der Bilder
              setIcon(getTileImage(row + column - 2));
         }
         return this;
    }

    public ImageIcon getTileImage(final int i)
    {
         // Direkte Weiterleitung an loadTileImage()
         return loadTileImage(i);
    }

    public ImageIcon loadTileImage(final int i)
    {
        final int imageNo = (i % getNumOfTiles());
```

```
        final File imageFile = new File( PATH_TO_IMAGES,
tileFileNames[ imageNo]);
        try
        {
             // Dateisystemzugriff
             return new ImageIcon( ImageIO. read( imageFile));
        }
        catch ( final IOException e)
        {
            return EMPTY_IMAGE;  // FALLBACK: EMPTY IMAGE
        }
    }

    public final int getNumOfTiles()
    {
        return tileFileNames. length;
    }
}
```

Eine Analyse dieser Klasse ergibt, dass bei jedem Zeichnen der Tabelle mit diesem Renderer durch die Aufrufe von `getTileImage( int)` und danach `loadTileImage( int)` Zugriffe auf das Dateisystem erfolgen. Das ist ein Ansatzpunkt – insbesondere weil lediglich eine feste Anzahl an Grafiken verwendet wird. Folglich kann als Optimierung das Laden der Grafiken besser einmalig bei der Konstruktion des Renderers erfolgen. Ein Caching der Bilder erlaubt, anschließend performant darauf zuzugreifen. Dazu erweitern wir die Renderer-Klasse dahingehend, dass eine Speicherung der Grafiken im Attribut `tileIcons` in Form eines Arrays von `ImageIcon`-Objekten erfolgt. Es entsteht die folgende, optimierte Renderer-Klasse `CachedImageTableCellRenderer`, die das Scrollen deutlich flüssiger macht. Ein Start des Programms LISTOPTIMIZATIONEXAMPLEIOIMPROVED zeigt dies.

```
public final class CachedImageTableCellRenderer extends
                                                SimpleImageTableCellRenderer
{
    private final ImageIcon[ ] tileIcons;

    public CachedImageTableCellRenderer()
    {
        final int numOfBackgrounds = getNumOfTiles();
        tileIcons = new ImageIcon[ numOfBackgrounds];

        for ( int i = 0; i < numOfBackgrounds; i++)
        {
             tileIcons[ i] = loadTileImage( i);
        }
    }

    public ImageIcon getTileImage( final int i)
    {
        final int index = i % getNumOfTiles();
        return tileIcons[ index];
    }
}
```

Listing 22.2 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEIOIMPROVED'**

Möchte man weitere Optimierungen vornehmen, so ist zu bedenken, dass sich als

Folge von Optimierungen sehr häufig das Laufzeitprofil der Applikation verändert und es zu einer Verschiebung von Hotspots kommt: Zuvor erkannte Hotspots können an Bedeutung verlieren und andere Programmteile können zu Hotspots werden. Es ist daher wichtig, erneute Messungen durchzuführen und nicht direkt mit der Optimierung des zuvor ermittelten zweiten Hotspots – hier der Methode `getValueAt(int, int)` – fortzufahren. Eine Folgemessung ergibt die in Abbildung 22-8 dargestellten Hotspots und zeigt, dass in diesem Fall die Methode `getValueAt(int, int)` auch weiterhin Performance-kritisch ist und sich nun sogar an erster Stelle befindet.

Hot Spots - Method	Self time [...	Self time	Invocations
optimierungen.ListOptimizationExample$PersonTableModel.**getValueAt** (int, int)		8956 ...(86,7%)	92071
optimierungen.SimpleImageTableCellRenderer.**getTableCellRendererComponent**...		340 ms (3,3%)	65765
javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.**run** ()		294 ms (2,9%)	1238
optimierungen.CachedImageTableCellRenderer.**getTableCellRendererComponent**...		165 ms (1,6%)	65765
optimierungen.ListOptimizationExample$PersonTableModel.**getColumnClass** (int)		150 ms (1,5%)	92071
optimierungen.CachedImageTableCellRenderer.**getTileImage** (int)		149 ms (1,4%)	131530
javax.swing.Timer$DoPostEvent.**run** ()		98.5 ms (1%)	20
sun.awt.GlobalCursorManager$NativeUpdater.**run** ()		78.9 ms (0,8%)	245
optimierungen.SimpleImageTableCellRenderer.**getNumOfTiles** ()		62.5 ms (0,6%)	131530
optimierungen.ListOptimizationExample$PersonTableModel.**getRowCount** ()		34.8 ms (0,3%)	19374

Abbildung 22-8 *Profiling einer* `LinkedList` *mit verbessertem I/O (50.000 Einträge)*

Die in der ersten Messung als primär problematisch erkannte Methode `loadTileImage(int)` spielt nach der Optimierung für die Performance praktisch keine Rolle mehr. Tatsächlich ist sie diesbezüglich sogar derart unbedeutend geworden, dass sie nicht einmal mehr in der abgebildeten Hotspot-Liste auftaucht.

Memory-bound-Optimierungen

Wie nun auch durch Messung nachgewiesen, stellt die Methode `getValueAt(int, int)` immer noch den limitierenden Faktor dar. Allerdings sind die Auswirkungen auf die Performance bei 50.000 Datensätzen relativ gering: Das Scrolling könnte noch ein wenig flüssiger erfolgen. Erst wenn man die Anzahl der Datensätze wie im folgenden Listing verdreifacht, also auf 150.000 `Person`-Objekte erhöht, wird das Scrolling zäher.

```
public static void main(final String[] args)
{
    final JFrame demoframe = new ListOptimizationExample(150_000,  // 150.000
                                 ListType.LinkedList,
                                 new CachedImageTableCellRenderer());

    demoframe.setSize(700, 500);
    demoframe.setVisible(true);
}
```

Listing 22.3 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEIOIMPROVED2'**

Um das Antwortzeitverhalten beim Scrolling noch weiter zu verbessern, sollten wir die Methode `getValueAt(int, int)` analysieren. Schauen wir auf deren Realisierung:

```
public Object getValueAt(final int rowIndex, final int columnIndex)
{
    final Person person = persons.get(rowIndex);

    if (columnIndex == 0)
        return person.getName();
    if (columnIndex == 1)
        return person.getGender();

    return ""; // "" => Kein Text für Bilder
}
```

Das Attribut `persons` ist eine `List<Person>` und verwendet hier eine `LinkedList<Person>`. Auf den ersten Blick ist daran nichts auszusetzen, aber die Ausführungszeit ist nicht zufriedenstellend. Die Grund ist einfach: Eine `LinkedList<E>` ist für indizierte Zugriffe nicht optimal. Aufgrund der Performance-Betrachtung verschiedener Listenimplementierungen in Abschnitt 22.2.1 wissen wir, dass eine `LinkedList<E>` zum Zugriff auf Elemente an einer beliebigen Position immer eine Iteration durch alle Vorgänger benötigt. Dadurch entstehen Kosten von $O(n)$. Beim Zeichnen der Tabelle fallen diese Kosten für jede sichtbare Zeile erneut an. Verwendet man statt einer `LinkedList<E>` eine `ArrayList<E>`, so sollte sich dies extrem positiv auswirken, da Zugriffe nur noch Kosten von $O(1)$ verursachen. Überprüfen wir diese Vermutung, indem wir eine `ArrayList<Person>` zur Datenspeicherung verwenden:

```
public static void main(final String[] args)
{
    final JFrame demoframe = new ListOptimizationExample(150_000,
                                 ListType.ArrayList, // ArrayList
                                 new CachedImageTableCellRenderer());

    demoframe.setSize(700, 500);
    demoframe.setVisible(true);
}
```

Listing 22.4 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEDSIMPROVED'**

Durch diese Modifikation ergibt sich die erwartete Verbesserung: Die CPU-Belastung bleibt selbst bei vielen Scroll-Aktionen immer unter 25% und außerdem ist kein wirklicher Hotspot mehr feststellbar. Selbst bei über 100.000 Aufrufen der jeweiligen Methoden bleiben die Ausführungszeiten unter 1 Sekunde (vgl. Abbildung 22-9).

Hot Spots - Method	Self time [...	Self time	Invocations
optimierungen.SimpleImageTableCellRenderer.**getTableCellRendererComponent**..		946 ms (33%)	132047
javax.swing.SystemEventQueueUtilities$ComponentWorkRequest.**run** ()		647 ms (22,6%)	3316
optimierungen.ListOptimizationExample$PersonTableModel.**getColumnClass** (int)		388 ms (13,6%)	184865
optimierungen.ListOptimizationExample$PersonTableModel.**getValueAt** (int, int)		290 ms (10,1%)	184865
optimierungen.CachedImageTableCellRenderer.**getTileImage** (int)		228 ms (8%)	132047
sun.awt.GlobalCursorManager$NativeUpdater.**run** ()		146 ms (5,1%)	509
optimierungen.SimpleImageTableCellRenderer.**getNumOfTiles** ()		121 ms (4,3%)	132047
optimierungen.ListOptimizationExample$PersonTableModel.**getRowCount** ()		95.2 ms (3,3%)	44595

Abbildung 22-9 *Profiling einer* `ArrayList` *mit 150.000 Einträgen*

Die erzielte Verbesserung wird noch deutlicher, wenn man nun die Anzahl der

Datensätze auf 1 Million erhöht. Die Performance ist nicht merklich schlechter als zuvor für 150.000 Einträge. Überprüfen Sie es selbst, indem Sie folgendes Programm LISTOPTIMIZATIONEXAMPLEONEMILLION ausführen:

```
public static void main(final String[] args)
{
   final JFrame demoframe = new ListOptimizationExample(1_000_000, //
1.000.000
                                ListType.ArrayList,
                                new CachedImageTableCellRenderer());

   demoframe.setSize(700, 500);
   demoframe.setVisible(true);
}
```

Listing 22.5 *Ausführbar als* **'LISTOPTIMIZATIONEXAMPLEONEMILLION'**

Fazit

Anhand eines Beispiels habe ich das grundsätzliche Vorgehen bei Optimierungen vorgestellt: Man beginnt mit Messungen, um die tatsächlich kritischen Bereiche zu bestimmen. Anschließend werden diese einzeln nacheinander bearbeitet. Durch nachfolgende Messungen stellt man dann sicher, dass es wirklich zu Verbesserungen gekommen ist. Dieses Vorgehen wiederholt man, bis eine zufriedenstellende Performance erreicht ist.

22.5 I/O-bound-Optimierungen

In diesem Abschnitt betrachten wir Möglichkeiten, die Ein- und Ausgabe zu beschleunigen. Interagiert ein Programm mit dem Dateisystem oder einem Netzwerk, so sind signifikante Performance-Steigerungen zu erreichen, wenn Zugriffe vermieden oder zu versendende Daten gepuffert werden. Im Folgenden wollen wir diese und einige weitere Techniken und ihre Auswirkungen auf die Performance genauer kennenlernen.

22.5.1 Technik – Wahl passender Strategien

Bei der Interaktion mit einem Netzwerk kann man sowohl durch Pufferung als auch durch eine geschickte Wahl der Repräsentationsform zu übertragender Daten die Performance verbessern. Beispiele dafür sind der Einsatz von Dekorierern zur Pufferung sowie die Anpassung des Serialisierungsvorgangs.

Optimierungen beim Einsatz von Streams

Das Problem beim direkten (naiven) Einsatz von Streams ist, dass sowohl das Lesen als auch das Schreiben ohne Pufferung erfolgt, d. h., es wird genau ein Byte zur Zeit

verarbeitet. Gleiches gilt analog für die Subklassen von `Reader` und `Writer`, die zeichenbasiert arbeiten. Eine Datei byte- bzw. zeichenweise zu lesen ist allerdings nicht sehr performant. Durch Einsatz von Dekorierern zur Pufferung werden jeweils größere Datenblöcke gelesen und geschrieben. Welch enorme Geschwindigkeitssteigerungen sich dadurch erzielen lassen, werden wir am Beispiel des Kopierens von Dateien kennenlernen.

Folgende Varianten zum Kopieren wurden untersucht:

- **Einsatz einer byteweisen Kopie (A)** – Nachfolgend wird die Hilfsmethode `copyBytewise(InputStream, OutputStream)` entwickelt, die Daten Byte für Byte kopiert, indem zunächst ein Byte gelesen und dann geschrieben wird:

```
public static void copyBytewise(final InputStream is, final OutputStream
os)
        throws IOException
{
    int data = -1;
    while ((data = is.read()) != -1)
    {
        os.write(data);
    }
    os.flush();
}
```

- **Einsatz eines gepufferten Streams (B)** – Eine Pufferung der Zugriffe erreicht man durch Ummantelung mit den Dekorierern `BufferedInputStream` bzw. `BufferedOutputStream`. Zum Ressourcen-Handling nutzen wir ARM, wodurch keine direkten Aufrufe von `close()`-Methoden der Streams erforderlich sind:

```
try (final InputStream bufferedIn = new BufferedInputStream(inStream),
     final OutputStream bufferedOut = new BufferedOutputStream(outStream))
{
    copyByteWise(bufferedIn, bufferedOut);
}
```

- **Einsatz eines eigenen Puffers (C)** – Statt die gepufferten Streams aus dem JDK zu benutzen, kann man eine Pufferung selbst implementieren. Diese Funktionalität kapselt man zweckmäßig innerhalb einer Methode. Dies wird durch die Methode `copyOwnBuffering(InputStream, OutputStream)` wie folgt realisiert:

```
public static void copyOwnBuffering(final InputStream is,
                                     final OutputStream os)
                    throws IOException
{
    final byte[] buffer = new byte[BUFFER_SIZE];
    int length = -1;
    while ((length = is.read(buffer, 0, BUFFER_SIZE)) != -1)
    {
        os.write(buffer, 0, length);
    }
    os.flush();
}
```

- **Einsatz der Klasse `FileChannel` (D)** – Zum Kopieren nutzen wir als letzte Variante die Klasse `FileChannel` und deren Methode `transferTo(long, long, java.nio.channels.WritableByteChannel)`, die direkt auf Betriebssystemebene arbeitet:

```
final FileChannel sourceChannel = inStream.getChannel();
final FileChannel destChannel = outStream.getChannel();

sourceChannel.transferTo(0, sourceChannel.size(), destChannel);
```

Als Eingabe dienen zwei PDF-Dateien unterschiedlicher Größe von ca. 710 KB und ca. 10,8 MB. In der folgenden Tabelle 22-4 sind die Ergebnisse verschiedener Performance-Messungen zusammengefasst.

Tabelle 22-4 *Performance-Gewinn durch Pufferung beim Kopieren*

Größe	Variante				Faktor			
	A	B	C	D	A-B	A-C	B-C	C-D
710 KB – JDK 7	4.116 ms	21 ms	1 ms	3 ms	196	4.116	31	0.33
710 KB – JDK 8	4.312 ms	22 ms	2 ms	3 ms	196	2.156	11	0.66
10,8 MB – JDK 7	83.479 ms	79 ms	22 ms	4 ms	1.057	3.795	3,6	5,5
10,8 MB – JDK 8	88.700 ms	90 ms	26 ms	5 ms	986	3.412	3.5	5,2

Wie man leicht sieht, ist bereits durch den Einsatz von Pufferung (Variante B) ein enormer Performance-Gewinn zu erzielen, der etwa zwischen Faktor 200 bis 1.000 im Vergleich zum ungepufferten Kopieren (Variante A) liegt. Eine deutliche Beschleunigung um zusätzlich etwa Faktor 3 bis 4 und deutlich mehr kann man nochmals erzielen, wenn man die Pufferung selbst durchführt (Variante C). Etwas verwunderlich ist, dass der Einsatz von Channels langsamer als der Einsatz der zuvor vorgestellten gepufferten Variante ist, obwohl hier direkt auf Betriebssystemebene gearbeitet wird. Hieran sieht man, dass sich der Aufwand für Spezialanpassungen nicht in jedem Fall lohnt. Hendrik Schreiber hat diesen Effekt bereits 2002 für das JDK 1.4 in seinem Buch »Performant Java programmieren« [78] beschrieben. Interessanterweise sind bis JDK 6 keine nennenswerten Verbesserungen erzielt worden – bei größeren Dateien beobachtet man indes einen negativen Effekt. Mit JDK 7 und insbesondere JDK 8 wurden hier deutliche Verbesserungen erzielt und Channels (Variante D) ziehen mit der Geschwindigkeit eigener Pufferungen gleich

bzw. übertreffen sie teilweise. Nach wie vor gilt trotzdem, dass sehr häufig die Variante mit eigener Pufferung eine sehr gute oder oft sogar die beste Wahl darstellt.

Kompression und Zip-Archive

Zur Optimierung von I/O ist eine weitere Idee, die Setup- und Transferzeiten zu reduzieren, indem man zu übertragende Daten oder Dateien in einem Zip-Archiv zusammenfasst. Jeder kennt es vom Kopieren von Dateien auf eine externe Festplatte oder auf einen USB-Stick: Im Dateisystem ist das Kopieren einer größeren Datei viel schneller als die Übertragung vieler kleinerer Dateien. Es lohnt sich in der Regel, zunächst diese Dateien in einem Zip-Archiv zusammenzufassen, selbst wenn man dabei keine Kompression vornimmt. Es sind dann zwar zwei zusätzliche Schritte (Zip-Archiv erzeugen und auslesen) notwendig, aber trotzdem ist diese Variante für viele kleine Dateien deutlich günstiger als einzelne Dateitransfers. Diese Ideen liegen auch der Speicherung in JAR-Dateien zugrunde.

Bei Übertragungen über ein Netzwerk kann man zusätzlich die Komprimierung einschalten. Im Normalfall sollten die Einsparungen durch die weniger zu übertragenden Bytes die Zeiten zum Komprimieren und Entpacken mehr als ausgleichen. Dafür können die im JDK im Package `java.util.zip` definierten Klassen `ZipInputStream` und `ZipOutputStream` genutzt werden, um einen Stream um genau diese Zip-Funktionalität zu erweitern.

Optimierungen der Serialisierung

Bei der Serialisierung werden Objekte in einen `ObjectOutputStream` geschrieben bzw. aus einem `ObjectInputStream` eingelesen. Die dazu notwendige Transformation von Objekten in eine Folge von Bytes bzw. umgekehrt erfolgt automatisch, wenn eine Klasse das Interface `Serializable` erfüllt (vgl. Abschnitt 8.3).

Diese Automatik ist für viele Fälle sehr praktisch, kann sich aber in einigen Situationen negativ auf die Performance auswirken. Der Grund dafür ist, dass die dahinterliegende Komplexität vor dem Aufrufer versteckt wird: So bleibt auch verborgen, dass immer der gesamte Objektgraph gespeichert bzw. erzeugt wird. Dieser kann beim Einsatz von Vererbung und vielen aggregierten Objekten durchaus komplex sein, wodurch auch der Aufwand zur Serialisierung steigt. Anders formuliert: Wenn Serialisierung eingesetzt wird, ist es sinnvoll, zu überlegen, ob wirklich alle Zustandsinformationen gespeichert werden müssen und welche Informationen bei einem späteren Einlesen eventuell ohne Probleme berechnet oder anderweitig ermittelt werden können.

Betrachten wir dies konkret für Swing-GUI-Komponenten. Deren Serialisierung ist im Vergleich zu anderen, normalen Objekten sehr teuer: GUI-Komponenten benötigen mehrere KB, da sie ihre komplette AWT- und Swing-Ableitungshierarchie

enthalten. Häufig ist aber eine Serialisierung dieser Daten nicht unbedingt notwendig bzw. sogar nicht sinnvoll. Für ein Label wäre es in vielen Fällen ausreichend, nur den Text zu speichern. Daraus leiten wir ab, dass es oftmals praktischer und effizienter ist, nur die Datenmodelle zu speichern. Werden GUI-Komponenten in eigenen, zu serialisierenden Objekten aggregiert, so sollten diese explizit vom Vorgang der Serialisierung per Schlüsselwort `transient` ausgeschlossen werden. Als Folge sind allerdings spezielle Anpassungen beim Einlesen durchzuführen. Details dazu beschreibt Abschnitt 8.3.2.

22.5.2 Technik – Caching und Pooling

Man setzt Caching ein, um häufig benötigte Daten in einem Speicher mit kurzen Zugriffszeiten vorrätig zu halten. Dadurch lassen sich teilweise enorme Verbesserungen der Ausführungszeit erreichen. Gleiches gilt für eine gepufferte Verarbeitung. Beide Themen wurden bereits ausführlich betrachtet (vgl. Abschnitt 22.4 für das Beispiel der Grafikdateien). Daher erfolgt in diesem Abschnitt keine weitere Beschreibung des Cachings für die Ein- und Ausgabe.

22.5.3 Technik – Vermeidung unnötiger Aktionen

Wie in Abschnitt 22.3 ausführlich am Beispiel der Lazy Initialization vorgestellt, kann das Vermeiden nicht benötigter Aktionen sowohl die Ausführungszeit als auch den Speicherbedarf einer Applikation reduzieren. Beispiele sind das Prüfen des Log-Levels vor Log-Ausgaben sowie die Reduktion von Remote Calls in verteilten Applikationen.

Optimierungen beim Logging

Verschiedene Techniken zur Optimierung beim Logging wurden bereits in Abschnitt 6.4.2 vorgestellt. Hier wird daher nur die grundsätzliche Idee nochmal kurz zusammengefasst: Man versucht, das Aufbereiten und das Schreiben von Log-Ausgaben möglichst zu reduzieren. Dazu wird vor komplexeren oder umfangreichen Log-Ausgaben zunächst das Log-Level überprüft, bevor Log-Ausgaben erfolgen. Somit vermeidet man einerseits eine Erzeugung der Texte zur Log-Ausgabe, wodurch weniger temporäre Stringobjekte entstehen. Andererseits sorgen sinnvoll gewählte Log-Level dafür, dass seltener in eine Log-Datei geschrieben wird. Dies kommt zudem der Lesbarkeit und Auswertbarkeit der entstehenden Log-Dateien zugute.

Reduktion von Remote Calls

Im Netzwerk und in verteilten Applikationen kann man durch Unachtsamkeit leicht

Performance-Probleme auslösen. Ich werde im Folgenden kurz vorstellen, welche Fallstricke auf dem Weg zu einem gelungenen und gleichzeitig performanten API für eine verteilte Applikation lauern.

Wenn eine Interaktion mit anderen Systemen erfolgen soll, so müssen die Klassen, die vom Netzwerk aus angesprochen werden sollen, dazu Remote-Interfaces anbieten. Wenn man diese externen Schnittstellen einer Applikation ähnlich zu den Schnittstellen innerhalb eines Programms definiert, so entstehen viele feingranulare Methodenaufrufe, die alle als einzelne Kommandos über das Netzwerk versendet werden müssen. Als Folge wird für jeden Methodenaufruf mindestens ein Netzwerktelegramm erzeugt. Abbildung 22-10 deutet dies an.

Analogie: Optimierung in der Realität

Mögliche Performance-Probleme von Remote Calls lassen sich sehr anschaulich an einem Beispiel aus der Realität verdeutlichen. Man stelle sich vor, es wäre ein Fußballabend mit einigen Freunden in vollem Gange. Irgendwann möchte jeder Gast natürlich auch etwas trinken. Ziemlich aufwendig wäre es, wenn man bei jeder Nachfrage nach einem gekühlten Getränk immer wieder zum Supermarkt fahren würde, um dieses dort zu kaufen. In der Realität ist sofort einsichtig, dass diese Verhaltensweise extrem unsinnig und zeitlich zudem ungünstig ist. Selbstverständlich fährt man einmal zum Supermarkt und deckt sich mit einer Menge an Getränken ein. So offensichtlich diese Vorgehensweise in der Realität auch ist, so ist eine Umsetzung in entsprechende Remote Calls nicht ganz so trivial.

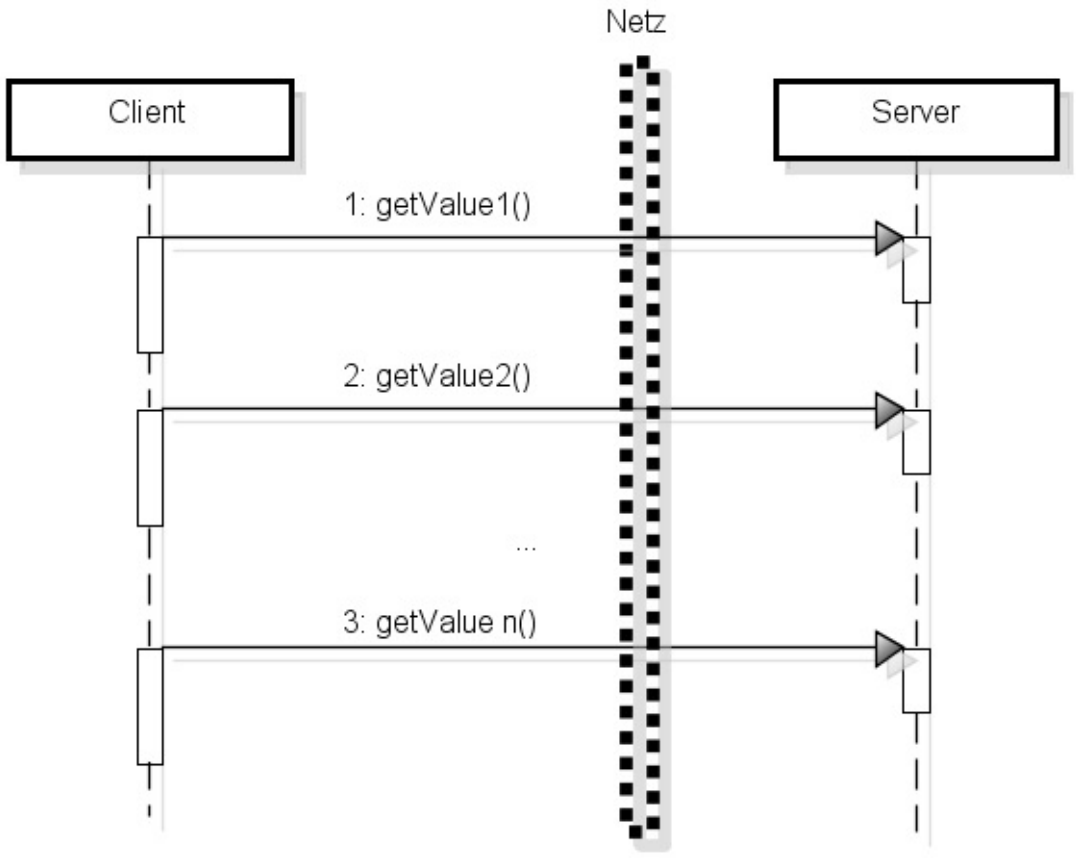


Abbildung 22-10 *Remote Calls*

Folgende Abhilfemaßnahmen bieten sich an:

- **Aufrufe zusammenfassen** – Nachrichten werden zu einer größeren Nachricht zusammengefasst, wie wir dies am Beispiel der Klasse `MessageConcatenator` im Kapitel 20 kennengelernt haben.
- **Daten zusammenfassen** – Daten werden zu einem größeren Datenpaket zusammengefasst. Dies kann gemäß dem Muster VALUE OBJECT (vgl. Abschnitt 3.4.5) in seiner Ausprägung als DATA TRANSFER OBJECT (DTO) geschehen.

Aufrufe zusammenfassen

Um die Anzahl von Remote Calls möglichst zu reduzieren, kann man Methodenaufrufe zusammenfassen. Dies geht besonders gut für `set()`-Methoden, denen man dann statt eines Parameters mehrere übergibt:

```
interface SimpleWriteIF
{
    void setValue1(int value1)
    void setValue2(int value2)
    void setValue3(int value3)
}

interface OptimizedRemoteWriteIF
{
    void setValues(int value1, int value2, int value3)
}
```

Bei `get()`-Methoden ist diese Technik nicht ganz so einfach einzusetzen, da nur ein Wert zurückgegeben werden kann.

Daten zusammenfassen

Man kann sich ein Data Transfer Object (DTO) bauen, das die Ergebnisse mehrerer `get()`-Aufrufe bündelt. Im einfachsten Fall speichert man dort die Attribute einzeln. Eine alternative Technik besteht darin, eine `HashMap<K, V>` mit benannten Attributen zu nutzen. Für beide Arten der Umsetzung werden dann mehrere feingranulare Aufrufe in einen Aufruf mit einem DTO umgewandelt und dann die entsprechenden Antworten generiert, wie dies in Abbildung 22-11 zu sehen ist.

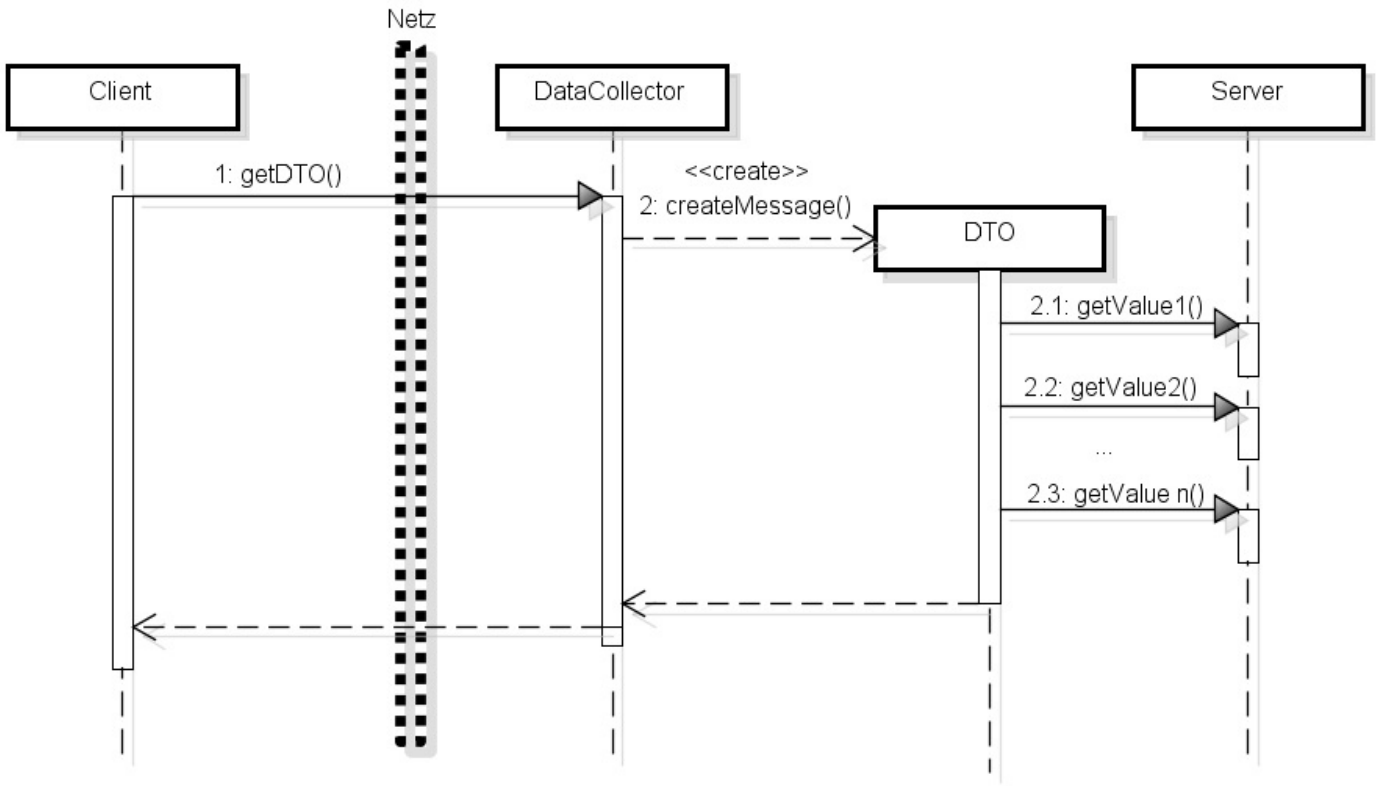


Abbildung 22-11 *Optimierung von Remote Calls*

Zum Transport über ein Netzwerk muss das DTO das Interface `Serializable` erfüllen und sollte zudem möglichst leichtgewichtig sein, indem bevorzugt primitive Typen als Datenattribute Anwendung finden. Abschnitt 22.6.4 beschreibt dies genauer.

Auswirkungen

Fasst man `set()`-Aufrufe zusammen, so reduziert man *n* Remote Calls auf lediglich einen Aufruf. Allerdings wird dadurch das Interface recht stark durch technische Gegebenheiten beeinflusst. In solchen Fällen bietet sich eine Abstraktionsebene vor dem eigentlichen Netzwerkzugriff an. Dies könnte durch ein Proxy-Objekt gemäß dem PROXY-Muster (vgl. Abschnitt 18.3.6) realisiert werden.

Fasst man `get()`-Aufrufe zusammen, so kann man auch hier wieder *n* Remote Calls auf lediglich einen Aufruf reduzieren. Allerdings muss dann ein DTO eingesetzt und während der Kommunikation gefüllt werden. Dazu sind nun statt *n* `get()`-Aufrufen die doppelte Anzahl notwendig: zum einen *n* Aufrufe aufseiten des Servers zum Befüllen des DTO, zum anderen bis zu *n* weitere Aufrufe aufseiten des Klienten zum Ermitteln der Werte. Da es sich in beiden Fällen um vergleichsweise günstige lokale Aufrufe handelt, ist dies ein geringer Preis für die eingesparten Remote Calls.

22.6 Memory-bound-Optimierungen

Die Größe des Hauptspeichers, der einer Applikation zur Verfügung steht, kann großen Einfluss auf die Performance haben. Dies gilt vor allem für Systeme mit wenig

Arbeitsspeicher oder für Applikationen, die mit sehr großen Datenmengen arbeiten. Die Größe des ausführbaren Programms und dessen Speicherverbrauch zur Laufzeit sind daher weitere wichtige Aspekte bei Optimierungen.

Abschnitt 22.6.1 geht kurz auf die Auswirkungen von Größe und Nutzung des Hauptspeichers ein und zeigt einige Möglichkeiten, die Garbage Collection zu optimieren. Abschnitt 22.6.2 beschreibt die Vorteile, die sich ergeben, wenn man häufig benutzte Objekte zwischenspeichert (***Objekt-Caching***) bzw. alternativ bereits erzeugte Objekte recycelt und mit geänderter Parameterbelegung wiederverwendet (***Objekt-Pooling***). Konkret wird dies für die Stringverwaltung von Java genutzt. Abschnitt 22.6.3 stellt weitere mögliche Optimierungspotenziale bei Strings vor. Abschließend wird in Abschnitt 22.6.4 erläutert, warum man für Attribute in der Regel primitive Datentypen den entsprechenden Wrapper-Klassen vorziehen sollte.

22.6.1 Technik – Wahl passender Strategien

Größe und Nutzung des Hauptspeichers

Steigt der von einer Applikation verbrauchte Speicher ständig bis nahe an die der JVM zugewiesene Grenze, so kann sich dies äußerst negativ auf die Programmantwortzeit auswirken. Wurde die maximale Größe des benötigten Speichers initial falsch eingeschätzt und ist deswegen das Programm infolge häufiger Garbage Collections träge, kann man eine Beschleunigung erreichen, indem man per Aufrufparameter `-Xmx` beim Programmstart mehr Speicher bereitstellt. Dieses Vorgehen ist hilfreich, stellt jedoch keine allgemeingültige Lösung dar. Man vermeidet damit lediglich, sich mit vermeintlichen Performance-Problemen in Programmteilen zu beschäftigen, die nur zufällig durch schlecht gewählte Rahmenbedingungen entstanden sind. Nach dieser, in der Regel einmalig wirksamen Optimierungsmaßnahme kann man sich dann tatsächlich vorliegenden Performance-Problemen widmen. Ist eine Applikation aber unersättlich bezüglich des Speichers, so sollte man nicht einfach iterativ die Speichergröße anpassen, sondern die verursachenden Stellen aufspüren und beheben. Einige dafür passende Techniken wurden bereits in Abschnitt 22.2 als Hinweise zur Wahl geeigneter Datenstrukturen und Rückgabewerte bei großen Datenmengen diskutiert.

Mythos: Viel hilft viel

Um eine gute Performance zu erzielen, ist es in der Regel nicht sinnvoll, der JVM einfach möglichst viel Speicher zuzuteilen. Dies führt teilweise nur dazu, dass die Garbage Collection mehr Zeit für die Speicherbereinigung benötigt. Ratsam ist es, zuvor den tatsächlichen Speicherverbrauch zu analysieren.

Tipp: Auswirkungen von Threads

Wenn ein Programm sehr viele Threads zur Verarbeitung mehrerer unabhängiger Aufgaben startet, so wirkt sich dies auf den Speicherbedarf aus. Threads werden normalerweise auf Betriebssystem-Threads abgebildet und sind dadurch in ihrer Anzahl begrenzt. Zudem benötigen Threads weitere Systemressourcen sowie eigenen Speicher, etwa zur Bereitstellung eines Stacks.

Bei Tausenden zu startenden Threads kann man einer Applikation dann weniger Speicher über die Option `-Xmx` zur Verfügung stellen. Es gilt: Je weniger Threads, desto größer kann der Wert von `-Xmx` gewählt werden und umgekehrt. Die Ursache liegt darin, dass Threads und genutzte Stacks nicht im Speicherbereich der JVM, sondern im Speicher des Betriebssystems verwaltet werden.

Optimierungen der Garbage Collection

Java befreit den Entwickler mit der automatischen Garbage Collection von der manuellen Freigabe des Speichers. Allerdings hat man durch den Automatismus nur wenig Einflussmöglichkeiten auf den Ablauf bzw. den Zeitpunkt von Garbage Collections. Man kann die Garbage Collection aber durch diverse Aufrufparameter der JVM parametrieren, etwa verschiedene Aufräumalgorithmen wählen.

Statt aufwendige Überlegungen anzustellen, ist es einfacher und pragmatischer, die verschiedenen Implementierungen der Garbage-Collection-Algorithmen und deren Einfluss auf die Performance auszuprobieren, indem man die Applikation mit den folgenden Parametern startet und beobachtet:

- `-XX: +UseSerialGC` – Serial Collector
- `-XX: +UseParallelGC` – Parallel Collector
- `-XX: +UseParNewGC` – Compacting Parallel Collector
- `-XX: +UseConcMarkSweepGC` – Concurrent Mark and Sweep Collector
- `-XX: +UseG1GC` – Garbage First (G1) Collector[9]

Um die Auswirkungen sichtbar zu machen, sollte man über `-verbose: gc` die Protokollierung der Garbage-Collection-Vorgänge aktivieren. Den Detailgrad der Ausgabe kann man über folgende Parameter steuern:

- `-XX: +PrintGCDetails`
- `-XX: +PrintGCTimeStamps`
- `-XX: +PrintGCDateStamps`

Damit eine Auswertung sinnvoll möglich wird, kann die Ausgabe in eine Datei umgeleitet werden, etwa in die Datei `gc.log` durch die JVM-Aufrufparameter `-Xloggc: gc.log`. Eine solche Ausgabe könnte etwa wie folgt aussehen:

```
01-02 15:57:57 INFO (MenuActionHandler    ) - executing MenuAction:
load_layout
[Full GC 9052K->2614K(260160K), 0.0818330 secs]
```

```
[GC 36492K->21178K(260224K), 0.0045101 secs]
…
[GC 248885K->232630K(260224K), 0.0025241 secs]
01-02 15:58:26 INFO (MenuActionHandler    ) - executing MenuAction:
load_layout
…
[GC 399533K->391320K(433224K), 0.0257720 secs]
[GC 434840K->408968K(701164K), 0.0258944 secs]
01-02 15:59:01 INFO (MenuActionHandler    ) - executing MenuAction: close
01-02 15:59:16 INFO (MenuActionHandler    ) - executing MenuAction:
      load_template
[Full GC 482060K->2647K(701164K), 0.1094170 secs]
```

Ein erster Überblick und eine manuelle Auswertung ist so bereits möglich, wenn man die Bedeutung der jeweiligen Werte kennt. Betrachten wir dies exemplarisch an folgenden zwei Zeilen:

```
[Full GC 9052K->2614K(260160K), 0.0818330 secs]
[GC 434840K->408968K(701164K), 0.0258944 secs]

Typ (Full GC oder GC), [Heap vorher]->[Heap nachher] (Gesamtheap), Zeitdauer
```

Jede Zeile liefert zunächst die Information über die Art der Speicherbereinigung (`GC` oder `Full GC`, die für eine Minor GC bzw. Major GC stehen). Anschließend wird der belegte Speicher vor und nach der Garbage Collection ausgegeben. Für den markierten Eintrag `Full GC` wird der Speicher von `9052K` auf `2614K` reduziert. Der JVM standen zu dem Zeitpunkt ein Heap-Speicher der Größe `260160K` zur Verfügung. Abschließend wird die Dauer der Garbage Collection angegeben, hier `0.0818330 secs`.

Mit diesem Wissen kann man erste Auswertungen vornehmen. Weitere Informationen finden Sie unter `http://www.oracle.com/technetwork/java/gc-tuning-5-138395.html` und `http://java.sun.com/developer/technicalArticles/Programming/t`

Viel bequemer als eine manuelle Auswertung ist es allerdings, Tools zu benutzen. Es empfiehlt sich, mit dem in Abschnitt 22.1.4 erwähnten Tool `VisualGC` die Garbage Collection zu beobachten. Die Kenntnis der in Abschnitt 8.5 beschriebenen Details zur Garbage Collection ist dabei sehr hilfreich.

22.6.2 Technik – Caching und Pooling

Objekt-Caching

Wie bereits für die Ein- und Ausgabe erwähnt, setzt man Caching ein, um häufig benötigte Daten in einem schnell zugreifbaren Speicher zwischenzuspeichern. Bei umfangreichen Datenmengen und dadurch bedingt längeren Zugriffszeiten auf Objekte kann man sich einen LRU-Cache bauen, der die zuletzt benutzten Objekte zwischenspeichert.

Zur Demonstration des Einsatzes von Caching auf die Performance betrachten wir hier eine Liste `persons` von `Person`-Objekten. Darin erfolgt auf zwei verschiedene Arten eine Suche nach Objekten anhand ihres Namens. Die erste Realisierung nutzt eine lineare Suche und wird durch die nachfolgend gezeigte Methode `findPersonByName(String)` implementiert:

```
private Person findPersonByName(final String name)
{
    for (final Person current : persons)
    {
        if (current.getName().equals(name))
        {
            return current;
        }
    }
    return null;
}
```

Im ungünstigsten Fall ist dabei ein vollständiges Durchlaufen der Liste nötig, um festzustellen, ob ein passendes `Person`-Objekt vorhanden ist oder nicht. Zur Optimierung setzen wir die aus Abschnitt 5.1.10 bekannte generische Klasse `LruLinkedHashMap<K, V>` ein. Als Abstraktion erfolgen optimierte Zugriffe auf die gespeicherten `Person`-Objekte über eine Zugriffsmethode `getPerson(String)`. Diese nutzt wiederum initial die Methode `findPersonByName(String)`, um den Cache transparent für die Applikation zu füllen. Ein erster Zugriff auf ein `Person`-Objekt kostet daher maximal $O(n)$. Alle Folgezugriffe auf dieses oder ein anderes im Cache gespeichertes `Person`-Objekt sind dann aber sehr schnell in $O(1)$ möglich, da hier ein Look-up in der `LruLinkedHashMap<String, Person>` erfolgt:

```
public Person getPerson(final String name)
{
    Person cachedPerson = cacheMap.get(name);

    if (cachedPerson != null)
    {
        return cachedPerson;
    }

    final Person person = findPersonByName(name);
    if (person != null)
    {
        cacheMap.put(name, person);
    }
    return person;
}
```

Mit folgender `main()`-Methode erzeugen wir 10.000 `Person`-Objekte. Anschließend werden als Testszenario 1.000, 10.000 und 100.000 Zugriffe auf eine Menge von immer denselben 10 `Person`-Objekten durchgeführt. Ein Cache der Größe 10 wäre demnach ausreichend, um die Zugriffe extrem zu beschleunigen.

Das folgende Listing zeigt die Klasse `LruListCacheExample`, die die zuvor beschriebene Funktionalität der Performance-Messung umsetzt:

```
public final class LruListCacheExample
{
    private final LruLinkedHashMap<String, Person> cacheMap;
    private final List<Person>                     persons;

    public LruListCacheExample( final List<Person> persons, final int
cacheSize)
    {
        this.cacheMap = new LruLinkedHashMap<>( cacheSize);
        this.persons = persons;
    }

    public static void performTests( final LruListCacheExample cache,
                                     final String info, final boolean useCache)
    {
        final long[] maxIterationCount = { 1_000, 10_000, 100_000 };
        for ( long max : maxIterationCount)
        {
            System.out.println( "Element count " + max);

            PerformanceUtils.startMeasure( info);
            for ( long i = 0; i < max; i++)
            {
                for ( int j = 0; j < 10; j++)
                {
                    final String name = "Person " + ( j* 1000);
                    if ( useCache)
                    {
                        cache.getPerson( name);
                    }
                    else
                    {
                        cache.findPersonByName( name);
                    }
                }
            }
            PerformanceUtils.stopMeasure( info);
            PerformanceUtils.printTimingResult( info);
        }
    }

    public static void main( final String[] args)
    {
        final int PERSON_COUNT = 10000;

        final List<Person> persons = new ArrayList<>();

        for ( int i = 0; i < PERSON_COUNT; i++)
        {
            final Person newPerson = new Person( "Person " + i, Gender.MALE);
            persons.add( newPerson);
        }

        final int[] cacheSizes = { 10, 20, 7 };
        for ( int cacheSize : cacheSizes)
        {
            System.out.println( "CacheSize = " + cacheSize);
            final LruListCacheExample cache = new LruListCacheExample( persons,
                  cacheSize);

            performTests( cache, "findPersonByName", false);
```

```
            performTests(cache, "getPerson", true);
        }
    }
    // …
```

Listing 22.6 *Ausführbar als* **'LRULISTCACHEEXAMPLE'**

Führen wir nun das Programm LRULISTCACHEEXAMPLE aus, um zu untersuchen, welchen Einfluss verschiedene Cache-Größen tatsächlich haben. Als Ergebnis erhalte ich mit meinem Quad-Core 2,6 GHz Notebook und JDK 7 die in Tabelle 22-5 aufgelisteten Zeiten, wobei – wie bei den vorherigen Messungen auch – nicht die absoluten Werte, sondern die Relationen von Interesse sind.

Tabelle 22-5 *Performance-Gewinn Caching*

	Cache-Größe 10			Cache-Größe 7		
	Anzahl Elemente			Anzahl Elemente		
Zugriffsvariante	**1.000**	**10.000**	**100.000**	**1.000**	**10.000**	**100.000**
`findPersonByName()`	258 ms	2.347 ms	22.026 ms	241 ms	2.160 ms	21.826 ms
`getPerson()`	2 ms	11 ms	78 ms	246 ms	2.257 ms	23.249 ms

Anhand der Werte erkennt man, dass die Zeiten des indizierten Zugriffs über die Methode `findPersonByName(String)` erwartungsgemäß linear wachsen. Weiterhin zeigt sich deutlich, dass sich Caching, hier durch Zugriffe über die Methode `getPerson(String)`, nicht in jedem Fall positiv auswirkt. Stattdessen sind die Faktoren Cache-Größe, Auswahl der zu cachenden Daten und Lokalität der Zugriffe ganz entscheidend. Erfolgen ständig Zugriffe auf beliebige Elemente und werden diese immer sofort in den Cache übernommen, so profitiert man nicht durch den Einsatz von Caches. Zum Teil ist sogar ein negativer Effekt zu messen. Dies macht die in diesem Beispiel bewusst zu klein gewählte Cache-Größe von 7 deutlich. ***Sind Zugriffsstrategie, Cache-Größe usw. passend gewählt, so kann man extreme Performance-Gewinne erzielen***. Die Zugriffszeit ist dann nahezu konstant und nicht von der Anzahl der gespeicherten Daten abhängig. Eine Vergrößerung des Cache auf 20 Einträge bringt – wie Sie sicher schon vermuteten – allerdings keine Performance-Gewinne mehr. Die gemessenen Ausführungszeiten sind (abgesehen von den Messungenauigkeiten) identisch mit denen für einen Cache der Größe 10.

Tipp: Probleme von Caches

Beim Einsatz von Caches muss man große Sorgfalt walten lassen, dies gilt insbesondere bei Nebenläufigkeit und bei Mehrprozessormaschinen. Man benötigt

dann ein spezielles Protokoll, um einen Abgleich mehrerer Caches sicherzustellen. Diese Problematik ist auch unter dem Begriff ***Cache-Kohärenz*** bekannt. Auch besteht das Problem der Datenkonsistenz, wenn einige Programmteile am Cache vorbei auf die Datenquellen zugreifen.

Objekt-Pools

Die Idee beim Einsatz von sogenannten ***Objekt-Pools*** ist es, Objekte nicht ständig neu zu erzeugen. Stattdessen werden Objekte einer initial erzeugten Menge von Objekten verwendet und nach Gebrauch wieder an den Pool zurückgegeben.

Im Apache Commons-Framework[10] findet sich folgendes Interface zur Realisierung von Objekt-Pools:

```
public interface ObjectPool
{
    Object borrowObject();
    void returnObject( Object borrowed);
}
```

Aus einem solchen Objekt-Pool wird eine momentan unbenutzte Instanz ausgewählt. Diese wird dann vom Aufrufer für den Einsatz initialisiert, sodass auf eine (möglicherweise aufwendige) Konstruktion des Objektes selbst verzichtet werden kann. Diese Form des Objekt-Recyclings ist hilfreich, wenn zu erzeugende Objekte komplex und daher in ihrer Instanziierung teuer sind. Ein weiteres Einsatzgebiet ist die Einschränkung der Anzahl erzeugter Objekte, z. B. weil die durch diese Objekte verwalteten Hardwareressourcen beschränkt sind. Dies gilt auch für den Fall, dass die Anzahl der Objektzustände relativ gering ist und daher eine begrenzte Anzahl unveränderlicher Objekte existieren kann, die sämtliche Einsatzgebiete abdecken. Das einfachste Beispiel dafür ist die Klasse `Boolean` aus dem JDK, die lediglich zwei Zustände besitzt.

Wie schon in Abschnitt 4.2 besprochen, existieren auch für dir Wrapper-Klassen `Integer` und `Long` spezielle Caches und es findet ein Objekt-Pooling statt. Performance-technisch ist ein `Long.valueOf(x)` einem `new Long[x]` vorzuziehen.

Bewertung

War dieses Objekt-Pooling in älteren JVMs noch von größerem Nutzen für die Performance, so sind die JVMs im Laufe der letzten Jahre im Bereich der Objekterzeugung und der Garbage Collection deutlich schneller geworden und machen den Einsatz von Objekt-Pools in der Regel überflüssig.

Abgesehen von der Optimierung bei der Konstruktion, setzt Objekt-Pooling zwingend voraus, dass sich alle Programmteile korrekt verhalten, d. h. die Objekte vollständig freigeben und keine Referenzen darauf zwischenspeichern. Gibt ein Programmteil ein genutztes Objekt nach Gebrauch zwar an den Pool zurück,

speichert aber weiterhin eine Referenz darauf, so ist dies problematisch: Objektreferenzen, die zwischengespeichert werden, zeigen auf mittlerweile neu initialisierte Objekte, die von anderen Programmteilen verwendet werden.

Meistens ist im Voraus nicht bekannt, wie viele Objekte in einem solchen Pool vorgehalten werden sollten. Werden weniger Objekte angefragt als erwartet und bereitgestellt, so verschwendet man sowohl Speicherplatz als auch Rechenzeit (für die initiale Konstruktion). Werden jedoch mehr Objekte angefordert, als im Pool verfügbar sind, so muss eine spezielle Verwaltung einer Warteliste selbst programmiert werden. Als Abhilfe kann man auch dynamisch wachsende Pools selbst realisieren. ***Aufgrund der genannten Einschränkungen sollte der Einsatz von Objekt-Pools gut jedoch überlegt werden***.

Beispiele im JDK

Die JVM nutzt für Strings mit dem Stringliteral-Pool einen Objekt-Pool. Dieses Pooling erfolgt für die Applikation transparent. Insbesondere sind keine zusätzlichen Methodenaufrufe notwendig, sodass die zuvor genannten Nachteile allgemeiner Objekt-Pools nicht gelten. Weiterhin stellt die Klasse `Boolean`, wie bereits erwähnt, die simpelste Form eines Objekt-Pools dar. Aufgrund der Beschränkung auf zwei unveränderliche Objekte existieren bei dieser Realisierung keine negativen Auswirkungen durch nicht zurückgegebene oder veraltete Referenzen.

Ich habe Messungen angestellt, die jeweils 10 Millionen `Boolean`-Werte in einem Array speichern. Einmal erfolgt dabei eine Objektkonstruktion per `new Boolean(true)`, die andere Variante speichert gemeinsame Instanzen aus dem Pool per `Boolean.TRUE`. Folgendes Listing zeigt einen Ausschnitt aus dem Programm:

```
public static void main(final String[] args)
{
    final Boolean[] testArray = new Boolean[10_000_000];
    reportMemory();
    performTests(testArray, "Boolean.TRUE", false);
    reportMemory();
    performTests(testArray, "new Boolean(true)", true);
    reportMemory();
}

public static void performTests(final Boolean[] testArray,
                                final String info,
                                final boolean createNew)
{
    PerformanceUtils.startMeasure(info);
    for (int i = 0; i < testArray.length; i++)
    {
        if (createNew)
            testArray[i] = new Boolean(true);
        else
            testArray[i] = Boolean.TRUE;
    }
    PerformanceUtils.stopMeasure(info);
```

```
    PerformanceUtils.printTimingResult(info);
}

private static void reportMemory()
{
    MemoryInfo.gcAndSleep5s();
    System.out.println(MemoryInfo.statistics());
}
```

Listing 22.7 *Ausführbar als* **'BOOLEANPOOLINGOPTIMIZATION'**

Diese Optimierung deckt mehrere Ebenen und Techniken ab: Es erfolgt eine Optimierung bezüglich des Speicherverbrauchs durch den Verzicht der Konstruktion von 10 Millionen `Boolean`-Objekten und eine Optimierung bezüglich der Laufzeit der Objekterzeugungen. Tabelle 22-6 zeigt die Ergebnisse der Messungen für den Speicherverbrauch und die Laufzeit.

Tabelle 22-6 *Performance-Gewinne durch* `Boolean`*-Pooling*

	`new Boolean(true)`	`Boolean.TRUE`
Ausführungszeit	1276 ms	15 ms
Speicherverbrauch	160.008.328 Bytes	397.204 Bytes

Anhand der Zahlen kann man zwei Dinge sehr deutlich ablesen: Die JVM ist extrem performant für Objekterzeugungen, hier am Beispiel für `Boolean`-Objekte gezeigt. Daher lohnt sich ein Objekt-Pooling zur Einsparung von CPU-Zyklen lediglich für ganz spezielle Anwendungsfälle. Wir erkennen hier zwar einen Faktor ca. 85, aber relativ zu 10 Millionen Objektkonstruktionen sind die absoluten Zeiten von 1.276 ms zu 15 ms fast immer vernachlässigbar. Interessanterweise schwanken die Werte sogar sehr zwischen den Updates des JDK. Ich habe für JDK 8 Update 5 ähnliche Zeiten gemessen, Update 20 ist mit 3.327 ms deutlich langsamer, JDK 7 Update 51 benötigt sogar 4.649 ms. Was ich damit sagen will ist: ***Messen Sie unbedingt mehrmals und auf der Version der JVM, die Sie verwenden.***

Entscheidender ist jedoch der Einfluss auf den Speicherbedarf. Hier kann es durch Objekt-Pooling zu deutlichen Einsparungen kommen. Wir sehen in diesem Fall etwa den Faktor 400, wobei das sehr stark von der Speicherfreigabe abhängt und noch deutlich stärker sein kann. Führen wir eine Überschlagsrechnung durch und teilen den verbrauchten Speicher durch die Anzahl der erzeugten Objekte, so erkennen wir, dass jedes `Boolean`-Objekt etwa 16 Bytes verbraucht. Weitere Details zum Einfluss der Wahl von Datentypen auf den Speicherverbrauch werden in Abschnitt 22.6.4 besprochen.

Hinweis: Messungenauigkeiten beim Speicherverbrauch

Für gemessene Ausführungszeiten habe ich bereits bei der Diskussion der für Datenstrukturen ermittelten Messwerte auf Ungenauigkeiten aufmerksam gemacht.

Ähnliches gilt für Messungen des Speicherverbrauchs. Diese sind aufgrund der Arbeit des Garbage Collectors nur Annäherungswerte, können aber häufig trotzdem auf Probleme aufmerksam machen.

22.6.3 Optimierungen der Stringverarbeitung

In diesem Abschnitt wollen wir analysieren, warum der massive Einsatz von Strings und Stringoperationen zu Performance-Problemen führen kann. Häufig liegt dies daran, dass Strings intuitiv wie veränderbare Objekte – auch bedingt durch ihr API – eingesetzt werden, aber intern als unveränderliche Objekte implementiert sind. Was heißt das genau? Jede kleinste Veränderung an einem String führt zu einer neuen Instanz. So entstehen schnell viele temporäre Stringobjekte, die sofort wieder verworfen werden, wenn sie nur ein Teilergebnis einer komplexeren Operation sind. Wie bereits erwähnt, ist die JVM bezüglich der Objekterzeugung und der Garbage Collection seit ihren Anfängen viel performanter geworden. Strings bergen jedoch ein weiteres Problem: Sie werden in den Stringliteral-Pool (vgl. Abschnitt 4.3.1) eingetragen und belegen Speicher im Perm-Bereich (vgl. Abschnitt 8.5.3), der extrem selten aufgeräumt wird. In JDK 8 wird stattdessen ein Bereich namens Metaspace genutzt. Teilergebnisse von Stringoperationen belegen so diesen Bereich für lange Zeit.

Tipp: Eingabeverwaltung von `Console.readPassword()`

Die Klasse `Console` bietet eine Methode `readPassword()`. Folgt man der obigen Argumentation bezüglich der Speicherung temporärer Stringresultate im Perm- bzw. Metaspace-Bereich, so würde man durch Aufruf der Methode `readPassword()` ein Passwort im Speicher finden können. Um dies zu vermeiden, arbeitet die `readPassword()`-Methode mit einem `char[]`. Dies ist übrigens eine beliebte Fangfrage bei OCPJP-Prüfungen, da die überladenen `readLine()`-Methoden der Klasse `java.io.Console` einen String zurückliefern.

`StringBuffer` / `StringBuilder` als Abhilfe?

Häufig wird als Optimierung von Stringoperationen der Einsatz der Klasse `StringBuffer` bzw. `StringBuilder` vorgeschlagen. Das Problem temporär erzeugter `String`-Instanzen lässt sich dadurch mildern. Das dazu notwendige Vorgehen wurde in Abschnitt 4.3 vorgestellt. Manchmal werden diese Klassen bereits genutzt, obwohl nur einige wenige Verknüpfungen von Strings durchgeführt werden müssen. Darunter leidet die Lesbarkeit, und außerdem wird der Sourcecode länger.

Betrachten wir ein Programm, das die Laufzeit für verschiedene Stringoperationen innerhalb einer Schleife mit 1 Million Durchläufen ermittelt und so eine Abschätzung von Performance-Gewinnen beim Einsatz einer speziellen

Methode erlaubt:

```
public static void main(final String[] args)
{
    final String name = "Sarah vom Auetal";
    final int counter = 1_000_000;
    String result = "";

    // Messung mit String +
    result = measureStringPlus(name, counter);

    // Messung mit String +=
    result = measureStringPlusEquals(name, counter);

    // Messung mit StringBuilder
    result = measureStringBuilderAppend(name, counter);

    // Messung mit StringBuilder und initialer Kapazität
    result = measurePresizedStringBuilderAppend(name, counter);

    // Messung mit StringBufferr
    result = measureStringBufferAppend(name, counter);

    // Messung mit StringBufferr und initialer Kapazität
    result = measurePresizedStringBufferAppend(name, counter);

    // Messung mit String und Formatter
    result = measureStringAndFormatter(name, counter);

    // Messung mit StringBuilder, initialer Kapazität und Formatter
    result = measureStringBuilderAndFormatter(name, counter);

    // Ausgabe des Ergebnisses vermeidet Weg-Optimierung der Berechnungen
    System.out.println(result);
}

private static String measureStringPlus(final String name, final int counter)
{
    String result = "";
    PerformanceUtils.startMeasure("String +");
    for (int i = 0; i < counter; i++)
    {
        result = "Mein Hund ist " + i + " Jahre alt und heißt " + name + ".";
    }
    PerformanceUtils.stopMeasure("String +");
    PerformanceUtils.printTimingResultWithAverage("String +", counter);
    return result;
}

private static String measureStringPlusEquals(final String name,
                                              final int counter)
{
    String result = "";
    PerformanceUtils.startMeasure("String +=");
    for (int i = 0; i < counter; i++)
    {
        result = "Mein Hund ist ";
        result += i;
        result += " Jahre alt und heißt ";
        result += name;
        result += ".";
```

```
    }
    PerformanceUtils.stopMeasure("String +=");
    PerformanceUtils.printTimingResultWithAverage("String +=", counter);
    return result;
}
// …
```

Listing 22.8 *Ausführbar als* **'STRINGBENCHMARK'**

In Tabelle 22-7 sind die Ergebnisse des Benchmarks zusammengefasst, wobei weitere Varianten der Konkatenation auf ihre Performance untersucht wurden, etwa die Auswirkungen der Angabe einer passenden initialen Größe für einen `StringBuffer`. Das wird in der Tabelle als »Presized« gekennzeichnet.

Tabelle 22-7 *Performance-Vergleiche von Stringkonkatenationen*

Variante	Dauer	Durchschnitt
+	81 ms	0,000081 ms
+=	276 ms	0,000276 ms
`StringBuilder`	153 ms	0,000154 ms
`StringBuilder` »Presized«	108 ms	0,000108 ms
`StringBuffer`	163 ms	0,000163 ms
`StringBuffer` »Presized«	130 ms	0,000130 ms
`String.format()`	1.470 ms	0,001470 ms
`String` und `Formatter`	1.485 ms	0,001445 ms
`StringBuilder` und `Formatter`	1.354 ms	0,001354 ms

Diskussion

Wie man eindrucksvoll sieht, sind alle Varianten der Konkatenation mit Ausnahme von ' +=' nahezu gleich schnell. Nach Lektüre von Abschnitt 4.3.2 ist dies leicht nachvollziehbar, denn der Operator ' +' auf Strings wird intern durch eine Konkatenation von `append()`-Methoden einer Instanz der Klasse `StringBuilder` abgebildet. Die um etwa den Faktor 3,3 langsamere Verarbeitung von ' +=' ist durch die mehrfachen Konstruktionen von temporären `StringBuilder`- und `String`-Instanzen zu erklären. Anhand der Messwerte sieht man auch, dass Stringaufbereitungen durch die `Formatter`-Klassen bzw. die `format()`-Methode

etwa um den Faktor 18 langsamer sind als die Konkatenation mit dem Operator ' +'.

Nur in tatsächlich Performance-kritischen Situationen sollte man einen `StringBuilder` der übersichtlichen Schreibweise mit dem Operator ' +' vorziehen. Wie immer gilt: Vor Optimierungen sollte man gründliche Messungen vornehmen, um sicherzustellen, dass die Maßnahmen erforderlich sind und sich lohnen. Ansonsten hat Übersichtlichkeit nahezu immer Priorität vor geringen Performance-Gewinnen.

22.6.4 Technik – Vermeidung unnötiger Aktionen

Zur Vermeidung einer aufwendigen Konstruktion schwergewichtiger Objekte haben wir in Abschnitt 22.3 bereits die Technik Lazy Initialization kennengelernt. Hierbei werden komplexe, schwergewichtige Objekte in mehrere kleinere, leichtgewichtigere Objekte aufgeteilt, um optionale Teile nur bei Bedarf nachzuladen bzw. zu initialisieren. Der aktuelle Abschnitt führt die Gedanken des Speichersparens und der Vermeidung unnötiger Aktionen auf feingranularer Ebene fort.

Reduziere die Kosten der Objekterzeugung

Fast alles ist in Java ein Objekt. Demnach werden in Applikationen oftmals viele Objekte angelegt, was sich negativ auf die Performance auswirken kann: Die Objektkonstruktionen kosten etwas Laufzeit. Die JVM ist diesbezüglich sehr performant geworden, sodass dies heutzutage normalerweise kein Problem mehr darstellt. Allerdings besitzt die Erzeugung von Objekten indirekten Einfluss auf die Performance: Zum einen kommt es dadurch später zu einer erhöhten Aktivität des Garbage Collectors, um diese Objekte wieder freizugeben. Zum anderen kann auch der von den Objekten belegte Speicher zu einem Mangel an freiem Speicher führen. Daraus resultieren eventuell Garbage Collections, die (wahrnehmbare) Pausen in der Abarbeitung verursachen.

Je weniger Objekte (unnötig) erzeugt werden, desto mehr Speicher steht der Applikation zur Verfügung. Dazu kann man versuchen, die Kosten der Objekterzeugung und die Anzahl der erzeugten Objekte zu minimieren, indem man Lazy Initialization nutzt und primitive Datentypen anstelle korrespondierender Instanzen der Wrapper-Klassen bevorzugt. Außerdem kann man die Freigabe von Objekten erleichtern, indem Referenzvariablen möglichst schnell auf `null` gesetzt werden, wenn diese nicht mehr benötigt werden. Dadurch kann der Garbage Collector diese Objekte einsammeln und eventuell (etwas) früher freigeben.

Hinweis: Ausführungszeit von `new`

Ein `new` in Java führt im Moment des Aufrufs nicht unbedingt zu einer Bereitstellung von Speicher durch das Betriebssystem, da der Speicher in der Regel

bereits in der JVM verfügbar ist. Daher kann ein `new` in Java sogar schneller sein als in C++.

Bevorzuge primitive Datentypen

Bei sehr großen Datenmengen kann die Performance positiv beeinflusst werden, wenn man primitive Typen anstelle der zugehörigen Wrapper-Klassen verwendet. Das liegt an folgenden Eigenschaften primitiver Typen:

- Sie sind schnell zu erzeugen (kein Aufwand zur Laufzeit).
- Sie sind schnell im Zugriff (keine Referenzen).
- Sie verbrauchen etwas weniger Speicher als Instanzen der Wrapper-Klassen, z. B. 4 Byte (`int`) zu 16 Byte (`Integer`).
- Sie bergen nicht die Gefahr uninitialisierter Referenzvariablen und daraus resultierender `NullPointerExceptions`.

Am Beispiel der Realisierung von Value Objects (VOs) (vgl. Abschnitt 3.4.5) kann man sich die Vorteile des Einsatzes primitiver Typen verdeutlichen. Nachfolgendes Beispiel zeigt zwei mögliche Realisierungen der Klassen `BadVO` und `GoodVO` zum Austausch von Daten zwischen Applikationen. Die Daten bestehen hier zur Vereinfachung der Darstellung nur aus einer Kennung `id` und zwei Werten `value1` sowie `value2`.

Die Klasse `BadVO` nutzt diverse Wrapper-Klassen zur Definition ihrer Attribute und speichert demnach Objektreferenzen:

```
class BadVO implements Serializable
{
    Long Id;
    Integer value1;
    Double  value2;
}
```

Eine solche Realisierung kann erforderlich sein, um für optionale Attribute »kein Wert« ausdrücken zu können. Dazu nutzt man `null` als Wert. Benötigt man diese Funktionalität nicht, so verbraucht der Ansatz deutlich mehr Speicher als die im Folgenden gezeigte `GoodVO`-Variante, die primitive Datentypen für die Attribute verwendet:

```
class GoodVO implements Serializable
{
    long Id;
    int value1;
    double value2;
}
```

Betrachten wir dies im Detail, indem wir die Größe beider VOs anhand der verwendeten Datentypen berechnen: Ein `BadVO` benötigt 48 Bytes, ein `GoodVO` belegt lediglich 20 Bytes (vgl. folgenden Praxistipp). ***Anhand dieser Größenangaben kann man abschätzen, dass das maximale Einsparpotenzial bei etwa 50 – 75% liegt,***

d. h., ein lediglich aus primitiven Typen zusammengesetztes Objekt ist nur etwa ein Viertel bis halb so groß wie ein VO, bestehend aus Wrapper-Klassen.

Generell gilt, dass sich diese Optimierung vor allem beim Einsatz von Zahlentypen lohnt. Bei umfangreichen VOs, die nicht nur Zahlen, sondern auch Objektreferenzen und Strings speichern, sind die Unterschiede deutlich geringer. Neben dem reinen Speicherverbrauch ist häufig noch ein weiterer Aspekt von Interesse. Bei der Übertragung von VOs zwischen verschiedenen JVMs über ein Netzwerk sind die Auswirkungen solcher Einsparungen in einer gesteigerten Geschwindigkeit bei der Kommunikation wahrnehmbar, weil die zu übertragende Datenmenge geringer ist.

Tipp: Größenabschätzungen für Objekte

Der Speicherverbrauch eines Objekts lässt sich anhand seiner Attribute berechnen. Für primitive Typen gilt: `byte` ≡ 1, `short` ≡ 2, `int` ≡ 4, `long` ≡ 8, `float` ≡ 4, `double` ≡ 8 Bytes. Referenzvariablen benötigen 4 Bytes (für 32-Bit-JVMs, 8 Bytes für 64-Bit-JVMs). Jede Instanz der Klasse `Object` belegt 8 Bytes. Die Wrapper-Klassen belegen konstant 16 Bytes anstatt der unterschiedlichen Größen der primitiven Typen. Eine `String`-Instanz verbraucht bereits ohne textuellen Nutzinhalt 40 Bytes. Zusatzinformationen findet man im Buch »Java Platform Performance – Strategies and Tactics« von Steve Wilson und Jeff Kesselman [91] sowie online unter `http://www.javaworld.com/javaworld/javatips/jw-javatip130.html`.

22.7 CPU-bound-Optimierungen

In wenigen Fällen wird die Ausführungsgeschwindigkeit eines Programms maßgeblich durch die reine Bearbeitungszeit der Anweisungen in der CPU aufgrund von nicht optimalen Berechnungen verursacht. Im Folgenden werden verschiedene CPU-bound-Optimierungstechniken lediglich kurz vorgestellt, weil derartige Optimierungen nur nach allen anderen Optimierungen überhaupt in Betracht gezogen werden sollten. In Ihrer täglichen Praxis wird das vermutlich eher selten der Fall sein.

22.7.1 Technik – Wahl passender Strategien

Manchmal existiert ein Performance-technisch offensichtlich besser geeigneter Algorithmus für ein zu lösendes Problem als der derzeit gewählte. Bei Performance-Problemen kann dann durch die Wahl des besseren Algorithmus für Abhilfe gesorgt werden. Dies haben wir für die Berechnung einer Summe bereits in Abschnitt 22.1.3 kennengelernt. Ein weiteres Beispiel ist der Einsatz bzw. der Verzicht von Rekursion:

Die Berechnung von Fibonacci-Zahlen ist rekursiv einfach möglich – allerdings nicht performant:

```
public static long fibonacciRecursive(long n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    return fibonacciRecursive(n - 1) + fibonacciRecursive(n - 2);
}
```

Es existiert eine deutlich effizientere, nicht rekursive Möglichkeit zur Berechnung:

```
public static long fibonacciIterative(final long n)
{
    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

    long fib_n_2 = 0;
    long fib_n_1 = 1;
    long fib_n = 1;

    for (long i = 2; i <= n; i++)
    {
        fib_n = fib_n_2 + fib_n_1;

        fib_n_2 = fib_n_1;
        fib_n_1 = fib_n;
    }

    return fib_n;
}
```

Schreiben wir ein kleines Programm, um die Performance-Unterschiede zwischen beiden Varianten der Berechnung zu demonstrieren:

```
public static void main(final String[] args)
{
   final int[] values = { 10, 30, 35, 36, 37, 38, 39, 40,
                          41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 55, 57 };

   for (final int currentValue : values)
   {
      System.out.println("currentValue: " + NumberFormat.getIntegerInstance().
                                                          format(currentValue));
      System.out.println("-------------- ");

      final long start1 = System.nanoTime();
      final long fibRecursive = fibonacciRecursive(currentValue);
      final long end1 = System.nanoTime();

      System.out.println("fibonacciRecursive took: " +
                         TimeUnit.NANOSECONDS.toMillis(end1 - start1) + "
ms");
      System.out.println("fibonacciRecursive is: " + fibRecursive);
```

```
        final long start2 = System.nanoTime();
        final long fibIterative = fibonaccieIterative(currentValue);
        final long end2 = System.nanoTime();

        System.out.println("fibonaccieIterative took: " +
                           TimeUnit.NANOSECONDS.toMillis(end2 - start2) + "
ms");
        System.out.println("fibonaccieIterative is: " + fibIterative);

        System.out.println("-------------- ");
    }
}
```

Listing 22.9 *Ausführbar als* **'FIBONACCIEXAMPLE'**

Führen wir das Programm FIBONACCIEXAMPLE aus, so sehen wir, dass bis etwa zur 30. Fibonacci-Zahl die Berechnungen recht zügig vonstatten gehen. Danach wird deutlich, wie extrem schnell die Laufzeit zunimmt (vgl. folgende Tabelle 22-8).

Tabelle 22-8 *Vergleich rekursive und iterative Berechnung von Fibonacci-Zahlen*

	Eingabewert und Ausführungszeit in ms							
Berechnung	**10**	**30**	**35**	**40**	**45**	**50**	**55**	**57**
Rekursiv	0	5	47	519	5.769	63.987	718.080	1.866.418
Iterativ	0	0	0	0	0	0	0	0

Bereits für Zahlen (ab ca. 40 bis 45) macht sich die exponentielle Zeitzunahme drastisch bemerkbar – bei der iterativen Variante ist die Laufzeit zwar theoretisch $O(n)$, aber praktisch unbedeutend. Um auch viel größere Fibonacci-Zahlen berechnen zu können, muss man das obige Programm nur ein wenig abwandeln und für Berechnungen die Klasse `BigInteger` nutzen:

```
public static BigInteger fib(final int n)
{
    if (n == 0)
        return BigInteger.ZERO;
    if (n == 1)
        return BigInteger.ONE;

    BigInteger fib_n_2 = BigInteger.ZERO;
    BigInteger fib_n_1 = BigInteger.ONE;
    BigInteger fib_n = BigInteger.ONE;

    for (int i = 2; i <= n; i++)
    {
        fib_n   = fib_n_2.add(fib_n_1);
        fib_n_2 = fib_n_1;
        fib_n_1 = fib_n;
    }
    return fib_n;
}
```

```
public static void main(final String[] args)
{
    System.out.println(fib(10));    // 55
    System.out.println(fib(20));    // 6.765
    System.out.println(fib(30));    // 832.040
    System.out.println(fib(40));    // 102.334.155
    System.out.println(fib(50));    // 12.586.269.025
    // über 200 Ziffern: 43466557686937456435688527675040…849228875
    System.out.println(fib(1000));
    // über 2.000 Ziffern: 364476487643178326662161200510 75…947366875
    System.out.println(fib(10_000));
}
```

Listing 22.10 *Ausführbar als* **'FIBONACCIBIGINTEGEREXAMPLE'**

Selbst für riesige Fibonacci-Zahlen bzw. Ergebnisse erhält man beim Start des Programms FIBONACCIBIGINTEGEREXAMPLE nur Laufzeiten im Millisekundenbereich.

Hinweis: Rekursion als Problemindikator

Rekursion ist immer ein potenzieller Ansatzpunkt für Optimierungen. Nicht immer sollte man beim Verzicht auf Rekursion aber enorme Gewinne wie bei der Berechnung der Fibonacci-Zahlen erwarten.

22.7.2 Technik – Caching und Pooling

Häufiger liest man, dass man Berechnungen dadurch optimieren sollte, dass Berechnungsergebnisse oder Werte vorausberechnet und in Caches oder Look-up-Tabellen gespeichert werden. Früher wurde eine solche Optimierung häufig für Berechnungen von Sinus- oder Kosinuswerten vorgeschlagen. Ich habe Messungen angestellt, und diese zeigen nur geringe Auswirkungen auf die Performance. Das gilt selbst für aufwendigere Berechnungen durch Aufrufe von `Math.exp(double)`. Eine Million Aufrufe benötigen inklusive Speicherung in einem Array auf meinem Rechner 251 ms, sodass sich eine Zwischenspeicherung bei einer derartigen Berechnungsgeschwindigkeit fast niemals lohnt. Diese Messung verdeutlicht nochmals die eingangs des Kapitels gemachte Aussage, dass heutzutage Performance-Optimierung auf Sourcecode-Ebene nur minimale Auswirkungen auf die Gesamtperformance besitzt.

Vergleichen Sie die Ausführungszeiten auf Ihrem Rechner durch Aufruf des folgenden Programms MATHEXPEXAMPLE:

```
public static void main(final String[] args)
{
    final int counter = 1000 * 1000;
    final double[] testArray = new double[counter];

    PerformanceUtils.startMeasure("MathExpExample");
    for (int i = 0; i < counter; i++)
    {
```

```
            testArray[i] = Math.exp(i);
        }
        PerformanceUtils.stopMeasure("MathExpExample");
        PerformanceUtils.printTimingResult("MathExpExample");
    }
```

Listing 22.11 *Ausführbar als* **'MATHEXPEXAMPLE'**

Gerade beim Einsatz von Look-up-Tabellen für mathematische Berechnungen muss man sich darüber im Klaren sein, dass dies auch negative Seiten besitzt:

- **Genauigkeit** – Man wird sich auf eine gewisse Menge an vordefinierten Werte beschränken, etwa die Sinuswerte auf ein Grad genau. Für einige Berechnungen mag diese Genauigkeit ausreichend sein, für andere wiederum nicht.
- **Speicherverbrauch** – Je genauer bzw. umfangreicher die Look-up-Tabellen werden, desto mehr Speicher belegen sie zwangsläufig. Hier existiert ein Tradeoff zwischen verschiedenen Optimierungszielen. Wahrscheinlich wirken sich aber riesige Look-up-Tabellen ungünstiger aus, als das, was sie an CPU-bound-Optimierung herausholen. Hier sind wieder umfangreiche Messungen gefragt.

Nichtsdestotrotz ist die Idee der Optimierung durch eine Berechnung im Voraus an sich korrekt und immer dann sinnvoll einzusetzen, wenn man schwergewichtige Objekte wiederverwenden kann oder aber ein Caching nicht mit einer Einschränkung der Lesbarkeit verbunden ist. Insbesondere bei Datenbankzugriffen kann man durch ein geschicktes Caching einiges an Performance-Optimierung erzielen.

22.7.3 Technik – Vermeidung unnötiger Aktionen

Eine Technik, die sowohl die Lesbarkeit erhält als auch Vorteile in der Ablaufgeschwindigkeit und bezüglich des Speicherverbrauchs bewirkt, ist der Einsatz von primitiven Datentypen anstelle von Wrapper-Klassen. Geschieht dies konsequent, so vermeidet man Auto-Boxing und Auto-Unboxing bei Konvertierungen. Auch für diese Optimierung findet man in der Literatur häufig übertriebene Aussagen zu deren Auswirkung auf die Performance. Um überhaupt Effekte messen zu können, müssen extrem viele Auto-Boxing-Vorgänge stattfinden.

In den ersten beiden Auflagen versuchte ich, dies an einer `equals(Object)`-Methode zu verdeutlichen. Für diese 3. Auflage habe ich mich für ein artifizielles Beispiel entschieden, das die Dinge besser verdeutlicht. Im folgenden Programm wird eine Summe mithilfe einer Schleife und einer Addition mit dem Wert 1 berechnet. Einmal benutzen wir dazu einen `long` und für die zweite Summation ein `Long`-Objekt. Dadurch kommt es beim Addieren zu Auto-Boxing und Auto-Unboxing:

```
public static void main(final String[] args)
{
   final long[] loopCounts = {1_000_000, 10_000_000, 100_000_000,
                              1_000_000_000, 10_000_000_000L,
100_000_000_000L};
```

```
    for (final long loopCount : loopCounts)
    {
        System.out.println("LoopCount: " + NumberFormat.getIntegerInstance().
                                                    format(loopCount));
        System.out.println("-------------- ");

        final long start1 = System.nanoTime();

        long sumAsPrimitive = 0;
        for (long i = 0; i < loopCount; i++)
        {
            sumAsPrimitive += 1;
        }

        final long end1 = System.nanoTime();
        System.out.println("Primitive sum took: " +
                           TimeUnit.NANOSECONDS.toMillis(end1 - start1));
        // Zugriff auf die Variable, um Optimierung zu vermeiden
        System.out.println("Primitive sum is: " + sumAsPrimitive);

        final long start2 = System.nanoTime();

        Long sumAsWrapper = 0L;
        for (long i = 0; i < loopCount; i++)
        {
            // Hier kommt es zu Auto-Boxing und Auto-Unboxing
            sumAsWrapper += 1;
        }

        final long end2 = System.nanoTime();
        System.out.println("Wrapper/Auto-/Un-)Boxing sum took: " +
                           TimeUnit.NANOSECONDS.toMillis(end2 - start2));
        // Zugriff auf die Variable, um Optimierung zu vermeiden
        System.out.println("Wrapper sum is: " + sumAsWrapper);

        System.out.println("-------------- ");

        final double factor = (end2 - start2) / (double) (end1 - start1);
        System.out.println(String.format("Factor %.2f", factor));
        System.out.println("-------------- ");
    }
}
```

Listing 22.12 *Ausführbar als* **'CPUBOUNDOPTIMIZATIONEXAMPLE'**

Führen wir das Programm CPUBOUNDOPTIMIZATIONEXAMPLE aus, um ein wenig mehr Gewissheit über den Einfluss von Auto-Boxing auf die Performance zu erhalten. Wie performant Berechnungen trotz Auto-Boxing und Auto-Unboxing sind, zeigt folgende Tabelle 22-9.

Tabelle 22-9 *Performance-Vergleich zwischen Wrapper-Objekten und primitiven Typen*

Berechnung	Anzahl Summationen				
	1.000.000	10.000.000	100.000.000	1.000.000.000	10.000.000.000
Mit Auto-Boxing	14 ms	63 ms	359 ms	3.019 ms	29.812 ms
Mit primitiven Typen	5 ms	5 ms	40 ms	401 ms	4.028 ms

Insgesamt sind Berechnungen mit primitiven Typen natürlich schneller, im Durchschnitt um etwa den Faktor 5–10. ***Allerdings zeigen die absoluten Zeiten, dass durch diese Optimierung tatsächlich nur sehr wenig Laufzeitgewinn zu erzielen ist***. Dies wird insbesondere klar, wenn man bedenkt, wie viele Durchläufe nötig sind, um die gemessenen Zeiten zu produzieren: Selbst 1 Milliarde Additionen benötigen nur etwa drei Sekunden.

Werden extrem große Datenmengen verwaltet, kann man durch Vermeiden von Auto-Boxing einen kleinen Performance-Gewinn erzielen. Ansonsten muss man sich über solche Minimaloptimierungen bezüglich der Wrapper-Objekt-Erzeugung keine Gedanken machen.

Fazit

Bitte erinnern Sie sich nochmals an meine eingangs gemachte Aussage: ***Low-Level-Optimierungen sollten Sie im Normalfall nicht durchführen müssen***. Bevor Sie daran denken, so etwas zu tun, schauen Sie, ob nicht auf höherer Ebene eine zweckmäßigere Optimierung vorgenommen werden kann.

22.8 Weiterführende Literatur

Weiterführende Informationen finden sich in folgenden Büchern sowie auf folgender Internetseite:

- »**Performant Java programmieren**« von Hendrik Schreiber [78]

  Dieses Buch ist sehr lesenswert und gibt viele interessante Einblicke in das Gebiet der Performance-Optimierung. Es verliert sich nicht in Details, sondern bringt die Ideen und Hinweise klar auf den Punkt. Daher empfehle ich dieses Buch ausdrücklich.

- »**Java Performance Tuning**« von Jack Shirazi [81]

  Dieser Klassiker unter den Performance-Tuning-Büchern beleuchtet das Thema

Performance-Optimierung auf breiter Front, geht in einigen Bereichen allerdings sehr auf Optimierungen auf Sourcecode-Ebene ein.

- »**Java Platform Performance – Strategies and Tactics**« von Steve Wilson und Jeff Kesselman [91]

  Dieses Buch ist neben dem von Jack Shirazi ein weiterer Klassiker unter den Performance-Tuning-Büchern und gibt einen guten Überblick über das Thema Performance-Optimierung, steigt allerdings nicht so detailliert in die jeweiligen Themen ein wie das zuvor genannte Buch.

- »**Java Tuning White Paper**« unter `http://java.sun.com/performance/reference/whitepapers/tun`

  Diese Dokumentation enthält einige nützliche Hinweise und auch Links auf weitere Seiten, etwa zur Optimierung der Garbage Collection.

Anhang

A Einführung in die UML

Die Unified Modeling Language, kurz UML, ist eine grafische Notationsform, die bei der Entwicklung und Planung von Softwaresystemen helfen kann. Sie wird häufig für die objektorientierte Analyse (OOA) und das objektorientierte Design (OOD) eingesetzt. Im Rahmen dieses Kapitels wird die UML nur so weit vorgestellt, wie dies zum Nachvollziehen der Beispiele dieses Buchs notwendig ist und den Einstieg für den Gebrauch im praktischen Softwareentwurf erleichtert. Weiterführende Informationen finden Sie in diversen Büchern – beispielsweise im Standardwerk »Das UML-Benutzerhandbuch« [11] oder im Buch »UML 2.0: Das umfassende Handbuch« [53].

Einen einführenden Überblick gibt Abschnitt A.1. Die Möglichkeiten zur Modellierung der statischen Struktur eines Softwaresystems stellt Abschnitt A.2 vor. Notationen zur Beschreibung der Dynamik werden in Abschnitt A.3 beschrieben.

A.1 Die UML im Überblick

Die UML entstand schrittweise durch Zusammenführung[1] und Überarbeitung der bereits bestehenden Objektmodellierungssprachen (vgl. Tabelle A-1) der drei Experten Grady Booch, James Rumbaugh und Ivar Jacobson.

Tabelle A-1 *Experte und Modellierungstechnik*

Experte	Modellierungstechnik
Grady Booch	Object-Oriented Analysis and Design (OOAD)
James Rumbaugh	Object Modelling Technique (OMT)
Ivar Jacobson	Object-Oriented Software Engineering (OOSE)

Die Entwicklung beginnt im Jahr 1994, als Booch und Rumbaugh beschließen, ihre Ansätze zu vereinigen, um einen Notationsstandard zu schaffen. Aus den ersten Ansätzen entsteht 1995 die Unified Method Version 0.8, basierend auf der OMT mit Erweiterungen durch die OOAD. Jacobson bringt später OOSE ein. Es entsteht die UML, die 1997 von der OMG (Object Management Group) in Version 1.1 zum Standard für objektorientierte Modellierungssprachen erklärt wird. In den folgenden Jahren erscheinen die UML 1.3 und 1.4. Diese stellen bis zur Veröffentlichung der Version 2.0 im Jahr 2005 den Standard dar. Aktuell ist Version 2.4 aus dem Jahr 2011. In diesem Buch verwende ich die UML 2.0, weil die Neuerungen der Folgeversionen 2.x für diese Einführung nicht relevant sind. Auf Änderungen in der

Notation von Version 2.0 gegenüber der Vorgängerversion 1.4 weise ich bei Bedarf hin.

Diagramme der UML

Ziel bei der Entwicklung der UML war es, eine universelle ***Beschreibungssprache*** für objektorientierte Softwaresysteme und damit eine standardisierte Notation zu schaffen: Eine Modellierungstechnik, mit der Entwickler ihre Modelle beschreiben und mit anderen Entwicklern austauschen können, ohne eine spezielle Programmiersprache vorauszusetzen. Zur Modellierung von Software wurden vor Einführung der UML verschiedene Notationen eingesetzt. Im Extremfall hat jeder Entwickler seine eigene Notation erfunden.[2] Ein Austausch gemeinsamer Architekturgedanken wird dadurch erschwert und ist eventuell sogar mit Missverständnissen verbunden. Die UML vereinfacht dies durch verschiedene vorwiegend grafische Notationen (Diagramme). Es existieren sechs *Strukturdiagramme*[3] und sieben ***Verhaltensdiagramme***, die im Anschluss nur kurz vorgestellt werden. Einen Überblick über die Diagramme gibt Abbildung A-1. Die im Praxiseinsatz und für dieses Buch wichtigsten davon beschreiben die Abschnitte A.2 und A.3 ausführlicher.

Strukturdiagramme

Strukturdiagramme beschreiben die *statische* Sicht auf ein Softwaresystem und umfassen hauptsächlich Klassen- und Objektdiagramme. Den größeren Zusammenhang, die Aufteilung in Komponenten, Pakete und Subsysteme, stellen Komponenten- und Paketdiagramme dar und ermöglichen somit eine Art Gliederungssicht. Die Aufteilung auf Rechner und andere Systeme wird mit Verteilungsdiagrammen modelliert.

Verhaltensdiagramme

Verhaltensdiagramme beschreiben *dynamische* Abläufe innerhalb eines modellierten Systems. Anforderungen aus Sicht eines Benutzers lassen sich mit Anwendungsfällen festhalten. Die Interaktion von Systemkomponenten oder ausgewählter Objekte kann mithilfe von Sequenz- und Kommunikationsdiagrammen beschrieben werden. Stehen nicht die Kommunikationspartner, sondern Aktivitäten oder Zustände im Fokus, so verwendet man dazu Aktivitäts- und Zustandsdiagramme.

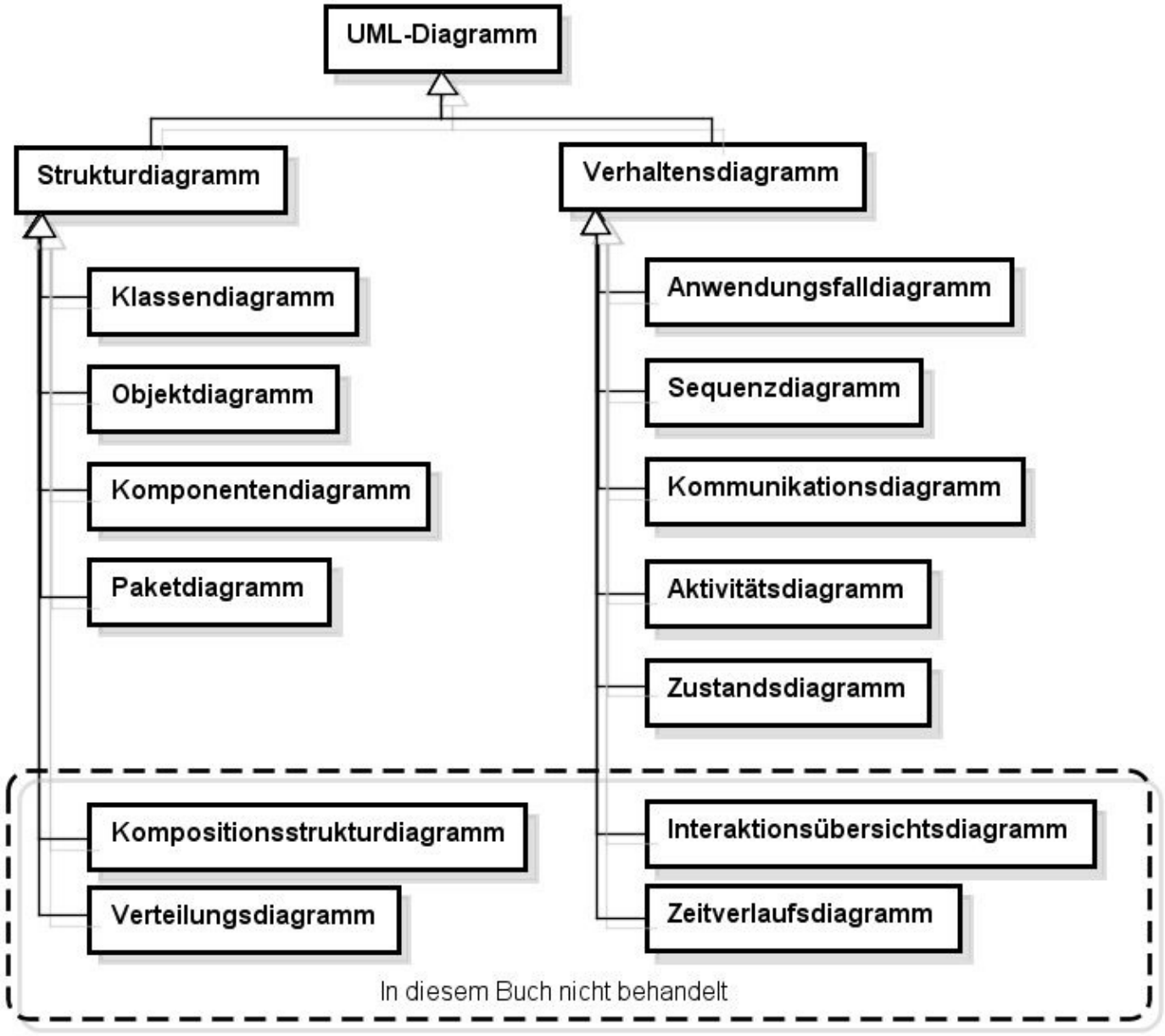


Abbildung A-1 *Alle Diagramme der UML 2.0*

Modellierung und Diagramm-Granularität

Der Entwurf eines Softwaresystems und dessen Modellierung beginnen mit der Aufnahme von Benutzeranforderungen und enden mit einer detaillierten Spezifikation der logischen und zeitlichen Abläufe des Systemverhaltens. In Gesprächen mit Anwendern lassen sich gewünschte Abläufe und Anforderungen ermitteln und mit Anwendungsfalldiagrammen beschreiben. Um die Modellierung näher zu erläutern, können optional Kommentare beliebiger Länge in die Diagramme eingefügt werden. Über Zustandsund Aktivitätsdiagramme sowie Sequenzdiagramme können Abläufe in den beschriebenen Anwendungsfällen bei Bedarf konkretisiert werden. Diese Diagramme können neben einer Modellierung auf einer eher fachlichen Ebene auch zur Darstellung und Umsetzung in ein technisches Modell genutzt werden.

Zudem kann eine erste, grobe Beschreibung der statischen Struktur in Form von

Komponenten- und Klassendiagrammen entwickelt werden. Letztere bilden meistens das zentrale Modell und den Ausgangspunkt der objektorientierten Modellierung.

Achtung: Objektmodellierung und Wiederverwendbarkeit

Ein Modell stellt eine Vereinfachung (Abstraktion) des modellierten Systems dar und konzentriert sich auf die für den jeweiligen Anwendungsfall benötigten Aspekte.

Es entstehen leicht Missverständnisse und die unerfüllbare Hoffnung bezüglich Objektmodellierung und Wiederverwendbarkeit: Aber die für einen Anwendungszweck modellierten Klassen sind nicht für alle möglichen Applikationen wieder zu verwenden. Viele Applikationen benötigen komplett andere Sichtweisen und Details der modellierten Gegenstände.

Im Projektverlauf können die Diagramme iterativ vervollständigt und angepasst werden. Hilfreich ist es, mehrere Versionen der Diagramme abhängig von der gegenwärtigen Projektphase (Konzept, Spezifikation, Implementierung) zu erstellen. Diese Modelle unterscheiden sich dann nicht nur in ihrem Detaillierungsgrad – sie können auch grundsätzliche Änderungen enthalten. Während der Konzeption kann es beispielsweise unbedeutend sein, wie eine Relation zwischen zwei Klassen umgesetzt wird, wichtig ist nur, dass es überhaupt eine Verbindung zwischen den Klassen gibt. In der Implementierungsphase kann es aber von Belang sein, ob diese Relation durch Referenzen oder eine Abbildungsklasse realisiert wird. Weiterhin ist es manchmal sinnvoll, eine zunächst als gerichtete Referenz modellierte Relation dann doch bidirektional umzusetzen. Dies deutet Abbildung A-2 an. Die dort eingesetzten Elemente werden im Detail in Abschnitt A.2.1 vorgestellt – hier ist zunächst die Idee wichtig, nicht die konkrete Notation.

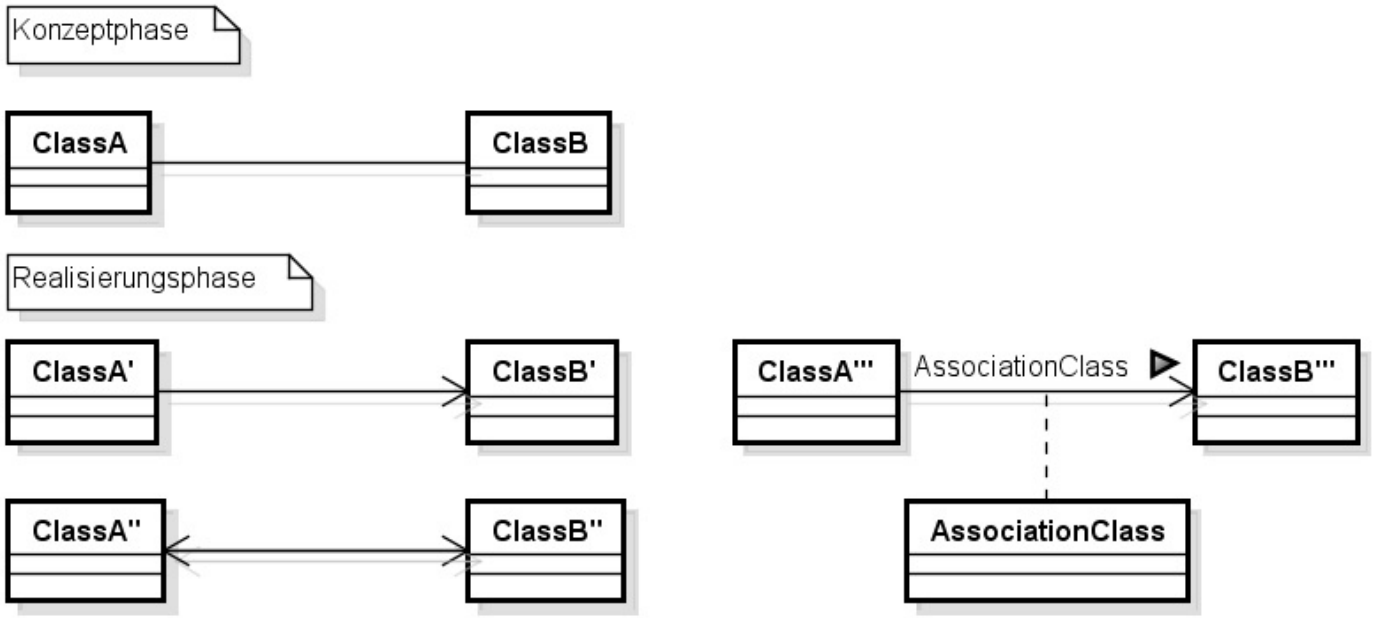


Abbildung A-2 *Beispiel für verschiedene Detaillierungsgrade*

Die Diagramme helfen nicht nur bei Analyse und Dokumentation, sondern sie können verschiedenen Tools als Basis für eine Sourcecode-Erzeugung dienen. Einige Tools können Änderungen am Sourcecode auch wieder zurück ins Modell

übernehmen (***Round-Trip Engineering***).

Modellierungstools

Um direkt ein wenig mit der UML vertraut zu werden, können Sie folgende Modellierungstools aus dem Internet herunterladen und mit diesen etwas experimentieren.

astah

Das Tool astah ermöglicht das Erstellen der wichtigsten UML-Diagramme. Zusätzlich besitzt es viele kleine hilfreiche Funktionen zum Zeichnen der UML-Diagramme und deren Export. Als Highlight erlaubt es, Verzeichnisbäume zu analysieren und daraus Klassendiagramme zu erzeugen. Auch Round-Trip Engineering wird unterstützt. Es steht kostenfrei unter `http://astah.net/editions/` als Community Edition zur Verfügung. Es ist mein Favorit, um schnell, einfach und bequem UML-Diagramme zu erstellen, die auch noch professionell aussehen.

ArgoUML

Ein weiteres frei verfügbares Tool ist ArgoUML. Vorteilhaft ist, dass man keine Installation vornehmen muss. Es existiert eine Version, die sich als Java Web Start ausführen lässt und damit auf jedem Rechner mit einer JVM und einem Webbrowser lauffähig ist. Ein Download ist unter `http://argouml.tigris.org/` möglich.

Enterprise Architect

Enterprise Architect ist ein kommerzielles Tool, für das unter `http://www.sparxsystems.com.au/products/ea/index.html/` eine Probeversion frei verfügbar ist. Der Enterprise Architect bietet viel, erfordert aber auch eine recht lange Einarbeitungszeit.

A.2 Strukturdiagramme – statische Modelle

A.2.1 Klassendiagramme

Ein Klassendiagramm dient zum Modellieren der statischen Struktur eines Softwaresystems und beschreibt beteiligte Klassen, Schnittstellen und deren Beziehungen.

Elemente von Klassendiagrammen

Klassen

Klassen werden als Rechteck dargestellt, das normalerweise in drei Abschnitte aufgeteilt wird. Im oberen Teil stehen der Klassenname und eventuell weitere, ergänzende Informationen, darunter folgen der Attribut- und der Methodenabschnitt. Beide sind optional. Abbildung A-3 zeigt den grundsätzlichen Aufbau.

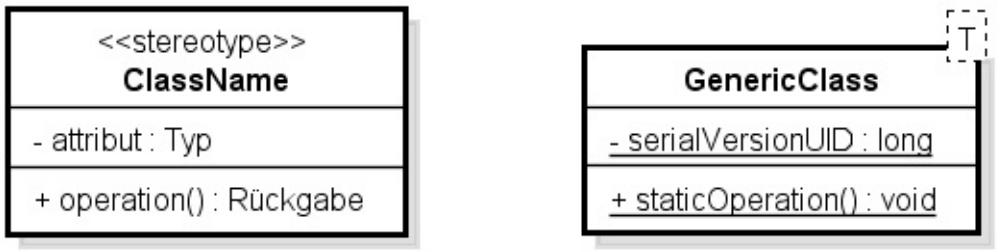


Abbildung A-3 *Struktur von Klassendiagrammen*

Im oberen Abschnitt können gewisse Anmerkungen, sogenannte ***Stereotypen***, notiert werden. Diese spezifizieren etwa, ob es sich um eine Aufzählung (Stereotyp ≪`enum`≫) oder um eine abstrakte Klasse (Stereotyp ≪`abstract`≫) handelt. Bei Letzterer werden sowohl der Name als auch deren abstrakte Methoden kursiv dargestellt.

Für Attribute und Methoden sind verschiedene Sichtbarkeiten möglich, wobei + für `public`, `#` für `protected` und – für `private` steht. Statische Elemente werden unterstrichen. Für generische Klassen erfolgt in der rechten oberen Ecke die Angabe ihrer Typparameter umrandet von einem gestrichelten Kästchen.

Tipp: Sichtbarkeiten

- In einem Klassendiagramm sollten zur besseren Übersicht hauptsächlich `public`-Methoden und evtl. einige `protected`-Methoden dargestellt werden. Damit sollte die Schnittstelle einer Klasse vollständig beschrieben sein. Ist dies nicht der Fall, so existieren direkte Zugriffe auf Attribute und es erfolgt keine vollständige Datenkapselung. Das ist zu vermeiden, um unerwartete Seiteneffekte zu verhindern.
- Methoden mit der Sichtbarkeit `private` spielen bei der Analyse und im Grobdesign kaum eine Rolle. Private Methoden sind selbst im Detaildesign und zur Dokumentation eher selten in Diagrammen sinnvoll. Das liegt daran, dass in frühen Entwurfsphasen solche Details noch nicht interessieren und während der Erstellung eines Softwaresystems auf dieser niedrigen Implementierungsebene einfach zu viel und in zu schneller Folge Änderungen stattfinden. Zum einen sind diese Details kaum relevant für die Dokumentation und zum anderen würde es einen ungeheuren Aufwand erzeugen, Dokumentation und Sourcecode jeweils aktuell und konsistent zueinander zu halten.

Schnittstellen

Für Schnittstellen (Interfaces) existieren verschiedene Darstellungsformen. Die Notation erfolgt normalerweise wie bei Klassen als Rechteck, jedoch nur mit Methodenabschnitt. Alle Methoden sind dort `public` und `abstract`. Im Unterschied zu Klassen wird im Kopfbereich das Stereotyp ≪`interface`≫ notiert. Als alternative Darstellungsform für Interfaces existieren die Symbole des »Lolli« (Kreis mit Linienverbindung zur Klasse) und der »Antenne« (offener Halbkreis mit Linienverbindung zur Klasse). Der »Lolli« wird verwendet, wenn man lediglich darstellen will, dass eine Klasse das Interface implementiert, ohne auf die konkreten Methoden einzugehen. Die »Antenne« zeigt an, dass dieses Interface benötigt wird, um gewisse Aufgaben durchzuführen. Vor allem im Zusammenspiel zwischen verschiedenen Klassen in unterschiedlichen Paketen kann diese Darstellungsweise nützlich sein, da sie die Details einzelner Methoden verbirgt. Abbildung A-4 zeigt die beschriebenen Varianten der Interface-Darstellung.

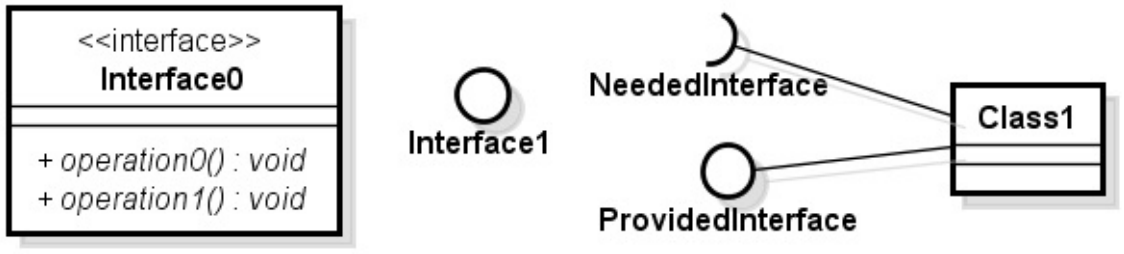


Abbildung A-4 *Verschiedene Darstellungen von Interfaces*

Beziehungen zwischen Klassen und Interfaces

Klassen und Schnittstellen stehen in verschiedensten Beziehungen zueinander. Klassen können nicht nur andere Klassen erweitern oder Interfaces realisieren, sondern diese auch referenzieren, um dort Funktionalität aufzurufen.

Vererbungsbeziehungen

Die Implementierung einer Schnittstelle wird über einen Pfeil mit gestrichelter Linie symbolisiert. Dabei zeigt die Spitze auf das implementierte Interface. Für Ableitungen verwendet man durchgezogene Linien. Hier zeigt die Pfeilspitze auf die Basisklasse. Abbildung A-5 stellt beide Beziehungen dar.

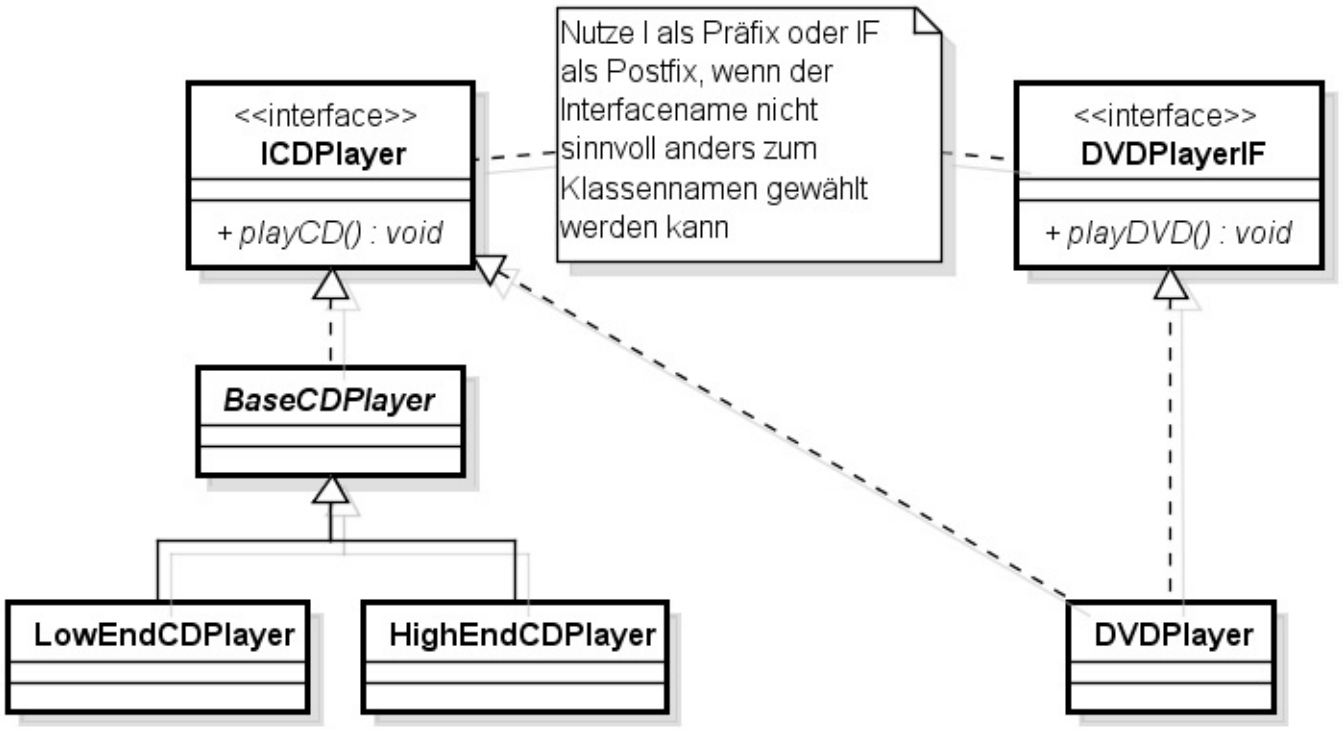


Abbildung A-5 *Interfaces, abstrakte Klassen und Vererbung*

Assoziation

Die Assoziation ist die allgemeinste Form der Beziehung zwischen Klassen oder Schnittstellen. Damit wird lediglich ausgedrückt, dass diese Elemente miteinander verbunden sind. Im Speziellen kann eine solche Assoziation auch wieder auf die ursprüngliche Klasse zurückverweisen, um eine Selbstreferenz auszudrücken.

Eine Assoziation kann mit einem Namen versehen werden und optional auch eine Richtung haben, d. h., die eine Seite »kennt« die andere, aber nicht andersherum. Des Weiteren können auf beiden Seiten einer Assoziation eine ***Multiplizität*** und ein ***Rollenname*** angegeben werden. Die Multiplizität legt die Anzahl möglicher Instanzen fest und kann beliebige Werte (1, 7) oder Wertebereiche (3-5; 7-9) umfassen. Im Speziellen steht ein * für beliebig viele Instanzen. Der Rollenname wird eher selten angegeben, da es in der Regel schwierig ist, dem Assoziationsnamen dadurch noch wertvolle Informationen hinzuzufügen.

Aggregation

Die Aggregation ist ein Spezialfall einer Assoziation, allerdings gibt es in der UML keine exakte Definition von deren Bedeutung. Gewöhnlich versteht man darunter eine Ganzes-Teile-Beziehung, deren verbundene Elemente unabhängig voneinander sind. Eine Aggregation verwende ich meistens dann, wenn ein Objekt eine Menge von anderen Objekten referenziert.

Die Darstellung erfolgt mit einem Rautensymbol an der Klasse, die das Ganze repräsentiert. Abbildung A-6 zeigt sowohl eine Assoziation als auch die Aggregation als eine Spezialisierung davon.

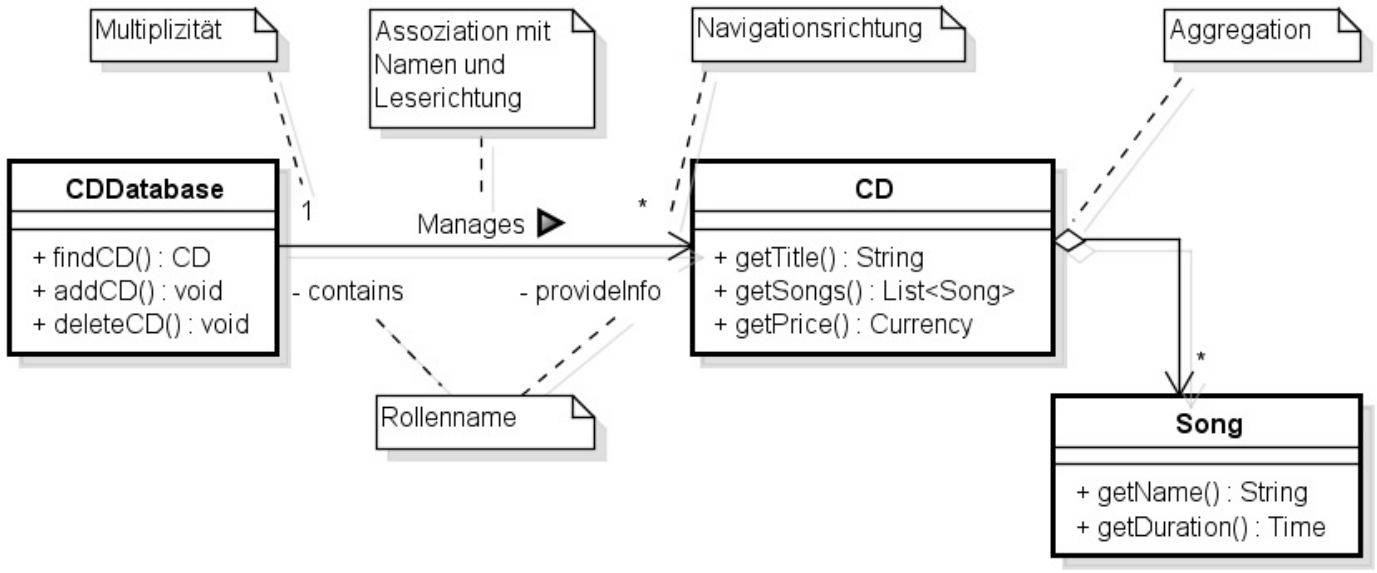


Abbildung A-6 *Assoziation und Aggregation*

Komposition

Die Komposition wiederum ist ein Spezialfall einer Aggregation mit noch stärkerer Bindung. Das referenzierte bzw. die referenzierten Objekt(e) gehören zu genau einer Instanz der besitzenden Klasse (sind »dinglich« enthalten). Endet die Lebenszeit eines Containerobjekts, so müssen für Kompositionen auch die enthaltenen Objekte terminiert werden. Diese Forderung ist in Java nicht direkt umzusetzen, da es keine vom Programmierer aktivierbare Objektlöschung gibt. Stattdessen werden Objekte nur dann durch die Garbage Collection gelöscht, wenn diese von keinem anderen Objekt mehr referenziert werden (vgl. Abschnitt 8.5). ***Komposition erfordert daher in Java, dass nur die Containerobjekte Referenzen auf die enthaltenen Objekte besitzen***. Nur dann ist es dem Garbage Collector möglich, den Speicher der enthaltenen Objekte freizugeben, wenn diese nicht länger von dem Containerobjekt referenziert werden.

Zur Darstellung wird ein gefülltes Rautensymbol verwendet. Abbildung A-7 zeigt dies anhand einer Klasse `Document`. Deren Instanzen sich aus `Paragraph`-Objekten zusammensetzen.

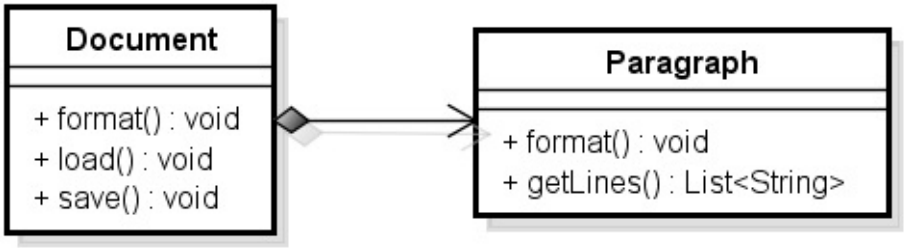


Abbildung A-7 *Komposition*

A.2.2 Objektdiagramme

Objektdiagramme ähneln den Klassendiagrammen. Sie zeigen einen speziellen

Zustand eines Klassendiagramms zu einem definierten Zeitpunkt, d. h. die aktuelle Wertebelegung der Attribute von Objekten. Da einerseits die Anzahl der Attribute groß sein kann und andererseits häufig nur eine gewisse Teilmenge für einen betrachteten Fall interessant ist, erlaubt die Notation, nur eine Auswahl bestimmter Attribute aufzulisten.

Die Darstellung der Objekte erfolgt ähnlich zu den Klassen aus dem Klassendiagramm als Kasten mit der Angabe eines unterstrichenen Instanznamens im Format `Instanzname : Klassenname`.[4] Der Methodenabschnitt entfällt. Ein einfaches Beispiel zweier Objekte ist in Abbildung A-8 visualisiert.

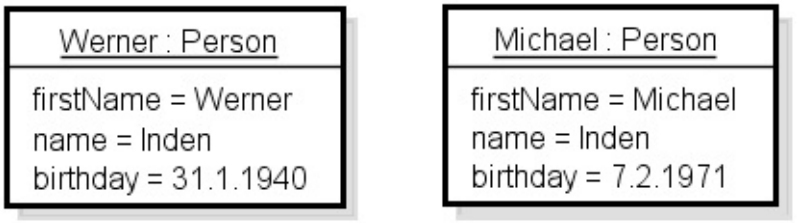


Abbildung A-8 *Objektdiagramm*

A.2.3 Komponentendiagramme

Komponentendiagramme dienen zur Modellierung von Komponenten und deren Zusammenspiel. Komponenten bestehen aus Klassen und Interfaces und zeichnen sich unter anderem dadurch aus, dass mehrere Komponenten zu größeren Komponenten zusammengefügt werden können.

Zwei Komponenten können miteinander arbeiten und verbunden werden, wenn die angebotenen Schnittstellen einer Komponente zu den benötigten Schnittstellen einer anderen Komponente passen. Zur Beschreibung von Abhängigkeiten werden die bereits vorgestellten »Lolli«- und »Antennen«-Symbole verwendet. Vor der UML 2 existierte das »Antennen«-Symbol noch nicht und man musste eine Abhängigkeitsbeziehung nutzen, die als gestrichelter Pfeil gezeichnet wird. Im Beispiel aus Abbildung A-9 ist die Komponente `ReportGenerator` mit den Komponenten `CSVEingang` als Datenlieferant und den Komponenten `HtmlReporter` bzw. `MailReporter` als Datenausgabe verbunden, indem die passenden Schnittstellen zusammengeführt sind.

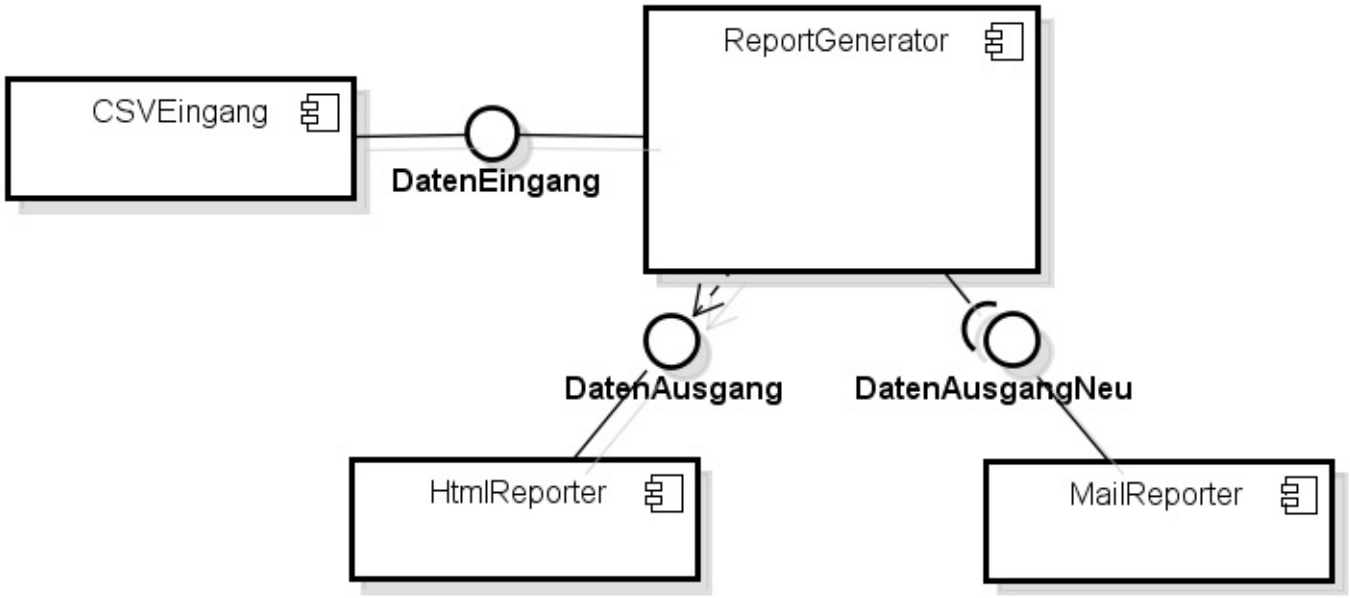


Abbildung A-9 *Komponentenbeispiel*

Komponentendiagramme können verschiedene Detaillierungsstufen enthalten. Die sogenannte ***Blackbox-Sicht*** zeigt lediglich Komponenten und ihre Schnittstellen. Die *Whitebox-Sicht* oder Komponenten-Modellierungssicht zeigt auch die innere Struktur, also Klassen, Interfaces und ggf. weitere enthaltene Komponenten.

A.2.4 Paketdiagramme

Ein Paketdiagramm stellt die Struktur eines modellierten Systems mithilfe von Paketen und Abhängigkeitsbeziehungen dar und zeigt die Unterteilung der Software in einzelne Pakete. Paketdiagramme können sehr nützlich sein, um unerwünschte und unerwartete Abhängigkeiten in einer Analyse eines bestehenden Systems, z. B. vor einem Redesign, zu visualisieren. Nehmen wir an, ein Paket `displaycontrol` hätte drei Abhängigkeiten und eine davon wäre eine unerwartete Abhängigkeit von einem Paket `services`. Abbildung A-10 stellt ein Paketdiagramm dar, das diese Situation beschreibt.

A.3 Verhaltensdiagramme – dynamische Modelle

Das dynamische Verhalten eines Systems lässt sich in verschiedenen Detaillierungsgraden betrachten. Auf sehr abstrakter Ebene kann es durch Anwendungsfalldiagramme beschrieben werden. Auf feingranularer Ebene hilft der Einsatz verschiedener anderer Diagrammtypen, das dynamische Verhalten genauer zu spezifizieren.

A.3.1 Anwendungsfalldiagramme

Ein Anwendungsfalldiagramm beschreibt Anforderungen an ein System aus Sicht

eines Benutzers. Es bietet dazu eine relativ simple Darstellung der gewünschten Abläufe in Form von Strichmännchen-***Akteuren*** und oval dargestellten Anwendungsfällen (sogenannten ***Use Cases***) sowie deren Beziehungen untereinander. Ein Anwendungsfall beschreibt dabei auf grobgranularer Ebene, Anforderungen und Arbeitsabläufe eines zukünftigen Benutzers. Bei Diskussionen mit Kunden oder dem Management über die Systemfunktionalität kann diese Betrachtungsweise sinnvoll sein. ***Durch die starke Abstraktion ist ein Anwendungsfalldiagramm aber eher ein Hilfsmittel zur Anforderungsermittlung und weniger zur Verhaltensbeschreibung und zum Systemdesign***.

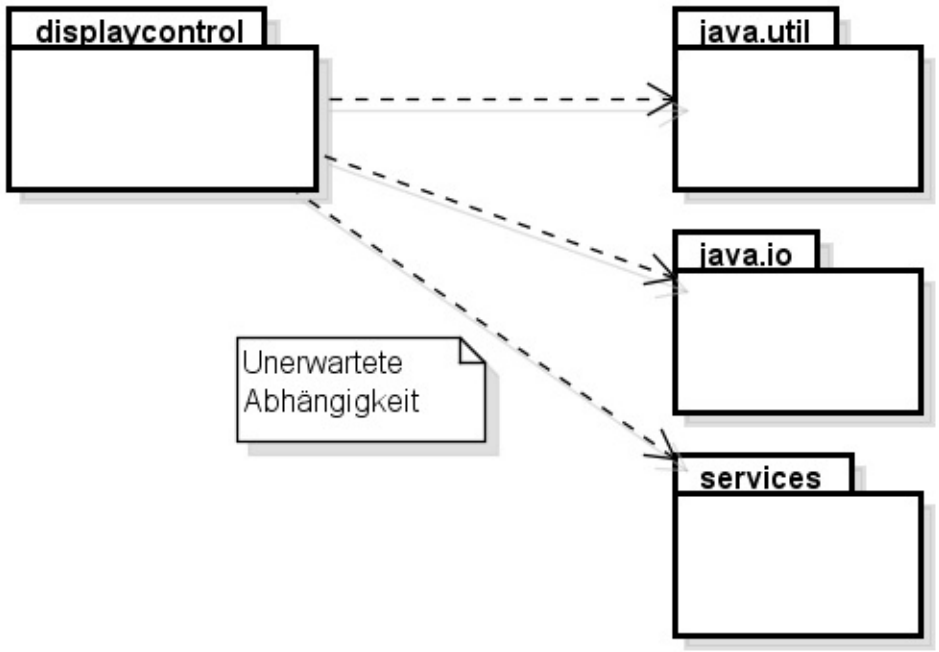


Abbildung A-10 *Abhängigkeiten in Paketdiagrammen*

In Abbildung A-11 wird eine Bestellung in einem Restaurant als Anwendungsfall modelliert.

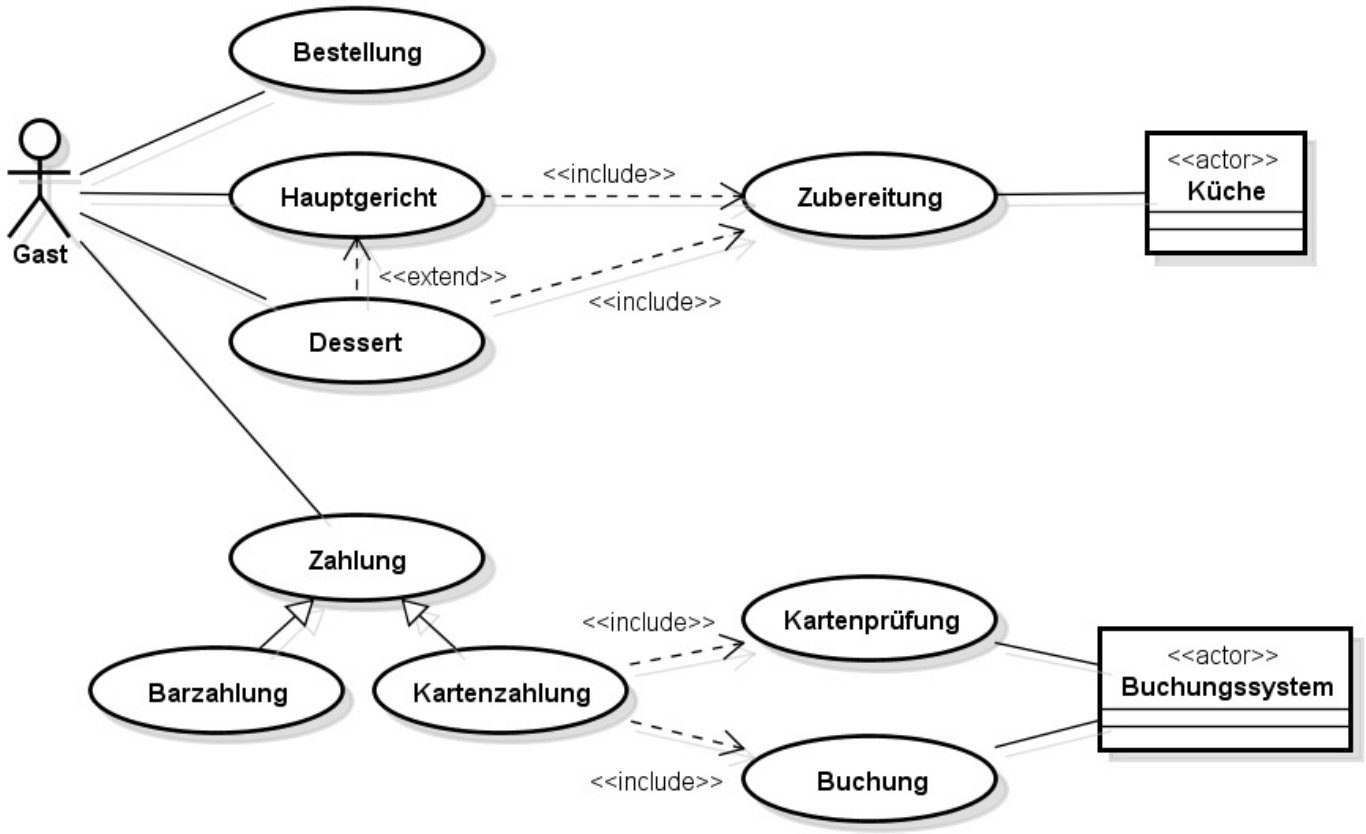


Abbildung A-11 *Anwendungsfall: Bestellung im Restaurant*

Die Idee habe ich aus dem Buch »UML 2.0 – Das umfassende Handbuch« [53] aufgegriffen und hier ein wenig erweitert, sodass sich verschiedene Abhängigkeitsbeziehungen intuitiv verdeutlichen lassen.

Es können also wie auch beim Klassendiagramm sowohl Akteure als auch Anwendungsfälle in Abhängigkeitsbeziehungen stehen. Auch Vererbung ist möglich: Ein Akteur oder Anwendungsfall kann eine Spezialisierung eines anderen Akteurs bzw. Anwendungsfalls sein, hier am Beispiel der verschiedenen Zahlungsweisen gezeigt.

Mit der Notation ≪`include`≫ wird ein Anwendungsfall B in einen bestehenden Anwendungsfall A integriert, d. h., während der Anwendungsfall A abläuft, wird der Anwendungsfall B abgearbeitet und dessen Ergebnis in einem Teilschritt des Anwendungsfalls A verwendet. Die Aktionen im inkludierten Anwendungsfall finden entweder vor oder während des eigentlichen Anwendungsfalls statt. Im Beispiel ist dies für den Anwendungsfall »Hauptgericht« gegeben. Wir erkennen ein weiteres Modellierungsdetail: Der inkludierte Anwendungsfall »Zubereitung« wird durch einen anderen Akteur ausgeführt.

Über die Notation ≪`extend`≫ wird ein Anwendungsfall A unter bestimmten Bedingungen durch einen anderen Anwendungsfall B erweitert. Der Anwendungsfall A ist dabei normalerweise in sich vollständig beschrieben. In diesem Beispiel kommt es je nach Laune des Gasts nach der Abarbeitung des Anwendungsfalls »Hauptgericht« noch zu der Bestellung eines Nachtischs, beschrieben durch den Anwendungsfall »Dessert«. Diese optionale Abarbeitung lässt sich mithilfe einer ≪`extend`≫-Beziehung modellieren.

Granularität von Anwendungsfalldiagrammen

Sollen kompliziertere Sachverhalte dargestellt werden, reicht die abstrakte Ebene der Abbildungen nicht mehr aus: Die in den Anwendungsfalldiagrammen dargestellten Aktionen bestehen tatsächlich aus vielen kleineren Verarbeitungsschritten und komplexen Interaktionen. Die Abbildungen sollten daher lediglich einen Teil der Modellierung mit Anwendungsfällen darstellen. Hört die Modellierung allerdings hier auf, so ist deren Wert relativ gering. Deshalb sollten die Anwendungsfälle mit weiteren detaillierten textuellen Beschreibungen konkretisiert werden. Aus diesen Erläuterungen können mögliche Abfolgen von Verarbeitungsschritten innerhalb eines Anwendungsfalls beschrieben werden. Deren Spezifikation erfolgt durch speziellere Verhaltensdiagramme, die im Folgenden vorgestellt werden.

A.3.2 Sequenzdiagramme

Sequenzdiagramme dienen dazu, Abläufe, sogenannte ***Szenarien***, zu beschreiben, in denen mehrere Objekte per Nachrichtenaustausch kommunizieren. Der Schwerpunkt liegt auf der konkreten chronologischen Abfolge der Nachrichten zwischen beteiligten Objekten bzw. Klassen[5]. Die Systemkomponente, die die Aktion startet, wird häufig zur Unterscheidung von den anderen Objekten als gefüllter Kreis oder als Strichmännchen links oben im Diagramm repräsentiert. Hier beginnt die Abarbeitung der Aktionen durch eine Nachricht an ein rechts daneben befindliches Objekt. Die zeitliche Reihenfolge der Kommunikation und damit die Abfolge der Nachrichten ergibt sich implizit durch die vertikale Anordnung: Die Zeit schreitet in einem Sequenzdiagramm von oben nach unten voran.[6]

Assoziationen zwischen Klassen werden in Sequenzdiagrammen nicht explizit dargestellt. Damit der Nachrichtenaustausch zwischen Kommunikationspartnern erfolgen kann, müssen sich diese kennen. Im Klassendiagramm muss dazu entweder eine Assoziation zugrunde liegen oder zumindest eine Abhängigkeit (Dependency) existieren. Eine Abhängigkeit kann beispielsweise durch Übergabe einer Referenz auf die zu verwendende Klasse ausgedrückt werden.

Elemente von Sequenzdiagrammen

Klassen und Objekte

Die an einem Ablauf beteiligten Objekte oder Teilkomponenten werden als Rechteck mit Namen visualisiert. Die Lebensdauer eines Objekts wird im Sequenzdiagramm als senkrechte gestrichelte Linie, die sogenannte ***Lebenslinie***, unterhalb des Objektsymbols dargestellt. Wenn das Objekt während des im Diagramm abgebildeten Zeitabschnitts erzeugt oder zerstört wird, beginnt oder endet diese Lebenslinie an der entsprechenden Stelle, ansonsten verläuft sie durchgehend von

oben nach unten.

Wenn ein Objekt erzeugt wird, zeigt der Pfeil der erzeugenden Nachricht auf das Rechtecksymbol des neu erzeugten Objekts. Die Zerstörung eines Objekts wird durch ein großes X am Ende seiner Lebenslinie kenntlich gemacht. Die auf der Lebenslinie eines Objekts platzierten Rechtecke kennzeichnen Perioden, in denen dieses Objekt aufgrund einer empfangenen Nachricht eine Aktion ausführt und aktiv ist. Es kann für jedes Objekt mehrere solche Aktivitätsperioden geben. Abbildung A-12 visualisiert dies.

Nachrichten

Der Austausch von Nachrichten zwischen Objekten bedeutet im Grunde nur, dass ein Objekt eine Aktion eines anderen Objekts aufruft – im Normalfall sind dies Methodenaufrufe.[7] Eine Nachricht wird als durchgezogener waagerechter Pfeil von der Lebenslinie des aufrufenden Objekts zu der des empfangenden Objekts gezeichnet, wobei Objekte auch Nachrichten an sich selbst senden können. Der Pfeil wird häufig mit dem Namen der Nachricht und optional mit Parametern versehen. Rückgabewerte können durch eine gestrichelte Linie symbolisch an den Aufrufer zurückgeliefert werden. Beim Nachrichtenaufruf unterscheidet man folgende Formen:

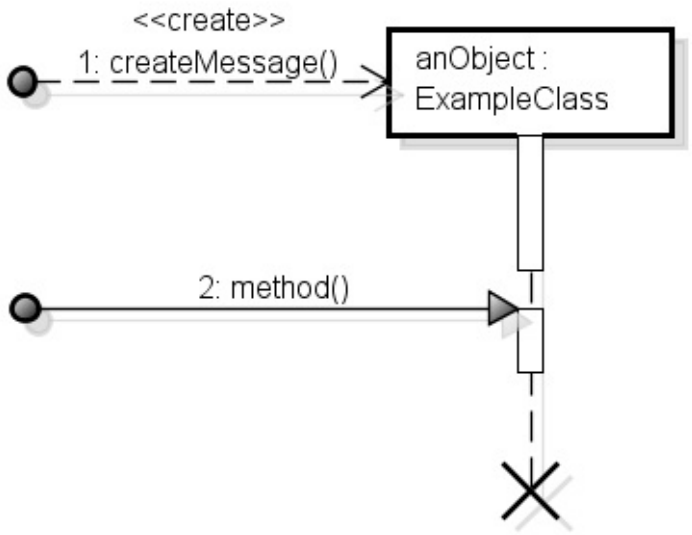


Abbildung A-12 *Objekt in einem Sequenzdiagramm*

- **synchron** – Der Aufrufer wartet so lange, bis der Empfänger die Nachricht verarbeitet hat, und setzt seine Abarbeitung anschließend fort.
- **asynchron** – Der Aufrufer stößt die Aktion nur an. Danach läuft die Abarbeitung parallel. Asynchrone Nachrichten setzen mehrere Threads voraus und können zur Modellierung von Nebenläufigkeit verwendet werden.

Synchrone Aufrufe werden mit einer gefüllten Pfeilspitze gezeichnet. Asynchrone Aufrufe werden mit einem offenen Pfeil markiert. Abbildung A-13 zeigt beide Varianten.

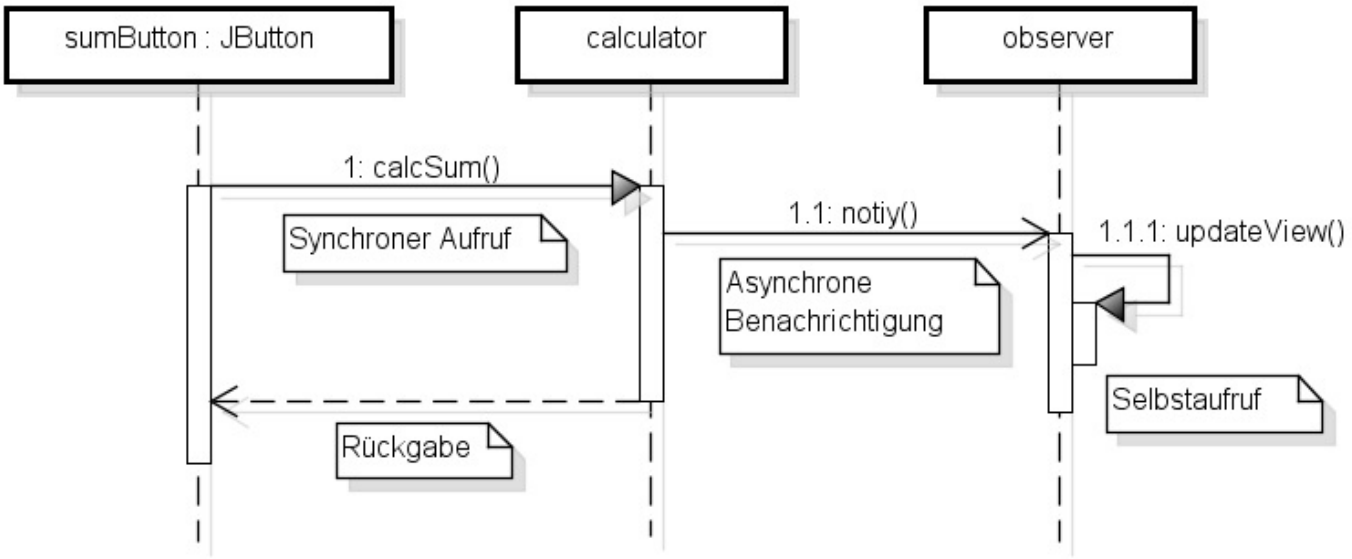


Abbildung A-13 *Nachrichten in Sequenzdiagrammen*

Bedingungen und Schleifen in der UML 1.4

Zum Teil sollen Nachrichten nur ausgeführt werden, wenn eine gewisse Bedingung erfüllt ist. Darüber kann man `if-else`- bzw. `switch-case`-Strukturen abbilden. In der UML 1.4 wird dies durch mehrere von einem Punkt ausgehende Pfeile dargestellt. Zur Auswahl der Nachricht wird eine Bedingung in eckigen Klammern angegeben.

Zur Darstellung von Schleifen bzw. der Wiederholung von Nachrichten wird in der UML 1.4 ein Sternsymbol '*' vor dem Namen der Nachricht gezeichnet. Um die Anzahl der Iterationen festzulegen, kann hinter dem Sternsymbol zusätzlich eine Abbruchbedingung spezifiziert werden. Abbildung A-14 zeigt beide Notationsformen.

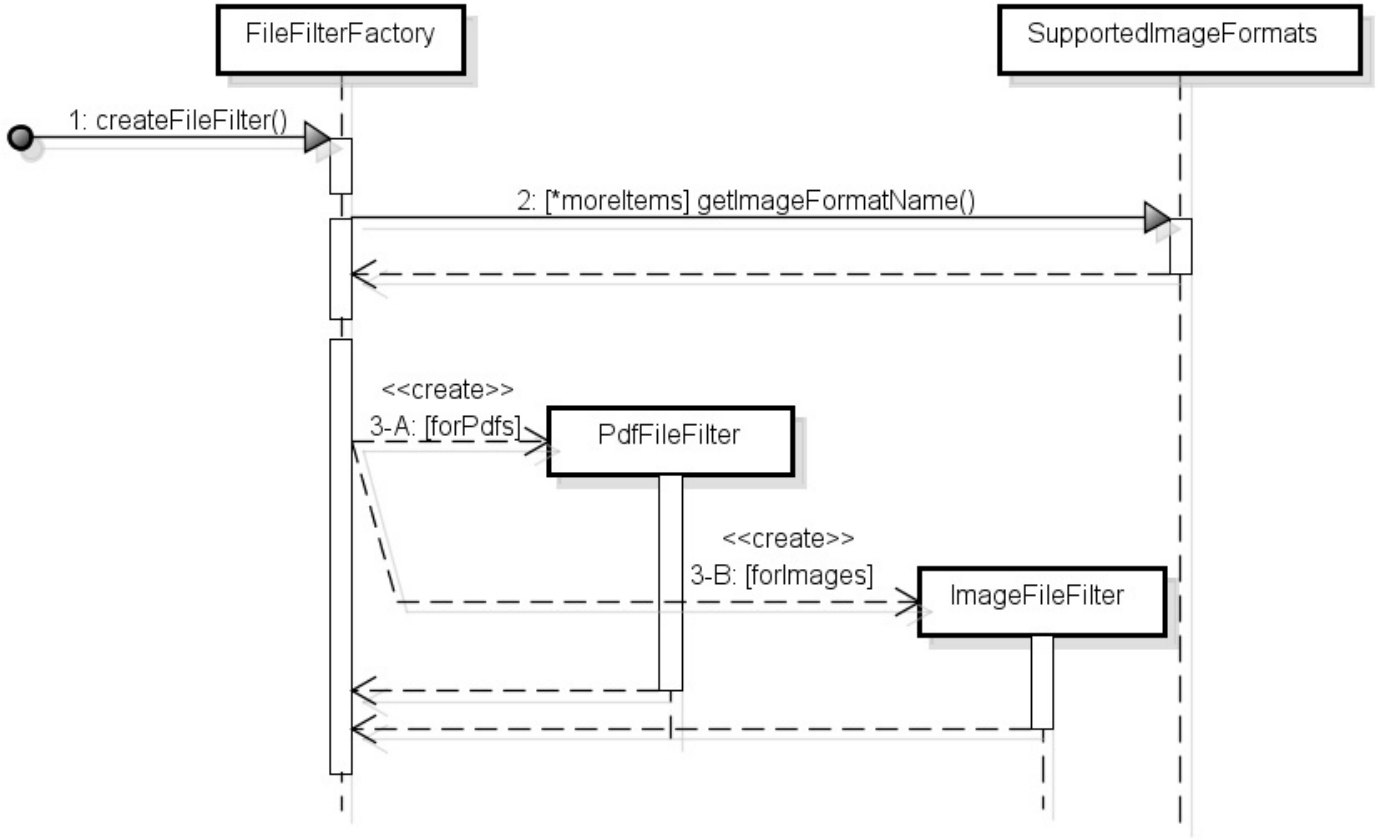


Abbildung A-14 *Schleifen und Bedingungen in Sequenzdiagrammen in der UML 1.4*

Bedingungen und Schleifen in der der UML 2

In der UML 2 werden Bedingungen und Schleifen in separaten Kästchen mit dem Zusatz '`alt`' bzw. '`loop`' dargestellt. Abbildung A-15 zeigt dies.

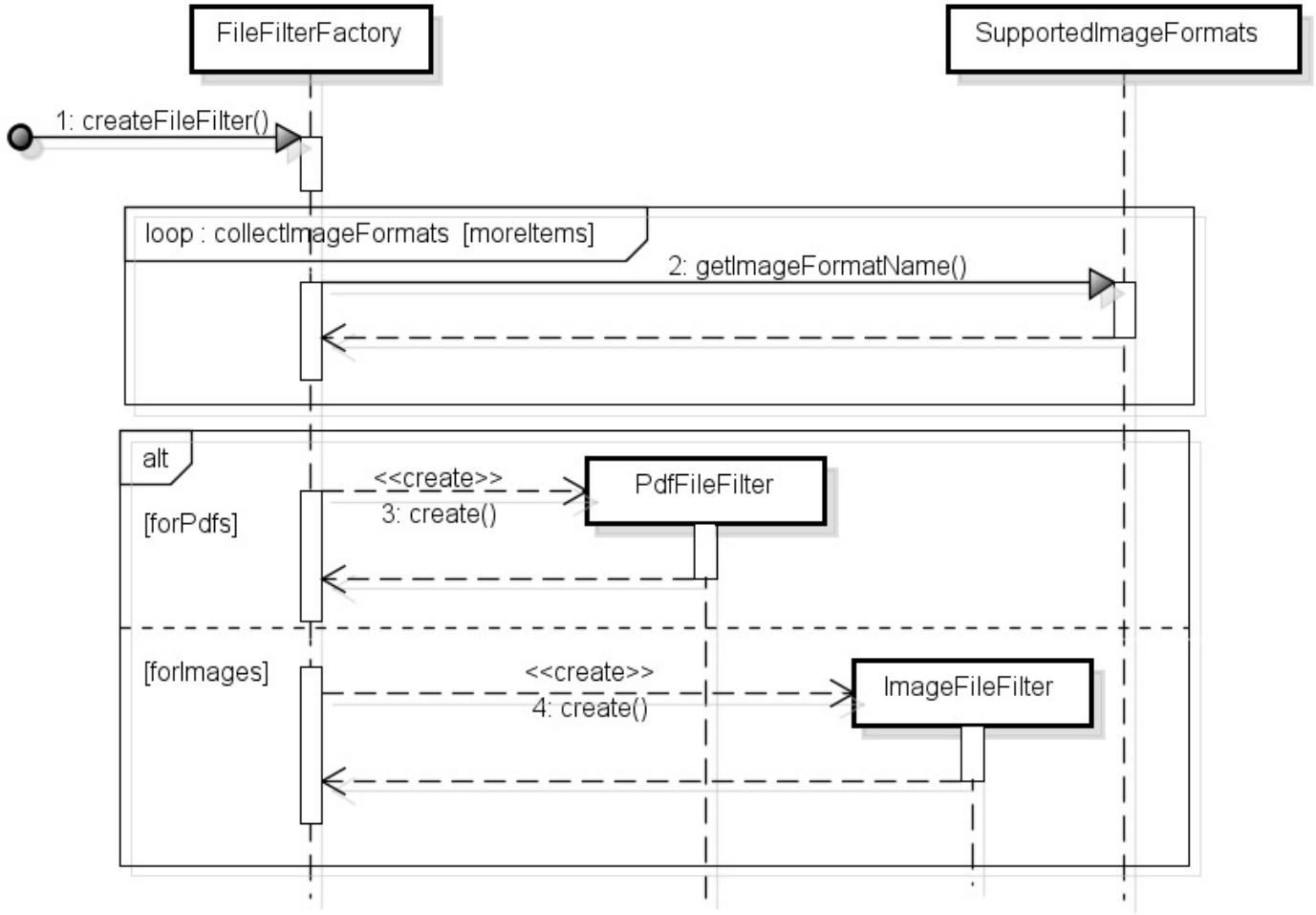


Abbildung A-15 *Schleifen und Bedingungen in Sequenzdiagrammen in der UML 2*

Hinweis: Zeitlicher Verlauf bei Bedingungen und Schleifen

Streng genommen widerspricht die Notation dem Gedanken, dass die Zeit im Sequenzdiagramm von oben nach unten läuft. Die angegebenen Alternativen finden konkurrierend statt und haben keinen zeitlichen Versatz, wie dies durch die Notation angedeutet wird.

Angabe zeitlicher Randbedingungen

Zur Spezifikation zeitlicher Randbedingungen lassen sich gewisse Zeitpunkte angeben und mit Bedingungen versehen, wie dies Abbildung A-16 zeigt.

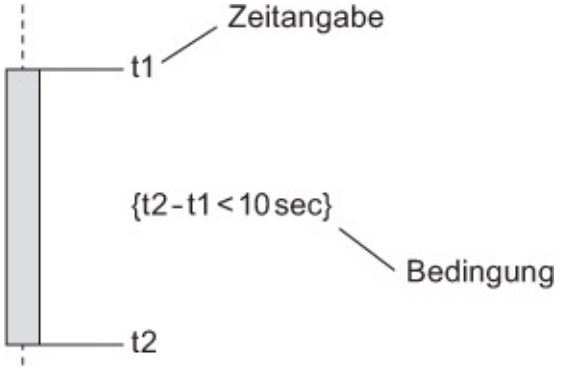


Abbildung A-16 *Zeitliche Randbedingungen in Sequenzdiagrammen*

A.3.3 Kommunikationsdiagramme

Kommunikationsdiagramme, früher unter dem Namen ***Kollaborationsdiagramme*** bekannt, modellieren, wie auch Sequenzdiagramme, Szenarien. Kommunikationsdiagramme betonen allerdings mehr den Nachrichtenfluss und die Aufrufhierarchie bei der Zusammenarbeit und Interaktion von Objekten. Die zeitliche Reihenfolge der Nachrichten steht weniger im Fokus.

Kommunikationspartner und Nachrichten werden analog zum Sequenzdiagramm als Rechtecke und Linien mit den Namen der Operationen dargestellt.[8] Es hat sich in der Praxis bewährt, die zentralen Objekte entweder mittig im Diagramm oder links oben zu positionieren. Die dargestellten Interaktionen gehen dann hauptsächlich aus der Diagrammmitte zum Rand oder in der bevorzugten Leserichtung von links nach rechts. Dies zeigt Abbildung A-17.

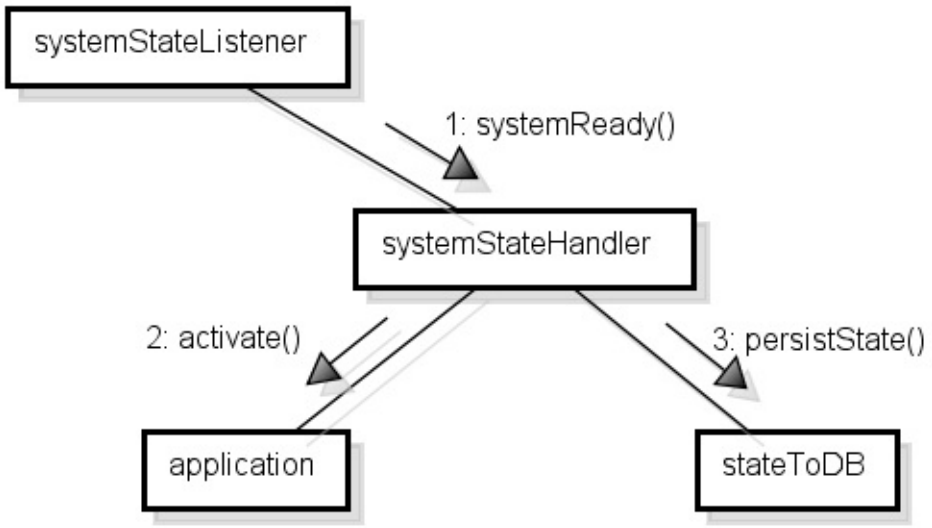


Abbildung A-17 *Beispiel eines Kommunikationsdiagramms*

Über Pfeile und eine Nummerierung der Nachrichten wird die Richtung und die zeitliche Abfolge ausgedrückt – der zeitliche Ablauf ist dadurch manchmal jedoch nicht so klar wie in Sequenzdiagrammen zu erkennen. Ein Rückpfeil als Symbol einer Rückantwort existiert hier ebenso wenig wie eine Unterscheidung zwischen synchronen und asynchronen Nachrichten. Teilweise ist es hilfreich, die Nachrichten mit einem Hinweis für die Aufrufart (synchron, asynchron) zu versehen.

A.3.4 Zustandsdiagramme

In der theoretischen Informatik beschreibt man Zustände und Übergänge zwischen diesen mit sogenannten Automaten. Das lässt sich in der UML mithilfe von Zustandsdiagrammen modellieren. Zustände besitzen einen Namen und werden als Rechteck mit runden Ecken dargestellt. Aus einem momentanen Zustand kann in einen Zielzustand gewechselt werden (***Zustandsübergang*** bzw. ***Zustandstransition*** genannt). Ein solcher

Übergang kann an Bedingungen gebunden sein oder durch Ereignisse ausgelöst werden. Weiterhin kann beim Übergang eine Aktion erfolgen. All dies ist optional und kann bei der Modellierung bei Bedarf angegeben werden. Abbildung A-18 zeigt die entsprechende Notation.

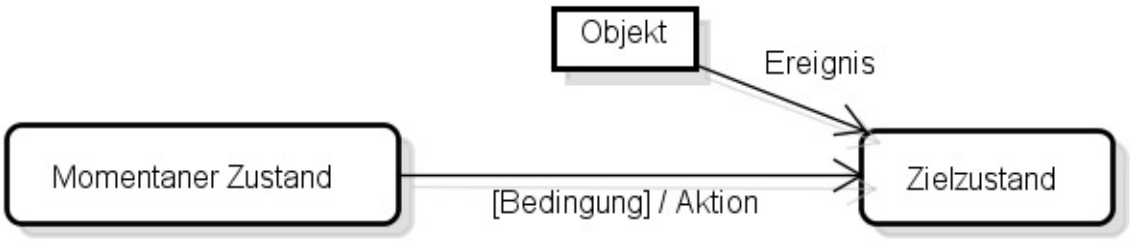


Abbildung A-18 *Zustand mit Transition*

Im Detailentwurf können für jeden Zustand zusätzlich noch spezielle Aktionen spezifiziert werden, und zwar für das Eintreten in diesen Zustand, das Verlassen des Zustands und Aktionen, die zyklisch während des Zustands ausgeführt werden. Zusätzlich zu den »normalen« Zuständen gibt es noch zwei explizite künstliche Zustände: den Start- und den Endzustand. Der durch einen gefüllten Kreis dargestellte Startzustand modelliert immer einen definierten Ausgangspunkt. Endzustände geben das Ende der Zustandsübergänge und damit des modellierten Verhaltens an. Diese werden als ausgefüllter Kreis mit Umrandung dargestellt. Abbildung A-19 zeigt dies.

Abbildung A-19 *Zustand mit Aktionen*

Beispiel

Betrachten wir als komplexeres Beispiel ein verteiltes System, in dem jede Komponente sich zunächst selbsttätig initialisiert und bei einem »NameService« registriert. Danach benachrichtigen sich die einzelnen Komponenten gegenseitig über die Bereitschaft zur Zusammenarbeit mit einer `SystemReady`-Meldung. Erst dann wechseln sie in den Zustand `SystemReadyAndRunning`. In diesem Zustand verweilen sie, bis entweder das Applikationsfenster geschlossen wird oder aber eine Exception auftritt. In letzterem Fall startet sich das System automatisch wieder neu und der Ablauf im Zustandsdiagramm beginnt wieder mit dem Zustand `Initial`. Den Ablauf verdeutlicht Abbildung A-20.

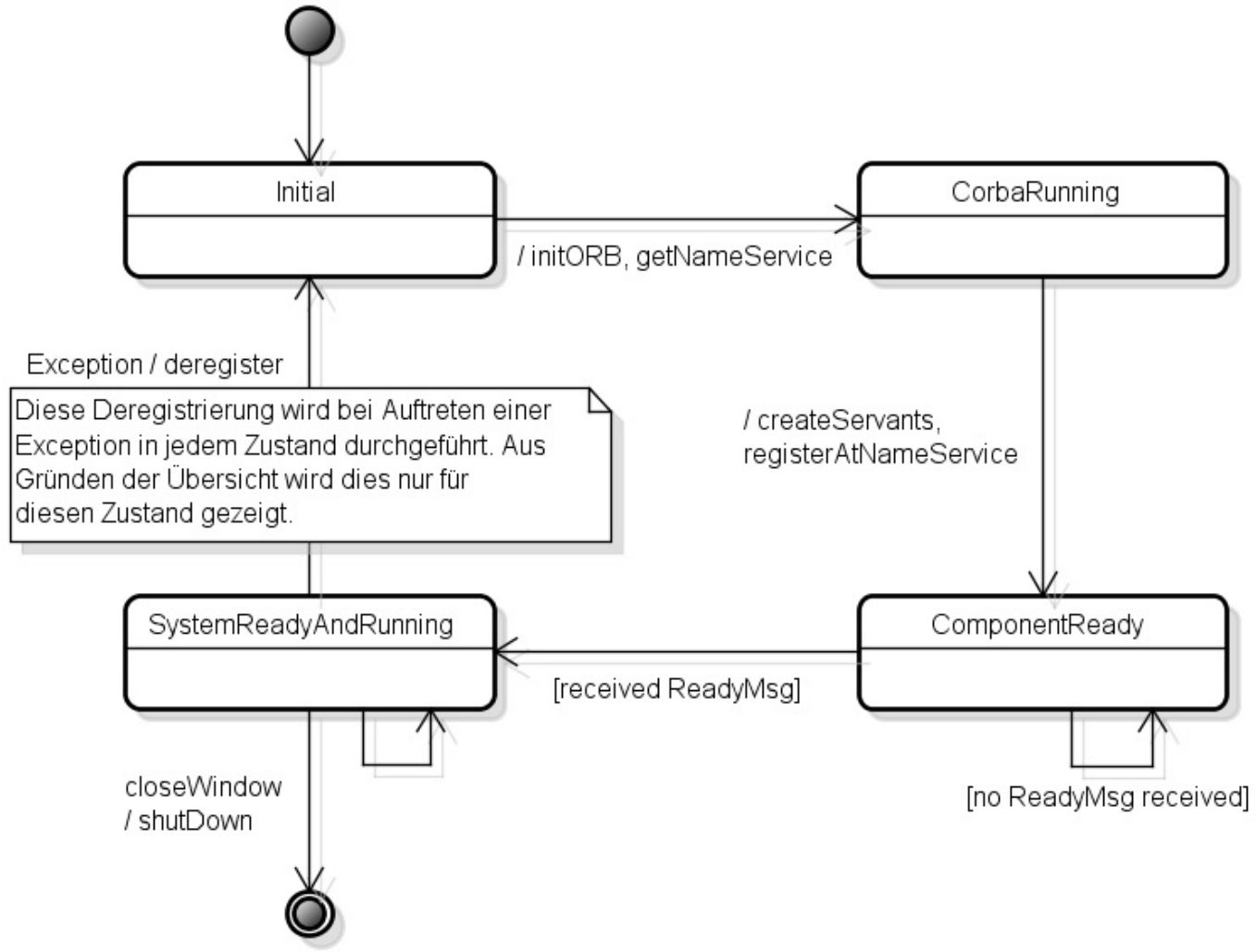


Abbildung A-20 *Zustandsdiagramm eines verteilten Systems*

A.3.5 Aktivitätsdiagramme

Aktivitätsdiagramme dienen der Darstellung von Abläufen auf der Basis von Aktivitäten und sind den Zustandsdiagrammen ähnlich. Aktivitätsdiagramme modellieren allerdings eher die Dynamik in Form von Aktionen – wohingegen in Zustandsdiagrammen mehr die Ergebnisse der Aktionen, d. h. die erreichten Zustände, modelliert werden.

Aktivitäten besitzen einen Namen und werden als Rechteck mit runden Ecken dargestellt (Abbildung A-21). Nach Abschluss einer Aktivität kann eine neue Aktivität ausgeführt werden. Dieser Übergang (Transition) kann an gewisse Bedingungen gebunden sein. Mehrere Aktivitäten können später wieder in einer Aktivität zusammengeführt werden.

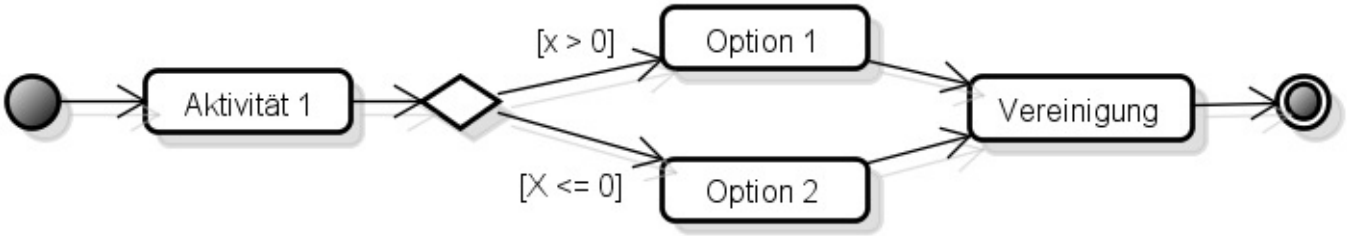


Abbildung A-21 *Aktivitäten und Transitionen*

Zur Modellierung von Nebenläufigkeit dienen Synchronisierungspunkte. Dadurch lässt sich das synchronisierte Aufsplitten und Zusammenführen von Aktionen darstellen. An einer Verzweigung können die Nachfolgeaktivitäten unabhängig voneinander starten. An einer Zusammenführung wird die Nachfolgeaktivität erst dann begonnen, wenn alle Vorgängeraktivitäten abgeschlossen sind. Die Notation zeigt Abbildung A-22.

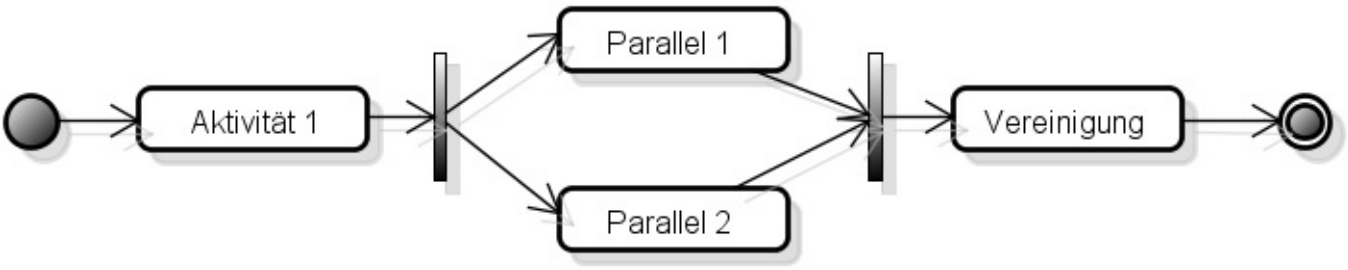


Abbildung A-22 *Aktivitäten und Synchronisation*

Mit Aktivitätsdiagrammen lassen sich, im Gegensatz zu Zustandsdiagrammen, sehr gut mehrere beteiligte Systeme modellieren. Als Hilfsmittel zur Unterteilung dienen sogenannte »***Swimlanes***«. Diese partitionieren die Aktivitäten verschiedener beteiligter Partner, um z. B. bestimmte Schritte eines Ablaufs unterschiedlichen Akteuren zuzuordnen.

Beispiel

Als Beispiel modellieren wir eine Kaffeebestellung in einem Café. Die Idee dazu habe ich aus dem Buch »Objektorientierte Softwareentwicklung mit UML« von Peter Forbrig [25] aufgegriffen und hier leicht abgewandelt.

In diesem Beispiel sind die involvierten Rollenspieler ein Gast, ein Kellner und die Küche. Deren Aktivitäten werden in eigenen Swimlanes modelliert. Neben der Bestellung des Kaffees beim Kellner und der anschließenden Zubereitung durch die Küche, möchte der Gast während der Wartezeit gern Zeitung lesen. Abbildung A-23 visualisiert eine Umsetzung in Form eines Aktivitätsdiagramms.

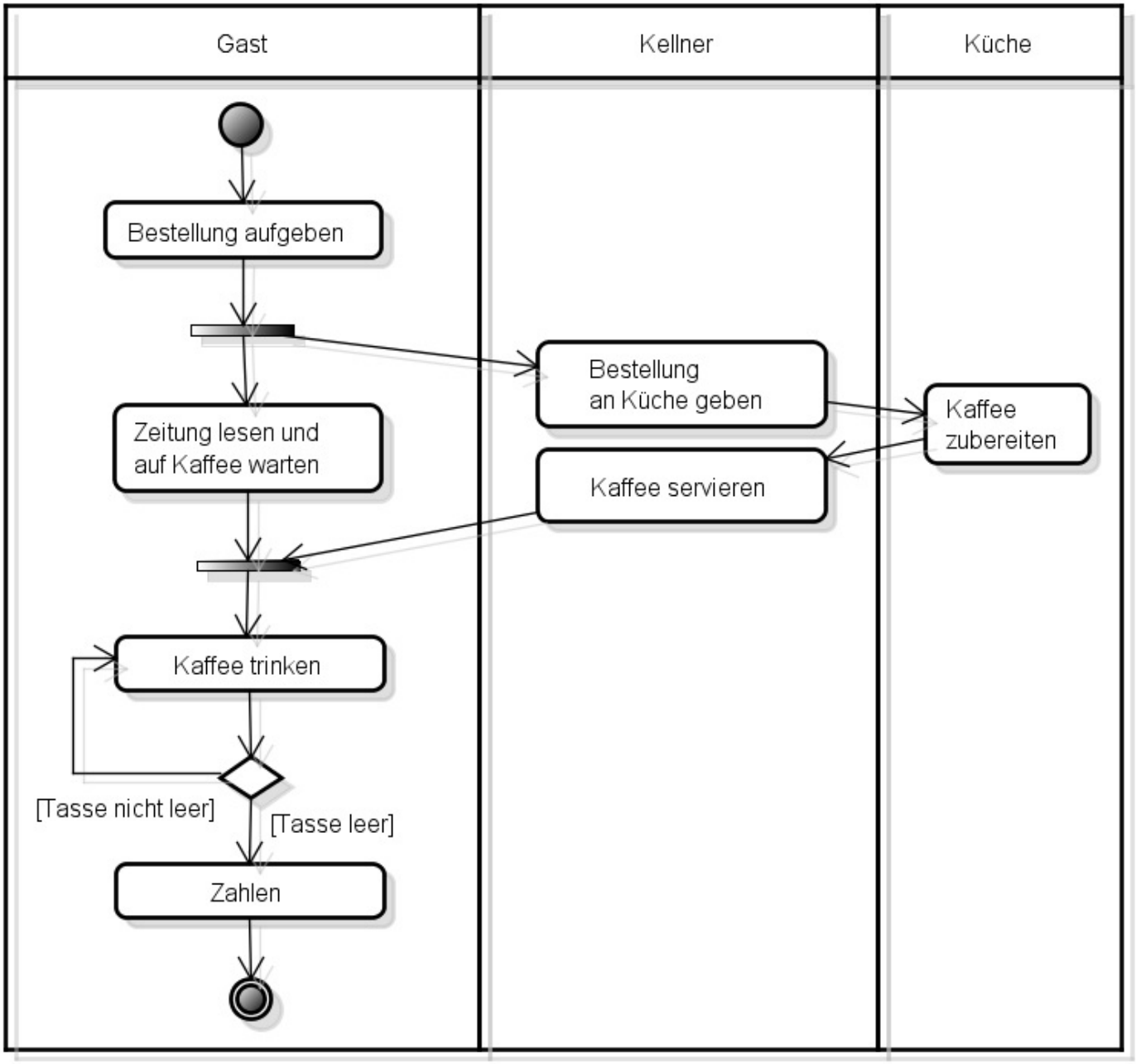


Abbildung A-23 *Aktivitätsdiagramm Kaffeebestellung*

A.4 Weiterführende Literatur

Dieser Anhang hat einen einführenden Überblick in die UML gegeben. Detaillierte Informationen finden Sie unter anderem in folgenden Büchern:

- »**Das UML-Benutzerhandbuch**« von Grady Booch, Jim Rumbaugh und Ivar Jacobson [11]
- »**UML 2.0: Das umfassende Handbuch**« von Christoph Kecher [53]

B Überblick über den Softwareentwicklungsprozess

Die grundsätzlichen Aktivitäten und Tätigkeiten in der Softwareentwicklung können durch sogenannte ***Vorgehensmodelle*** beschrieben werden. Ziel ist es, den komplexen Prozess des Softwareentwurfs in einzelne handhabbare Phasen zu unterteilen. Dieses Kapitel gibt einen kurzen Einblick in einige bekannte Vorgehensmodelle.

B.1 Vorgehensmodelle

Als Grundlage für die im Anschluss vorgestellten Vorgehensmodelle betrachten wir zunächst eine Aufteilung des Softwareentwurfs in verschiedene Phasen. Einige Aktivitäten müssen nicht unbedingt nacheinander durchgeführt werden, sondern können durchaus parallel erfolgen. Gerade aktuellere Vorgehensmodelle betonen das iterative Entstehen von Software und propagieren eine Überschneidung einzelner Phasen.

B.1.1 Aufgaben und Phasen beim Softwareentwurf

Beim Softwareentwurf werden die in einer Anforderungs- und Analysephase gesammelten Erkenntnisse in ein Programm abgebildet. Das geschieht in einer Design- und Implementierungsphase. In einer Testphase prüft man das entstandene Programm auf die Realisierung der gewünschten Funktionalität. Nach einer Auslieferung an Kunden beginnt die Wartungsphase, in der eventuell verbliebene Fehler ausgemerzt werden. Klassischerweise werden folgende Phasen beim Softwareentwurf unterschieden:

- **Anforderungsanalyse** – In dieser Phase erfolgt eine ***Istanalyse*** des zu modellierenden Systems mit einer Beschreibung von Vorgängen und Abläufen der umzusetzenden Funktionalität. Hierbei nutzt man normalerweise die Sprache und Begriffe des Kunden. Als Resultat entsteht ein ***Sollkonzept*** mit erwarteten Eigenschaften an die zu erstellende Software – hier geht es zunächst um das »Was«, nicht um das »Wie«.
- **Analyse und Design** – Während der Phase der technischen Analyse und des Designs wird eine Verfeinerung des Sollkonzepts und eine Beschreibung der wichtigsten Abläufe vorgenommen. Des Weiteren wird die Struktur des zu erstellenden Systems festgelegt. Es entsteht ein erstes Modell in Form von Paketen, Klassen sowie Interaktionen und dabei ausgetauschten Informationen. Im Design wird dann konkretisiert, wie die einzelnen Aufgaben und Probleme gelöst werden können.

- **Implementierung** – In der Implementierungsphase erfolgt eine schrittweise Umsetzung des zuvor entworfenen Systems in Sourcecode. Entwicklungsbegleitend bieten sich kleinere Tests einzelner Methoden und Klassen an. Bei größeren Systemen erfolgt zu bestimmten Zeitpunkten eine Integration neu entstandener Programmteile in bereits bestehende Komponenten.
- **Qualitätssicherung und Test** – Während der Phase des Tests wird die Software auf ein mögliches Fehlverhalten oder Verletzungen anderer Anforderungen, etwa nicht angemessener Antwortzeiten, untersucht. Diese Phase sollte bevorzugt entwicklungsbegleitend erfolgen. Somit lassen sich die Testaufwände und Fehlerkorrekturen vor der Fertigstellung eines Release in der Regel deutlich reduzieren.
- **Auslieferung und Wartung** – Die Software wird beim Kunden installiert und nach einem Abnahmetest eingesetzt. Anschließend ist die Software zu pflegen, d. h., in Tests oder im Praxiseinsatz gefundene Fehler sind zu beheben. Weiterhin sind häufig kleinere Anpassungen und Erweiterungen vorzunehmen.

Nachdem wir kurz die Phasen des Softwareentwurfs kennengelernt haben, betrachten wir nun einige Vorgehensmodelle.

B.1.2 Wasserfallmodell und V-Modell

Aus dem Software Engineering sind das ***Wasserfallmodell*** und das ***V-Modell*** zwei bekannte Vertreter. Beide Verfahren beschreiben die verschiedenen Phasen der Projektdurchführung und in welcher Reihenfolge diese durchlaufen werden.

Das Wasserfallmodell

Das in Abbildung B-1 dargestellte Wasserfallmodell hat seinen Namen daher, dass jeder neue Schritt bzw. jede neue Phase im Entwicklungsprozess zeitlich später und bildlich eine Stufe tiefer angesiedelt ist. Der Projektverlauf folgt damit Wasser, das Stufen hinabfließt. Den Abschluss einer Phase bilden immer spezielle Abnahmen, d. h. Prüfungen, ob gewisse Ziele erreicht wurden. Die ersten Versionen des Wasserfallmodells besaßen keine Rückkopplungen zwischen den Phasen. Demnach verweilte man so lange in einer Phase, bis alle Prüfungen bestanden wurden. Dies war jedoch inkonsistent: Wurden beispielsweise erst beim Testen Designfehler aufgedeckt, so verblieb man in der Testphase und es wurden neue Designs entworfen oder Implementierungen durchgeführt. Um diese Inkonsistenz von tatsächlicher Tätigkeit und Phase aufzulösen, wurden später Rückkopplungen in das Modell eingefügt. Diese ermöglichen es, bei neuen Erkenntnissen, auftretenden Unstimmigkeiten oder Problemen während der Durchführung einer Phase wieder eine oder mehrere Phasen zurückzuspringen und dort an Erweiterungen oder Verbesserungen zu arbeiten.

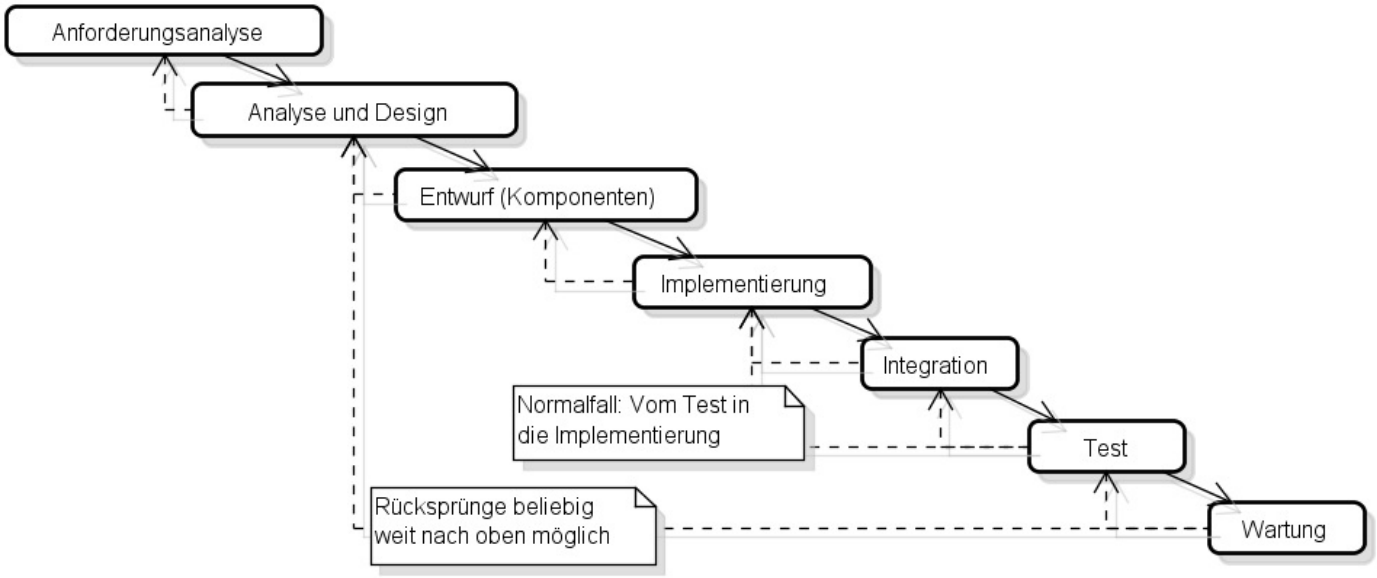


Abbildung B-1 *Wasserfallmodell*

Das V-Modell

Das V-Modell ist eine Weiterführung des Wasserfallmodells. Der Name ergibt sich aus der V-förmigen Darstellung der jeweiligen Aktivitäten des Softwareentwicklungsprozesses (vgl. Abbildung B-2). Die Aktivitäten aus den Phasen des Wasserfallmodells werden dabei auf der linken Seite des Vs abgebildet. Die Tests auf der rechten Seite des Vs korrespondieren mit den Abnahmen, die eine Phase des Wasserfallmodells abschließen.

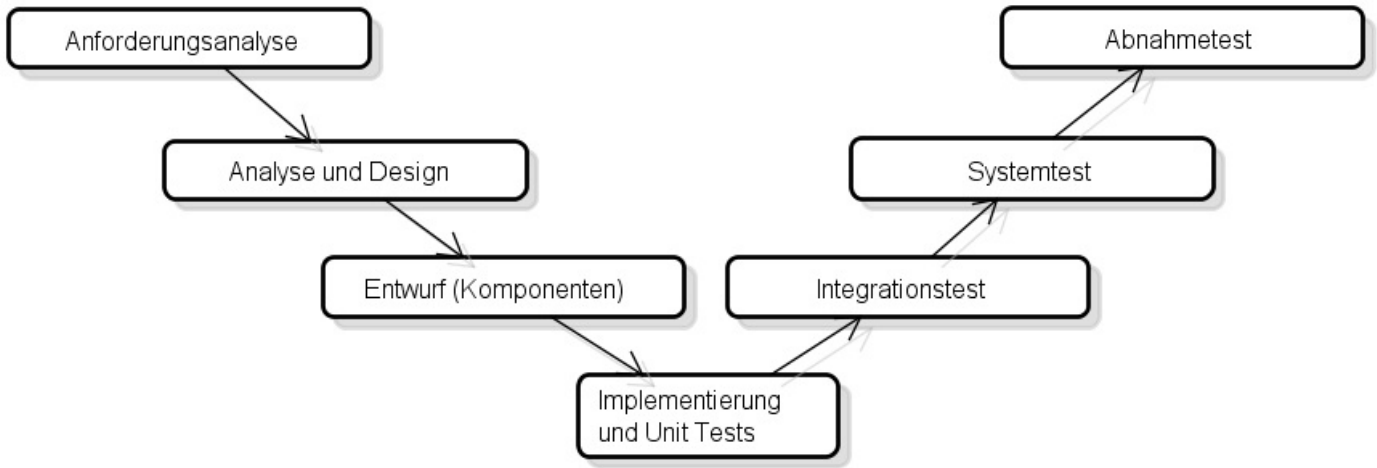


Abbildung B-2 *V-Modell*

Weiterführende Informationen finden Sie etwa im Buch »Software Engineering« von Bernd-Uwe Pagel und Hans-Werner Six [87].

Probleme beider Modelle

Viele gescheiterte Softwareprojekte haben gezeigt, dass die zuvor beschriebenen Modelle manchmal zu schwergewichtig und starr sind und der Dynamik der Softwareentwicklung nicht genug Rechnung tragen. ***Ironischerweise beschreibt der***

»Erfinder« des Wasserfallmodells Winston Royce in seiner Publikation »Managing the Development of Large Software Systems«[1] die Starrheit von Softwareentwicklungsprozessen bereits 1970 als einen häufig praktizierten Irrweg und ein fehleranfälliges Vorgehen bei der Softwareentwicklung. Schon damals schlägt Royce iterative Vorgehensweisen vor. Einige Jahrzehnte später sind die Erkenntnisse dann zum Allgemeinwissen geworden. Peter Coad und Mark Mayfield propagieren in ihrem Buch »Design mit Java« [15] den iterativen agilen OO-Entwurf und schreiben zwei sehr passende Sätze:

- »Änderungen sind unvermeidlich. Die frühen Methoden der Softwareentwicklung bekämpften Änderungen noch.«
- »Effektives objektorientiertes Design akzeptiert ständigen Wechsel.«

Die klassischen Modelle besitzen aufgrund der sequenziellen Abarbeitung der Phasen das Problem, dass bei Zeitdruck häufig versucht wird, »am Test zu sparen«. Weniger Tests führen in der Regel aber zu schlechterer Qualität. Außerdem ist bei diesen klassischen Modellen problematisch, dass Kunden nur zu Beginn und zur Auslieferung in den Erstellungsprozess involviert sind. Da ansonsten keine Kommunikation mit dem Kunden vorgesehen ist, besteht die Gefahr, dass sich das gewünschte Produkt stark von dem tatsächlich gelieferten unterscheidet. Abbildung B-3 deutet dies an.

Abbildung B-3 *Diskrepanz zwischen Wunsch und Realität*

Abhilfe wollen sogenannte agile Vorgehensmodelle schaffen, indem sie fortlaufend neue Releases und damit gebrauchsfähige Softwarestände erstellen, die inkrementell durch Rückmeldungen des Kunden weiterentwickelt werden. Daher ist das Thema agiler Softwareentwurf momentan aktuell. Ein empfehlenswertes Buch zu dem Thema ist »Agile Java-Entwicklung in der Praxis« von Michael Hüttermann [47]. Bekannt wurden die Ideen der agilen Softwareentwicklung unter anderen durch die von Kent Beck vorgeschlagenen Vorgehensmodelle ***Extreme Programming*** (XP) und ***Test-Driven Development*** (TDD), die in den folgenden Abschnitten B.1.3 und B.1.4 vorgestellt werden. Diese Modelle berücksichtigen die Dynamik beim Softwareentwurf besser als das zu starre Wasserfall- bzw. V-Modell. Es werden jedoch nicht komplett neue Ideen entwickelt, sondern vielmehr die Phasen der anderen Modelle in kleinere Zyklen unterteilt. ***Das Revolutionäre daran ist allerdings, dass ein solch iteratives, sich ständig neuen Gegebenheiten anpassendes Vorgehen im Gegensatz zu anderen Ingenieurdisziplinen derart nur beim Softwareentwurf praktiziert werden kann***. In anderen Disziplinen sind zwar auch gewisse iterative Verfeinerungen möglich, jedoch sind diesen viel engere Grenzen

gesetzt.

B.1.3 Extreme Programming (XP)

Unter XP versteht man das von Kent Beck entwickelte Extreme Programming, das im Folgenden kurz vorgestellt wird. Eine umfassende Darstellung bietet sein Standardwerk »Extreme Programming. Das Manifest« [4].

XP propagiert einfaches Design und unterteilt die Entwicklung einer Anwendung in sogenannte Releases, eigenständige, funktionierende Versionen der Anwendung. Diese enthalten zu Beginn der Entwicklung zunächst nur einen kleinen Teil der Funktionalität. Jedes neue Release erweitert das vorherige um etwas neue Funktionalität. Auf diese Weise werden kurze Releasezyklen möglich, und man kann schnell auf neue oder geänderte Anforderungen reagieren. Zusätzlich können Akzeptanztests mit Kunden durchgeführt werden, um Rückmeldungen möglichst schnell in kommende Releases aufnehmen zu können. Somit können dort nur tatsächlich benötigte Funktionalitäten einfließen. Man spricht hier von *Lean Management*.

Implementierungsbegleitend finden zur Qualitätssicherung ständig Unit Tests statt. Um die Entwicklung im Team zu vereinfachen, propagiert XP ein häufiges Integrieren der Sourcen in das Repository. Dies sollte zumindest einmal täglich erfolgen. Die kontinuierliche Integration im Zusammenspiel mit Unit Tests sorgt dafür, dass mögliche Programmfehler umgehend während der Implementierung erkannt werden können. Da als Vorbedingung gilt, dass alle Unit Tests bisher funktionierten, kann ein Defekt nur durch das Einspielen der neuen Sourcen verursacht worden sein. Auch die Anzahl potenziell Fehler verursachender Dateien ist durch die kontinuierliche Integration überschaubar. Außerdem fällt die Integration in das Repository leichter, weil die Wahrscheinlichkeit für Konflikte in Dateien gering ist. Selbst wenn es zu Konflikten kommt, betreffen sie meistens nur kleinere Abschnitte des Sourcecodes.

In Diskussionen und Abstimmungen in täglichen Stand-up-Meetings, kurzen Informationsrunden, berichtet jeder Entwickler von seinen aktuellsten Entwicklungen und Problemen. Zum einen können so Doppelentwicklungen und Konflikte beim Integrieren vermieden werden. Zum anderen kann das Wissen aller Entwickler angezapft werden. Ist ein Problem allerdings nicht in kurzer Zeit lösbar, so sollte sich ein kleineres Team zur Problemlösung im Anschluss an ein solches Meeting separat damit auseinandersetzen. Eine weitere wichtige XP-Technik ist die sogenannte Retrospektive. Damit ist gemeint, aus den bisher gemachten Erfahrungen und Fehlern zu lernen und die Erkenntnisse bei weiteren Entwicklungen zu berücksichtigen.

Der programmierte Sourcecode wird ständig kritisch betrachtet und entwickelt sich durch fortlaufende Refactorings immer weiter. Es wird größter Wert auf SourcecodeSauberkeit gelegt. Dazu benötigt man einen Satz an sogenannten Coding

Conventions. Ausführliche Informationen dazu liefert Kapitel 19. XP empfiehlt ***Pair Programming***. Darunter versteht man, dass zwei Entwickler gemeinsam programmieren. Dabei ist jeweils eine Person der aktive Part und implementiert. Der andere Entwickler schaut zu und dient als Diskussionspartner und überwacht gleichzeitig die Einhaltung der Coding Conventions. Dadurch findet einerseits ein Mini-Codereview (siehe Kapitel 21) statt und andererseits lernen beide Entwickler voneinander. Weiterhin entwickelt sich ganz nebenbei ein gemeinsames Qualitäts- und Sourcecode-Verständnis sowie ein Verantwortungsgefühl. Dies wird ausdrücklich von XP propagiert und ich empfinde es als einen großen Vorteil, denn es kann sich kein Entwickler mit Sätzen wie: »Das ist doch nicht mein Sourcecode …« herausreden, wie dies zum Teil bei »normalen« Projekten der Fall ist. Es ist zudem wenig teamorientiert, wenn Entwickler im Sourcecode ihre Claims abgesteckt haben und bei Änderungen in »ihrem« Sourcecode verärgert reagieren.

Achtung: Besitzdenken bezüglich des Sourcecodes

Wenn Entwickler Sourcecode als ihr Eigentum betrachten und versuchen, diesen möglichst vor Änderungen durch andere Teammitglieder zu verteidigen, zeigen diese ein wenig teamorientiertes Besitzdenken. Dieses Verhalten kann ausufern und problematisch werden, wenn Änderungen zur Sicherstellung der Produktqualität erfolgen müssen, sich aber einige Entwickler durch Änderungen an »ihrem« Sourcecode persönlich angegriffen fühlen.

Die beschriebene Einstellung ist absurd. Der entstehende Sourcecode gehört in der Regel niemanden – wenn überhaupt der Firma. Ein solches Besitzdenken ist zum Teil nur schwer zu durchbrechen. Am einfachsten kann man mit Pair Programming versuchen, das Teamdenken und die gemeinsame Verantwortung zu stärken.

B.1.4 Test-Driven Development (TDD)

Die Erkenntnisse mit XP hat Kent Beck zu den Ideen der testgetriebenen Entwicklung fortgeführt und im Buch »Test-Driven Development« [5] vorgestellt.

Die Kernidee bei der testgetriebenen Entwicklung ist es, vor dem Implementieren der Anwendungsfunktionalität mit dem Schreiben von Testfällen zu beginnen. Erst nachdem mindestens ein Test entworfen ist, wird mit der Entwicklung der eigentlichen Programmfunktionalität begonnen. Um TDD in der Praxis einsetzen zu können, ist ein Unit-Test-Framework Voraussetzung, das die automatisierte Ausführung der Testfälle und deren Auswertung ermöglicht. Dieses prüft, ob alle vorhandenen Tests erfolgreich durchlaufen werden (Grün) oder ob dabei ein Fehler auftritt (Rot).

Beim Entwickeln sollen sich Programmierer auf möglichst kleine Schritte fokussieren, um lesbaren, konkreten und gut testbaren Sourcecode zu schreiben. Die Entwicklung erfolgt dazu iterativ immer in folgenden drei einfachen Schritten:

1. **Rot** – Schreibe einen kleinen Test. Dieser darf anfangs fehlschlagen und lässt sich häufig nicht übersetzen, wenn beispielsweise die Implementierung der zu testenden Klasse oder Methode noch fehlt.
2. **Grün** – Sorge dafür, dass der Test so schnell wie möglich erfolgreich bestanden wird. TDD erlaubt dazu auch einige »Programmiersünden«. Dieses Vorgehen halte ich allerdings für gefährlich und übertriebenen Aktionismus. ***Meiner Meinung nach sollte die Maxime besser lauter: Sorge mit der einfachst möglichen, sauberen Lösung für einen funktionierenden Test.***[2]
3. **Refactoring** – Eliminiere duplizierten und unschönen Sourcecode, der im zweiten Schritt eingeführt wurde.

Durch den Test-First-Ansatz sind selbst bei Projektstress immer bereits Tests vorhanden und das Testen wird nicht in der Projekthektik vernachlässigt. Es kann gar nicht erst zu einem Teufelskreis kommen, indem durch erhöhten Stress weniger getestet wird und so mehr Stress entsteht, wodurch wiederum weniger Zeit für Tests verbleibt. Droht eine solche Situation, so können beim Vorgehen nach TDD einfach die bereits vorhandenen Tests durchgeführt werden. Im Idealfall bekommt man unmittelbar die Rückmeldung, dass alles noch funktioniert. Ein solches »Unit-Test-Sicherheitsnetz« sorgt für mehr Ruhe und Gelassenheit, wodurch sich automatisch die Gefahr von stressbedingten Flüchtigkeitsfehlern reduziert.

Testgetriebene Entwicklung bezieht sich allerdings nur auf solche Tests, die Entwickler selbst schreiben und durchführen, d. h. auf Unit Tests. Funktionstests durch ein Testteam bzw. weiter gehende Systemtests sind nicht Bestandteil der eigentlichen TDDProzesse, aber auch nicht ausgeschlossen. Weitere Informationen zum Thema Testen liefert Kapitel 20.

Persönliche Erfahrungen: XP und TDD in der Praxis

Sowohl XP als auch TDD sind in der Theorie sehr schön. Im Prxaiseinsatz habe ich vor allem mit vielen XP-Techniken sehr gute Erfahrungen gemacht und kann dieses Vorgehen empfehlen. Für TDD gilt dies allerdings mit Einschränkungen – insbesondere für Legacy-Code. Zwar ist es bei kurzen Methoden oder Neuentwicklungen durchaus möglich, TDD zu nutzen. Existierender, etwas älterer Sourcecode zeichnet sich aber in aller Regel dadurch aus, dass er schlecht lesbar und häufig kompliziert und unstrukturiert ist. Dies erschwert es enorm, gemäß TDD zu arbeiten bzw. überhaupt erst einmal testbare Einheiten zu extrahieren. Tipps dazu finden Sie im Buch »Working Effectively With Legacy Code« [24] von Michael Feathers.

Die hier genannten Hindernisse sind jedoch kein Argument, auf Tests komplett zu verzichten. Vielmehr gilt: ***Teste alles, was sich testen lässt!*** Darüber hinaus sollte man versuchen, die schwierig testbaren Bereiche so umzustrukturieren, dass man dort mit dem Testen beginnen kann.

C Grundlagen zur Java Virtual Machine

Dieser Anhang stellt einige Grundlagen zur Java Virtual Machine und zur Ausführung von Java-Programmen vor.

C.1 Wissenswertes rund um die Java Virtual Machine

Java ist eine Programmiersprache, die in einer Laufzeitumgebung, der ***Java Virtual Machine*** (***JVM***), ausgeführt wird. Im Gegensatz zu anderen Programmiersprachen, etwa C++, wird der Sourcecode beim Kompilieren mit dem Java-Compiler `javac` nicht in die Maschinensprache des jeweiligen Computers übersetzt, sondern in eine Zwischensprache, den plattformunabhängigen ***Bytecode***. Der Name rührt daher, dass die Instruktionen in Form von Bytes codiert sind. Dieser Bytecode wird nicht direkt vom Prozessor des Rechners ausgeführt. Stattdessen handelt es sich beim Bytecode um Befehle für einen speziellen virtuellen Computer, nämlich für die Java Virtual Machine. Diese stellt einen Computer im Computer dar und ermöglicht so eine Abstraktion von der darunterliegenden Hardware. Die Plattformunabhängigkeit von Java wird dadurch erreicht, dass für jedes Betriebssystem eine eigenständige JVM existiert, die Befehle im Bytecode-Format ausführt. Abbildung C-1 deutet den prinzipiellen Ablauf an.

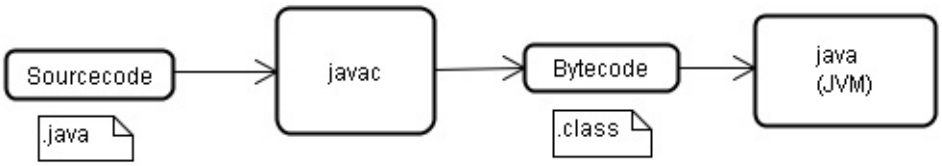


Abbildung C-1 *Ablauf beim Kompilieren und Ausführen eines Java-Programms*

C.1.1 Ausführung eines Java-Programms

Für die Ausführung eines Java-Programms haben früher die ersten JVMs den Bytecode Instruktion für Instruktion interpretiert und abgearbeitet. Aufgrund der durchaus effizienten Arbeitsweise des Bytecode-Interpreters konnte damit bereits eine einigermaßen akzeptable Ausführungsgeschwindigkeit erzielt werden, die jedoch deutlich unter der Geschwindigkeit kompilierter Programme lag. Als Folge davon haben die Ausführungszeiten dieser ersten JVMs lange Zeit das Gerücht genährt, Java-Programme würden (zu) langsam ablaufen. Kompilierte C++-Programme, die in die jeweilige Maschinensprache des Zielrechners übersetzt werden, waren bis zum Erscheinen von JDK 1.4 bzw. JDK 5 performanter. In den letzten Jahren wurden aber immer leistungsfähigere JVMs entwickelt, sodass sich die Ausführungsgeschwindigkeit von Java-Programmen immer mehr derjenigen kompilierter C++-Programme angenähert bzw. mit aktuellen JVMs diese sogar teilweise überflügelt hat.

Heutzutage sind JVMs also extrem leistungsfähig. Das wird unter anderem dadurch erreicht, dass während bzw. parallel zu der eigentlichen Programmausführung der Bytecode in Maschinensprache übersetzt wird. Man spricht von einem Just-in-Time-Compiler (kurz JIT). Führt man diese Transformation für den gesamten Bytecode durch, so kann das allerdings recht aufwendig werden. Für selten durchlaufene Programmteile wiegt somit der erzielte Geschwindigkeitsgewinn bei der Ausführung der kompilierten Anweisungen nicht den zeitlichen Aufwand zur Transformation auf. Demnach ist diese Form der Optimierung manchmal sogar kontraproduktiv und langsamer als eine Ausführung per Interpreter. Aktuelle JVMs nutzen daher eine intelligentere Vorgehensweise bei der Programmausführung, nämlich das sogenannte HotspotOptimierungsverfahren.[1] Hierbei werden die häufig durchlaufenen Programmteile (die Hotspots) erkannt und nur diese kompiliert und optimiert.[2] Abschnitt 22.1.3 geht auf einige Optimierungen im Detail ein.

C.1.2 Sicherheit und Speicherverwaltung

Um Java-Programme möglichst robust zu machen, wurden verschiedene Sicherheitsmechanismen in die Sprache integriert. Insbesondere wurde die Speicherverwaltung so gestaltet, dass man sich als Entwickler kaum darum kümmern muss: Der per `new` angeforderte Speicher für Objekte muss nicht explizit freigegeben werden. Das wird stattdessen automatisch durch eine spezielle Komponente der JVM, den ***Garbage Collector***, erledigt. Fehler durch zu früh oder mehrmals freigegebene Speicherbereiche sind damit ausgeschlossen. Allerdings verbleibt das Problem von nicht freigegebenem Speicher. Derartige ***Memory Leaks*** sind in Java im Vergleich zu Sprachen mit manuellem Speichermanagement eher selten, da die JVM diese automatisch sehr zuverlässig verhindert. Es gibt jedoch Spezialfälle, für die dies nicht gilt, nämlich genau dann, wenn nicht alle Referenzen auf Objekte korrekt freigegeben werden, etwa weil eine andere Programmkomponente eine Referenz weiterhin speichert. Details zur Garbage Collection beschreibt Abschnitt 8.5.

Achtung: Speicherverwaltung

Zwar befreit der Garbage Collector den Entwickler von der expliziten Speicherverwaltung und vermeidet mögliche Fehler einer manuellen Speicherfreigabe. Allerdings existieren verschiedene Varianten der Garbage Collection, die sich auf die Performance auswirken und bei Bedarf mit Bedacht gewählt werden sollten. Bei nicht zufriedenstellender Performance ist die Analyse der Garbage-Collection-Vorgänge mithilfe eines geeigneten Tools, etwa dem zuvor im Buch vorgestellten VisualVM (vgl. Abschnitt 22.1.4), sehr hilfreich, um mögliche Probleme aufspüren und beheben zu können.

Ein Schutz vor Fehlern beim Speicherzugriff wird durch verschiedene Sicherheitsmechanismen erreicht. Zum einen kann man mit Referenzen nur auf Objekte zugreifen, nicht aber mit dem Referenzwert rechnen, wie dies bei einigen

anderen Sprachen möglich ist. Auch Bereichsüberschreitungen bei Array-Zugriffen werden von der JVM verhindert – als Folge werden automatisch `IndexOutOfBoundsExceptions` ausgelöst. Diese Sicherheitsmechanismen verhindern sowohl das versehentliche als auch das mutwillige Auslesen oder Beschreiben von Speicherbereichen, die eigentlich nicht adressierbar sein sollten. Selbst beim Auftreten derartiger Zugriffsprobleme verbleibt die JVM in einem definierten, arbeitsfähigen Zustand, und man kann kontrolliert über Exception Handling auf die Fehlersituation reagieren. Im besten Fall existiert dazu ein `catch`-Block zur Fehlerbehandlung. Dieser wird durch die JVM angesprungen, sodass die dortigen Anweisungen ausgeführt werden. Eine Fehlersituation bewirkt demnach nur, dass die Anweisungen, die der Exception auslösenden Programmstelle folgen, nicht mehr ausgeführt werden und stattdessen die Ausführung des Programms mit dem umgebenden `catch`-Block fortgesetzt wird. Findet sich kein passender `catch`-Block zur Fehlerbehandlung, d. h., wird die Exception nicht im Programm behandelt, so führt dies zu einem Abbruch des Programms, jedoch nicht zu einem Absturz der JVM.

C.1.3 Sicherheit und Classloading

Beim Start einer JVM ist immer eine Klasse anzugeben, deren `main()`-Methode ausgeführt werden soll, wie hier für eine Klasse `MyClass` gezeigt:

```
java MyClass
```

Zunächst muss der Bytecode der entsprechenden Klasse in die JVM geladen werden. Dazu dient eine Instanz eines sogenannten `ClassLoaders`, der selbst eine Java-Klasse ist. Diese spezielle Klasse lädt die Klassendateien (. `class`-Dateien). Was so alles an Klassen geladen wird, wenn Sie ein Java-Programm starten, sehen Sie durch Angabe der JVM-Option `-verbose` beim Aufruf:

```
java -verbose MyClass
```

Anhand der Ausgabe erkennt man, dass zunächst die wichtigsten Interfaces und Klassen des JDKs geladen werden:

```
[Opened C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.lang.Object from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.io.Serializable from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.lang.Comparable from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.lang.CharSequence from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.lang.String from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
…
[Loaded java.lang.Class from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.lang.Cloneable from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
[Loaded java.lang.ClassLoader from C:\Programme\Java\jdk1.7.0\jre\lib\rt.jar]
…
```

Bei diesen Ladevorgängen der Klassendateien finden einige Prüfungen statt, um zu verhindern, dass Systemklassen verändert werden – insbesondere der `ClassLoader` selbst. Anschließend wird durch die Komponente ***Bytecode Verifier*** sichergestellt, dass keine ungültigen Bytecode-Instruktionen in der . `class`-Datei enthalten sind.

Derartige Instruktionen könnten durch Übertragungsfehler oder mutwillige Veränderungen am Bytecode entstehen. Beispielsweise besitzt die JVM den Befehl `iadd` zum Addieren von `int`-Zahlen. Würde dieser mit zwei `float`-Werten ausgeführt, käme es zu Problemen in der JVM. Der Bytecode Verifier prüft nun den gesamten Bytecode nach derartigen Verstößen. Wird ein solcher entdeckt, so wird ein `java.lang.VerifyError` ausgelöst und das Programm beendet. Damit wird verhindert, dass Instruktionen ausgeführt werden, die möglicherweise eine Fehlfunktion der JVM auslösen könnten.

Werden Klassen per Netzwerk oder aus anderen potenziell gefährlichen Quellen geladen, so ist es aus Sicherheitsgründen zum Schutz vor externen Angriffen wünschenswert, dass für diese Klassen eingeschränkte Rechte gelten. Genau das wird innerhalb der JVM durch eine Instanz der Klasse `SecurityManager` sichergestellt. Dieser kann einem ausgeführten Programm (bzw. dessen Klassen) zur Laufzeit gewisse Aktionen erlauben oder verbieten. Beispielsweise sind für derart geladene Klassen standardmäßig Dateisystemoperationen verboten. Das ist sinnvoll, um den Rechner, der das Programm ausführt, vor Angriffen aus dem Netz zu schützen.

Literaturverzeichnis

[1] Scott W. Ambler. *The Elements of Java Style*. Cambridge University Press, Cambridge, Mass., 1995.

[2] Ken Arnold, James Gosling und David Holmes. *The Java Programming Language*. Addison-Wesley, 4. Auflage, 2006.

[3] Joachim Baumann. *Gradle*. dpunkt.verlag, 2013.

[4] Kent Beck. *Extreme Programming. Das Manifest*. Addison-Wesley, 2000.

[5] Kent Beck. *Test-Driven Development*. Addison-Wesley, 2003.

[6] Jon Bentley. *Perlen der Programmierkunst. Programming Pearls*. Addison-Wesley, 2000.

[7] Cedric Beust und Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts*. Addison-Wesley, 2008.

[8] Joshua Bloch. *Effective Java*. Addison-Wesley, 2001.

[9] Joshua Bloch. *Effective Java*. Addison-Wesley, 2. Auflage, 2005.

[10] Joshua Bloch und Neil Gafter. *Java Puzzlers*. Addison-Wesley, 2005.

[11] Grady Booch, James Rumbaugh und Ivar Jacobson. *Das UML-Benutzerhandbuch*. Addison-Wesley, 2006.

[12] Kenneth L. Calvert und Michael J. Donahoo. *TCP/IP Sockets in Java*. Morgan Kaufmann, 2. Auflage, 2008.

[13] Shyan R. Chidamber und Chris F. Kemerer. *A Metrics Suite for Object Oriented Design*. *IEEE Transactions on Software Engineering*, 20:476–493, 1994.

[14] Mike Clark. *Pragmatisch Programmieren: Projekt-Automatisierung*. Hanser Verlag, 2006.

[15] Peter Coad und Mark Mayfield. *Design mit Java*. Prentice Hall, 2. Auflage, 1999.

[16] Ben Collins-Sussman, Brian W. Fitzpatrick und C. Michael Pilato. *Versionskontrolle mit Subversion*. O'Reilly, 2. Auflage, 2006.

[17] James O Coplien. *Why Most Unit Testing is Waste*. *http://www.rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf*, 2014.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest und Clifford Stein. *Introduction to Algorithms*. McGraw-Hill, 2. Auflage, 2001.

[19] Michael C. Daconta, Eric Monk, J. Paul Keller und Keith Bohnenberger. *Java Pitfalls: Time-Saving Solutions and Workarounds to Improve Programs*. Wiley, 2000.

[20] Edsger W. Dijkstra. *The humble programmer*. *Communications of the ACM*, 15(10):859–866, 1994.

[21] Robert Eckstein, Marc Loy und Dave Wood. *Java Swing*. O'Reilly, 1998.

[22] Friedrich Esser. *Java 2 – Designmuster und Zertifizierungswissen*. Galileo Computing, 2001.

[23] Friedrich Esser. *Java 6 Core Techniken*. Oldenbourg, 2008.

[24] Michael Feathers. *Working Effectively With Legacy Code*. Prentice Hall, 2007.

[25] Peter Forbrig. *Objektorientierte Softwareentwicklung mit UML*. Fachbuchverlag Leipzig, 2. Auflage, 2002.

[26] Ira R. Forman und Nate Forman. *Java Reflection in Action*. Manning, 2005.

[27] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[28] Eric Freeman, Elizabeth Freeman, Kathy Sierra und Bert Bates. *Head First Design Patterns*. O'Reilly, 2004.

[29] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Design Patterns – Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[30] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 1996.

[31] Brian Goetz et al. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[32] VJavier Fernández González. *Java 7 Concurrency Cookbook*. Packt Publishing, 2012.

[33] Pete Goodlife. *Code Craft: The Practice of Writing Excellent Code*. No Starch Press, 2007.

[34] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3. Auflage, 2005.

[35] Markus Gumbel, Marcus Vetter und Carlos Cardenas. *Java Standard Libraries*. Addison-Wesley, 2000.

[36] Chet Haase und Romain Guy. *Filthy Rich Clients*. Addison-Wesley, 2008.

[37] Vincent J. Hardy. *Java 2D API Graphics*. Prentice Hall, 2000.

[38] Elliotte Rusty Harold. *Java Network Programming*. O'Reilly, 3. Auflage, 2005.

[39] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, 1995.

[40] Ron Hitchens. *Java NIO*. O'Reilly, 2002.

[41] Allen Holub. *Taming Java Threads*. Apress, 2000.

[42] Allen Holub. *Holub on Patterns*. Apress, 2004.

[43] Steve Holzner. *Ant: The Definitive Guide*. O'Reilly, 2. Auflage, 2005.

[44] Cay S. Horstmann. *Java SE 8 for the Really Impatient*. Addison-Wesley, 2014.

[45] Cay S. Horstmann und Gary Cornell. *Core Java 2 – Band 1 Grundlagen*. Addison-Wesley, 7. Auflage, 2005.

[46] Cay S. Horstmann und Gary Cornell. *Core Java 2 – Band 2 Expertenwissen*. Addison-Wesley, 7. Auflage, 2005.

[47] Michael Hüttermann. *Agile Java-Entwicklung in der Praxis*. O'Reilly, 2008.

[48] Andrew Hunt und David Thomas. *Der Pragmatische Programmierer*. Hanser Verlag, 2003.

[49] Andrew Hunt und David Thomas. *Unit-Tests mit JUnit*. Hanser Verlag, 2004.

[50] Andrew Hunt und David Thomas. *Pragmatisch Programmieren: Versionsverwaltung mit CVS*. Hanser Verlag, 2005.

[51] Xiaoping Jia. *Object-Oriented Software Development Using Java*. Addison-Wesley, 2. Auflage, 2002.

[52] Tomasz Kaczanowski. *Practical Unit Testing with JUnit and Mockito*. kaczanowscy.pl, 2013.

[53] Christoph Kecher. *UML 2.0: Das umfassende Handbuch*. Galileo Computing, 2. Auflage, 2006.

[54] Joshua Kerievsky. *Refactoring To Patterns*. Addison-Wesley, 2005.

[55] Jonathan Knudsen. *Java 2D API Graphics*. O'Reilly, 1999.

[56] Donald E. Knuth. *Structured Programming with go to Statements*. *ACM Journal Computing Surveys*, 6(4):268, Dezember 1974.

[57] Lasse Koskela. *Test Driven*. Manning, 2008.

[58] Lasse Koskela. *Effective Unit Testing: A Guide for Java Developers*. Manning, 2013.

[59] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 2. Auflage, 2000.

[60] Johannes Link. *Softwaretests mit JUnit*. dpunkt.verlag, 2. Auflage, 2005.

[61] Joshua Marianacci und Chris Adamson. *Swing Hacks*. O'Reilly, 2005.

[62] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[63] Mike Mason. *Pragmatic Version Control Using Subversion*. Pragmatic Programmers, 2. Auflage, 2006.

[64] Brett McLaughlin. *Java & XML*. O'Reilly, 2. Auflage, 2001.

[65] Steven J. Metsker. *Design Patterns Java Workbook*. Addison-Wesley, 2002.

[66] Sun Microsystems. *Java Look and Feel Design Guidelines*. Addison-Wesley, 2. Auflage, 2001.

[67] Khalid A. Mughal und Rolf W. Rasmussen. *A Programmer's Guide to Java SCJP Certification*. Addison-Wesley, 3. Auflage, 2009.

[68] Benjamin Muschko. *Gradle In Action*. Manning, 2014.

[69] Maurice Naftalin und Philip Wadler. *Java Generics and Collections*. O'Reilly, 2007.

[70] Johannes Nowak. *Fortgeschrittene Programmierung mit Java 5*. dpunkt.verlag, 2005.

[71] Scott Oaks und Henry Wong. *Java Threads*. O'Reilly, 3. Auflage, 2004.

[72] Rainer Oechsle. *Parallele und verteilte Anwendungen in Java*. Hanser Verlag, 2. Auflage, 2007.

[73] Bryan O'Sullivan. *Mercurial: The Definitive Guide*. O'Reilly, 2009.

[74] René Preißel und Bjørn Stachmann. *Git*. dpunkt.verlag, 2. Auflage, 2014.

[75] Bruno R. Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Wiley, 2000.

[76] Matthew Robinson und Pavel Vorobiew. *Swing*. Manning, 2. Auflage, 2003.

[77] Stefan Roock und Martin Lippert. *Refactorings in großen Softwareprojekten*. dpunkt.verlag, 2004.

[78] Hendrik Schreiber. *Performant Java programmieren*. Addison-Wesley, 2002.

[79] Robert Sedgewick. *Algorithmen*. Addison-Wesley, 1992.

[80] Alan Shalloway und James R. Trott. *Design Patterns Explained*. Addison-Wesley, 2. Auflage, 2005.

[81] Jack Shirazi. *Java Performance Tuning*. O'Reilly, 2. Auflage, 2003.

[82] Kathy Sierra und Bert Bates. *SCJP Sun Certified Programmer & Developer for Java 2*. Osborne, 2003.

[83] Kathy Sierra und Bert Bates. *SCJP Sun Certified Programmer for Java 6 Study Guide*. McGraw-Hill, 2008.

[84] Bernhard Stephan. *Einstieg in Java 6 – Sonderausgabe: Verständliche und umfassende Einführung*. Galileo Computing, 3. Auflage, 2008.

[85] Venkat Subramaniam. *Programming Groovy 2: Dynamic Productivity for the Java Developer*. O'Reilly, 2013.

[86] Kim Topley. *Core Swing: advanced programming*. Prentice Hall, 2000.

[87] Bernd-Uwe Pagel und Hans-Werner Six. *Software Engineering*. Addison-Wesley, 1994.

[88] Jennifer Vespermann. *CVS: Versionskontrolle und Quellcode-Management*. O'Reilly, 2004.

[89] Uwe Vigenschow und Björn Schneider. *Soft Skills für Softwareentwickler*. dpunkt.verlag, 2006.

[90] Frank Westphal. *Testgetriebene Entwicklung mit JUnit & FIT*. dpunkt.verlag, 2006.

[91] Steve Wilson und Jeff Kesselman. *Java Platform Performance – Strategies and*

Tactics. Addison-Wesley, 2000.

Index

Symbole

@Deprecated, 113, 256, 431
@Override, 204
@TODO, 997
@deprecated, 113, 256, 431

A

Abarbeitung
    iterative, 822
    parallele, 854
    sequenzielle, 822, 854, 857
Abhängigkeit
    zirkuläre, 639
Ableitung, 370
Ableitungshierarchie, 972, 988
    von Exceptions, 299
Abnahmetest, 1207
Abstract Window Toolkit, 644
AbstractMap<K,V>, 362, 367
AbstractSet<E>, 336
Abstraktion, 1185
Abstraktionsebene, 1069, 1186
    einheitliche, 1070, 1168
    Implementierungsebene, 1169
    unterschiedliche, 1168
accept(), 811, 812
Access Control Proxy, 1151
Accessor, 89
ActionEvent, 885

ActionListener, 806
Adapter, 1117
Aggregation, 91, 370, 1382
Aktivitätsdiagramm, 1376, 1377, 1393
Akzeptanztest, 1207
Algorithmus, 1318
allMatch(), 836, 849
AmbientLight, 922
Analyse
    dynamische, 1191
    statische, 1191
and(), 826, 827
Animation, 909
Annotation, 37, 593
    @Deprecated, 113, 256, 431, 964, 1081, 1179
    @Override, 204, 372
    auslesen, 599
    Auto-Complete-, 598
    Definition einer eigenen, 596
    Meta-, 598
    Standard-, 594
Annotation Processor, 593
Annotations, 577
Anomalie
    MIN_VALUE, 223
Ant, 11, 59
Anti-Pattern, 1093
    Beobachter, 1160
    Observer, 1160
    Singleton, 1109
Antialiasing, 702

Anwendungsfalldiagramm, 1376
    Akteur, 1385
anyMatch(), 836, 849
AOP, 1117
Apache Commons, 428
Apache Commons CLI, 467, 472
Apache Commons Configuration, 488
API, 102, 446, 1214
    Calendar-, 256, 257
    Date-, 255
    Design, 1214
    File-, 273
    fremdes, 478
API-Design, 1214
API-Problem, 446
Appender, 456
Application Programming Interface, 102
Applikationstest, 1207
apply(), 811, 828, 830
Äquivalenzklasse, 1224
Äquivalenzklassentest, 1012, 1223
Arbeitsumgebung, 11
Architekturproblem, 1191
ARM, 305, 308
Array, 317, 320
    Anpassung der Größe, 321, 322
    Kapazität, 321
ArrayStoreException, 181
ArrayBlockingQueue<E>, 558
ArrayIndexOutOfBoundsException, 322, 974, 1001
ArrayList

sort(), 814
ArrayList<E>, 206, 327, 330, 1320, 1322
add(), 331
get(), 331
Größenanpassung, 332
Kapazität, 330
remove(), 332
size(), 330
Speicherverbrauch, 332
trimToSize(), 332
Arrays, 204, 322, 832, 934
asList(), 204, 392
binarySearch(), 375
copyOf(), 322
copyOfRange(), 322
deepEquals(), 418
deepHashCode(), 419
deepToString(), 419
equals(), 418
hashCode(), 419
parallelPrefix(), 937
parallelSetAll(), 936
parallelSort(), 934
stream(), 831
toString(), 205, 419
Artefakt, 65
ASCII, 954
asLongStream(), 833
AspectJ, 1117
aspektorientierte Programmierung, 1117
Assert, 36, 37

Assertion, 294, 310
-da, 311
-ea, 311
aktivieren, 311
deaktivieren, 311
AssertionError, 310
Assoziation, 91, 1381
atomare Variable, 553
Atomarität, 536, 537
AtomicInteger, 538
AtomicLong, 538
AtomicInteger, 834
getAndIncrement(), 834
Attribut, 86
Präfix, 1173
Typpräfix, 1173
Attributermittlungsfilter, 390
Aufzählung, 154
Ausdruck
regulärer, 246
Ausgabestream, 275
Auslassungszeichen, 891
Auslieferung, 16
äußere Qualität, 1211
Austauschbarkeit, 1185
Auto-Boxing, 225
Wissenswertes zu, 228
Auto-Unboxing, 225, 228
Wissenswertes zu, 228
AutoClosable, 309
Automatic Resource Management, 305, 308

average(), 848, 931
AWT, 644
AWT-Event-Queue, 662, 675
    Einstellen von Aktionen, 675

B

Backup, 20
Balancierung, 356
Base64, 954
Basisklasse, 993
    abstrakte, 128, 129
Baum, 1125
    binärer, 356
    Blatt, 356
    Tiefe, 356
    Wurzel, 356
Baumdarstellung
    dynamische, 755
Baumstruktur, 747
    Blätter, 747
    Knoten, 747
    Wurzel, 747
Bedeutung von this, 807
Bedienelement, 648
    natives, 644
Bedingung
    boolesche, 825
behaves-like-Beziehung, 87
Beobachter, 1071, 1153
    als Anti-Pattern, 1160
Berechnung

dynamisch ausführen, 950
Bereich
kritischer, 503, 553
Beschreibungssprache, 1376
Betriebsblindheit, 1209
between(), 869, 870, 878
Bézierkurve, 693
BiConsumer, 811
accept(), 811
BiFunction, 811
apply(), 811
BigInteger, 1037
binärer Baum, 356
Binärsuche, 375, 1316
BinaryOperator, 858
Binden
dynamisches, 96, 988, 1020
Bindings, 949
Blackbox-Sicht, 1384
Blackbox-Test, 1208
Blatt, 356, 1125
BlockingDeque<E>, 560
BlockingQueue<E>, 530, 558, 559
offer(), 558
poll(), 558
put(), 558
take(), 558
Boilerplate-Code, 804, 923
Boolean, 225
BorderLayout, 651
BorderPane, 886

Box, 920
boxed(), 833
Boxing, 225
Branch, 19, 24
Breakpoint, 43
Bucket, 341
BufferedInputStream, 276
BufferedReader
    readLine(), 487
BufferedWriter
    newLine(), 487
Build, 59
Builder, 1102
Bulk Operations
    on Collections, 821
Business-Methode, 90, 104, 1062, 1064, 1186
    Auswirkung auf Beobachter-Muster, 1160
Busy Waiting, 520
Button, 885, 908
ByteBuffer, 288
Bytecode, 1405
Bytecode Verifier, 1408

C

Cache, 363
    Integer, 229
    Long, 229
    LRU-, 1354
    Thread-lokaler, 502
Cache-Kohärenz, 537, 1357
Calendar, 252, 253, 256, 866, 867, 873

add(), 257
get(), 257
getTime(), 253
roll(), 257
set(), 257
Calendar-API, 256, 257
Call-by-Reference, 97
Call-by-Value, 97
Callable, 567, 944
Callback-Interface, 822
Callback-Methode, 266
CamelCase-Schreibweise, 1172
can-act-like, 140
can-act-like-Beziehung, 87, 125
CardLayout, 652
CAS, 538
Cascading Style Sheets, 901
Cast, 96, 972
ClassCastException, 972
CBO, 1194
Character, 225
isDigit(), 1030
CharBuffer, 286, 288
CharSequence, 234, 832
Charset, 278
availableCharsets(), 280
Cp1252, 279
Cp850, 279
forName(), 280
ISO-8859-1, 279
UTF-16, 279

UTF-8, 279
CharsetDecoder, 278
CharsetEncoder, 278
Checked Exception, 299, 305, 1002
Checkstyle, 1196
ChronoUnit, 869
Class<T>, 199
ClassCastException, 336, 399, 972
CLASSPATH, 16
Client, 123
Client-Server-Kommunikation, 301
Clock, 877
clone()
Kontrakt, 619
clone()-Kontrakt, 619
Cloneable, 618
Closure, 801
Clustering, 343
CMS, 637
Cobertura, 1280
Code as Data, 824
Codereview, 1170, 1174, 1289
Meeting, 1289
Probleme, 1291
psychologische Aspekte, 1290
Tipps, 1291
Tool, 1296, 1297
Codereview-Meeting, 1289
Codereview-Tool
Code Collaborator, 1296
Crucible, 1296

Jupiter, 1297
Coding Conventions, 1038, 1165, 1170
Akzeptanz, 1170
Formatierung, 1171
Namensgebung, 1171
Colebourne, 867
CollationKey, 783
Collator, 779, 925
getInstance(), 925
collect(), 836
Collection, 810, 827, 830
Mengenoperation, 324
parallelStream(), 831
removeIf(), 827, 828
stream(), 831
Collection<E>, 318, 323, 424, 1328
add(), 206, 323
addAll(), 323
contains(), 323
containsAll(), 323
isEmpty(), 323
iterator(), 324
next(), 325
remove(), 323, 326
removeAll(), 323
retainAll(), 323
size(), 323
Collections, 813
binarySearch(), 375
checkedCollection(), 399
checkedList(), 399

checkedMap(), 399
checkedSet(), 399
checkedSortedMap(), 399
checkedSortedSet(), 399
emptyList(), 394
emptyMap(), 394
emptySet(), 394
frequency(), 401
max(), 402
min(), 402
nCopies(), 401
replaceAll(), 403
shuffle(), 403
singleton(), 394
singletonList(), 394
singletonMap(), 394
sort(), 807
unmodifiableList(), 327, 397
Collector, 847
Collectors
counting(), 852
groupingBy(), 852
joining(), 852
partitioningBy(), 852
toCollection(), 847
toList(), 847
Command, 1143
CommandLineParser parse(), 474
Commons CLI, 467, 472
Commons Configuration, 488
Comparable, 924

Comparable<T>, 350, 351
compareTo(), 351
Comparator, 803, 804, 836, 923, 924, 926
compareTo(), 923, 925
comparing(), 924, 925
comparingDouble(), 924
comparingInt(), 924, 926
comparingLong(), 924
Hintereinanderschaltung, 925
naturalOrder(), 926
null-Werte, 927
nullsFirst(), 927
nullsLast(), 927
primitive Typen, 926
reversed(), 926
reverseOrder(), 926
thenComparing(), 924, 925
thenComparingDouble(), 924
thenComparingInt(), 924
thenComparingLong(), 924
Comparator<T>, 350, 354
compare(), 355
Comparators
naturalOrder(), 924
reversed(), 924
reverseOrder(), 924
compare(), 803–805
Compare-and-Swap-Operation, 538
compareTo(), 923, 925
comparing(), 925
comparingInt(), 926

CompletableFuture, 944, 945
supplyAsync(), 945, 946
thenAccept(), 945
thenApply(), 945
thenApplyAsync(), 946
thenCombine(), 946
Component, 645
computeIfAbsent(), 940
computeIfPresent(), 940
Concurrency, 944
Concurrent Mark and Sweep Collector, 637
ConcurrentHashMap, 939, 942, 944
computeIfAbsent(), 944
forEach(), 944
forEachEntry(), 944
forEachKey(), 944
forEachValue(), 944
merge(), 944
reduce(), 944
search(), 944
ConcurrentHashMap<K,V>, 555, 556, 1324
ConcurrentModificationException, 326, 396, 555
ConcurrentNavigableMap<K,V>, 560
ConcurrentSkipListMap<K,V>, 320, 556, 560, 1325
ConcurrentSkipListSet<E>, 320, 1325
Condition, 512, 529, 553
await(), 529, 536
signal(), 529, 536
signalAll(), 529, 536
Conditional-Operator, 1190
Console, 1360

Consumer, 811, 824
accept(), 812
Container, 646–648
hashbasierte, 340
JPanel, 649
JScrollPane, 649
JSplitPane, 649
nützliche, 649
Containerklasse, 317
ContentPane, 882
Continuous Integration, 61
Convenience-Methode, 971
Convention over Configuration, 65
Copy-Konstruktor, 358, 627
Copy-Paste, 113, 970, 971, 1011
Erkennung von, 1198
Probleme durch, 970
Wiederverwendung, 91
Copy-Paste-Ansatz, 93
CopyOnWriteArrayList<E>, 517, 555, 556
CopyOnWriteArraySet<E>, 517, 555, 556
count(), 836, 848
CountDownLatch, 534, 553
await(), 534
countDown(), 534
counting(), 852
Cp1252, 279
Cp850, 279
CPU-bound, 1301
CPU-bound-Optimierungen, 1365
Cross-Cutting Concern, 1116, 1117

CSS, 901
currentTimeMillis(), 865, 877
CVS, 11
CyclicBarrier, 534, 553
Cyclomatic Complexity, 1193
Cylinder, 920

D

Daemon-Thread, 543
DAO, 1175
Data Access Object, 1175
Data Binding, 916
Data Transfer Object, 159
DatagramChannel, 288
DataInputStream, 284
    readLong(), 284
    readUTF(), 284
DataOutputStream, 284
    writeLong(), 284
    writeUTF(), 284
Date, 201, 252, 865–868, 873, 879, 1057
    after(), 255
    before(), 255
    deprecated, 865
    Fallstricke, 252
    getDate(), 254
    getDay(), 254
    getMonth(), 254
    getTime(), 253
    getYear(), 254
    setTime(), 253

Date And Time API, 5, 865
Date-API, 255
DateFormat, 488, 768
    setLenient(), 775
DateFormatter, 866
Datenkapselung, 111, 1055, 1062
Datenmodell, 704
Datensicherung, 20
Datenstruktur, 1318
    Array, 320
    Baum, 1318
    Einsatz geeigneter, 1318
    Liste, 327
    Menge, 334
Datentyp, 87
    primitiver, 87
DatePicker, 913
DateTimeFormatter, 878
DayOfWeek, 873
Deadlock, 490, 510, 511
Debugger, 41
Debugging, 41
    Java Debug Wire Protocol, 46
    JDWP, 46
Debuggingzwecke, 836
Decoder, 278
Decorator, 1119
Deep Copy, 148
default, 19, 29
Default-Renderer, 709
Defaultmethode, 263, 809, 810

API-Erweiterung, 810, 815
forEach(), 810
Gefahren, 815
Konflikt, 814
Rückwärtskompatibilität, 810, 815
sort(), 810
Spezialbehandlung, 813, 815
Standardverhalten, 812, 815
Vorteile, 815
Zugriff auf Attribute, 816
DefaultMutableTreeNode, 748, 1123
hasChildren(), 1123
isLeaf(), 1123
Definition, 87
Deklaration, 87
Delayed, 559
Delayed Exception, 1009
DelayQueue<E>, 559
Delegation, 114, 1119
Delegation (Based) Event Model, 659
Dependency Injection, 176, 1148
Dependency Inversion Principle, 174
Deployment, 50
Deque<E>, 319, 421, 424, 560
Deserialisierung, 600
Design by Contract, 115, 1181
Designpattern, 1093
Designproblem, 1191
Destruktor, 358
Dialog, 646
Diamond Operator, 188, 805, 806

Dictionary, 360
Differenzmenge, 324
distinct(), 835, 836, 845
DIT, 1194
divide and conquer, 106, 572
Domain, 1207
Double, 218
    doubleToLongBits(), 232
    longBitsToDouble(), 232
    parseDouble(), 227
Double Checked Locking, 1109
DoubleBuffer, 288
DoubleStream, 833
    boxed(), 833
Down Cast, 96, 146, 972, 991
DropShadow, 908
DRY-Prinzip, 303, 1068, 1087
DTO, 1328
    Hilfsmethoden, 161
Duration, 869, 870, 872, 876, 878
Dynamic Binding, 96
dynamische Analyse, 1191
dynamisches Binden, 1020

E

EasyMock, 1250
EclEmma, 1284
Eclipse, 11
    IDE, 12
    Refactoring
        Encapsulate Field, 1056

Extract Interface, 1065
Extract Local Variable, 1022
Extract Method, 1068
Eden-Bereich, 631
EDT, 674
Effect, 908
effectively final, 261, 310, 808
Effekt, 907
Eiffel, 115
Eigenschaften
orthogonale, 139
unabhängige, 139
Eingabestream, 275
Ellipsis, 891
Encoder, 278
Endzustand
Zustandsdiagramm, 1392
Entropie, 1167
Entwicklungsumgebung
integrierte, 11
Entwurfsmuster, 1093
Adapter, 1117
Builder, 1102
Command, 1143
Decorator, 139, 1119
Erbauer, 1102
Erzeugungsmethode, 1015, 1017, 1096
Fabrikmethode, 772, 775, 992, 1099, 1322
Fassade, 1114
Iterator, 1126
Kompositum, 1122, 1318

Listener, 1152
Null-Objekt, 851, 1005, 1128, 1331, 1333
Observer, 1152
Prototyp, 1110
Proxy, 1150, 1334, 1336
Publisher/Subscriber, 1152
Schablonenmethode, 1131
Singleton, 479, 502, 1105
Strategie, 1135
Template-Methode, 1131
Value Object, 159
Entwurfsstil, 106
Enum-Muster, 154
Enumeration<E>, 244
hasMoreElements(), 244
nextElement(), 244
equals()
Behandlung optionaler Attribute, 212
in Subklassen, 214
Kontrakt, 207
nullSafeEquals(), 213
StringBuffer, 242
StringBuilder, 242
Typische Fehler, 210
EqualsUtils
nullSafeEquals(), 213
equalTo(), 1274
Erbauer, 1102
Ereignis, 659
Ereignisbehandlung, 647, 659
Ereignisverarbeitung

Varianten, 669
Erreichbarkeit, 630
Erweiterbarkeit
um Methoden, 807
Erzeugungsmethode, 1015, 1017, 1096
Escapen, 248
Escaping Reference, 985
Event, 659
Event Dispatch Thread, 674
Event Listener, 659, 664
Event Source, 659
Event-Queue
Einstellen von Aktionen, 675
EventHandler, 885, 907
Lambda als, 911
Exception, 294
Checked, 299, 300
IllegalArgumentException, 295
IllegalStateException, 295
in Threads, 543
NullPointerException, 295
printStackTrace(), 296
Unchecked, 299, 300
UnsupportedOperationException, 295
Exception Handling, 300
Besonderheiten, 306
Erleichterung in JDK 7, 1002
Final Rethrow, 309
Multi Catch, 306
unspezifisches, 1001
Vereinfachung, 1002

Exchanger, 553
Exchanger<V>, 534
    exchange(), 534
Executor, 561, 817
Executors, 563, 817
    newCachedThreadPool(), 564
    newFixedThreadPool(), 564
    newScheduledThreadPool(), 564
    newSingleThreadExecutor(), 564
ExecutorService, 563
    submit(), 567
Explosion
    kombinatorische, 138
Externalizable, 615
Extraktion, 856
Extreme Programming, 1208, 1400, 1401
Extremwerttest, 1012

F

Fabrikmethode, 772, 775, 992, 1099, 1322
fail-fast, 555
Fail-fast-Iterator, 326, 327, 330, 814
Fall-Through-Assertion, 313
False Positives, 1267
Fassade, 1114
FCFS, 422
Fehler, 1206
Fehlerbehandlung
    offensive, 1004
Fehlermaskierung, 1196, 1208
Fehlersuche, 43

mit Debugger, 43
FIFO, 422
FIFO-Prinzip, 319
File, 268, 286
createNewFile(), 270
delete(), 270
exists(), 270
getAbsolutePath(), 269
getCanonicalPath(), 269
getName(), 269
isDirectory(), 270
isFile(), 270
list(), 270
listFiles(), 270
mkdir(), 270
mkdirs(), 270
File-API, 273
FileChannel, 288
FileFilter, 271
accept(), 271
FileInputStream, 276
FilenameFilter, 271
accept(), 271
FileOutputStream, 276
Files, 289
readAllLines(), 947
Files.lines(), 942
Files.list(), 942
Files.readAllLines(), 942
Files.write(), 943
filter(), 837, 839

Filter-Map-Reduce, 855, 863
    im Einsatz, 858
Filter-Map-Reduce-Framework, 821
Filterbedingung, 947
final
    effectively, 808
Finalizer, 640
findAny(), 836, 850
FindBugs, 1199
findFirst(), 836, 850
First-Come-First-Serve, 422
First-In-First-Out, 422
flache Kopie, 148
flatMap(), 836, 839, 853, 947
Float, 218
    floatToIntBits(), 232
    intBitsToFloat(), 232
    parseFloat(), 227
Flow, 1216
FlowLayout, 652
FlowPane, 885, 886
FontMetrics, 689
    Abmessungen der Schriftart, 689
for-each, 822
forEach(), 810, 822, 823, 836, 846
FOREVER, 869
Fork-Join-Framework, 821, 946
Format, 243, 769
Formatierung
    Ausnahmen, 1172
    grundlegende Regeln, 1171

Formatter, 769
Frame, 646
freeMemory(), 1311
Füllgrad, 321, 348
Full GC, 633
Function, 811, 828, 835, 838
    apply(), 811, 828, 830, 838
Functional Interface, 803
    besondere Methoden, 803
    Implementierung von, 804
FunctionalInterface
    Annotation, 803
Future, 566, 944
    isDone(), 569
FXMLLoader, 897

G

Garbage Collection, 89, 200, 577, 629, 630, 641
    alte Generation, 631, 632
    Eden-Bereich, 631
    finalize(), 640
    Full GC, 633
    Grundlagen zur, 630
    isolierte Inseln, 639
    junge Generation, 631
    Major GC, 633
    Mark-and-Compact-Algorithmus, 634
    Mark-and-Sweep-Algorithmus, 633
    Memory Leak, 638
    Old Generation, 632
    Perm, 632

Permanent Generation, 632
SoftReference, 642
Stop-and-Copy-Algorithmus, 634
Survivor-Bereich, 631
Ternured Generation, 632
WeakReference, 642
Young Generation, 631
Garbage Collector, 200, 332, 629, 641, 1184, 1406
CMS, 637
G1, 637
GaussianBlur, 908
GC, 629
Geheimnisprinzip, 162
Generalisierung, 93, 1185
Generation
alte, 631
junge, 631
Generics, 85, 144, 184, 399
Gesetz der Ähnlichkeit, 1167
Gesetz der Nähe, 1169
Gesetz von Demeter, 163
Gestaltung
mit CSS, 901
get(), 811
getAvailableZoneIds(), 880
getOrDefault(), 939
getSeconds(), 878
Google Guava, 428
Gradle, 11, 59
Grafikkontext, 685
Graphen, 1125

Graphical User Interface, 644
Graphics
    Zeichenmethoden, 685
GregorianCalendar, 256, 879
GridBagConstraints, 654
GridBagLayout, 653
GridLayout, 653
GridPane, 886
groupingBy(), 852
GUI, 644
    Container, 646, 648
    Datenmodell, 704
GUI-Komponente
    native, 644

H

Hamcrest, 1273
    assertThat(), 1274
Happens-before, 513, 541
    Antisymmetrisch, 542
    Irreflexiv, 542
    Transitiv, 542
Happens-before-Ordnung, 546, 560
has-a-Beziehung, 994
hashCode(), 346
    Kontrakt, 345
    Primzahl, 346
Hashcontainer
    Bucket, 341
    Füllgrad, 348
    Kollision, 348

Load Factor, 348
HashMap<K,V>, 362, 555, 1323
HashSet<E>, 336, 1323
Hashtabelle, 342
Hashtable<K,V>, 480
HashUtils, 346
Hauptast, 19
HBox, 886
HEAD, 19
HelpFormatter
printHelp(), 475
heterogene Liste, 417
High-Level-Event, 663
High-Level-Refactoring, 1055
Hilfsmethoden, 161
homogene Liste, 418
Hook, 1131
Hyperassertions, 1262

I

I/O-bound, 1302
I/O-bound-Optimierungen, 1344
IDE, 11, 12
Identität, 205
identity(), 828
Idiom
Null-sichere Vergleiche mit der Utility-Klasse Objects, 213
Performante Stringkonkatenation, 241
Prüfung auf inhaltliche Gleichheit, 242
Thread-sichere Iteration, 396
Traversierung von Collections mit dem Interface Iterator, 324

Warten auf eine Bedingung, 523
IEEE-754-Format, 232
ifPresent(), 930
IllegalThreadStateException, 493
IllegalArgumentException, 1088
IllegalMonitorStateException, 506
unerwartete, 534
IllegalStateException, 327, 424, 842, 1086
Konfigurationsfehler, 1004
Image, 368, 908
ImageView, 908
Implementierungsphase, 1182
Implementierungsvererbung, 94, 114, 422, 476
Indirektionen, 1116
Infinitest, 1279
Information Hiding, 89, 162
Initialisierung
korrekte, 1086
Initialisierungsprüfung
zentrale, 1087
Initializer
Instanz-, 988
innere Klasse, 259
innere Qualität, 1211
inneres Interface, 262
Inplace Edit, 741
InputStream, 275, 286
available(), 277
close(), 277
read(), 276, 277
ready(), 277

InputStreamReader, 275
Inspektion
    von Verarbeitungsschritten, 842
instanceof, 208
Instant, 868–870, 872, 874, 879
Instanz, 86
Instanz-Initializer, 988
IntConsumer, 811
Integer, 218, 805
    Cache, 229
    parseInt(), 226, 227, 488, 1036
Integrationstest, 1207, 1209
IntelliJ IDEA
    IDE, 12
Interface, 86, 123, 262
    Angebot von Verhalten, 86, 123
    Aufspalten eines, 1066
    Cloneable, 618
    Comparable<T>, 351
    Einführen eines, 1065
    Erweiterung, 810
    inneres, 262
    Kompatibilitätsproblem, 810
    Namensgebung von, 126
    Postfix, 1172
    Präfix, 126, 1172
    Read-only-, 1067, 1068
    Read-Write-, 1067
    statische Methoden, 816
    Veröffentlichung, 809
    Write-, 1068

Interface Segregation Principle, 173, 1186
Interleaving, 540
Intermediate Operation, 835
    zustandsbehaftete, 845
    zustandslose, 837
Internationalisierung, 759, 925
    ResourceManager, 790
Interprozesskommunikation, 490
InterruptedException, 499, 501
IntFunction, 811, 937
IntPredicate, 811
IntStream, 832, 833
    asLongStream(), 833
    boxed(), 833
    iterate(), 834
    mapToObj(), 833, 863
    range(), 863
IntSupplier, 811
IntToDoubleFunction, 937
IntToLongFunction, 937
IntUnaryOperator, 834, 936
Invariante, 115, 150
Invarianz, 180, 405, 407
IOException, 269
is-a-Beziehung, 93, 96, 126, 262, 423, 994
isEmpty(), 828
isLeap, 874
ISO-8859-1, 279
isolierte Inseln, 639
ISP, 1186
Istanalyse, 1397

it, 812
Iterable, 810
    forEach(), 846
Iterable<T>, 1128
iterate(), 834
Iteration, 324, 821
    externe, 821–823
    interne, 821–823
Iterator, 324, 812, 821, 822, 827, 1126
    fail-fast, 555
    fail-fast-, 326, 327, 330, 814
    remove(), 812
    weakly consistent, 557
Iterator<E>, 1128, 1328
    ConcurrentModificationException, 326
    hasNext(), 324
    next(), 325, 326
    remove(), 325, 327
    UnsupportedOperationException, 325

J

JAR, 51
Java 2D, 645, 691
    Bedienelement selbst erstellen, 696
    Einführung, 691
    Figuren, 692
    Grundlagen, 694
Java Debug Wire Protocol, 46
Java Foundation Classes, 644
Java Language Specification, 803
Java Virtual Machine, 1405

Java Virtual Machine Debug Interface, 47
Java-Memory-Modell, 513, 536
Javadoc, 1169, 1178
    @author, 1179
    @deprecated, 113, 256, 431, 964, 1179
    @inheritDoc, 1179
    @param, 1179
    @return, 1179
    @see, 1179
    @since, 1179
    @throws, 1179
    @version, 1179
    Tag, 1178
JavaFX
    Animation, 909
    Effekt, 907
    Neuerungen in Version 8, 911
javap, 241
JavaScript
    ausführen, 949
JavaScript-Engine, 948
    Nashorn, 948
    Rhino, 948, 950
JComponent, 645
JConsole, 1312
JDepend, 1201
JDialog, 646
JDWP, 46
JFC, 644
JFrame, 646
JList, 713

Datenmodell, 715
dynamischer Inhalt, 720
Renderer, 717
Selektion, 714
JLS, 313, 345, 803
JMM, 536
Atomarität, 536, 537
Reordering, 536, 539
Sichtbarkeit, 536, 537
joining(), 852
JPanel, 649, 655
JSeparator, 658
JSplitPane, 658
JSR-310, 865
JTable, 727
Datenmodell, 728
Editoren, 741
Filterung, 738
Renderer, 732
Selektion, 728
Sortierung, 734
Spalten anpassen, 731
JToolBar, 658
JTree, 747
Baumstruktur, 747
Datenmodell, 749
Renderer, 753
Selektion, 751
Zyklenfreiheit, 749
JUnit, 1273
@After, 1222

@Before, 1222
Annotation, 37
Assert, 37
assertEquals(), 38, 1274
assertFalse(), 37, 1031
assertNotNull(), 38
assertNotSame(), 38
assertNull(), 38
assertSame(), 38
assertTrue(), 37, 1031
Extreme Programming, 1208
fail(), 38
TDD, 1208
Test-Driven Development, 1208
XP, 1208
JUnit 4, 35
JUnit-Framework, 37
Just-in-Time-Compiler, 1307
JVM, 1405
JVMDI, 47

K

Kanten, 1125
Kapazität, 321, 342
Kapselung, 85, 89, 106, 111, 216, 1062, 1194
bessere, 1057
Keep It Clean, 1167
Keep It Human-Readable, 1166
Keep It Natural, 1166
Keep It Simple And Short, 1166
Key-Extractor, 925, 928

Klasse, 86
abgeleitete, 92
Basis-, 92
innere, 259
konkrete, 129
Ober-, 92
Sub-, 92
Super-, 92
Klassenattribut, 816
Klassendiagramm, 1376, 1379
Stereotyp, 1380
Klassenhierarchie, 93, 139, 216, 382
Klassenlänge
maximale, 1187
Klassenname
voll qualifizierter, 978
Knoten, 333, 334, 1125
Kohäsion, 90, 106, 112, 1194, 1214
Kollaborationsdiagramm, 1391
Kollision, 341, 342, 348, 1323
kombinatorische Explosion, 382
Kommentar
Überschrift-, 1177
@TODO, 997
Kommunikationsdiagramm, 1376, 1391
Komparator, 354
Hintereinanderschaltung, 925
Kompiliertyp, 95, 1020
Komponente, 1383
Komponentendiagramm, 1376
Komponententest, 1207

Komposition, 91, 1382
Kompositum, 1122, 1318
Konsistenz, 207
Konstantensammlung, 154
Kontrakt
    clone(), 619
    equals(), 207
    hashCode(), 345
Kontravarianz, 180, 182, 409
Kopie
    tiefe, 147, 151
Kopplung, 92, 1214
    lose, 1057
Kosteneinsparungen
    durch Qualität, 1212
Kovarianz, 180, 405, 409
kritischer Bereich, 490, 503
    Lock, 512
    Synchronisationsobjekt, 505
    synchronized, 504, 512, 553
Kurzzeitgedächtnis, 1187

L

Label, 883, 885
Lambda, 262, 801, 802, 804
    als Parameter, 807
    als Rückgabewert, 807
    Bedeutung von this, 807, 808
    Erweiterbarkeit, 808
    im Java-Typsystem, 802
    Kurzschreibweisen, 806

vs. anonyme innere Klasse, 807
Zugriff auf Variablen, 808
Lambda-Ausdruck, 801
Last-In-First-Out, 421
lastDayOfMonth(), 873
Laufzeit, 857
Laufzeittyp, 95
launch(), 884
Law of the Big Three, 358
Layout, 456
Layoutmanagement, 647, 650
LayoutManager
BorderLayout, 651
CardLayout, 652
FlowLayout, 652
GridBagLayout, 653
GridLayout, 653
Layoutmanager, 648
Kombination, 655
Motivation für, 650
vordefinierte, 651
Lazy Init Proxy, 1150
Lazy Initialization, 960, 1107, 1329
Lebenslinie, 1387
Legacy-Code, 399, 879, 1205
LIFO, 421
LightBase, 922
limit(), 834, 836
lines(), 942
LinkedBlockingQueue<E>, 558
LinkedHashMap<K,V>, 363

removeEldestEntry(), 365
LinkedList<E>, 327, 333, 424, 1320, 1322
add(), 333, 334
get(), 333
Größenanpassung, 334
Knoten, 333, 334
Node, 333
remove(), 334
Speicherverbrauch, 334
Liskov Substitution Principle, 168
List, 810, 830
replaceAll(), 830
sort(), 807
list(), 942
List<E>, 318, 327
add(), 328, 423
get(), 328, 423
indexOf(), 328, 423
isEmpty(), 423
lastIndexOf(), 328
listIterator(), 330
remove(), 328, 423
set(), 328
subList(), 329
Liste, 327
heterogene, 417
homogene, 418
Listener, 1152
als (statische) innere Klasse, 670
als anonyme Klasse, 669
als separate Klasse, 671

in Komponentenklasse, 672
ListIterator<E>, 330
hasPrevious(), 330
nextIndex(), 330
previous(), 330
previousIndex(), 330
Literal, 154
Load Factor, 348
LOC, 1193
LocalDate, 868, 872, 878
LocalDateTime, 872, 877
Locale, 761
getAvailableLocales(), 762
getCountry(), 762
getLanguage(), 762
LocalTime, 868, 869, 872, 878
Lock, 503, 512, 553
Grundlagen, 512
lock(), 512, 529
unlock(), 512, 529
LoD, 163
Log-Konfiguration, 458
Log-Level, 458
log4j, 455
Layout, 456
Log-Konfiguration, 458
Log-Level, 458
Logger, 456
Logger, 456
Logging, 454
Logging-Framework, 454, 1180

log4j, 455
Lokalität, 1303
Long, 218
Cache, 229
parseLong(), 227
toString(), 282
LongStream, 833
boxed(), 833
Look-up-Tabelle, 360
Low-Level-Event, 663
Lower Type Bound, 409
LRU-Cache, 366, 1354

M

Magic Number, 154, 469, 961, 1180, 1298
Magic String, 469, 477, 1180
Main-Branch, 24
main-Thread, 491, 543
Major Garbage Collection, 632
Major GC, 633
Manifest-Datei, 54
Map, 938
computeIfAbsent(), 940
computeIfPresent(), 940
getOrDefault(), 939
größenbeschränkte, 364
merge(), 941
putIfAbsent(), 939
replace(), 939
replaceAll(), 942
map(), 835, 838, 840, 947

Map.Entry<K,V>, 360
Map<K,V>, 318, 319, 360
clear(), 361
containsKey(), 361
containsValue(), 361
entrySet(), 361
get(), 361
isEmpty(), 361
keySet(), 361
put(), 360
putAll(), 360
remove(), 360
size(), 361
values(), 361
mapToObj(), 833, 863
Mark-and-Compact-Algorithmus, 635
Mark-and-Sweep-Algorithmus, 633
Marker-Interface, 153
Maschinenzeit, 868
master, 19, 29
Matcher, 1273, 1274
Maven, 11, 59
Maven Central, 67
max(), 836, 931
McCabe, 1193
Mehrfachvererbung, 816
Mehrschrittoperation, 944
Membervariable, 86
Memory Leak, 332, 638, 1406
Garbage Collection, 638
Memory-bound, 1301

Memory-bound-Optimierungen, 1351
Menge, 334
Mengenoperation, 324
Merge, 19
merge(), 941
MessageFormat, 768
Messaging, 490
Metadaten, 579
Metainformation, 579, 592
Metaspace, 951
Methode, 86
    abstrakte, 86, 125
    Ausgangspunkt, 1181
Methodenlänge
    maximale, 1187
Methodenreferenz, 818
Metrik, 1165, 1192
    CBO, 1194
    Coupling Between Objects, 1194
    Cyclomatic Complexity, 1193
    DIT, 1194
    Lines Of Code, 1193
    LOC, 1193
    McCabe, 1193
    Method Lines Of Code, 1193
    MLOC, 1193
    NCSS, 1193
    NOC, 1194
    NOM, 1195
    Non Commented Sourcecode Statements, 1193
    Number Of Methods, 1195

Metriken, 1191
Middleware, 490
MIME, 954
min(), 836, 931
Minor Garbage Collection, 632
Minor GC, 632
MLOC, 1193
Mock, 1247
Mockito, 1250
Model-View-Controller-Architektur, 645, 703
Modena, 911
Modulo-Operator, 1275
Monitor, 503, 504
Month, 873
MonthDay, 874
MoreUnit, 1279
Multi Catch, 307, 1002
Multi Map, 940
Multicore, 821, 863
Multiplizität, 1381
Multitasking, 489
Multithreading, 489, 944
Murphy's Law, 1332
MutableTreeNode, 748
Mutator, 89
MVC, 645, 703

N

Nachbedingung, 115
Nachricht
    asynchron, 1388

synchron, 1388
Namen
aussagekräftige, 1174
Lesbarkeit, 1175
Semantik, 1175
sinnvoll gewählte, 1175
sinnvolle, konsistente, 1175
Namensgebung, 1174
grundlegende Regeln, 1171
sinnvolle, 1174
Namenskonsistenz, 1175
Namenskürzel
vermeide, 1174
Namensregeln, 1172
CamelCase-Schreibweise, 1172
Narrowing, 222
Nashorn, 948
natürliche Ordnung, 350
naturalOrder(), 926
NavigableMap<K,V>, 320, 367
ceilingKey(), 367
floorKey(), 367
higherKey(), 367
lowerKey(), 367
NavigableSet<E>, 319, 320
NCSS, 1193
Nebenläufigkeit, 489
negate(), 826, 827
NetBeans
IDE, 12
New Input Output, 942

Nightly Build, 61
NIO, 942
NOC, 1194
Node, 882, 908
Nodeeclipse, 80
Noise, 804
NOM, 1195
noneMatch(), 836
NoSuchElementException, 424
now(), 868
Null-Akzeptanz, 207, 211
Null-Objekt, 851, 929, 1005, 1128, 1331, 1333
null-Wert
    Behandlung von, 927
    IllegalArgumentException, 1088
    NullPointerException, 1088
NullPointerException, 828, 927–929, 932, 999, 1024, 1088
    unerwartete, 1004, 1005
nullsFirst(), 927
    Fallstrick, 928
nullsLast(), 927
    Fallstrick, 928
Number, 218
NumberFormat, 768
    Format, 769
    getCurrencyInstance(), 769
    getInstance(), 769
    getIntegerInstance(), 769
    getNumberInstance(), 769
    getPercentInstance(), 769
NumberFormatException, 225, 488

O

O-Notation, 1315

Object, 199, 803

- equals(), 200, 205, 345
- finalize(), 200, 640
- getClass(), 199, 202, 208
- hashCode(), 200, 345, 346
- notify(), 200, 498, 521, 529, 534, 552, 553, 1021
- notifyAll(), 200, 498, 521, 529, 534, 553, 1021
- toString(), 200–202
- wait(), 200, 498, 521, 529, 534, 552, 553, 1021

Objects

- requireNonNull(), 825

Objekt, 86

- Lebenszeit, 631

Objekt-Caching, 1351, 1354

Objekt-Pool, 1357

Objekt-Pooling, 1351

Objekt-Spaghetti, 1194

Objektdiagramm, 1376, 1383

Objektgraph, 288

Objektintegrität, 1087

- Probleme bei Remote Calls, 1071
- Probleme durch Zwischenzustände, 1071

Objektmethoden

- gebräuchlichste, 446

Objektzustand, 86, 200, 205, 313, 989, 1071, 1083, 1173, 1184, 1185, 1326

- erwarteter, 1081
- gültiger, 1091, 1181
- konsistenter, 1077
- ungültiger, 1081

Observer, 1152, 1153
als Anti-Pattern, 1160
OCPJP
Frage, 1360
of(), 868
ofDays(), 868
ofMonths(), 868
ofSecondOfDay(), 878
Old Generation, 631
OO-Design
Aggregation, 91
Assoziation, 91
Attribut, 86
behaves-like-Beziehung, 87
Call-by-Reference, 97
Call-by-Value, 97
can-act-like-Beziehung, 87
Cast, 96
Datenkapselung, 111
Generalisierung, 93
Interface, 86, 123
is-a-Beziehung, 93
Kapselung, 89, 111
Kohäsion, 90, 112
Komposition, 91
Kopplung, 92
Membervariable, 86
Methode, 86
Objektzustand, 86
Overloading, 93, 94
Overriding, 93, 94

Polymorphie, 95
Realisierung, 87
Referenz, 88
Schnittstelle, 123
Spezialisierung, 93
Sub-Classing, 95
Sub-Typing, 95
Trennung von Zuständigkeiten, 111
Typ, 87
Type Cast, 96
Typkonformität, 87
Vererbung, 92
Vererbung und Wiederverwendbarkeit, 113
OO-Metrik, 1194
OO-Techniken
abstrakte Basisklasse, 128, 129
grundlegende, 123
Interface, 123
Marker-Interface, 153
Read-only-Interface, 142
Realisierung, 129
Schnittstelle, 123
Open Closed Principle, 167
OpenOption, 943
Optimierung, 1300
80-zu-20-Regel, 1309
CPU-bound-, 1365
Grundlagen, 1300
I/O-bound-, 1344
Inlining, 1306
Loop-Unrolling, 1306

Memory-bound-, 1351
Peephole-, 536, 1307
Speicherverbrauch, 1351
von Stringoperationen, 1360
Vorher-nachher-Messungen, 1308
Optional, 851, 929
empty, 930
get(), 929
ifPresent(), 930, 931
isPresent(), 929
of(), 929
ofNullable(), 929
orElse(), 930
orElseGet(), 930
orElseThrow(), 930
OptionalDouble, 848, 931
OptionalInt, 931
OptionalLong, 931
Options
addOption(), 472, 473
or(), 826, 827
Orthogonalität, 91, 857, 1070, 1166, 1187
Out-of-Memory-Situation, 332, 1184
OutOfMemoryError, 951
OutputStream, 275
OutputStreamWriter, 275
Overloading, 93, 94, 210
Overriding, 93, 94, 210
invariantes, 182
kovariantes, 182

P

Paging, 706, 846
Pair Programming, 1170, 1174, 1289, 1290, 1295, 1401, 1402
Paketdiagramm, 1376
parallel(), 832
Parallelisierbarkeit, 843
Parallelisierung, 844
    Möglichkeit zur, 821
parallelPrefix(), 937
parallelSetAll(), 936
parallelSort(), 934
Parallelverarbeitung, 832, 854
    Wissenswertes zur, 854
Parameter Object, 159
Parameter Value Object, 159
Parameterprüfung, 1181
parse(), 868, 878
partitioningBy(), 852
Path, 817
Paths, 817
PatternSyntaxException, 249
peek(), 836, 842
Peephole-Optimierung, 536
Period, 872, 875, 876
Permanent Generation, 951
PerspectiveCamera, 921
Philosophenproblem, 511
PhongMaterial, 920
Pipeline-Verarbeitung, 844
plus(), 871
PMD, 1198

PointLight, 922
POLA, 1027, 1180
POLS, 1027
Polymorphie, 95, 110, 988
Port, 301
Post-Commit-Review, 1290
Post-Condition, 1181
Post-Decrement, 973
Post-Increment, 973
Postfix, 1172
Prädikat, 825
    boolesche Bedingung, 825
    komplexe Bedingung, 826
Präfix, 1172
Pre-Commit-Review, 1290
Pre-Condition, 1181
Pre-Decrement, 973
Pre-Increment, 973
Precondition
    Einhaltung von, 825
Predicate, 811, 825, 836, 840, 851, 947
    and(), 826, 827
    boolesche Bedingung, 825
    negate(), 826, 827
    or(), 826, 827
    test(), 811, 825
Preferences, 467, 482
    systemNodeForPackage(), 483
    systemRoot(), 483
    userNodeForPackage(), 483
    userRoot(), 483

primitiver Datentyp, 218
boolean, 218
byte, 218
char, 218
double, 218
float, 218
int, 218
long, 218
short, 218
Principle of Least Astonishment, 1027, 1180
Principle of Least Surprise, 1027
PrintStream, 244, 285
print(), 285
printf(), 243, 244
println(), 285
PrintWriter, 244
printf(), 244
Prioritätswarteschlange, 422
PriorityBlockingQueue<E>, 558
PriorityQueue<E>, 422
Problem des Handlungsreisenden, 1316
Producer-Consumer-Problem, 518
Profiling-Messung, 1313, 1322
Profiling-Tools, 1311, 1312
JConsole, 1312
JProbe, 1313
JProfiler, 1313
VisualVM, 1312
Programmierstil
einheitlicher, 1170
Programmierung

funktionale, 801
Projektion, 856
Properties, 467
getProperty(), 475, 476
load(), 476
Prototyp, 1110
provides-Beziehung, 125
Proxy, 1150, 1334, 1336
Prozess, 490
Psychologie, 1187
Publisher/Subscriber, 1152
Pull
Beobachter, 1153
Observer, 1153
Push
Beobachter, 1153
Observer, 1153
putIfAbsent(), 939

Q

Qualität
äußere, 1211
innere, 1211
Qualitätssicherung, 1291
Qualitätssicherungsmaßnahme
etablieren, 1212
Mehraufwand durch, 1212
Queue<E>, 319, 421, 424, 558
add(), 424
element(), 424
IllegalStateException, 424

NoSuchElementException, 424
offer(), 424
peek(), 424
poll(), 424
remove(), 424

R

Race Condition, 500, 502, 538, 1106
range(), 863
Raw Type, 185, 1298
RCP, 922
Read-only-Interface, 142, 1067
Readable, 286
readAllLines(), 942
Reader, 275, 286
ReadWriteLock, 512, 513, 944
readLock(), 515
writeLock(), 516
Realisierung, 87
konkrete, 129
reduce(), 836, 851
reentrant, 510
ReentrantLock, 512, 513
ReentrantReadWriteLock, 512, 513
Refactoring, 12, 1029
Encapsulate Field, 1056
Extract Interface, 1065
Extract Local Variable, 1022
Extract Method, 1068
High-Level, 1055
Standardvorgehen, 1038

Reference Counting, 630
Referenzsemantik, 97, 149
Referenzvergleich, 205
Reflection, 199, 577, 598, 908, 916, 978
Reflexivität, 207, 211
Regel
    Keep It Clean, 1167
    Keep It Human-Readable, 1166
    Keep It Natural, 1166
    Keep It Simple And Short, 1166
Registry, 482
Regressionstest, 40, 1214, 1273, 1279
regulärer Ausdruck, 246
    Bereichsangaben, 248
    Darstellung von Varianten, 247
    Spezialzeichen, 248
    Wiederholungen, 248
Rehashing, 349
Reihenfolge
    korrekte, 854
Release, 16, 59
Releasedatum, 871
Remote Debugging, 46
    Parametrierung der JVM, 46
Remote Proxy, 1150
Remote Repository, 27
remove(), 812
removeIf(), 827, 828
Renderer, 703, 709, 710
Reordering, 536, 539
replace(), 939

replaceAll(), 828, 830, 942
Repository, 19, 22, 65
    lokales, 27
ResourceManager, 790
ReverseComparator, 384
    reverseOrder(), 384
reversed(), 926
reverseOrder(), 926
Rhino, 948, 950
Rich-Client Experience, 901
Rich-Client-Framework, 922
Rich-Client-Plattform, 922
Rolle, 125, 138, 263
Rollenname, 1382
Round-Trip Engineering, 1378
Rückgabewert
    kovarianter, 183
Runnable, 491, 546, 567, 675, 803, 804, 944
    run(), 491
Runtime, 1311
RuntimeException, 300

S

SAM-Typ, 262, 803, 804
    Erweiterbarkeit, 808
Scanner, 286, 287
    ioException(), 287
    useDelimiter(), 286
Scene, 882
SceneBuilder, 895
Scenegraph, 882

Schablonenmethode, 1131
ScheduledExecutorService, 570
    scheduleAtFixedRate(), 571
    scheduleWithFixedDelay(), 571
ScheduledThreadPoolExecutor, 570
Scheduler, 490
Schnittmenge, 324
Schnittstelle, 123
Schönwettersoftware
    unzuverlässige, 296
SCJP
    Frage, 1360
ScriptEngineFactory, 948
ScriptEngineManager, 948
    getEngineByName(), 949
Scripting Engine
    auflisten, 948
Seiteneffekt, 313, 1180
Select, 1328
Semaphore, 533, 553
Sequenzdiagramm, 1376, 1377
Serialisierung, 577, 600
Server, 123
Service Locator, 124
Set<E>, 318, 334
Shallow Copy, 148
Shape3D, 920
Short, 218
Short-circuiting Operations, 835
Shortcut-Return, 1181
Shut-down-Hook, 543, 981

Sichtbarkeit, 536, 537, 1186
    private, 89
    protected, 89
    public, 89
Signatur, 87
Silent Fail, 1182
SimpleDateFormat, 775, 866, 867
    Fallstricke, 776
Single Abstract Method, 262
Single Point of Failure, 32
Single Responsibility Principle, 166
Single-Thread-Regel, 676
Singleton, 479, 502, 1105
    als Anti-Pattern, 1109
    Lazy Initialization, 1107
    Race Condition, 1106
Skalierbarkeit, 1314
skip(), 836
sleep(), 498
Socket
    close(), 302
SocketChannel, 288
Softwarequalität, 1206
    Sichern der, 1206
SOLID, 99
SOLID-Prinzipien, 1186
Sollkonzept, 1397
SonarQube, 1202
sort(), 807, 810, 822
sorted(), 836, 841, 845, 947
SortedMap<K,V>, 320, 367

SortedSet<E>, 319, 320, 337
first(), 338
headSet(), 338
last(), 338
subSet(), 338
tailSet(), 338
Sortierung
ClassCastException, 336
Comparable<T>, 350
Comparator<T>, 350
TreeMap<K,V>, 350
TreeSet<E>, 350
Sourcecode-Checker, 1191
Sourcecode-Duplikation
Vermeiden von, 1002
Sourcecode-Duplizierung, 91
Erkennung von, 1198
Spezialisierung, 93
Sphere, 920
split(), 840
SQL
Select, 1328
Stack<E>, 421, 422
empty(), 422, 423
peek(), 422
pop(), 422
push(), 422
search(), 422
StackOverflowError, 1125
StackPane, 883, 886
Stage, 882

Staging Area, 29
Stamm, 19
StampedLock, 944
Standardverhalten
    vorgeben, 812
start(), 884
Startzustand
    Zustandsdiagramm, 1392
Starvation, 510, 511
statische Analyse, 1191
Stereotyp, 1380
Strategie, 1135
Stream, 275, 821, 830, 832
    allMatch(), 836, 849
    anyMatch(), 836, 849, 850
    collect(), 836, 847
    count(), 836, 848
    Create Operations, 831
    distinct(), 836, 845
    für primitive Typen, 832
    filter(), 835, 837, 839, 846
    findAny(), 836, 850
    findFirst(), 836, 850
    flatMap(), 836, 839, 853, 947
    forEach(), 836
    forEachOrdered(), 854
    Intermediate Operations, 835, 837, 845
    limit(), 834, 836, 846
    map(), 835, 838, 840, 947
    max(), 836
    min(), 836

noneMatch(), 836
parallelStream(), 854
peek(), 836, 842
reduce(), 836, 851
skip(), 836, 846
sorted(), 836, 841, 845, 947
Terminal Operations, 835, 836, 846
toArray(), 834, 836
unendlicher, 834
Stream-im-Stream, 839
String, 200, 234, 278, 286
charAt(), 1030
format(), 243
getBytes(), 279, 282
isEmpty(), 828
length(), 1034
split(), 246, 287, 840
toLowerCase(), 272
trim(), 311, 488
valueOf(), 203
StringBuffer, 204, 234, 239
delete(), 242
deleteCharAt(), 242
equals(), 242
insert(), 242
StringBuilder, 204, 234, 239
append(), 204
delete(), 242
deleteCharAt(), 242
equals(), 242
insert(), 242

StringIndexOutOfBoundsException, 1030, 1037
Stringliteral, 235
Stringliteral-Pool, 235, 1360
Stringobjekt, 235
StringTokenizer, 244, 246, 286, 287, 311
    countTokens(), 245
    hasMoreTokens(), 244
    nextToken(), 244, 311
Strukturdiagramm, 1376
Stub, 1247, 1248
Sub-Classing, 95
Sub-Typing, 95
Substituierbarkeit, 93, 262
Substitutionsprinzip, 93, 423
Subversion, 11
Suche
    Binärsuche, 374
    binarySearch(), 374
    contains(), 374
    containsAll(), 374
    containsKey(), 374
    containsValue(), 374
    indexOf(), 374
    lastIndexOf(), 374
Suchen
    Die Rolle von equals(), 206
Supplier, 811, 834
    accept(), 811
    get(), 811
supplyAsync(), 945
Survivor-Bereich, 631

Swimlanes, 1394
Swing, 644, 645
    Multithreading, 674
SwingUtilities, 675
    invokeAndWait(), 675
    invokeLater(), 675
    isEventDispatchThread(), 676
SwingWorker<T,V>, 679
    doInBackground(), 680
    done(), 680
    process(), 680
    publish(), 680
Symmetrie, 207
Synchronisationsobjekt, 505
Synchronisierung, 490
synchronized, 503, 504, 506, 512, 514, 539, 552, 553
Synchronizer, 553
SynchronousQueue<E>, 559
System
    arraycopy(), 322
    currentTimeMillis(), 865, 877
    getenv(), 481
    getProperty(), 481
System-Properties, 481
System.in, 276
System.out, 276
Systemtest, 1207
Szenarien, 1387

T

Tabelleninhalt

dynamischer, 745
Target
Ant, 62
Task, 561
Ant, 62
TCP, 301
TDD, 1208, 1211, 1215, 1400, 1402
technical debt, 1291
teile und herrsche, 106, 572
Template-Methode, 1131
Temporal, 878
TemporalAdjusters, 873
TemporalUnit, 871
Terminal Operations, 835
test(), 811, 825
Test-Driven Development, 1208, 1215, 1400, 1402
Testabdeckung, 1280
Testen, 1206
Testfixture, 1222
Testklasse, 1172
Postfix, 1172
TestNG, 1273
TestNG-Framework, 41
Texteffekt, 912
Textfixture, 1218
thenAccept(), 945
thenApply(), 945
thenApplyAsync(), 946
thenCombine(), 946
Thread, 490, 491, 546
Ableitung von der Klasse, 546

Daemon-, 543
dumpStack(), 466
getState(), 496
getPriority(), 494
getThreadGroup(), 495
Gruppen, 495
interrupt(), 499–501, 545, 548
interrupted(), 548
isAlive(), 493, 531
isInterrupted(), 499, 500, 548
join(), 530, 531, 553
main-, 491, 543
Priorität, 494
run(), 491, 531
setDaemon(), 543
setDefaultUncaughtExceptionHandler(), 545
setPriority(), 494
setUncaughtExceptionHandler(), 545
sleep(), 498, 548
start(), 492, 531
State, 496
stop(), 494, 545
ThreadUtils, 495
User-, 543, 551
yield(), 498
Thread-Pool, 563
Thread-Scheduler, 497
ThreadGroup, 495
activeCount(), 495
enumerate(), 495
ThreadPoolExecutor, 564

Throwable, 310
Tiefe des Baums, 356
tiefe Kopie, 147, 148
Time, 866
Timer, 548, 570
    cancel(), 551
    schedule(), 549
    scheduleAtFixedRate(), 550
TimerTask, 548, 570
    cancel(), 549
    run(), 549
    scheduledExecutionTime(), 549
Timestamp, 866
TimeUnit, 570
toArray(), 834, 836
toCollection(), 847
Token, 244, 286
toList(), 847
Tools
    Checkstyle, 1196
    FindBugs, 1199
    JDepend, 1201
    PMD, 1198
    SonarQube, 1202
    Sourcecode-Checker, 1191
totalMemory(), 1311
Transformation
    in Lambda, 804
Transitivität, 207
Transmission Control Protocol, 301
Traveling-Salesman-Problem, 1316

Traversierung, 324
TreeItem, 915, 916
TreeItemPropertyValueFactory, 916
TreeMap<K,V>, 350, 367, 560, 1325
TreeNode, 747
TreeSet<E>, 336, 337, 350, 1325
TreeTableCell, 918
TreeTableColumn, 916
TreeTableView, 914, 916, 918
TreeView, 915
Trennung von Zuständigkeiten, 85, 111, 216, 1077
trunk, 29
Typ, 87
Type Cast, 96
Type Erasure, 190, 399
Type Inference, 806
Typhierarchie
    Collection<E>, 318
    Map<K,V>, 319
Typkonformität, 87
Typkürzel
    ?, 323
    ? extends E, 323
    E, 323
    K, 323
    T, 323
    V, 323
Typumwandlung, 972

U

Überschreiben

invariantes, 182
kovariantes, 182
Umgebungsvariable, 481
UML
Aggregation, 1382
Aktivitätsdiagramm, 1376, 1377
Anwendungsfalldiagramm, 1376
Assoziation, 1381
Klassendiagramm, 1376, 1379
Stereotyp, 1380
Kommunikationsdiagramm, 1376
Komponentendiagramm, 1376
Komposition, 1382
Objektdiagramm, 1376, 1383
Paketdiagramm, 1376
Sequenzdiagramm, 1376, 1377
Strukturdiagramm, 1376
Verhaltensdiagramm, 1376
Verteilungsdiagramm, 1376
Zustandsdiagramm, 1376, 1377
UnaryOperator, 828, 829, 838
identity(), 828
Unboxing, 225
UncaughtExceptionHandler, 544
Unchecked Exception, 299
Unendliche Streams, 834
Unicode, 235, 278
Unit Test, 35, 251, 1031, 1038, 1039, 1207, 1211, 1291
als Regressionstest, 1273
Cobertura, 1280
EasyMock, 1250

EclEmma, 1284
für Legacy-Code, 1236
Hamcrest, 1273
Infinitest, 1279
Mockito, 1250
MoreUnit, 1279
Motivation für, 1227
reguläre Ausdrücke absichern, 251
Unit-Test-Framework, 1215, 1402
Unordnung, 1167
UnsupportedOperationException, 140, 325, 327, 370, 812, 999, 1122, 1129
Upper Type Bound, 409
Use Case, 1385
Use-Beziehung, 123
User-Thread, 543, 551
UTF-16, 279
UTF-8, 279
Utility-Klasse, 213
Arrays, 322
HashUtils, 346
SleepUtils, 492, 500

V

V-Modell, 1398, 1399
Value Object, 159, 1075
Optimierung, 1363
Varargs, 159, 785
Variable
atomare, 553
Zugriff auf, 807
Varianz, 179

VBox, 886
Vector<E>, 327, 330, 422, 1320, 1322
    add(), 331
    get(), 331
    Größenanpassung, 332
    Kapazität, 330
    remove(), 332
    size(), 330
    Speicherverbrauch, 332
    trimToSize(), 332
Verarbeitungskette, 843
Verarbeitungsschritte, 835
Vererbung, 92, 1194
Vererbung und Wiederverwendbarkeit, 113
Vererbungshierarchie, 138, 262, 993
    breite, 1191
    tiefe, 1191
Vergleich
    Referenz-, 205
Verhalten, 85, 205
Verhaltensdiagramm, 1376
Verhaltensweise, 125, 263
Verklemmung, 490
Versionsverwaltung, 11
    auschecken, 22
    Branch, 24
    Checkin, 22
    Checkout, 22
    commit, 22
    dezentrale, 21
    einchecken, 22

Konflikt, 22
Main-Branch, 24
Marke, 24
Merge, 22
Repository, 19, 22
Stamm, 24
Synchronize, 22
Tag, 24
Trunk, 24
Update, 22
zentrale, 21
Verteilungsdiagramm, 1376
Verzeichnisüberwachung, 273
VisualVM, 1312, 1407
volatile, 537–539, 546, 552, 553
Vorbedingung, 115
Vorgehensmodell, 1397

W

Wahrnehmungspsychologie, 1167
Gesetz der Ähnlichkeit, 1167
Gesetz der Nähe, 1169
Wartungsalbtraum, 971
Wasserfallmodell, 1398
weakly consistent, 557
Whitebox-Sicht, 1384
Whitebox-Test, 1208
Whitespace, 828
Widening, 222, 230, 811, 833
Wiederverwendbarkeit, 85, 1070, 1185, 1187
Wiederverwendung, 92

Wizard, 653
Working Copy, 22
Wrapper, 1119
Wrapper-Klassen, 218
write(), 943
Writer, 275, 278
    close(), 278
    flush(), 278
    write(), 278
Wurzel, 356

X

XP, 1208, 1211, 1400, 1401

Y

Year, 874
    isLeap(), 874
YearMonth, 874
Young Generation, 631

Z

Zeichnen
    in GUI-Komponenten, 683
ZonedDateTime, 877, 879
ZoneId
    getAvailableZoneIds(), 880
Zuständigkeit
    beim Testen, 1209
Zustand, 85, 205
Zustandsübergang, 1391
Zustandsdiagramm, 1376, 1377, 1392
Zustandsinformation, 1180

Zustandsprüfung, 825
    Einführen einer, 1086
    IllegalStateException, 1086
Zustandstransition, 1391
Zweierkomplement, 219, 232
Zyklen, 1125

Fußnoten

1 Einleitung

[1] http://www.oracle.com/technetwork/java/codeconv-138413.html

[2] http://www.ambysoft.com/essays/javaCodingStandards.html

2 Professionelle Arbeitsumgebung

[1] Wenn man in der IDE den Ordner `test` als zusätzliches Sourcecode-Verzeichnis wählt.

[2] Es müssen dann die gesamten Dateien eines Projekts überprüft werden, um herausfinden, welche Version jede Datei zum Zeitpunkt des gewünschten Stands hatte. Anschließend muss dann jede Datei einzeln anhand der ermittelten Versionsnummer wiederhergestellt werden.

[3] http://git-scm.com/downloads

[4] http://mercurial.selenic.com/

[5] Mercurial und Git legen dazu versteckte Ordner und Dateien an.

[6] Insbesondere muss kein zentraler Server mit Repository existieren – für verteilte Teams ist dies aber hilfreich.

[7] Mercurial bedeutet Quecksilber und hg ist das chemische Kürzel dafür.

[8] Bitte beachten Sie, dass dieser Befehl auch alle Unterverzeichnisse durchsucht, wodurch die Liste recht umfangreich werden kann. Möchte man die Treffermenge etwa auf PDF-Dateien begrenzen, so kann man für Mercurial Folgendes eingeben: `hg status *.pdf`. Es gibt weitere Möglichkeiten zur Einschränkung. Ergänzende Hinweise gibt `hg status -help`.

[9] Damit ein Abgleich mit Remote Repositories erfolgen kann, müssen diese entsprechend konfiguriert werden. Ich setze voraus, dass dies initial geschehen ist.

[10] Damit lassen auch ohne weiteres Integrations- und Systemtests schreiben.

[11] Annotations sind mit dem Zeichen '`@`' beginnende Markierungen im Sourcecode, die Metainformationen zu einem damit gekennzeichneten Element (Klasse, Methode usw.) beschreiben.

[12] Der merkwürdige Name »Entwanzer« geht historisch auf die großen wohnraumfüllenden ersten Computer zurück, in denen Insekten zu Fehlern in den Schaltungen geführt haben. Eine kurze Anekdote findet man im Buch »Der Pragmatische Programmierer« von Andrew Hunt und David Thomas [48] in Abschnitt 18 zum Thema Fehlersuche.

[13] Die folgenden Beschreibungen beziehen sich auf Eclipse. Die Abläufe sind aber in ähnlicher Weise auch mit anderen IDEs möglich.

[14] Wenn Programmzeilen mehrere Methodenaufrufe hintereinander etwa `calcSum(x, y) + calcFactorial(y)`, oder als verschachtelten Aufruf, z. B. `calcSum(x, calcFactorial(y))`, enthalten, werden diese der Reihe nach abgearbeitet.

[15] JDWP steht für Java Debug Wire Protocol. Das ist ein spezielles Protokoll zur Kommunikation zwischen einem Debugger und einer JVM. Eine Übersicht über alle Optionen bekommt man durch die Aufrufe von `java -agentlib:jdwp=help` bzw. `java -Xrunjdwp:help`.

[16] Tatsächlich kann man den Build auch mit Gradle sehr feingranular beeinflussen – als normaler Anwender wird man dies eher selten nutzen.

[17] Einige Build-Tools, wie das später vorgestellte Gradle, erlauben es, Projektdateien für die IDE zu generieren. Wenn man diese Funktion nutzt, kann man Build-Skript und IDE leichter synchron halten. Gerade bei vielen externen Abhängigkeiten ist das praktisch.

[18] `http://hudson-ci.org/`

[19] `http://jenkins-ci.org/`

[20] Maven unterscheidet zwischen Release- und (in der Entwicklung befindlichen) Snapshot-Versionen, wobei Letztere explizit durch das Postfix `-SNAPSHOT` gekennzeichnet werden.

[21] Dort finden Sie diverse weitere Informationen rund um Gradle – insbesondere gibt es dort auch verschiedene Tutorials, die als Ergänzung zu diesem Text dienen können.

[22] Vorsicht: Die Gradle-Tasks entsprechen in etwa den Ant-Targets und nicht den Ant-Tasks. Letztere repräsentieren eher einfache Kommandos im Build. Das wird in Gradle nicht direkt abgebildet, aber man kann eigene Tasks schreiben.

[23] Falls dies nicht reichen sollte, können weitere Tasks selbst definiert werden. Das erfolgt in Form von Groovy-Skripten und auf einer höheren Abstraktionsebene und deutlich besser lesbar als bei der Beschreibung des Builds mit Ant.

[24] Noch besser ist es, das Task-Autowiring zu nutzen und auf das explizite `dependsOn` zu verzichten. Dann schreibt man nur noch `classpath=jar.outputs.files`.

[25] Natürlich sind andere Aufteilungen denkbar.

3 Objektorientiertes Design

[1] Danke an meinen Freund Tim Bötzmeyer für diese schöne Wortschöpfung.

[2] Auch ohne explizite Beschreibung besitzt jede Klasse eine Schnittstelle in Form der von ihr definierten Methoden, die für andere Klassen aufrufbar sind.

[3] Mit Java 8 ändert sich diese strenge Unterscheidung, da es dann erlaubt ist, in Interfaces mithilfe sogenannter Defaultmethoden bereits ein Standardverhalten vorzugeben.

[4] Dies entspricht der Adresse im Speicher, wo das Objekt nach seiner Erzeugung abgelegt ist.

[5] Man kann diese Sichtbarkeit durch einen Kommentar der Form `/*private*/` bzw. `/*package*/` andeuten. Dies ist hilfreich, um versehentliche Sichtbarkeitserweiterungen zu verhindern. Man dokumentiert somit, dass bewusst *diese* Sichtbarkeit gewählt wurde.

[6] Beide Varianten der Namensgebung sind sinnvoll, und die konkrete Wahl ist eine Frage des Geschmacks (oder aber durch sogenannte Coding Conventions (vgl. Kapitel 19) festgelegt).

[7] Änderungen im API würden dann in allen Applikationen, die das API einsetzen, zu Inkonsistenzen oder Kompilierfehlern führen.

[8] Delegation beschreibt, dass eine Aufgabe einem anderen Programmteil übergeben wird, etwa indem eine Klasse eine andere über eine Assoziation kennt und deren Methoden aufruft.

[9] *=Angepasster Wert: Um sicherzustellen, dass nur Rasterpunkte eingenommen werden, erfolgt in beiden Fällen durch Aufruf der Methode `snapToGrid(int)` eine Modulo-Korrektur.

[10] Darunter versteht man beliebige andere Klassen, die Methoden der eigenen Klasse aufrufen.

[11] Eine zentrale Stelle wird häufig durch einen sogenannten Service Locator realisiert, meistens eine Klasse, die einen wohldefinierten Zugriff auf Services und Schnittstellen bereitstellt.

[12] Diese Einschränkung ist in vielen Fällen eher ein Vorteil als ein Nachteil. Außerdem können mithilfe der

Entwurfsmuster ERZEUGUNGSMETHODE und FABRIKMETHODE (vgl. Abschnitte 18.1.1 und 18.1.2) oder aber auch Frameworks wie Spring entsprechende Realisierungen des Interface erzeugt werden.

[13]Eine solche Leerimplementierung kann dann nützlich sein, wenn diese Methode nicht zwangsläufig von Subklassen mit spezifischer Funktionalität versehen werden muss.

[14]Beide Namen sind valide und besitzen Vor- und- Nachteile. Mit `BaseFigure` werden Signaturen besser lesbar – das Präfix `Abstract` ist dort eher störend, weil es suggerieren würde, dass Instanzen abstrakter Klassen übergeben würden. Allerdings wird durch Namen wie `AbstractGraphicsFigure` kommuniziert, dass man keine Instanz erzeugen kann.

[15]Teilweise möchte man Interfaces auch explizit zusammenfassen, um Aufrufern die Handhabung zu erleichtern. In Abschnitt 18.2.1 wird das für das Entwurfsmuster FASSADE diskutiert.

[16]Die Grundidee des Beispiels habe ich dem Buch »Einstieg in Java 6« von Bernhard Stephan [84] entnommen.

[17]Bemerkenswerterweise gibt es sogar einige Tierarten, die sich nicht fortbewegen, etwa Schwämme und Korallen. Daran sieht man, wie schwierig oder teilweise gar nicht realisierbar es ist, ein möglichst allgemeingültiges Modell zu definieren.

[18]Es gibt Programmiersprachen, wo dies möglich ist. In Java geht das jedoch nicht.

[19]Heutzutage wird man bei Neuentwicklungen eher auf Annotations setzen, Marker-Interfaces findet man jedoch noch vielfach im JDK, weshalb sie hier behandelt werden.

[20]Entsprechende Sichtbarkeiten und Zugriffsrechte vorausgesetzt.

[21]Natürlich könnte man diese Veränderung im Objekt selbst feststellen. Hier ist aber gemeint, dass keine Methode der Klasse aufgerufen wird, sondern die Änderungen indirekt erfolgen.

[22]Allerdings ist zu bedenken, dass bei sehr umfangreichen Datenstrukturen auch einiges an Speicher belegt und eine entsprechende Anzahl an Referenzen kopiert wird – nicht jedoch die Objekte selbst!

[23]Dies kann durch die Referenzsemantik zu Missverständnissen oder Fehlern führen. Diverse Methoden der Klasse `String`, des bekanntesten Vertreters dieser Technik aus dem JDK, suggerieren durch ihren Namen eine Veränderlichkeit. Dies ist jedoch nicht der Fall, stattdessen liefern die Methoden jeweils neu erzeugte Stringobjekte zurück (vgl. Abschnitt 4.3.1).

[24]Implementiert eine Klasse das Interface `java.io.Serializable`, so sind weitere Anstrengungen nötig, um die Unveränderlichkeit sicherzustellen. Details dazu finden Sie in der zweiten Auflage des Buchs »Effective Java« von Joshua Bloch [9].

[25]Mit Literalen bezeichnet man Zeichenfolgen, die Werte von Basistypen darstellen, etwa die Zeichenfolge '`123`' als `int`-Literal oder '`true`' als Literal vom Typ `boolean`.

[26]Das Namenssuffix `Enum` wurde hier nur zur Unterscheidung von der auf `int`-Konstanten basierenden zuvor vorgestellten Aufzählung gewählt und sollte normalerweise entfallen.

[27]Details zur generischen Definition von Klassen beschreibt Abschnitt 3.7.

[28]Die Herkunft der Begrifflichkeit wird unter `http://www.adam-bien.com/roller/abien/entry/value_object_vs_data_transfer` dargelegt.

[29]Eine Austauschbarkeit der Implementierung ohne Nebenwirkungen gilt für die Praxis nur dann, wenn keine zeitlichen Randbedingungen vom Aufrufer angenommen werden.

[30]Teilweise ist es hilfreich, wenn Objekte über klar definierte Schnittstellen (in Form eines Interface oder einer abstrakten Klasse) miteinander kommunizieren. Zudem sollten die Abhängigkeiten (Dependencies) in der Regel nicht von Klassen selbst aufgebaut, sondern besser von außen und möglichst spät »injected« (eingeimpft) werden.

[31]Diese Aussage lässt sich für Methoden, Klassen und Komponenten nutzen, wobei der Kontext der Aufgabe natürlich größer wird, wenn man von Methoden bis hin zu Komponenten geht.

[32] Als Bad Smell bezeichnet man eine (potenziell) problematische Programmstelle.

[33] Der Einsatz des Entwurfsmusters FASSADE (vgl. Abschnitt 18.2.1) führt auch recht leicht zu derartigen Interfaces. In diesem Fall ist das aber in Ordnung, weil eine Fassade eine Schnittstellensammlung und einen externen Zugriffspunkt als Abstraktion vieler interner Schnittstellen darstellt.

[34] Das Problem liegt teilweise darin, wie die Abhängigkeit erzeugt wird. Manchmal ist es auch akzeptabel, auf ein Interface zu verzichten, solange man die Klassen, die man benötigt, nicht selbst instanziiert.

[35] In der Praxis erfordert die Konstruktion von Objekten einiges an Aufwand, weil an Konstruktoren oft diverse Parameter übergeben und diese Informationen zuvor ermittelt werden müssen.

[36] Diese ist online unter `http://docs.oracle.com/javase/specs/` verfügbar.

[37] Es sind auch Subtypen davon erlaubt.

[38] Wodurch im Unterschied zur vorherigen Variante in dieser `T`-Variante die `BaseFigure`-Spezialisierung »festgehalten« wird.

[39] Allerdings verbleiben einige Hinweise zu generischen Definitionen von Attributen und Methodenparametern im Bytecode. Darauf gehe ich später in Abschnitt 8.1 ein.

[40] Das ist zunächst verwunderlich, da man hier eher erwarten würde, dass der Vergleich fehlschlagen und der Wert `false` zurückgegeben würde. Ist keine Typkompatibilität gegeben, kommt es jedoch zu einem Kompilierfehler mit der obigen Fehlermeldung.

4 Java-Grundlagen

[1] Außer die Klasse wird sicher niemals in Hashcontainern verwaltet.

[2] Details zu Listen und anderen Datenstrukturen beschreibt Kapitel 5 und geht dabei ausführlich auf das Collections-Framework ein.

[3] Weil sichergestellt ist, dass der zum Vergleich benutzte Attribut `name` nicht `null` ist.

[4] Unter einem Idiom versteht man eine beispielhafte Umsetzung eines speziellen Problems auf Ebene des Sourcecodes. Idiome sind also »Entwurfsmuster« auf Programmzeilenebene.

[5] `http://de.wikipedia.org/wiki/IEEE_754`

[6] Der obere Wert lässt sich über die System-Property `java.lang.Integer.IntegerCache.high` ändern. Verschiedene Werte können das Programmverhalten (unerwartet) ändern.

[7] Das erinnert an Probleme des Cachings bei den Wrapper-Klassen (vgl. Abschnitt 4.2.2).

[8] Übrigens sind genau dies auch die Fallstricke, die in der OCPJP/SCJP-Prüfung gern angewendet werden, um Fehler zu provozieren.

[9] Mit der Ausnahme, dass man bereits genutzte Teile aus Stringliteralen miteinander verknüpft, etwa `"A" + "BC"`, wenn das Stringliteral `"ABC"` schon vorher im Sourcecode vorkam.

[10] Seit JDK 8 muss eine Variable nicht mehr explizit `final` definiert werden, sondern es reicht aus, wenn sich diese nicht ändert und damit »***effectively final***« ist. Dieses Verhalten wurde auch in neuere Varianten des JDK-7-Compilers aufgenommen.

[11] Auch hierbei handelt es sich um einen Fallstrick beim OCPJP/SCJP.

[12] In diesem speziellen Fall kann daher genau nur das angegebene Interface erfüllt werden. Sind mehrere Interfaces zu erfüllen, so ist es erforderlich, dass man ein neues Interface einführt, das die benötigten Interfaces erweitert.

[13] Da ein `File`-Objekt lediglich einen Pfad im Dateisystem repräsentiert, wird durch einen Konstruktoraufruf weder eine Datei noch ein Verzeichnis angelegt. Das erfordert explizit einen Methodenaufruf.

[14] Diese Eigenschaft hat unter anderem zu den Erweiterungen bezüglich des nicht blockierenden I/O geführt.

[15] Zumindest sind seit JDK 7 in der Klasse `java.nio.charset.StandardCharsets` die sechs in jeder JVM verfügbaren Standard-Charsets definiert.

[16] Eine Liste gültiger Namen findet man unter `http://download.oracle.com/javase/8/docs/technotes/guides/intl/encoding.doc.html`.

[17] Es sei denn, alle Felder haben eine feste und konstante Länge.

[18] Würde man Typinformationen ebenfalls in den Stream schreiben, würde man damit beginnen, den in Java integrierten Serialisierungsautomatismus (vgl. Abschnitt 8.3) nachzubauen.

[19] Das gilt nur dann, wenn die Exception vom Typ Checked Exception (vgl. Abschnitt 4.7.2) ist und keine `RuntimeException` wie `IllegalArgumentException` ausgelöst wird. Allerdings kann man auch diese explizit mit einem `catch`-Block bearbeiten.

[20] Da meistens eine Ausnahmesituation vorliegt, ist der Mehraufwand oftmals irrelevant.

[21] Daher sind Unchecked Exceptions in der Regel nicht Bestandteil der Methodensignatur (obwohl man sie dort explizit aufführen kann).

[22] Darunter versteht man unerwartete Modifikationen am eigenen Objektzustand (oder noch schlimmer am Objektzustand eines anderen Objekts).

5 Das Collections-Framework

[1] Vermutlich weil dies für eine Enthaltensein-Prüfung eine zu starke Einschränkung gewesen wäre.

[2] Die von `Collections.unmodifiableList(List<? extends T>)` erzeugte Spezialisierung einer Liste stellt lediglich eine unveränderliche Sicht dar. Ein Aufruf von verändernden Methoden führt zu `UnsupportedOperationException`s.

[3] Es kommt zu einer `IndexOutOfBoundsException`, falls kein Element an dieser Position existiert. Für `add()` ist jedoch auch der nicht existierende Index `list.size()` erlaubt.

[4] Das neu entstehende Array ist um die Hälfte größer als die ursprüngliche Größe, weil diese Vergrößerung derart in der Implementierung der `ArrayList<E>` programmiert ist. Im Beispiel wäre die neue Größe also 750 MB.

[5] Werden diese nicht eingehalten, sollte dies unbedingt in der Javadoc vermerkt werden.

[6] Die in Eclipse eingebaute Automatik aus dem Menü SOURCE –> GENERATE HASHCODE() AND EQUALS()… erzeugt eine `hashCode()`-Methode, die potenziell zu viele Attribute nutzt.

[7] Über das »natürlich« kann man sich trefflich streiten, weil es nur um die Vorgabe einer Reihenfolge durch die Implementierung geht und die Ordnung auch unintuitiv oder unerwartet sein kann und sich somit möglicherweise sogar eher unnatürlich anfühlt.

[8] Allerdings ist die Reihenfolge für den Vergleich unbedingt zu beachten. Während diese für `equals(Object)` keinen Einfluss auf das Ergebnis besitzt, macht es für `compareTo(T)` möglicherweise einen großen Unterschied: Es ist entscheidend, ob erst die Namen und dann das Alter oder erst das Alter und dann die Namen verglichen werden.

[9] Für die Typen `float` und `double` sind Rundungsfehler zu bedenken (vgl. Abschnitt 4.1.2).

[10] Schnell stellt sich die Frage: Was ist mit gleichen Werten? In den baumbasierten Datenstrukturen `TreeSet<E>` und `TreeMap<K, V>` werden nicht mehrere gleiche Werte gespeichert, sondern es existiert jeweils

nur ein derartiger Eintrag. Für Mengen ist dies per Definition so, für Maps gilt dies, da hier die Eindeutigkeit von Schlüsseln gefordert wird. Versucht man trotzdem einen gleichen Wert zu speichern, so wird der alte Eintrag ersetzt, also für `TreeMap<K, V>` ein neuer Wert für den Schlüssel eingetragen.

[11] Ausnahmen davon sind entsprechend zu dokumentieren.

[12] Das gilt ebenso, wenn ein `Comparator<SimplePerson>` genutzt wird.

[13] Um die Eigenschaft der Zugriffsreihenfolge überhaupt setzen zu können, ist man durch die Konstruktoren der `LinkedHashMap<K, V>` dazu gezwungen, die Werte für Initialkapazität und Füllgrad (Load Factor) anzugeben.

[14] Bei einer Realisierung als Subklasse kann man dies nur durch Überschreiben und Auslösen von z. B. einer `UnsupportedOperationException` in der Implementierung ausdrücken.

[15] Das findet erst in JDK 8 mit dem Filter-Map-Reduce-Framework und den Stream-Klassen Einzug in Java und wird ausführlicher in Kapitel 12 besprochen.

[16] Erweitert man die Eingabemenge der Primzahlen um die Werte 17 und 19, so wird als Folge die Zahl 7 nicht mehr gefunden und die Binärsuche liefert die Rückgabe -1.

[17] Zudem ist eine Realisierung aufgrund der unsortierten Speicherung in einer Hashtabelle nicht möglich.

[18] Die Ausnahme bildet der Wert `Integer.MIN_VALUE`. Für diesen gilt die Anomalie – `Integer.MIN_VALUE == Integer.MIN_VALUE` (vgl. Abschnitt 4.2).

[19] Natürlich können Methoden nicht im Nachhinein aus einem Interface entfernt werden. Stattdessen löst ein Aufruf solcher Methoden eine `UnsupportedOperationException` aus.

[20] Älterem, meistens überarbeitungswürdigem Sourcecode.

[21] Eine Änderung des Typs ist durch das Casten zunächst auf `Object` und anschließend auf irgendeinen anderen Typ möglich und führt logischerweise zu Typfehlern.

[22] Für Interfaces existiert keine Notation mit `implements`, sondern es wird hier, analog zu den bereits bekannten Typeinschränkungen, das Schlüsselwort `extends` genutzt.

[23] Meiner Ansicht nach ist das wenig intuitiv und mitunter sogar problematisch. Joshua Bloch wollte in einer persönlichen Diskussion auf der Java One 2009 nicht ganz so weit gehen. Allerdings war ihm die Problematik bereits beim Entwurf des Collections-Frameworks bewusst und daher existiert `Arrays.deepEquals(Object[], Object[])`.

[24] Die Methode `add(E)` existiert bereits im Interface `Collection<E>`, ist hier aber aufgeführt, um jeweils Methodenpaare vorzustellen und auf eine Besonderheit hinzuweisen.

6 Applikationsbausteine

[1] `http://commons.apache.org/`

[2] `https://code.google.com/p/guava-libraries/`

[3] Allerdings sieht man durch die Indirektion nicht genau, was geprüft wird, und muss sich darauf verlassen, dass dort auch das Richtige geschieht.

[4] CSV steht für Comma Separated Values. Allerdings spricht man häufig auch von CSV, obwohl als Trennzeichen z. B. ein Semikolon genutzt wird. Den Vorteil durch Einsatz von Bibliotheken haben Sie natürlich analog bei ähnlichen Trennzeichen.

[5] Durch den Einsatz von statischen Imports in komplexeren Anwendungsklassen sind Abhängigkeiten manchmal schwieriger zu erkennen. Dieses Sprachfeature sollte daher mit Bedacht verwendet werden.

[6] Weiterführende Informationen zu den diversen Varianten der Erzeugung finden Sie unter `https://code.google.com/p/guava-libraries/wiki/RangesExplained`.

[7] Gleiches gilt auch für andere Zeichen, die im Dateisystem verboten sind.

[8] Für Windows findet man unter `http://gnuwin32.sourceforge.net/packages/grep.htm` eine freie Version.

[9] Beispielsweise weil dieser nicht verfügbar ist oder aber durch den Einsatz des Debuggers Timing-Probleme ungewollt entstehen oder aber verschwinden.

[10] Das sind String- bzw. Zahlenliterale im Sourcecode, etwa `-h=` oder der Wert `7`. Die Magie dieser Angaben liegt darin, dass die Intention normalerweise verborgen bleibt. Häufig ist es sinnvoller und besser verständlich, Konstanten zu definieren. Ausgenommen davon sind oftmals nur die Werte 0 und -1 als Startwert bzw. zur Berechnung der Abbruchbedingung in Schleifen.

[11] Man beachte hier den feinen textuellen, aber großen semantischen Unterschied zwischen `Options` ≡ Container und `Option` ≡ Parameter.

[12] Die JVM kann Objekte in eine Folge von Bytes transformieren, wenn diese das Interface `Serializable` erfüllen. Details dazu beschreibt Abschnitt 8.3.

[13] Das Verhalten kann manchmal problematisch sein, etwa wenn die Forderung nach »alles oder nichts« besteht.

7 Multithreading

[1] Vereinfachend wird hier kein Unterschied zwischen *n* CPUs und *n* Rechenkernen gemacht.

[2] Werte außerhalb dieses Wertebereichs führen zu einer `IllegalArgumentException`.

[3] Daher ist die Exception in den Signaturen der Methoden `sleep(long)` und `wait()` definiert.

[4] Wenn man den `catch`-Block derart implementiert, so kommt es zu einem unterschiedlichen Programmverhalten abhängig vom momentanen Thread-Zustand, wartend oder aktiv. Eine solche Inkonsistenz ist zu vermeiden. Hängt das Programmverhalten von der zeitlichen Abfolge der Anweisungen ab, so spricht man auch von ***Race Conditions***. Details beschreibt Abschnitt 7.2.1.

[5] Das bezieht sich auf die Semantik. Auf Ebene des Bytecodes gibt es deutliche Unterschiede. Schauen Sie es sich per `javap` an.

[6] Sofern alle Zugriffsmethoden dieses Synchronisationsobjekt nutzen.

[7] Dieser Abgleich ist auch unter dem Begriff ***Cache-Kohärenz*** bekannt.

[8] Durch das Interface kann dies nur gefordert, nicht aber sichergestellt werden.

[9] Die genaue Anzahl hängt von der verwendeten Stackgröße und dem zur Verfügung stehenden Hauptspeicher ab.

[10] Die Idee dazu basiert auf der API-Dokumentation zur Klasse `RecursiveTask`.

8 Fortgeschrittene Java-Themen

[1] `http://download.oracle.com/javase/tutorial/reflect/`

[2] Mithilfe einer Spezialisierung der abstrakten Klasse `SecurityManager` können verschiedene Berechtigungen erteilt oder verweigert werden, etwa Zugriffe auf Dateien. Existiert keine Berechtigung, so wird

eine `java.lang.SecurityException` ausgelöst.

[3] Statische Attribute werden nicht serialisiert, da sie keiner Objektinstanz zugeordnet sind.

[4] Dabei ist zu beachten, dass die Signatur exakt eingehalten wird, da diese Methoden per Reflection durch den Serialisierungsautomatismus ermittelt und aufgerufen werden.

[5] In der Praxis ist das häufig ein wenig komplizierter, aber vieles kann man auf wenige zu serialisierende Informationen zurückführen.

[6] Nicht als Listing gezeigt, aber analog zu der zuvor gezeigten `main()` -Methode realisiert.

[7] Daher wird in der Regel in den Klassen, die das Interface `Externalizable` erfüllen, ein Defaultkonstruktor angeboten.

[8] Wenn referenzierte Objekte unveränderlich sind, muss keine tiefe Kopie erstellt werden. Wenn ein referenziertes Objekt sehr schwergewichtig ist, kann auf eine Kopie in Ausnahmefällen verzichtet werden. Dieser Sachverhalt muss dann aber explizit im Javadoc-Kommentar erwähnt werden.

[9] In C++ muss zur Freigabe des von Objekten belegten Speichers explizit ein Destruktor aufgerufen werden.

[10] Aus Konsistenzgründen müssen alle Bereiche betrachtet werden, da sich die Speicherpositionen der Objekte durch die Garbage Collection ändern können. Es muss also eine Korrektur von Referenzen erfolgen, etwa für den Fall, dass Objekte aus der jungen Generation auf solche der alten Generation verweisen, die im Verlauf der Garbage Collection gerade im Speicher verschoben wurden. Die Referenzanpassungen sind für das Java-Programm und den Entwickler transparent.

9 Programmierung grafischer Benutzeroberflächen mit Swing

[1] Darunter versteht man eine Abfolge von Dialogen (oder Bildschirmmasken), die einen Benutzer durch eine bestimmte Sequenz von Verarbeitungsschritten führen. Die jeweiligen Seiten eines Schritts werden dabei in derselben Containerinstanz, z. B. einem `JDialog`, dargestellt.

[2] Falls Sie das Programm von der Konsole gestartet haben, sind dort keine Eingaben möglich.

[3] Oder wie bei Mac OS X in die Docking-Leiste minimiert werden. Auf keinen Fall ist es aber wünschenswert, das Fenster unsichtbar zu machen, ohne die Möglichkeit zu besitzen, später wieder mit dem Fenster interagieren zu können.

[4] Nur für extrem einfache Listener, die keinen Zugriff auf die Attribute der äußeren Klasse benötigen, ist das nicht der Fall.

[5] Hier gilt eine 0-basierte Positionsangabe und es gibt ein Leerzeichen im Text.

[6] Es gibt kein Package `java2d`. Stattdessen entstammen die Interfaces und Klassen dem Package `java.awt` sowie dem Subpackage `java.awt.geom`. Trotz des AWT in ihrem Namen sind diese nicht AWT, sondern Java 2D zugeordnet.

[7] Das erinnert etwas an die ersten Computergrafiken, die man mit der Programmiersprache Logo erstellen konnte. Dort bewegte man eine virtuelle Schildkröte, die den Zeichenstift symbolisierte, auch auf solchen Pfaden, die durch Kommandos bestimmt wurden.

[8] Ausnahmen bilden unter anderem `Border` und `JLabel`.

[9] Das Ganze folgt dem BEOBACHTER-Muster, das ich im Detail in Abschnitt 18.3.7 vorstelle.

[10] Diese Form der Speicherung kann zudem bei sehr umfangreichen Datenbeständen speicherintensiv sein und sich negativ auf die Performance auswirken. Als Lösung kann man das sogenannte ***Paging*** einsetzen, das nur einen kleinen Teil der Daten tatsächlich in den Speicher lädt (ähnlich wie bei Ergebnissen von Suchanfragen auf

Webseiten).

[11] Im Listing erkennt man einen erwähnenswerten Unterschied zu den vorherigen Beispielen: Der `JList<E>` wird ein `Person`-Array direkt an den Konstruktor übergeben. Dadurch wird nicht ein `DefaultListModel` erzeugt, sondern eine Modellinstanz basierend auf einem `AbstractListModel<E>`, die analog zu der zuvor gezeigten Klasse `StringListModel` arbeitet. Das so erstellte Modell erlaubt somit nachträglich keine Modifikationen!

[12] Die Verarbeitung von Zwischenzuständen kann beispielsweise für eine Vorschau eingesetzt werden oder aber um Aktionen beim Deselektieren von Einträgen vorzunehmen.

[13] Außer man ruft diese explizit aus einem beliebigen Thread selbst auf.

[14] Zwar kann man so für jede Zeile eine unterschiedliche Höhe festlegen – allerdings sollte man das vermeiden, da ansonsten das gesamte Erscheinungsbild sehr unruhig wird.

[15] Darüber hinaus kann man jeder Spalte explizit einen Renderer zuordnen und nicht nur typabhängig. Dazu muss man für das entsprechende `TableColumn`-Objekt mit dessen Methode `setCellRenderer(TableCellRenderer)` den gewünschten Renderer setzen.

[16] Genauere Daten über die Spaltentypen sind in der Basisklasse nicht verfügbar: Wie zuvor schon erwähnt, erfolgt in der Basisklasse keine Datenspeicherung.

[17] Allerdings wird immer nur ein Sortierpfeil dargestellt, und zwar in der Spalte, die durch das erste Element der Liste bestimmt ist.

[18] Dieses Verhalten bietet sich häufig an, damit man eine Selektion der Zeile vornehmen kann, ohne direkt den Editor zu aktivieren. Probieren Sie einfach mal einen anderen Wert aus.

10 Basiswissen Internationalisierung

[1] Jeweils eine für jede zu unterstützende Sprache und Region.

[2] Voraussetzung ist jedoch, dass die Property-Dateien im `CLASSPATH` liegen.

[3] Nur wenn man sich an diese Konvention bei der Benennung der Property-Dateien hält, ist ein zuverlässiges Einlesen durch die Methode `getBundle(String, Locale)` möglich. Weitere Informationen zur Arbeitsweise finden Sie in der API-Dokumentation zu dieser Methode.

[4] Zur besseren Lesbarkeit wurden die Methoden statisch importiert.

[5] In der Methode `compare(Object, Object)` werden beide Parameter direkt auf `String` gecastet. Korrekterweise müsste die Klasse `Collator` daher das auf die Klasse `String` typisierte Interface `Comparator<String>` erfüllen. Allgemeiner könnte man den Typ `CharSequence` statt `String` verwenden.

[6] In vielen Sprachen wird durch einen Akzent oder Umlaut kein wirklich eigenständiger Buchstabe beschrieben. In einigen Sprachen, etwa Dänisch, ist das anders. Dann wird ein Unterschied im Umlaut, z. B. ’ø’ statt ’o’, auch als primärer Unterschied gewertet.

[7] Häufig geschieht dies in Form von Property-Dateien für jede zu unterstützende Sprache und Region.

11 Lambda-Ausdrücke

[1] Außer natürlich über Copy-Paste, wodurch es dann aber schnell zu einem Wartungsalbtraum kommen kann. Ähnliche Probleme beschreibt BAD SMELL: UNVOLLSTÄNDIGE ÄNDERUNGEN NACH COPY-PASTE in Abchnitt 16.1.7.

12 Bulk Operations on Collections

[1]Leider wurde kein neues Schlüsselwort `foreach` eingeführt, sondern das Schlüsselwort `for` etwas missbraucht.

[2]Sie erleichtert lediglich die Schreibweise, stellt sogenannten syntaktischen Zucker dar und wird beim Kompilieren in eine externe Iteration mit Iterator umgewandelt.

[3]Wir könnten uns vorstellen, diese Zeilen entstammten einer Textdatei.

[4]Deswegen spricht man auch von »Terminal« Operations.

[5]Aufmerksamen Lesern fallen die hier genutzten Klassen bzw. Aufzählung `LocalDate` und `Month` und deren Konstanten, etwa `MAY` und `JULY`, auf. Diese sind neu im JDK 8 und werden später in Kapitel 13 genauer erläutert.

[6]`http://qaware.blogspot.ch/2013/09/lambdas-and-streams-in-jdk-8.html`

13 JSR-310: Date And Time API

[1]Zeiteinheiten mit variabler Länge, wie Monate, werden nicht unterstützt.

[2]Insbesondere interessiert uns dabei die Zeitzone, in der die Termine stattfinden, in der Regel nicht – mit Ausnahme von Telefonterminen, die man etwa mit Geschäftspartnern in Übersee hat.

[3]Dass dies problematisch ist, könnte man daran erkennen, dass diese Methoden statisch sind.

14 Einstieg JavaFX 8

[1]Allerdings besteht die Gefahr von Fehlkonfigurationen und Namensinkonsistenzen zwischen FXML und korrespondierendem Java-Code durch Tippfehler oder nach Umbenennungen.

[2]Ich habe lediglich die Größe des Fensters und die Referenz auf die CSS-Datei angepasst.

[3]Durch den Effekt allein wird das Bedienelement aber noch nicht deaktiviert. Die Deaktivierung muss selbst programmiert werden.

[4]Während es derartige Bedienelemente für Swing leider nicht im Standard gab, musste die Entwicklergemeinde für JavaFX bis zu dessen Version 8 darauf warten.

[5]Vereinfachend nach folgendem Modell: $Light_{out} = Light_{ambient} + Light_{diffus} + Light_{specular}$. Details finden Sie unter `http://de.wikipedia.org/wiki/Phong_Shading`.

[6]Diese finden Sie unter `http://docs.oracle.com/javase/8/javafx/api/javafx/scene/paint/PhongMaterial.html` und `http://docs.oracle.com/javafx/8/3d_graphics/materials.htm`.

[7]Geschieht dies nicht, so existiert immer eine Standardlichtquelle.

15 Weitere Änderungen in JDK 8

[1]Hier sieht man ein nützliches Beispiel für eine statische Methode in einem Interface. Es wäre aber ebenso gut möglich gewesen, dies durch eine separate Utility-Klasse bereitzustellen.

[2]Wobei ein »New« in einem Namen potenziell immer schwierig ist. In diesem Fall existiert das NIO bereits

seit Java 1.4.

16 Bad Smells

[1]Lazy Initialization beschreibt die Technik, ein Attribut nicht direkt zu initialisieren, sondern zunächst mit `null` zu belegen. Erst beim ersten Zugriff erfolgt dann eine Initialisierung. Dies kann man bei optionalen Programmbausteinen nutzen, um die Konstruktion komplexer Objekte zu vermeiden. Kapitel 22 beschreibt diese Technik im Detail.

[2]Tatsächlich kann man mit etwas Mühe einen sogenannten Shut-down-Hook realisieren, der Aufräumarbeiten vornimmt, wie dies im nachfolgenden Praxistipp angedeutet wird.

[3]Als letztes Sicherheitsnetz besitzen einige I/O-Klassen, im Speziellen die Klassen `FileInputStream` und `FileOutputStream`, eine überschriebene `finalize()` -Methode, die einen Aufruf an `close()` ausführt. Wie bereits bekannt, wird `finalize()` jedoch möglicherweise erst zu einem viel späteren Zeitpunkt im Programmablauf aufgerufen.

17 Refactorings

[1]`http://commons.apache.org/`

[2]`https://code.google.com/p/guava-libraries/`

[3]Normalerweise ist das nicht notwendig, weil Optimierungen auf dieser feingranularen Ebene durch die JVM automatisch erfolgen (vgl. Kapitel 22).

[4]Allerdings besitzen moderne IDEs mittlerweile eine sehr ausgefeilte farbliche Darstellung von Programmelementen, wodurch eine gute visuelle Trennung möglich wird, ohne dass man dazu `this` nutzen muss.

[5]Dabei wird auf `null`-Prüfungen verzichtet, weil es sich um eine interne Hilfsklasse handelt und wir uns hier auf die Refactoring-Schritte konzentrieren wollen.

[6]Bis JDK 8 ist das vermutlich die beste Wahl, wenn man Datumsarithmetik ausführen muss. Online verfügbar unter `http://www.joda.org/joda-time/`.

[7]Oftmals bietet die in Eclipse integrierte QUICK-FIX-Funktionalität, die man durch CTRL+1 aufruft, die Möglichkeit, die ersten drei Refactorings direkt auszuführen.

[8]Bitte beachten Sie, dass diese Umordnung hier zu keiner Verhaltensänderung führt, aber dass das in der Praxis z. B. wegen möglicherweise versteckter Seiteneffekte nicht immer so ist.

[9]Dabei können die Tastaturkürzel ALT+SHIFT+LEFT/RIGHT/DOWN hilfreich sein.

[10]Allerdings sollten wir uns dabei bewusst sein, dass sich die Verarbeitungsreihenfolge durch das INLINE ändern kann.

[11]Insbesondere gilt dies für das Refactoring CHANGE METHOD SIGNATURE, um Parameter umzuordnen, deren Typ zu ändern oder neue Parameter einzufügen, wodurch sich schnell Verhalten ändert. Ebenso kann ein Inlining problematisch sein, weil dadurch die Abarbeitungsreihenfolge leicht geändert wird. Man spricht bei dem Problem auch von »temporal coupling«.

[12]Das kann man ganz hervorragend auch für andere Rückgabetypen außer `boolean` machen, solange die Wertemenge nicht zu groß wird.

[13]Potenziell eine Verletzung des Single Responsibility Principle (SRP) (vgl. Abschnitt 3.5.3).

[14]Details finden Sie in Martin Fowlers Buch »Refactoring: Improving the Design of Existing Code« [27].

[15] Das gilt, sofern keine Zugriffe per Reflection erfolgen.

[16] Diese Aussage gilt allerdings nur unter der Voraussetzung, dass man allen Sourcecode vorliegen hat, der auf die Klasse zugreift.

[17] `http://www.ucdetector.org/`

[18] Reflection wird unter anderem von Dependency-Injection-Frameworks genutzt. In solchen Kontexten kommt es dann erst zur Laufzeit zu Problemen, wenn die aufzurufenden Methoden nicht die erwartete und benötigte Sichtbarkeit aufweisen.

[19] Das ist etwa bei Remote Calls möglich.

[20] Eine fehlende Auswertung von Rückgabewerten wurde in Abschnitt 16.1.6 als BAD SMELL: UNVOLLSTÄNDIGE BETRACHTUNG ALLER ALTERNATIVEN diskutiert.

[21] Je höher die Anzahl der `if`-Abfragen oder Verschachtelungen, desto eher sind mehrere `return`-Anweisungen zu vermeiden, weil sie dann den Programmfluss meistens schwieriger erkennbar machen. Für Shortcut-Returns am Methodenanfang oder kurze Methoden bis etwa 20 Zeilen können mehrere `return`-Anweisungen aber sogar für mehr Klarheit sorgen.

[22] Statische Methoden werden in der UML unterstrichen.

[23] Natürlich kann man nicht ausschließen, dass die Werte beim Methodenaufruf auch falsch belegt sind. Die Fehler werden jedoch schneller klar.

[24] Oder: Wenn man an der Grenze schon Ausweise und Waren kontrolliert, muss man das nicht im ganzen Land immer und überall wieder tun.

18 Entwurfsmuster

[1] Englische Schreibweise ist in Klammern angegeben, falls sie von der deutschen abweicht.

[2] MVC ist im strengen Sinn kein Entwurfsmuster, sondern ein Architekturmuster.

[3] Verallgemeinerung von Magic Numbers: Literale primitiver Typen, Strings und `null`-Referenzen.

[4] Hintergründe finden Sie online unter `http://www.ibm.com/developerworks/java/library/j-dcl.html`.

[5] Ein sogenannter Klassenlader vom Typ `ClassLoader` lädt benötigte Klassenbeschreibungen bei Bedarf in den Speicher und entscheidet vorher, ob eine Klasse überhaupt geladen werden darf. Zudem stellt er sicher, dass Anwendungen keine Systemklassen überschreiben.

[6] Dies ist die Namensgebung der Methoden im Interface `java.util.Iterator<E>` bzw. `java.util.Enumeration<E>`.

[7] Eine derartige Umsetzung des BEOBACHTER-Musters wäre nur dann sinnvoll gewesen, wenn Java Mehrfachvererbung unterstützen würde. Aber selbst in diesem Fall wäre die Einbindung über Vererbung kein guter Stil.

19 Programmierstil und Coding Conventions

[1] `http://www.oracle.com/technetwork/java/codeconv-138413.html`

[2] `http://www.ambysoft.com/essays/javaCodingStandards.html`

[3] Sofern nicht zuvor eine Zuweisung mit `null` erfolgt.

[4]Wenn es auf eine Reihenfolge bei der Benachrichtigung ankommt, wäre eine Liste besser geeignet.

[5]Hierunter fallen auch die von IDEs automatisch generierten Javadoc-Vorlagen ohne Inhalt.

20 Unit Tests

[1]Wenn man penibel ist, entspricht dies laut ISTQB-Glossar (International Software Testing Qualifications Board) der Fehlerwirkung. Nachfolgend unterscheide ich aber der Einfachheit halber nicht so genau.

[2]`https://www.cs.utexas.edu/ÊWD/transcriptions/EWD03xx/EWD340.html`

[3]Neuere Vorgehensmodelle, wie etwa Scrum, sehen das Testen als Aufgabe des Entwicklungsteams, dem auch Tester angehören, an.

[4]Das klingt zunächst abwegig, tritt aber durchaus manchmal auf – aus der Natur kennt man es: Wenn zwei Wellen gegeneinander verschoben sind, können sich Wellentäler und -berge auslöschen.

[5]In manchen regulierten Domänen wie z. B. bei Eisenbahnsystemen ist genau definiert, wann der Entwickler selbst testen darf bzw. wann dies ein Tester machen muss.

[6]Niemand wird das bewusst machen, aber manchmal kommt es bei komplexeren Änderungen doch zu solchen Problemen – gerade bei unübersichtlichem oder unverständlichem Sourcecode.

[7]Eine absolute Sicherheit hätte man nur dann, wenn eine Testabdeckung von 100 % vorliegen würde und die Tests absolut fehlerfrei und vollständig wären. Selbst diese Aussage ist nicht ganz korrekt: Für jedes komplexere Programm garantiert eine Testabdeckung von 100 % keine absolute Sicherheit (vgl. Abschnitt 20.8.4). Im Besonderen gilt dies beim Einsatz von Multithreading.

[8]Damit ist ohne Weiteres auch das Schreiben von Integrations- und Systemtests möglich.

[9]Die Erfahrung zeigt, dass Programmiersünden häufig für längere Zeit im Sourcecode verbleiben und sich später nur mühevoll korrigieren lassen. Obwohl TDD im dritten Schritt dafür sorgen will, diese umgehend zu eliminieren, wird dieser notwendige Korrekturschritt schnell mal übergangen.

[10]Man kann darüber diskutieren, ob eine `NullPointerException` oder eine `IllegalArgumentException` eine bessere Wahl darstellt.

[11]`http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html`

[12]Wollte man die Effekte durch Fehler im Betriebssystem oder der Ablaufumgebung (JVM, Application Server, Webcontainer) sicher ausschließen, so müsste man tatsächlich alle Werte durchprüfen. Das wäre aber schon etwas neurotisch.

[13]Bedenken Sie aber: Mit Tests kann man Fehlerfreiheit nicht sicherstellen, sondern nur Fehler aufdecken.

[14]Bitte beachten Sie, dass man verschiedenste abweichende Definitionen und Verwendungen der obigen Begriffe für Test-Doubles findet.

[15]`http://easymock.org/Downloads.html`

[16]`https://code.google.com/p/mockito/`

[17]http://www.rbcs-us.com/documents/Why-Most-Unit-Testing-is-Waste.pdf

[18]`https://github.com/Pragmatists/junitparams`

[19]Entgegen dem Namen basiert es nicht auf Emma, sondern auf JaCoCo als Checker und ist JDK-8-kompatibel.

[20]Frei unter `http://code.google.com/p/hamcrest/` herunterladbar.

[21] Detaillierte Informationen findet man unter http://moreunit.sourceforge.net/.

[22] Das Eclipse-Plugin kann unter http://infinitest.github.io bezogen werden.

[23] Zu einem ungünstigen Zeitpunkt kann man auch einmal einen Fehlalarm erhalten.

[24] Es kann unter http://cobertura.sourceforge.net/ heruntergeladen werden.

[25] http://www.ibm.com/developerworks/java/library/j-cq01316/

[26] Details finden Sie im Internet unter http://www.eclemma.org/index.html.

22 Optimierungen

[1] In diesem Buch legen wir den Fokus auf Datenstrukturen, da es für die Wahl passender Algorithmen diverse Literatur gibt. Daher wird hier lediglich der eine oder andere Tipp gegeben, um Algorithmusprobleme zu lösen.

[2] Die tatsächlichen Zahlen sind nicht fix, sondern schwanken etwas: 90 zu 10 oder 70 zu 30 passen auch sehr häufig.

[3] Die Auflösung der Zeitmessung liegt in Windows-Systemen nur etwa im Bereich von 10 bis 20 ms. Ein Zahlenwert von 0 ms ist damit durchaus mit einer Angabe von 15 ms vergleichbar.

[4] Die Bedeutung der Notation wird auf der nächsten Seite mit der Vorstellung von Beispielen für weitere Komplexitätsklassen verständlicher. Eine weiterführende Darstellung finden Sie unter http://www.linux-related.de/index.html?/coding/o-notation.htm.

[5] Dabei wird die Zeitkomplexität $O(2^n)$ nicht dargestellt, weil deren Wachstum zu stark ist, um es sinnvoll ohne den Einsatz von 10er-Potenzen auszudrücken.

[6] Werfen Sie einen Blick in die Sourcen des JDK, um solche internen Details kennenzulernen.

[7] Es werden drei Referenzen (Daten, Vorgänger und Nachfolger) statt einer in einer ArrayList<E> gespeichert. Dadurch wird im Vergleich zur ArrayList<E> etwa der dreifache Speicher für die Verwaltung verbraucht, allerdings nicht zusammenhängend.

[8] Die Idee zur Darstellung der Lazy Initialization anhand dieser Klassenstruktur stammt aus einem Vortrag von Peter Haggar auf der Java One 2001. Ich habe das gesamte Beispiel aber bezüglich Problembeschreibung und Lösungsmöglichkeiten stark erweitert.

[9] http://www.oracle.com/technetwork/java/javase/tech/g1-introjsp-135488.html

[10] http://commons.apache.org/

A Einführung in die UML

[1] Daher auch der Name UML, was auf Deutsch so viel wie »vereinheitlichte Modellierungssprache« bedeutet.

[2] In vielen Dokumentationen von Oracle wird leider nicht die UML verwendet. Dies macht es immer wieder kompliziert, die Beispiele auf einen Blick nachzuvollziehen.

[3] Mit Version 2.2 wurde mit dem Profildiagramm noch ein siebtes Strukturdiagramm eingeführt, was wir hier aber nicht weiter betrachten wollen.

[4] Der Instanzname ist optional und kann bei nicht benannten (anonymen) Objekten entfallen.

[5] Klassen können auch Bestandteil von Sequenzdiagrammen sein, wenn es sich um Utility-Klassen mit statischen Methoden handelt.

[6]Allerdings ist dies streng genommen bei bedingten Lebenslinien nicht ganz korrekt, da aus Platzgründen dann Alternativen untereinander dargestellt werden, obwohl sie eigentlich zeitgleich ablaufen.

[7]Bei geringem Detaillierungsgrad in frühen Entwurfsphasen können dies auch nur Nachrichten sein, die exemplarisch den gewünschten Verlauf der Kommunikation andeuten.

[8]Bei entsprechendem Detaillierungsgrad muss wieder jede Verbindung im Kommunikationsdiagramm durch eine Assoziation bzw. eine Dependency im Klassendiagramm abgebildet werden.

B Überblick über den Softwareentwicklungsprozess

[1]`http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf`

[2]Die Erfahrung zeigt, dass Programmiersünden häufig für längere Zeit im Sourcecode verbleiben und sich später nur mühevoll korrigieren lassen. Obwohl TDD im dritten Schritt dafür sorgen will, diese umgehend zu eliminieren, wird dieser notwendige Korrekturschritt schnell mal übergangen.

C Grundlagen zur Java Virtual Machine

[1]Die Ausführungsmodi der JVM lassen sich über JVM-Aufrufparameter steuern: `-Xint` aktiviert den Interpreter-Modus. Ohne diese Angabe erfolgt eine Ausführung im Hotspot-Modus.

[2]JDK 7 beinhaltet einige derart extreme Optimierungen, die sogar zum Teil Probleme bereiten: In namhaften Projekten wie Apache Lucene Core und Solr beobachtet man JVMAbstürze oder auch Berechnungsfehler. Diese Probleme entstehen dadurch, dass mitunter zu forsch optimiert wird und die erzielten Resultate nicht immer fehlerfrei sind (siehe dazu `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7044738`). Einige Fehler wurden mit dem ersten Update vom JDK 7 behoben. Andere Optimierungen sind immer noch mit Vorsicht zu genießen, etwa die Aufrufoption `-XX:AggressiveOpts`.