Springer-Lehrbuch

Robert Braun · Werner Esswein
Steffen Greiffenberg

Einführung in die Programmierung

Grundlagen, Java, UML

Mit 51 Abbildungen und 16 Tabellen

Springer

Dipl.-Wirtsch.-Ing. Robert Braun
Professor Dr. Werner Esswein

TU Dresden
Fakultät Wirtschaftswissenschaften
Lehrstuhl für Wirtschaftsinformatik,
insbesondere Systementwicklung
Münchner Platz 3
01062 Dresden

robert.braun@tu-dresden.de
werner.esswein@tu-dresden.de

Dr. Steffen Greiffenberg
semture GmbH
Gostritzer Straße 61–63
01217 Dresden

steffen.greiffenberg@semture.de

ISBN-10 3-540-32855-6 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-32855-1 Springer Berlin Heidelberg New York

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.

Springer ist ein Unternehmen von Springer Science+Business Media

springer.de

Printed in Germany

Umschlaggestaltung: Design & Production, Heidelberg

SPIN 11684985 88/3153-5 4 3 2 1 0 – Gedruckt auf säurefreiem Papier

Vorwort

Dieses Buch basiert auf der Lehrveranstaltung „Programmierung“ im Grundstudium an der Wirtschaftswissenschaftlichen Fakultät der TU Dresden. Es richtet sich daher primär an zukünftige Sozial- und Wirtschaftswissenschaftler, für die die Erlernung einer Programmiersprache zur (universitären) Grundausbildung gehört. Es vermittelt ein grundlegendes Verständnis für die Informationstechnologie, indem deren Funktionsweise aufgezeigt wird. Die Grundlagen von Programmiersprachen im Allgemeinen und von Java 5.0 im Besonderen werden zunächst in Grundzügen vorgestellt und anhand der Gestaltung von grafischen Benutzungsoberflächen praktisch erprobt. Mit Hilfe der Unified Modeling Language (UML 2.0) wird folgend die Nutzung von einfachen Algorithmen und Mustern bei der Gestaltung von Softwareentwürfen demonstriert, die im Anschluss mittels Java implementiert werden. Die Programmierung mit Java wird damit als die letzte Phase eines modellbasierten Vorgehens bei der Entwicklung von Anwendungssystemen aufgefasst.

Wir schulden allen Dank, die in irgendeiner Form an der Lehrveranstaltung und dem Zustandekommen des Buches mitgewirkt haben. Unser besonderer Dank gilt dabei Dipl.-Ing. B. Oestereich für die freundliche Genehmigung der Adaption seines Kuh-Beispiels, welches den Rahmen für viele Beschreibungen im Buch liefert, sowie Dr. A. Dietzsch für die Er- und Bereitstellung der zugehörigen Grafiken. Dipl.-Wirt.-Inf. A. Gehlert gebührt Dank für seine Beiträge zu dem Programmbeispiel und zahlreicher LaTeX-Tipps sowie Frau L. Gerstenberger für das Korrekturlesen des Skriptes, und nicht zuletzt sei allen Studenten gedankt für das Erdulden sämtlicher Skript-Rohfassungen, deren kritische Durchsicht und viele Verbesserungsvorschläge. Ein besonderer

Dank gilt darüber hinaus dem Springer-Verlag für die ausgezeichnete Zusammenarbeit und die Erfüllung sämtlicher LaTeX-Wünsche.

Das Programmbeispiel, Hinweise und Lösungsvorschläge zu den Übungsaufgaben sowie weitere Ressourcen sind unter `http://wiseweb.wiwi.tu-dresden.de/javabuch` im Internet abrufbar.

Auf alle Anregungen und Verbesserungsvorschläge an unsere Postanschrift oder `javabuch@wise.wiwi.tu-dresden.de` freuen wir uns sehr.

Dresden,
Februar 2006

Robert Braun
Werner Esswein
Steffen Greiffenberg

Inhaltsverzeichnis

1

Einleitung

> *„There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is a second theory that states that this has already happened.“*
>
> *Douglas Adams*
> *„The Hitchhiker's Guide to the Galaxy“*

Ein grundlegendes Verständnis für die Informationstechnologie und deren Funktionsweise wird aufgrund ihrer Omnipräsenz im Alltag heutzutage nicht nur Informatikern abverlangt. Entsprechend findet sich eine Lehrveranstaltung wie „Programmierung“ oder „Programmiersprache“ im Grundstudium auch vieler sozial- und wirtschaftswissenschaftlichen Fakultäten, nicht nur an deutschen Universitäten. Der Fokus liegt dabei, wie bereits erwähnt, auf der Schaffung eines grundlegenden Verständnisses und unterscheidet daher solche Lehrveranstaltungen von Vorlesungen mit überwiegend technisch (vor-)geprägter Zuhörerschaft. Dies spiegelt sich auch in diesem Buch wider, welches bewusst diesen Leserkreis adressiert. Es gliedert sich in folgende Kapitel:

Kapitel 2 Dieses Kapitel thematisiert die Grundlagen von Programmiersprachen im Allgemeinen und definiert durch die Einführung in das objektorientierte Paradigma das in diesem Buch verwendete Begriffssystem. In der Natur von Grundlagen – und die Grundlage einer Programmiersprache wird nun mal von sprachwissenschaftlichen Erkenntnissen gebildet – liegt es leider, manchmal eine gewisse Einstiegsbarriere in eine Thematik darzustellen. Dem primär an Java interessierten Leser wird es daher empfohlen, im Abschnitt 2.5 mit dem Lesen zu beginnen und die Abschnitte 2.1 bis 2.4 später nachzuholen.

Kapitel 3 Mit dem JAVA Development Kit (JDK) wird die Programmierumgebung vorgestellt, deren Programmiersprache JAVA auf dem erläuterten objektorientierten Paradigma beruht.

Kapitel 4 Mit der Vorstellung der Syntax und der Semantik von JAVA wird eine Einführung in die grundlegenden Elemente der Programmierung gegeben.

Kapitel 5 Dieses Kapitel zeigt die Möglichkeiten der Entwicklung grafischer Oberflächen unter Verwendung des JDK. Die Gestaltung mittels Layoutmanager und die Ereignisbehandlung bilden den Kern des Kapitels.

Kapitel 6 Das letzte Kapitel erläutert die Möglichkeiten der Wiederverwendung innerhalb der Softwareentwicklung. Dabei wird die Programmierung mit JAVA in die letzte Phase eines modellbasierten Vorgehens zur Entwicklung von Anwendungssystemen eingeordnet. Die Schwerpunkte liegen hier auf der Vorstellung der für das Buch relevanten Teile der verwendeten Modellierungssprache UML sowie in der Anwendung der UML zur Demonstration, wie einfache Algorithmen und Muster zur Gestaltung von Softwareentwürfen genutzt werden können und wie diese Entwürfe anschließend mittels JAVA implementiert werden.

Es wird mit diesem Buch nicht der Versuch unternommen, alle Aspekte der Programmiersprache JAVA oder der Modellierungssprache UML zu beleuchten oder die Klassenbibliothek im JDK vorzustellen. Vielmehr liegt der Fokus darauf, ein grundlegendes Verständnis für die Informationstechnologie zu vermitteln, indem deren Funktionsweise aufgezeigt wird.

Hervorhebungen von Definitionen erfolgen *kursiv*. JAVA-Sprachelemente im laufenden Text werden in `Schreibmaschine` gesetzt und ein längerer Abschnitt von Quelltext wird wie folgt eingerahmt:

```
public void einLaengererAbschnittQuelltext() { }
```

2

Grundlagen der objektorientierten Programmierung

> *„Ich meine, wenn man wirklich etwas begreifen möchte, ist es der beste Weg, dass man versucht, es jemand anderem zu erklären. Das zwingt einen, sich im eigenen Kopf darüber klar zu werden. Und je langsamer und dümmer der Schüler ist, desto weiter muss man das Problem in immer einfachere Gedanken unterteilen. Und das ist das eigentliche Grundprinzip des Programmierens.“*
>
> *Richard in Douglas Adams'*
> *„Dirk Gently's holistische Detektei“*

2.1 Klassifizierung und Spezifikationsmittel von Sprachen

Folgt man der Zeichentheorie (Semiotik), entspricht Sprache einem System von Zeichen (vgl. [Zema92], S. 72). Durch die Klassifikation von Zeichen lassen sich die daraus gebildeten Sprachen unterteilen. Man unterscheidet:

- *ikonische Zeichen* mit anschaulich bildhafter Darstellung des Originals und
- *symbolische Zeichen* ohne direkten Bezug zum Original. Letztere werden wiederum in kurz- und langsymbolisch unterteilt.
  - Auf Seiten der *kurzsymbolischen* (oder auch künstlichen) Sprachen, d. h. Systemen aus kurzsymbolischen Zeichen, werden
    - *formale* (d. h. mathematische) und
    - *nicht formale* Sprachen unterschieden.
  - Zu den *langsymbolischen* (oder auch natürlichen) Sprachen, d. h. Systemen aus langsymbolischen Zeichen, werden insbesondere Fachsprachen und die Umgangssprache gezählt.

Die Struktur eines einzelnen (Sprach-) Zeichens, also beispielsweise eines Wortes, lässt sich nach Eco, wie in Abbildung 2.1 dargestellt, beschreiben.

Das *Objekt* stellt das Element der realen oder fiktiven Welt dar, das durch ein (Sprach-) *Zeichen* im wahrsten Sinne „bezeichnet“ werden

soll.[1] Der *Begriff* bildet die Brücke zwischen dem Objekt und dem (Sprach-) Zeichen. Er stellt die (gedanklichen) Vorstellungen dar, die man über dieses Objekt besitzt. Diese Vorstellungen ergeben sich aus der (definierten) Bedeutung des Zeichens.[2]

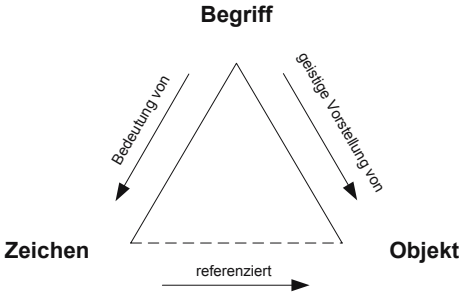


Abb. 2.1. Zeichenstruktur – Semiotisches Dreieck (in Anlehnung an [Eco02], S. 28)

Im Sprachgebrauch kann die Bezeichnung eines Objektes, d. h. man spricht mit jemandem über etwas, immer nur über den Begriff erfolgen, da mit einem (Sprach-) Zeichen nicht direkt über ein Objekt gesprochen werden kann, ohne dass eine Vorstellung darüber existiert, was überhaupt bezeichnet werden soll.[3] Die Bildung einer Vorstellung darüber, was bezeichnet werden soll, bedingt wiederum, dass eine Bedeutung für das (Sprach-) Zeichen festgelegt wurde, aus der sich diese ableiten lässt.

Die Einbeziehung des Spracherstellers bzw. -nutzers komplettiert das semiotische Dreieck zum sog. semiotischen Viereck (vgl. Abbildung 2.2).

Die Beziehungen zwischen Sprachersteller bzw. -nutzer und dem (Sprach-) Zeichen wird durch die *Pragmatik* der Sprache ausgedrückt. Sie legt fest, was derjenige, der das Zeichen benutzt bzw. entwickelt, damit meinen kann bzw. meint. Damit bestimmt sie das Ziel einer sprachlichen Handlung. Aus dieser Bedeutung der Pragmatik ergibt sich

[1] Fiktiven Welt deshalb, weil beispielsweise ein Einhorn real nicht existiert, es jedoch sowohl ein Wort als auch Bilder (im semiotischen Sinne ikonische Zeichen) von ihm existieren.

[2] Im herkömmlichen Sprachgebrauch werden häufig die Worte *„Begriff“* und *„Wort“* synonym verwendet. Es lässt sich jedoch merken: Ein Begriff umfasst all das, „. . . was gleichbleibt, wenn sich die verwendeten Wörter ändern.“ (vgl. Seiffert in [Schü98], S. 122) Allerdings wird sich in diesem Buch, im Hinblick auf dessen Verständlichkeit, diesem herkömmlichen Sprachgebrauch gebeugt.

[3] In diesem Falle würde man einfach nicht wissen, worüber gesprochen wird.

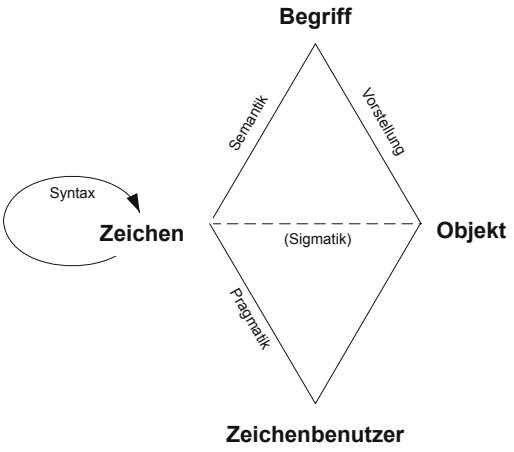


Abb. 2.2. Semiotisches Viereck (in Anlehnung an [WaHa97], S. 43f)

die Unmöglichkeit, Pragmatik als Mittel zur Spezifikation einer Sprache hinzuziehen zu können, da das Ziel einer sprachlichen Handlung von Zeichenbenutzer zu Zeichenbenutzer nicht ohne weiteres einfach als identisch unterstellt werden kann. Als Mittel zur Sprachspezifikation können daher nur Syntax und Semantik herangezogen werden (vgl. Abbildung 2.3).

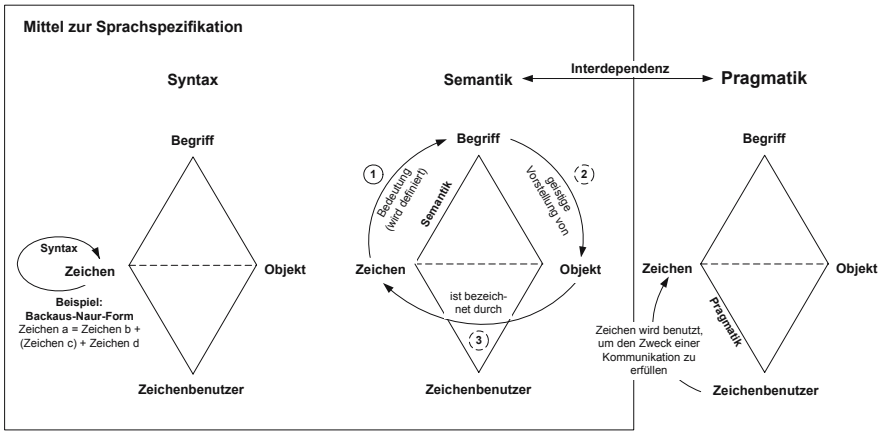


Abb. 2.3. Auf Basis der Semiotik abgeleitete Mittel zur Spezifikation einer Sprache

Die *Syntax* behandelt die Kombinierbarkeit der Zeichen ohne Rücksicht auf ihre spezielle Bedeutung oder ihre Beziehung zur Umwelt. In der Grammatik der Umgangssprache ist sie die Lehre vom Satzbau. In der Programmierung bildet sie die Vorschriften zum korrekten Aufbau von Programmen ab und legt die Grundlage dafür, dass ein Computer Programme verstehen kann.

Die *Semantik* behandelt die Bedeutung der Zeichen in Bezug auf die Objekte, auf die sie angewendet werden. Sie regelt die Wirkung einzelner oder zusammengesetzter Sprachelemente, also von Wörtern und Sätzen, und die Bedeutungsunterschiede, die sich aus dem Zusammenhang ergeben. In der Wirtschaftsinformatik (oder allgemeiner in den Wirtschaftswissenschaften) umfasst sie im weitesten Sinne die Sinnhaftigkeit von Begriffen oder Darstellungen.

Semantik und Pragmatik hängen wechselseitig voneinander ab. Was eine sprachliche Äußerung bedeutet, ist an den Rahmen dessen gebunden, was der Zeichenbenutzer mit ihr meint (Sprachkonvention), und was der Zeichenbenutzer mit einer sprachlichen Äußerung meinen (mitteilen) kann, lernt er über die Erfahrung, was die Äußerung in dieser Sprache bedeutet (vgl. [Bra$^+$99], S. 2).

In der Wirtschaftsinformatik wird zusätzlich der Begriff der *semiformalen* Sprache verwendet. So eine Sprache besitzt eine analog zu formalen Sprachen präzise formulierte Syntax, die Definition der Semantik wird jedoch vernachlässigt. Dieser Begriff dient vor allem der Abgrenzung nicht formaler Modellierungssprachen von langsymbolischen Sprachen wie der Umgangssprache. Die in Kapitel 6.2 vorgestellte Unified Modeling Language (UML) ist ein Beispiel für so eine Modellierungssprache.

2.2 Programmiersprache und Programm

Eine Programmiersprache stellt eine formale Sprache dar und dient wie natürliche Sprache der Kommunikation. Herkömmliche Computer sind leider selbst nach ihrer mittlerweile über 70-jährigen Geschichte immer noch nicht in der Lage, die menschliche Sprache zu verstehen. Ihr Umgang ist auf der untersten Ebene auf die Verarbeitung von `0` und `1` beschränkt. Um diese Maschinen trotzdem steuern zu können, ist es notwendig, ihnen klar verständliche Befehle (imperativ) oder klare Verhaltensregeln (deklarativ) zu geben. Allgemein kann definiert werden:

- Eine Programmiersprache ist eine formale Sprache zur Kommunikation von Mensch und Rechnersystem.

- Eine Programmiersprache dient zudem der Kommunikation zwischen Menschen.
- Die Spezifikation einer Programmiersprache muss festlegen, welche Zeichenfolgen berechenbare Funktionen beschreiben, unter den Restriktionen, dass diese von einem Rechner verarbeitet werden können (vgl. [Schn81], S. 14) und ihre Bedeutungen festgelegt sind.

Folgt man diesen Definitionen, so sind Programme:

- in einer Programmiersprache formulierte Sätze,
- die Beschreibung der Anforderungen, die ein Anwender an ein Anwendungssystem stellt,
- Anweisungen von Menschen für Maschinen.

2.2.1 Entwicklungsgeschichte von Programmiersprachen

Die Entwicklungsgeschichte der Programmiersprachen wird in fünf Generationen unterteilt:

1. *Generation - Maschinensprachen:* Die Programme werden in der Regel binär kodiert, d. h. ein Programmierer weiß, dass der Befehl „Addition von zwei Zahlen“ vom Rechner als `01011001` verarbeitet wird. Diese Sprache ist nur begrenzt von Menschen lesbar und führt zu erheblichen Schwierigkeiten bei der Wartung und Fehlerbeseitigung.
2. *Generation - Assemblersprachen:* Die in der Maschinensprache verwendeten Zahlenfolgen wurden durch lesbare Abkürzungen ersetzt. Ein Assembler übernimmt die Übersetzung in den Maschinencode. Anders als bei den folgenden Generationen existiert für den Quelltext genau eine Übersetzung in die Maschinensprache und umgekehrt. Die Nachteile der Lesbarkeit wurden also teilweise beseitigt, ohne die Vorteile der hohen Ausführungsgeschwindigkeit und des geringen Speicherbedarfs der Programme aufzuheben. Jedoch gilt auch hier, dass für eine Änderung der Hardware, also des ausführenden Computers, auch die Software verändert werden muss.
3. *Generation - Höhere Programmiersprachen:* In diese Gruppe lassen sich die prozedural problemorientierten und die objektorientierten Sprachen einordnen. Diese werden im nächsten Kapitel näher vorgestellt. Ab dieser Generation müssen die Programme zunächst in eine maschinenlesbare Form (Nullen und Einsen) übersetzt werden, bevor sie vom Rechner ausgeführt werden können. Die Werkzeuge für die Übersetzung werden als *Compiler* bzw. *Interpreter* bezeichnet, wobei ein Compiler vor der Ausführung einer Software deren

gesamten Quelltext und der Interpreter während der Ausführung die benötigten Teile übersetzt.

4. *Generation - Deklarative Sprachen:* Die Unterteilung der 4. und 5. Generation ist nicht allgemein anerkannt, jedoch wird zumeist eine Sprache, deren Fokus nicht auf der Beschreibung von Befehlsfolgen, sondern auf der Beschreibung des gewünschten Ergebnisses basiert, als deklarativ bezeichnet und der 4. Generation zugeordnet. Beispiele sind Sprachen zur Kommunikation mit Datenbanken (SQL oder ADABAS).
5. *Generation - Wissensbasierte Sprachen:* Wissensbasierte Sprachen bedienen sich der Beschreibung von Beziehungen zwischen komplexen Bedingungen und einer Konklusion in Form von Regeln. Einen wichtigen Beitrag leisten diese Sprachen auf dem Gebiet der Künstlichen Intelligenz (KI). Typische Vertreter sind PROLOG oder LISP.

Exkurs: Entwicklungsausblick

Ein weiterer Schritt in der Entwicklung von Programmiersprachen wurde mit der Konzeption von *Quantenprogrammiersprachen* getan. Diese sind jedoch nicht mehr für Computer im herkömmlichen Sinne gedacht (Verarbeitung von `0` und `1`), sondern für sogenannte *Quantenrechner*, welche die Erkenntnis des Quantenparallelismus der Quantentheorie ausnutzen (vgl. [Rüdi03], S. 406). Dabei werden auch „. . . durch Bibliotheken entsprechend erweiterte konventionelle Sprachen [..] dieser Kategorie zugerechnet ..“ ([Rüdi03], S. 407).

Ein regelrechter Forschungsboom um Quantenrechner ist seit Mitte der 90er Jahre des letzten Jahrhunderts durch die Entdeckung eines Algorithmus durch SHOR entstanden (heute Shor-Algorithmus genannt), durch den ein hinreichend großer Quantenrechner die Primfaktorzerlegung einer natürlichen Zahl in bezüglich der Stellenzahl polynominaler Zeit durchführen könnte, wohingegen die derzeit besten Verfahren (Siebverfahren) eine exponentielle Zeitkomplexität besitzen. Aufgrund der Tatsache jedoch, dass solche großen Quantenrechner, d. h. mit entsprechend großer Anzahl an *qubits*[4], nach vorherrschender Meinung auf absehbare Zeit nicht zur Verfügung stehen, stellt sich die Frage, warum sich zur Zeit mit diesen Programmiersprachen beschäftigt wird. Grund dafür stellt die beschriebene Eigenschaft von Programmiersprachen dar, auch als Kommunikationsmittel zwischen Menschen zu dienen. So erscheinen viele altbekannte Phänomene der Quantenphysik in neuem Licht, wenn sie mittels einer Quantenprogrammiersprache formuliert werden. Vor allem bisher bekannte Algorithmen aus diesem Bereich

[4] qubits ≡ Quantenbits, das Äquivalent zu *bits* in herkömmlichen Computern.

können wesentlich genauer und detaillierter formuliert werden, das hat insbesondere auch einen didaktischen Wert (vgl. [Rüdi03], S. 407).

Da sich der Gegenstandsbereich des Buches jedoch auf herkömmliche Computer beschränkt, werden Quantenprogrammiersprachen von den folgenden Betrachtungen explizit ausgeschlossen.

2.2.2 Klassifizierung von Programmiersprachen

Nach der Art, wie mit Hilfe einer Programmiersprache ein Problem gelöst wird, lassen sich vier Kategorien bilden:

- *Prozedurale Programmiersprachen:* Ein Programm ist aus prozeduraler Sicht eine Befehlsfolge. Es besteht aus einem Algorithmus für die Berechnung einer Funktion durch den Aktionsträger und Betriebsmittelanforderungen an die Umgebung (für ein Beispiel vgl. [FeSi01], S. 272ff) Beispiele: Pascal, C, Cobol.
- *Deklarative bzw. wissensbasierte Programmiersprachen:* Dienen in erster Linie der Beschreibung einer Aufgaben-Außensicht, wodurch der WAS-Aspekt in den Vordergrund gestellt wird. Lösungen werden durch ein einheitliches Lösungsverfahren ermittelt. Beispielsweise bestehen Prolog-Programme aus Fakten über Objekte und deren Beziehungen und Regeln über Objekte und deren Beziehungen. Hinzu kommt ein Frage- und Antwortsystem, mit dem die in den Fakten und Regeln beschriebenen Objekte abgefragt werden können (Sql).
- *Objektorientierte Sprachen:* Ihnen liegt die Idee zugrunde, die Trennung zwischen Daten und Prozeduren/Funktionen aufzuheben. Es existieren nur noch Objekte, die Daten und Operationen auf diese Daten vereinen. Die Objekte kommunizieren mit Hilfe von Nachrichten miteinander. Beispiele: Smalltalk, C++, Java und C#.
- *Funktionale Sprachen:* Bereits 1959 wurde die Sprache Lisp entwickelt. Einsatzgebiete waren nicht-numerische Probleme wie Mustererkennung oder mathematisch-logische Beweisverfahren. Zentrales Element ist Funktion im traditionellen mathematischen Sinne $y = f(x)$. Ein weiteres Beispiel ist Fortran.

2.3 Die semantische Lücke

Bereits in den einführenden Definitionen von Programmiersprache und Programm wurde auf den unterschiedlichen Sprachschatz von Menschen

und Maschinen verwiesen, welcher für Nicht-Programmierer zu Kommunikationsproblemen führt. Dieser Komplexitätsabstand aus Anforderungen einer Anwendung (Nutzermaschine) und umsetzbaren Möglichkeiten (Basismaschine) wird von FERSTL und SINZ als *semantische Lücke* bezeichnet (vgl. [FeSi01], S. 289). Diese zu überwinden, ist Ziel der Entwicklung von Software. Folgende Möglichkeiten werden hierfür vorgeschlagen:

1. Absenken der Anforderungsebene durch Detaillierung und Präzisierung und Zerlegung der Spezifikation
2. Anheben der Programmierebene durch höhere Programmiersprachen und deklarative Programmierung
3. Methodische Unterstützung bei schrittweiser Überwindung in überschaubaren Komplexitätsspannen

Die Überbrückung der semantischen Lücke soll beispielhaft an folgendem Sachverhalt demonstriert werden: Gegeben sei die quadratische Gleichung $x^2 + 8x + 7 = 0$. Gesucht werden die Werte für x, für die die Gleichung gilt. In der Mathematik wird dazu folgende Formel verwandt:

- Allgemeine Form: $x^2 + px + q = 0$
- Nach x aufgelöst: $x = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q}$

Die Lösung für unser Beispiel beträgt: $x_1 = -7$ und $x_2 = -1$. Zu der Lösung gelangt man dadurch, dass man weiß, WIE (in welchen Schritten) man vorgehen muss. Dabei spielt es keine Rolle, WARUM die Formel gilt. Solch ein Vorgehen, welches allgemein gesprochen zur Lösung eines Problems führt, wird als *Algorithmus* bezeichnet. Die nun folgende erste Beschreibung der Schritte zur Lösung des Beispiels benennen wir daher als Algorithmus 1 (A1).[5] A1 hat folgendes Aussehen:

1. Lies die Zahlen p und q
2. Berechne die Zahl $w = \sqrt{\frac{p^2}{4} - q}$
3. Berechne die Zahl $x_1 = -\frac{p}{2} + w$
4. Berechne die Zahl $x_2 = -\frac{p}{2} - w$
5. Schreibe x_1 und x_2 als Ergebnis auf

Der so beschriebene Algorithmus A1 besteht aus 5 einzelnen Schritten. Diese müssen eigentlich so verständlich sein, dass der Leser weiß,

[5] Das Vorgehen zur Lösung von quadratischen Gleichungen stammt vermutlich von einem arabischen Mathematiker namens AL-CHWARIZMI und wurde ca. 800 entdeckt. Das Wort *„Algorithmus“* selbst ist durch eine Abwandlung oder Verballhornung des Namens AL-CHWARIZMI entstanden.

was er zu tun hat. Ansonsten entsteht ein Kommunikationsproblem. Die Bestimmtheit der Schritte in A1 ist aber bspw. aufgrund ihrer Komplexität noch nicht eindeutig genug. Der Algorithmus wird also durch ein „Auflösen“ der komplizierten Schritte vereinfacht und somit in Algorithmus 2 (A2) überführt:

1. Lies p und q
2. Berechne $\frac{p}{2}$ und nenne das Ergebnis a
3. Berechne a^2 und nenne das Ergebnis b
4. Berechne $b - q$ und nenne das Ergebnis c
5. Berechne $\sqrt{c}$ und nenne das Ergebnis d
6. Berechne $-a$ und nenne das Ergebnis e
7. Berechne $e + d$ und nenne das Ergebnis x_1
8. Berechne $e - d$ und nenne das Ergebnis x_2
9. Schreibe x_1 und x_2

A2 enthält am Ende Schritte, die auf jeden Fall einfacher auszuführen sind als die von A1. Allerdings ist der Gesamtalgorithmus A2 unübersichtlicher und schwerer zu verstehen als A1. Zusätzlich bleibt die Frage ungeklärt, was geschehen soll, wenn c kleiner oder gleich 0 ist, d. h. unser Algorithmus führt nicht zu einem vorhersehbaren Endzustand, ist also nicht eindeutig. Um ihn weiter zu verbessern, muss Algorithmus 3 (A3) also sowohl den Wertebereich (die Menge der gültigen Werte) eindeutig festlegen als auch alle möglichen Eingaben berücksichtigen:

1. Lies p, q
2. Berechne $a = \frac{p}{2}$
3. Berechne $b = a^2$
4. Berechne $c = b - q$
5. Wenn $c \leq 0$ Abbruch
6. Berechne $d = \sqrt{c}$
7. Berechne $e = -a$
8. Berechne $x_1 = e + d$
9. Berechne $x_2 = e - d$
10. Schreibe x_1, x_2

Ein Mittel zur eindeutigen Beschreibung des Algorithmus sind formale Sprachen. Programmiersprachen sind, wie bereits erwähnt, solche formale Sprachen. Demnach müsste es möglich sein, die Berechnung in Java durchzuführen. Auf eine Erklärung der verwendeten Java-Syntax wird an dieser Stelle zunächst verzichtet:

```
public class QuadratischeGleichung {
  public static void main(String[] args) {
    System.out.println("Mein erstes Programm");
    int p = 8;
    int q = 7;
    double x1 = −p/2 + Math.sqrt(p ∗ p /4 − q);
    double x2 = −p/2 − Math.sqrt(p ∗ p /4 − q);
    System.out.println("Ergebnis x1=" + x1 + ", x2=" + x2);
  }
}
```

Dass die Implementierungen von Programmiersprachen mangelhaft sind (beispielsweise sind den Werten von x_1 und x_2 Grenzen nach oben und unten gesetzt), soll an dieser Stelle nicht weiter interessieren.

2.4 Qualitätsanforderungen an Programme

Im Rahmen des Qualitätsmanagements bei der Entwicklung von Programmen (Softwareprodukten) muss man sich folgenden Kernfragen stellen (vgl. [Boe+78]):

- Wie ist Software-Qualität zu definieren? Das heißt, ist es möglich, Eigenschaften und Merkmale eines Software-Produktes zu definieren, die messbar sind und sich nicht überschneiden?
- Wie kann Software-Qualität geprüft werden? Das heißt, wie gut können die Eigenschaften bzw. Merkmale, die die Qualität des Software-Produktes bestimmen, gemessen werden?
- Wie gut können Anforderungen mit einem Instrumentarium formuliert werden? Kann eine Qualitätslenkung erfolgen? Wie können Informationen über die Qualität des Produktes zur Verbesserung des Produktes im Lebenszyklus eingesetzt werden?

Die Antworten auf diese Fragen werden nur in Ansätzen in diesem Abschnitt diskutiert, da sie sich über alle Phasen der Softwareentwicklung erstrecken. Die Konsequenzen für die Implementierung werden im Abschnitt 2.4.4 vorgestellt.

2.4.1 Definition von Qualität

Der Begriff der *Qualität* kann nach dem Deutschen Institut für Normung e. V. (DIN) wie folgt definiert werden: „Grad, in dem ein

Satz inhärenter Merkmale [...] Anforderungen [...] erfüllt“ ([DIN05], S. 18).[6] Eine *Anforderung* ist dabei eine „Erfordernis oder Erwartung, das oder die festgelegt, üblicherweise vorausgesetzt oder verpflichtend ist“ ([DIN05], S. 19).[7] Ein qualitätsrelevantes und damit auch inhärentes *Merkmal* wird in diesem Zusammenhang als kennzeichnende Eigenschaft eines Produkts, Prozesses oder Systems verstanden, das sich auf eine Anforderung bezieht (vgl. [DIN05], S. 26).[8] Qualitätsrelevante Merkmale sind beispielsweise der Benzinverbrauch eines Fahrzeuges oder die Zuverlässigkeit einer Software.

Zur Bestimmung, inwieweit die spezifizierten Anforderungen an ein Produkt, Prozess oder System durch die qualitätsrelevanten Merkmale erfüllt werden, wird, wie in anderen Bereichen ebenfalls üblich, im Bereich der Softwareentwicklung das Produkt (hier das Programm) geprüft (vgl. [Wall90], S. 168). Dies dient der Qualitätssicherung (vgl. dazu Abschnitt 2.4.3). Es werden dabei folgende Arten von Qualitätsmängeln unterschieden (vgl. [Wall90], S. 168):

- Ein *Fehler* (error) beschreibt eine Abweichung zwischen dem berechneten, beobachteten oder gemessenen Wert oder einem Zustand der Betrachtungseinheit und dem entsprechenden spezifizierten oder theoretisch richtigen Wert.

Beispielsweise liegt ein Fehler vor, wenn ein Programm einen Wert von `45` ausgibt, aber der tatsächliche Wert bei `40` liegt.

- Ein *Defekt* (defect) beschreibt die Abweichung von der festgelegten (erwarteten) Ausprägung eines Merkmales der Betrachtungseinheit. Im Unterschied zu einem Fehler muss dieser nicht berechenbar sein.

[6] *Inhärent* bedeutet im Gegensatz zu *zugeordnet* einer Einheit innewohnend, insbesondere als ständiges Merkmal. (vgl. [DIN05], S. 18).

[7] *Üblicherweise vorausgesetzt* bedeutet, „... dass es für die Organisation, ihre Kunden und andere interessierte Parteien üblich oder allgemeine Praxis ist, dass das entsprechende Erfordernis oder die entsprechende Erwartung vorausgesetzt ist.“ ([DIN05], S. 19) Eine *festgelegte Anforderung* ist eine Anforderung, die beispielsweise in einem Dokument angegeben ist. Anforderungen können von verschiedenen interessierten Parteien aufgestellt werden. Darüber hinaus darf ein Bestimmungswort verwendet werden, um eine spezifische Anforderungsart zu bezeichnen, z. B. Produktanforderung, Qualitätsmanagementanforderung, Kundenanforderung (vgl. [DIN05], S. 19).

[8] Daher stellt ein einem Produkt, einem Prozess oder einem System lediglich zugeordnetes Merkmal (z. B. der Preis eines Produkts, der Eigentümer eines Produkts, usw.) kein Qualitätsmerkmal dieses Produkts, Prozesses oder Systems dar (vgl. [DIN05], S. 26).

Defekte können, aber müssen nicht, die Funktionstüchtigkeit der Betrachtungseinheit beeinträchtigen. Sie können in Zusammenhang mit jedem qualitätsrelevanten Merkmal auftreten. Beispielsweise würde ein Defekt vorliegen, wenn die Verfügbarkeit einer Software bei 95% liegen soll, sie jedoch tatsächlich nur bei 40% liegt.

- Ein *Ausfall* (failure) entspricht der Beendigung der Fähigkeit einer Betrachtungseinheit, die geforderte Funktion auszuführen.

Ein Ausfall kann eine Betriebsbeeinträchtigung einer übergeordneten Betrachtungseinheit bewirken (bspw. einen Defekt oder Fehler) oder aber wesentlich katastrophalere Folgen haben. Ein Ausfall wirkt sich auf die Zuverlässigkeit der Betrachtungseinheit aus.

- Eine *Störung* (fault) ist die Unfähigkeit einer Betrachtungseinheit, ihre geforderte Funktion auszuführen.

Eine Störung kann durch einen Ausfall, einen Defekt oder einen Fehler verursacht werden. So kann bspw. der Ausfall einer Datenbank die Störung eines Programms bewirken, welches monatliche Abschlussarbeiten durchführt.

2.4.2 Ansätze zur Qualitätsmessung

Es existieren fünf Ansätze zur Messung von Softwarequalität (vgl. [Wall90], S. 7f).

1. Der *transzendente* Ansatz basiert auf der Annahme, dass Qualität universell erkennbar und ein Synonym für kompromisslos hohe Standards und Ansprüche an die Funktionsweise eines Produktes ist. Demnach kann Qualität nicht präzise definiert und gemessen werden, sondern nur durch Erfahrung bewertet werden.
2. Nach dem *produktbezogenen* Ansatz ist Qualität präzise messbar. Dabei auftretende Abweichungen spiegeln Unterschiede in der vorhandenen, beobachtbaren Quantität bestimmter Eigenschaftsausprägungen wider. Diese Betrachtungsweise erlaubt es, Rangordnungen von verschiedenen Produkten gleicher Kategorien anzugeben.
3. *Anwenderbezogen* wird die Qualität durch den Benutzer und weniger durch das Produkt selbst festgelegt. Verschiedene Benutzer haben unterschiedliche Wünsche und Bedürfnisse. Produkte, die diese am besten befriedigen, werden als qualitativ hochstehend angesehen.
4. Der *prozessbezogene* Ansatz setzt Qualität mit der Einhaltung von Spezifikationen gleich. Er basiert auf der Idealvorstellung, dass eine Tätigkeit beim ersten Mal richtig ausgeführt wird. Er ist auf die

heutige Wirtschaft und Industrie ausgerichtet und spielt eine bedeutendende Rolle bei der Automatisierung, um den Produktionsprozess bezüglich geringerer Ausschuss- und Nachbearbeitungskosten zu optimieren.

5. Im *Preis-Nutzen-bezogenen* Ansatz wird eine Beziehung zwischen Preis und Qualität hergestellt. „Ein Qualitätsprodukt ist in dieser Denkweise ein Erzeugnis, das einen bestimmten Nutzen zu einem akzeptablen Preis oder eine Übereinstimmung mit Spezifikationen zu akzeptablen Kosten erbringt." ([Wall90], S. 7f)

2.4.3 Qualitätssicherungsmaßnahmen

Maßnahmen zur Sicherung von Qualität können in folgende Kategorien eingeteilt werden ([Wall90], S. 24):

- *planerisch-administrative*: Maßnahmen für Aufbau, Einführung und Pflege eines Qualitätssicherungssystems
- *konstruktive*: Maßnahmen zur Qualitätsgestaltung
  - *technische*: z. B. Verwendung von Prinzipien, Methoden und Werkzeugen des Software-Engineering
  - *organisatorische*: z. B. Verwendung von Vorgehensmodellen, Pläne zur Konfigurationsverwaltung
  - *menschliche*: z. B. Schulungen und psychologisch orientierte Maßnahmen (Maßnahmen für die Menschen als Entwickler, Projektleiter und Projektmanager)
- *analytische*: Maßnahmen zum Erkennen und Lokalisieren von Mängeln (vgl. Abschnitt 2.4.1)

2.4.4 Einfluss der Objektorientierung auf die Qualität

Nach Untersuchungen von Baker, Fagan, Walston und Felix am Ende der 70er Jahre des letzten Jahrhunderts ist durch den Einsatz geeigneter Methoden der Softwareentwicklung ein Produktivitäts- und Qualitätsgewinn von ca. 50% – 150% (je nach Methode) möglich. Nach einer weiteren Untersuchung von Jones besitzen Projekte auf Grundlage von Methoden der objektorientierten Analyse, des objektorientierten Entwurfs und der objektorientierten Programmierung zudem eine bis zu 30% höhere Produktivität im Vergleich zu Projekten, die auf einem anderem Paradigma basieren. Gebert und Wiese berichten überdies von einer Aufwandsreduzierung von 50% – 70% in einem Projekt, in welchem durch die von der *OMG (Object Management Group)* postulierte

modellgetriebene Softwareentwicklung *MDA (Model Driven Architecture)* „nahtlos“ aus objektorientierten Analysemodellen über (plattformunabhängige und plattformspezifische) objektorientierte Entwurfsmodelle Software in einer objektorientierten Programmiersprache entwickelt wurde, gegenüber herkömmlichen Projekten, in denen mit objektorientierten Methoden gearbeitet wird (vgl. [GeWi05], S.29).

Allein diese Ergebnisse rechtfertigen also den Einsatz der Objektorientierung bis hin zur Programmierung. Nach der Vorstellung dieses Paradigmas (Abschnitt 2.5) und seiner Umsetzung in Java (Teil 4) schließen sich mit dem Abschnitt über die (Programmier-) Methodik (Teil 6) einige weiterführende Betrachtungen an.

2.5 Das objektorientierte Paradigma

Bevor das in Java verwendete Smalltalk-artige Modell in diesem Abschnitt näher vorgestellt wird, wird ein Überblick über existierende Variationen von Objektorientierung gegeben.

Smalltalk-*artige Modelle*

Im Mittelpunkt dieses Paradigmas steht das Konzept der Klasse. Die Instanzen einer Klasse werden als Objekte bezeichnet und stets genau einer Klasse zugeordnet. Alle Objekte einer Klasse besitzen den selben strukturellen Aufbau (Instanzvariablen). Zu jeder Klasse gehört eine Menge von Methoden (Operatoren), die auf den Instanzen ausführbar sind. Objekte können über Nachrichten miteinander kommunizieren. Jeder dieser interpretierbaren Nachrichten wird mit einem *Method Directory* eine Methode zugeordnet. Die Klasseninstanz repräsentiert die Klasse selbst und besitzt keine Instanzvariablen. Zwischen den Klassen können Generalisierungs- und Spezialisierungsbeziehungen definiert werden. Nachteile sind vor allem die Beschränkung auf die Repräsentation realer Objekte und die Überführung dieser.

Delegationsmodelle

Dieses Paradigma stammt aus dem Bereich der Wissensmodellierung und hat das Ziel, nicht typidentische Objekte zu verwalten, sondern ähnliche Objekte abzugrenzen. Ausgangspunkt hierfür ist die Objekttyphierarchie. Innerhalb dieser entsprechen die Objekte jedoch nicht notwendigerweise ihrer Typbeschreibung. Vielmehr kann diese bei der Objektbildung beliebig ergänzt oder überschrieben werden. Anders als

im Smalltalk-artigen Modellen, besitzen die in den Klassen beschriebenen Eigenschaften keinen Typcharakter, sondern beschreiben lediglich Prototypen. Daraus folgt unter anderem, dass dieses Modell nicht für die Verwendung in Datenbanksystemen geeignet ist.

Objekt-Rollenmodell (ORM)

Esswein schlägt insbesondere zur aufbauorganisatorischen Regelung in Unternehmen das Objekt-Rollenmodell (ORM) vor (vgl. [Essw93], S. 551). Dieses Modell vereinigt die Vorteile beider oben beschriebenen Modellgruppen. Es ist streng typgebunden.

Die Beschreibung der Typen erfolgt in einer Rollentyphierarchie (RTH). Im Unterschied zu smalltalk-artigen Modellen beschreiben die Knoten dieser Hierarchie nur einen speziellen Aspekt von Objekttypen. Objekte besitzen nicht den Charakter einer Klasseninstanz. Ihre Typbeschreibung ist ein zusammenhängender Teilbaum der Wurzel innerhalb der RTH. Diese Wurzel bildet das Systemobjekt und enthält Methoden zum Erzeugen und Löschen von Objekten und zum Verändern des Typs des Objekts (Typmigration). Rollentypen können mittels

- Typerweiterung,
- obligate Spezialisierung oder
- wahlweise Spezialisierung

in Subrollentypen spezialisiert werden. Aus diesem Vorgehen ergeben sich Objektrollenbäume, in denen jede Typbeschreibung einen zusammenhängenden Teilbaum bildet.

2.5.1 Objekt

Bei einem *Objekt* handelt es sich um ein physisches oder gedankliches Konstrukt. Bei der Programmierung werden diese Objekte in Informationssystemen abgebildet. Es fasst eine Menge von Daten und Methoden, die auf diese Daten zugreifen, zusammen. Jedes Objekt ist durch:

- sein Verhalten (welches durch Methoden beschrieben wird),
- seinen Zustand (der mit Daten beschrieben wird, die mit Hilfe von Attributen gespeichert sind) und
- seine Identität

gekennzeichnet. Ein Beispiel (in Anlehnung an [Oest01], S. 38ff):

> Bauer Bertram besitzt drei Kühe - Elsa Euter, Vera Vollmich und Anja v. d. Alm. Jede Kuh ist ein Objekt, da sie eine Identität (also

einen eindeutigen Namen), einen Zustand (satt oder hungrig) und ein Verhalten (sie laufen über die Weide und lassen sich melken) besitzen. Vera besitzt außer ihrem Zustand satt/hungrig wie jede Kuh vier Beine, ein Euter und einen Schwanz.

2.5.2 Kapselung

Kapselung bedeutet, dass der Zustand und das Verhalten eines Objekts hinter einer Schnittstelle verborgen werden kann, d. h. sie sind nur für die Methoden dieses Objekts zugänglich, nicht für andere Objekte direkt. Die Implementierung der Methoden ist ebenfalls gekapselt, d. h. nach außen ist nur deren *Signatur* (bestehend aus Methodenname und Liste der Parameter) sichtbar, nicht deren Aufbau. Für das Beispiel:

Für unsere drei Kühe bedeutet dies, dass wir nur die Eigenschaften sehen und verändern können, die jede Kuh uns zeigt, bzw. verändern lässt. Beispielsweise könnte sie sich beim Melken wehren, wenn wir ihr nicht vorher gut zureden. Wir müssen schon die Erlaubnis der Kuh einholen, sie also streicheln und über ihre Methode „gib Milch“ die Eigenschaft „Euter“ vom Zustand „voll“ in den Zustand „leer“ überführen. Wie eine Kuh von innen funktioniert, also wie sie eigentlich das Euter jeden Tag füllt, ist dem Bauer Bertram ebenfalls nicht klar. Er weiß nur, dass die Kuh eine Methode „friss Gras“ besitzt und dass beides irgendwie zusammenhängt – wie genau ist ihm eigentlich egal und wird von der Kuh sowieso geschickt verborgen.

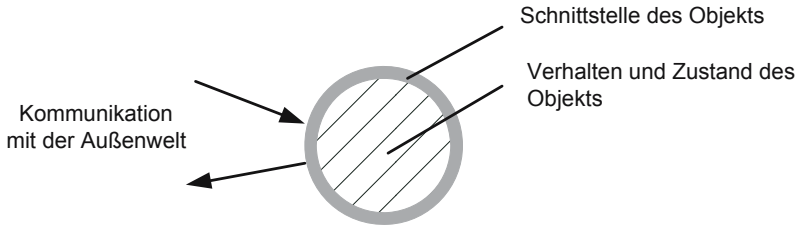


Abb. 2.4. Prinzip der Kapselung

2.5.3 Klassen

Für gleichartige Objekte werden Baupläne erstellt. Beispielsweise erscheint es logisch, alle drei Damen aus der Abbildung 2.5 als Kuh zu beschreiben. Sie sind Individuen (jede hat spezielle Eigenschaften - Vera

z. B. Flecken). Ihre Gemeinsamkeiten kann man zur Klasse `Kuh` zusammenfassen. Eine Klasse kann man im Gegensatz zu ihren Objekten (im Beispiel die drei) nicht „anfassen“, da sie „nur“ eine Beschreibung darstellt.

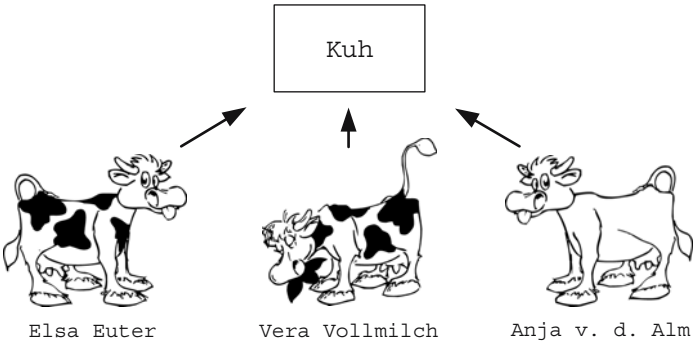


Abb. 2.5. Von der Kuh zur Klasse (vgl. [Oest01], S. 39)

Korrekt definiert sind *Klassen* Schablonen, die verwendet werden, um gleichartige Objekte zu erzeugen. Diesen Vorgang nennt man *Instanziierung* einer Klasse. Speziellen Eigenschaften der Objekte werden für jede Instanz einmal definiert, Methoden jedoch nur einmal für jede Klasse. Die Instanzen der Klasse (Objekte) nutzen diese Methoden gemeinsam.

Abb. 2.6. Bildung von Klassen

Die Klassen enthalten somit eine Beschreibung aller Eigenschaften ihrer Objekte. Während des objektorientierten Programmierens werden also Klassen definiert und deren Beschreibung in der jeweiligen Sprache verfasst (in unserem Fall Java). Während der Laufzeit des Programms werden von diesen Klassen Objekte erzeugt.

2.5.4 Vererbung

Als *Vererbung*[9] wird die Fähigkeit von Klassen bezeichnet, die Eigenschaften ihrer Superklasse zu übernehmen. Subklassen werden gebildet, wenn das Verhalten einer Klasse verändert oder erweitert werden soll. Die Subklasse kann auf die Attribute und Methoden der Superklasse zurückgreifen. Vererbung ermöglicht ebenfalls die Wiederverwendung von Programmcode. Die Subklasse erbt alle Attribute und Methoden der Superklasse, kann jedoch geerbte Attribute neu definieren bzw. geerbte Methoden verändern (dieses Verfahren wird als *Überschreiben* oder *Overwriting* bezeichnet, vgl. auch Abschnitt 2.5.6) oder neue Variablen und neue Methoden hinzufügen.

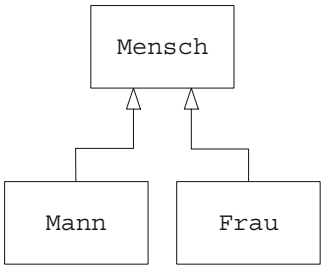


Abb. 2.7. Vererbung

Der Datentyp (und damit der Wertebereich) geerbter Attribute kann aber nicht verändert werden. Im Beispiel der Abbildung 2.7 stellen die Klassen `Mann` und `Frau` Subklassen der Klasse `Mensch` dar. Eine Superklasse, die zudem nur Subklassen und keine Instanzen besitzt bzw. besitzen soll, wird als *abstrakte Klasse* bezeichnet. Vererbung führt zur Herausbildung von *Klassenhierarchien*, innerhalb derer Strukturen aus über- und untergeordneten Klassen gebildet werden.

2.5.5 Nachrichten

In Fortsetzung des Beispiels verständigen sich die Kühe in ihrer eigenen Sprache mit ihren eigenen Nachrichten, wenn sie beispielsweise ihre „Mitkühe“ auffordern wollen, einen Grashalm in Ruhe zu lassen, um ihn selbst zu fressen.

[9] engl. *inheritance*

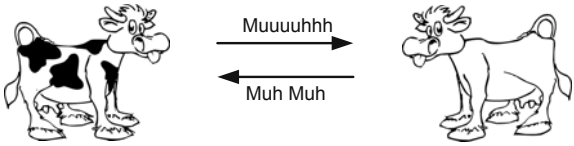


Abb. 2.8. Nachrichtensystem der Kühe

Objekte kommunizieren also über Nachrichten. Eine *Nachricht* ist eine Aufforderung von einem Objekt (Sender) an ein anderes Objekt (Empfänger) eine Methode auszuführen.

Ein fertiges Programm besteht zu seiner Laufzeit aus einer Vielzahl von Objekten mit unterschiedlichen Aufgaben (Ausgabe auf den Bildschirm, Verwalten von Daten, Durchführen von Berechnungen etc.), die über Nachrichten miteinander und mit dem Nutzer des Programms kommunizieren, wie es durch die Definition ihrer Klassen vorgeschrieben wurde.

2.5.6 Polymorphismus

Polymorphismus[10] entsteht durch Vererbung. Dadurch ist es möglich, Methoden mit gleichem Namen, aber unterschiedlichem Verhalten, zu definieren und geerbte Attribute zu überschreiben. Bezüglich gleichnamiger Methoden bedeutet dies: Die gleiche Nachricht, an verschiedene Objekte gesandt, löst unterschiedliches Verhalten aus.

Man stelle sich hierzu eine Klasse `GeometrischeFigur` vor, welche auf die Nachricht „Stelle Dich dar“ damit reagiert, die Methode `anzeigen()` auszuführen. Des Weiteren existieren eine Klasse `Kreis` und eine Klasse `Rechteck`, die von `GeometrischeFigur` erben, d. h. Subklassen von ihr sind. Damit Objekte der Klasse `Kreis` als auch der Klasse `Rechteck` auf dem Bildschirm richtig dargestellt werden, wenn sie die Nachricht „Stelle Dich dar“ empfangen, muss die von der Klasse `GeometrischeFigur` geerbte Methode `anzeigen()` in den Klassen `Kreis` und `Rechteck` entsprechend *überschrieben* und somit also durch ein neues (anderes bzw. spezielleres) Verhalten ersetzt werden.

2.5.7 Interfaces

Wie bereits im Abschnitt 2.5.2 erläutert wurde, können Objekte ihren Zustand und ihr Verhalten hinter einer Schnittstelle nach außen verber-

[10] Vielgestaltigkeit; engl. *polymorphism*

gen. Die englische Übersetzung *Interface* wird jedoch nicht nur in dieser Bedeutung verwendet. Es existieren somit zwei alternative Definitionen:

- Interface als Schnittstelle, die jedes Objekt besitzt, um Methoden oder Daten zu verbergen oder nach außen zu propagieren
- Interface als „klassenähnliche“ Einigung über die „Minimalfunktionalität“ eines Objekts

Während der erste Punkt bereits auf Seite 18 erklärt wurde, bedarf der zweite einer näheren Erläuterung. Man denke sich hierfür ein Objekt der Klasse `Auto`, das mit Objekten der Klasse `Fahrer` interagieren soll. Dem Auto sollte es dabei egal sein, ob der Fahrer männlich oder weiblich ist, ob er lange oder kurze Haare oder einen Bauernhof hat. Das Auto wird stets nur bemüht sein, mit Objekten zu kommunizieren, welche die Nachricht „Der Tank ist fast leer“ oder „Ich fahre jetzt 120 km/h“ verstehen. Gefordert wird also ein Interface `Fahrer`, welches Methoden für diese Nachrichten vorsieht (Minimalfunktionalität) und beispielsweise von den Klassen `Mann` und `Frau` umgesetzt (implementiert) wird.

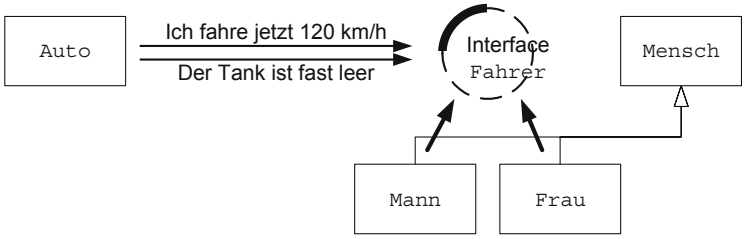


Abb. 2.9. Kommunikation mit Interfaces

Wenn also eine Klasse (und damit alle ihre Objekte) ein Interface implementiert, ist gewährleistet, dass sie auf bestimmte Nachrichten reagiert. Für unser Beispiel spricht man davon, dass die Klassen `Mann` und `Frau` das Interface `Fahrer` implementieren müssen, damit ihre Objekte mit dem Auto kommunizieren können.

Generell stellen abstrakte Klassen (also Klassen, die keine Objekte besitzen können, da eine Methode keinen Inhalt besitzt) solche Schnittstellen dar. Da in JAVA keine Mehrfachvererbung (eine mehrfach erbende Klasse erbt von mehreren übergeordneten Klassen) möglich ist, müssen Klassen, die mehreren Schnittstellen entsprechen sollen, diese als Interfaces implementieren.

2.5.8 Pakete

Pakete[11] dienen der logischen Strukturierung einer größeren Menge von Klassen. Gerade umfangreiche Sammlungen (so genannte Klassenbibliotheken oder Frameworks) erfordern eine Strukturierung, um für den Programmierer handhabbar zu werden. Jede Klasse wird einem Paket zugeordnet, wobei dieses sinnvollerweise mehr als eine Klasse beinhalten sollte. Um die Klassen des Bauernhofes zu strukturieren, könnten folgende Pakete gebildet werden:

- `bauernhof`: Dieses Paket enthält die Pakete `tiere`, `menschen` und `einrichtung`.
- `menschen`: Enthält die Klassen `Bauer` und `Bäuerin`.
- `tiere`: Enthält die Klassen `Kuh` und `Schwein`.
- `einrichtung`: Enthält die Klassen `Stall` und `Melkeimer`.

2.5.9 Zeiger und Referenzen

Zeiger sind Verweise auf bestimmte Bereiche im Speicher des Rechners, auf dem das Programm ausgeführt wird. Im Gegensatz dazu werden in *Referenzen* Verweise auf Objekte abgelegt (vgl. Abschnitt 4.5). Zeiger können also ebenfalls auf Objekte verweisen, da auch diese zwangsläufig in einem bestimmten Speicherbereich liegen. Aus folgenden Gründen werden in JAVA jedoch stets nur Referenzen verwendet:

- Zeiger gestatten den Zugriff auf Teile von Objekten, ohne deren Methoden (also die Schnittstellen nach außen) zu nutzen → Verletzung der Kapselung!
- Zeiger können auf ungültige Bereiche im Speicher verweisen, was zu zahlreichen Fehlern in der Programmierung führen kann (die „Allgemeine Schutzverletzung ...“ dürfte hinreichend bekannt sein).

Übergibt eine Methode einer anderen eine Objektreferenz, so wird keine Kopie des Objekts innerhalb der zweiten Methode erstellt, sondern die Veränderungen am Objekt selbst durchgeführt. Grundsätzlich gilt also: Ein Objekt ist im Gegensatz zu seinen Referenzen nur einmal im Programm (bzw. Arbeitsspeicher des Rechners) enthalten.

2.5.10 Konstruktor

Ein *Konstruktor (constructor)* ist eine Methode einer Klasse, die neue Objekte erzeugt. Genauso wie ein Konstruktionsplan die Anzahl der

[11] engl. *packages*

Etagen und den Aufbau des Daches für ein Gebäude festlegt, beschreiben die Klassen die Eigenschaften und das Verhalten der Objekte. In Analogie zu einem Bauarbeiter, der anhand dieser Pläne ein Gebäude baut, erzeugt ein Konstruktor die Instanzen einer Klasse, also die Objekte.

Demnach können die speziellen Methoden nicht den Objekten, sondern den Klassen zugeordnet werden. Man definiert zur Unterscheidung der Begriffe deshalb die *Klassenmethoden*. Neben der Aufgabe, Objekte zu erzeugen, können Klassenmethoden auch andere Aufgaben übernehmen.

2.6 Zusammenfassung

Das in diesem Abschnitt definierte Begriffssystem wird zusammenfassend in Abbildung 2.10 dargestellt. Eine kurze Beschreibung ausgewählter Begriffe ist Tabelle 2.1 zu entnehmen.

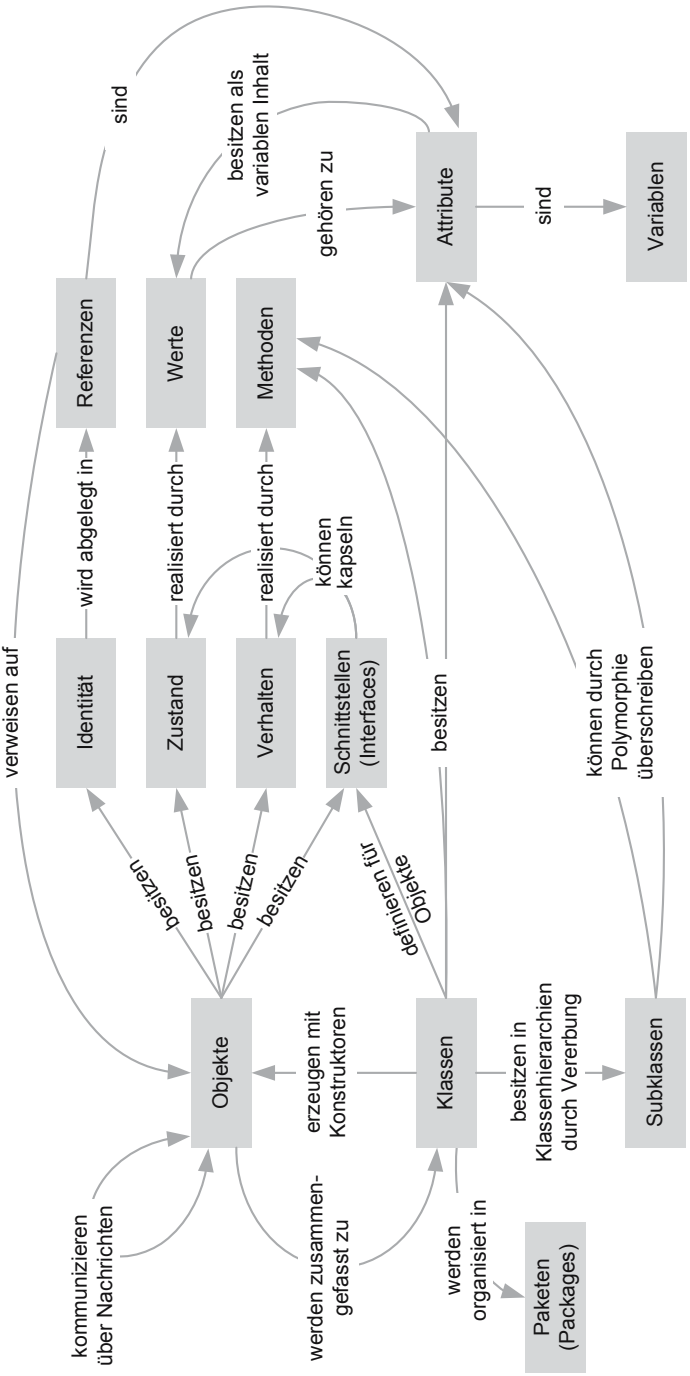


Abb. 2.10. Begriffssystem des objektorientierten Paradigmas

Begriff	**Erklärung**
Programmiersprache	Formale Sprache zur Kommunikation zwischen Mensch und Rechner (siehe S. 6).
Objektorientierung	Denkansatz (Paradigma), der davon ausgeht, dass sich Zusammenhänge der Realwelt als Objekte abbilden lassen (siehe S. 16).
Objekt	Ist ein physisches oder gedankliches Konstrukt, welches sich durch Identität, Zustand und Verhalten auszeichnet (siehe S. 17).
Klasse	Bauplan für eine Menge gleichartiger Objekte. Definiert dabei das Verhalten sowie die Attribute (Speichern den Zustand des Objektes) (siehe S. 18).
Kapselung	Zustand und Verhalten eines Objekts werden hinter einer Schnittstelle verborgen und sind für andere Objekte nicht „einsehbar“ (siehe S. 18).
Vererbung	Klassen übernehmen Eigenschaften (Attribute) und Verhalten (Methoden) von ihrer Superklasse (siehe S. 20).
Instanziierung	Erzeugen eines Objektes mittels eines Konstruktors seiner Klasse.
Attribute	Variablen eines Objektes, die dessen Zustand speichern. Diese Variablen werden bereits in der Klasse definiert und durch den Konstruktor mit sinnvollen Werten initialisiert (belegt).
Methoden	Bestimmen das Verhalten eines Objektes zur Laufzeit und werden in der Klasse definiert. Werden durch Nachrichten aufgerufen.
Nachrichten	Hilfsmittel zur Kommunikation zwischen Objekten Objekte versenden Nachrichten und können diese mittels ihrer Methoden empfangen (siehe S. 20).
Overwriting (Überschreiben)	Vererbte Attribute und Methoden können redefiniert werden. Für Methoden gleichen Namens bedeutet dies, dass sie unterschiedliche Implementierungen und damit unterschiedliches Verhalten aufweisen können (siehe S. 20 und S. 21).

Tabelle 2.1. Begriffe und ihre Bedeutung

2.7 Übungsaufgaben

Aufgabe 1. Ordnen Sie eine Programmiersprache Ihrer Wahl (außer JAVA) in die Klassifizierung der Programmiersprachen ein und begründen Sie Ihre Entscheidung!

Aufgabe 2. Welche Anforderungen stellen Sie persönlich an „gute“ Software? Wie lassen sich diese einordnen und realisieren?

Aufgabe 3. Identifizieren Sie in Ihrer näheren Umgebung Objekte! Beschreiben Sie deren Schnittstelle nach außen sowie Verhalten und Eigenschaften, die vor Ihnen verborgen bleiben!

Aufgabe 4. Fassen Sie die gefundenen Objekte aus Aufgabe 3 zu Klassen zusammen! Versuchen Sie Hierarchien zu bilden, indem Sie Gemeinsamkeiten der Klassen identifizieren und daraus Vererbungsbeziehungen generieren!

Aufgabe 5. Warum ist ein Konstruktor eine Klassenmethode?

3

Das JAVA Development Kit

„Gordon's wunderbarer Einfall war, ein Programm zu entwickeln, das es einem erlaubte, schon vorher genau anzugeben, zu welcher Lösung man gelangen möchte, und ihm erst dann alle Fakten einzugeben. Die Aufgabe des Programms, die es mit absoluter Leichtigkeit zu lösen vermochte, war schlicht die, die plausible Reihe logisch klingender Schritte zu vollziehen, um die Voraussetzungen mit der Schlussfolgerung zu verbinden. Und ich muss sagen, es funktionierte prächtig."

Richard über Gordon in Douglas Adams'
„Dirk Gently's holistische Detektei"

3.1 Historie

Die Entwicklung von JAVA kann zu den Versuchen zurückverfolgt werden, Software zur Steuerung von Elektronik aus dem Heimbereich zu entwickeln. Im April 1991 begann eine Gruppe von Angestellten des Unternehmens SUN MICROSYSTEMS in einem Projekt names *Green Project* (vgl. [Gosl98]) zusammenzuarbeiten. „The goal of the Green project was to develop a system for developing consumer consumer electronics' logic." ([SiWe97], S. 2)

JAMES GOSLING versuchte als Mitglied des *Green Projects* eine Alternative für die Programmiersprache C++ zu entwerfen, nachdem zahlreiche Versuche, diese an die Anforderungen für den Einsatz in Heimelektronik anzupassen, fehlschlugen (vgl. [SiWe97], S. 2). Man hatte festgestellt, dass in Hinblick auf Plattformunabhängigkeit[1], Sicherheit[2] und Robustheit[3] eine Anpassung oder Erweiterung von C++ nie das gewünschte Ziel erreichen würde. Die neue Sprache sollte zuerst OAK genannt werden, allerdings war der Name bereits für eine andere experimentelle Programmiersprache reserviert. Bei einem Treffen in einem Café einigte man sich dann auf JAVA (vermutlich nach dem gleichnamigen Kaffee).

[1] Unabhängigkeit vom Betriebssystem (z. B. Linux, Solaris oder Windows).
[2] Möglichkeit, die Rechner bei der Ausführung von Programmen vor unerlaubten Zugriffen zu schützen.
[3] Toleranz gegenüber Fehlern.

Das erste Programm in JAVA entstand 1992 im Rahmen dieses *Green Projects* und sollte ein *PDA (Personal Digital Assistant)* ähnliches Gerät namens **7 („StarSeven“)* mittels einer einfachen Benutzeroberfläche steuern (vgl. [Byou98]). Das Maskottchen DUKE war Bestandteil einer dafür entworfenen Animation und ist heute eines der eingetragenen Warenzeichen von SUN MICROSYSTEMS.

**7* war allerdings kein Erfolg beschieden. Jedoch wurde Mitte der neunziger Jahre des letzten Jahrhunderts das Internet für den „Endanwender“ immer interessanter. Das *World Wide Web* führte zu den heute bekannten Antwortzeiten mancher Server. SUN erkannte das Potenzial von JAVA für das *World Wide Web* und entwickelte einen Webbrowser namens WEBRUNNER, später umbenannt in HOTJAVA, der dieses Potenzial ausschöpfen sollte (vgl. [SiWe97], S. 3). Der Browser wurde 1995 auf der „Sun World ’95“, einer Messe des Herstellers, vorgestellt. Die in JAVA geschriebenen Animationen auf den im Browser dargestellten Webseiten, die bis dahin rein statisch waren, entfachten riesige Begeisterung. Das Interesse der Computer-Fangemeinde war vollständig geweckt (vgl. [Byou98]). Zusammenfassend besitzt JAVA folgende Eigenschaften:

- *Objektorientierung*: JAVA ist „vollständig“ objektorientiert. Unterstützt werden alle Elemente des objektorientierten Paradigmas.
- *Interpretation*: JAVA-Programme werden interpretiert. Durch die Interpreter auf verschiedenen Plattformen wird die Unabhängigkeit gewährleistet.
- *Robustheit/Sicherheit*: Beispiele für das Vermeiden von Laufzeitfehlern sind die strenge Prüfung bei Typumwandlung, das Vermeiden von direkten Speicherzugriffen durch Zeiger, das automatische Freigeben von nicht mehr benötigten Objekten und die Ausnahmebehandlung.

Von den meisten Webbrowsern wird die Sprache JAVASCRIPT unterstützt.[4] Diese Skriptsprache[5] wurde von NETSCAPE entwickelt und kann in HTML-Code eingebettet werden. Mit dieser Sprache können einige Funktionen der Browser ausgeführt werden (Banner scrollen, Bilder flackern etc.). JAVASCRIPT ist aber nicht gleich JAVA, obwohl die Syntax an JAVA angelehnt ist und auch einige Schlüsselworte identisch

[4] JAVASCRIPT wurde unter Einbeziehung von MICROSOFTS JSCRIPT unter dem Namen *ECMAScript Language* standardisiert (vgl. [Ecm99]).

[5] Skriptsprachen sind Programmiersprachen, die vor allem für kleine, überschaubare Programmieraufgaben gedacht sind. Sie verzichten oft auf bestimmte Sprachelemente, deren Nutzen erst beim Lösen größerer Aufgaben zum Tragen kommt.

sind, unterscheiden sich die beiden Sprachen von der Semantik teilweise beträchtlich. Im Gegensatz zu JAVA ist JAVASCRIPT:

- nicht vollständig objektorientiert (bspw. werden keine Klassen einsetzt, sondern statt dessen Objekte als Prototypen verwendet)
- nicht plattformunabhängig[6],
- nicht mit den notwendigen Sicherheitsmechanismen ausgestattet.

Das JAVA Development Kit (JDK) wird mit folgendem System der Nummerierung von SUN MICROSYSTEMS ausgeliefert (vgl. [Sun06b]):

- Bug-Fixes werden in Form von Updates innerhalb einer Familie veröffentlicht (5.0 Update x) und dienen zur Korrektur von Fehlern.
- Funktionelle Freigaben innerhalb einer Familie können bspw. neue Sprachzeichen enthalten (5.x).
- Major Releases sind wesentlich erweiterte Versionen (x).

3.2 Funktionsweise

„Normales“ JAVA bedient sich eines Interpreters zur Ausführung der Programme. Diese Idee ist nicht neu (man denke an die Basic-Dialekte der Heimcomputer) und begründet vor allem die Performance-Probleme, die sichtbar werden, wenn JAVA-Programme mit rechenintensiven oder Echtzeit-Problemen konfrontiert werden. Der Mechanismus des Interpreters wird in der folgenden Abbildung dargestellt:

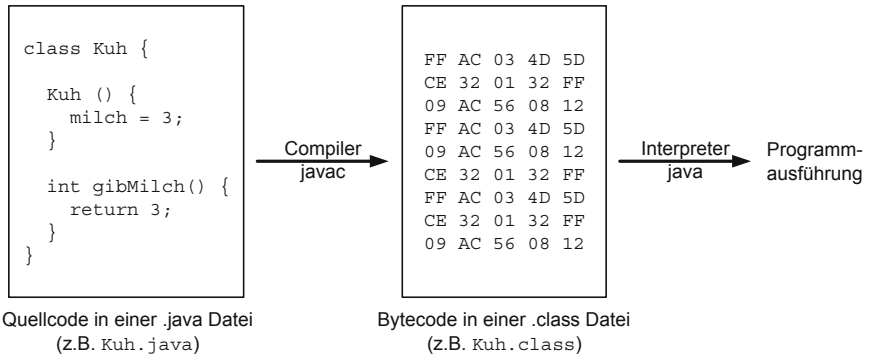


Abb. 3.1. Interpretermechanismus

[6] da an die Plattform der Browser gebunden

Funktionelle Freigaben und Major Releases (vgl. Abschnitt 3.1) sind bezüglich des Quellcodes nicht abwärtskompatibel. Beispielsweise sind JAVA-Programme, die mit JAVA des JDK 5.0 geschrieben wurden, nicht bzw. nur eingeschränkt mit einem Compiler des JDK 1.4 compilierbar. Bezüglich des Bytecode besteht jedoch bis auf Ausnahmen eine Aufwärtskompatibilität, d. h. dass bspw. ein Bytecode erzeugt mit einem Compiler des JDK 5.0 auch korrekt mit einem Interpreter des JDK 6.0 ausgeführt wird (vgl. [Sun06b]).

3.3 Aufbau

Die von SUN MICROSYSTEMS vorgestellte JAVA 2 Plattform versteht sich sowohl als Entwicklungs- als auch Laufzeitumgebung für JAVA-Anwendungen. Eine Übersicht über alle Bestandteile liefert die Abbildung 3.2.

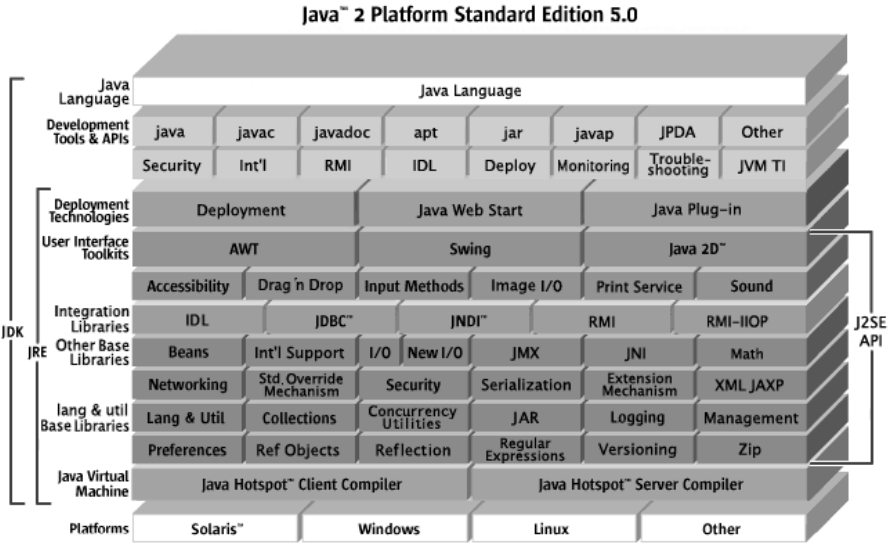


Abb. 3.2. Aufbau der JAVA 2 Plattform ([Sun04])

Seit der Version 1.2 wird JAVA als JAVA 2 bezeichnet, da im Vergleich zu der Version 1.1 nochmal grundlegende Änderungen vorgenommen wurden. Ab der Version 1.2.2 wurde die ursprüngliche Bezeichnung JAVA Development Kit (JDK) zwischenzeitlich durch Software Develop-

ment Kit (SDK) ersetzt. Die aktuelle Plattform heißt wieder JDK und liegt in der (Produkt-) Version 5.0 vor.[7]

3.4 Pakete

Im JDK werden Klassen und Interfaces bestimmter Themengebiete zu Paketen zusammengefasst. Die vollständige Spezifikation dieser Anwendungsprogrammierschnittstelle des JDK (kurz API, Application Programming Interface) kann bei SUN Microsystems eingesehen bzw. abgerufen werden.[8] In der Tabelle 3.1 werden die wichtigsten Pakete aufgeführt und nachfolgend drei von ihnen noch kurz etwas näher erläutert:

Name des Paketes	**Inhalt**
`java.applet`	*Interfaces für Applets und die Klasse Applet* Damit können Programme in einem Webbrowser-Fenster ausgeführt werden.
`java.awt`	*User Interface Design* siehe auch Kapitel 5
`java.beans`	*Komponentenmodell* Enthält Klassen und Interfaces zur Entwicklung von JavaBeans. Diese dienen dem Austausch und der Speicherung von Daten in einem Application-Server, also einem Webserver, der komplexe Anwendungen bereitstellt.
`java.io`	*Ein- und Ausgabe* Schreiben und Lesen von Dateien, Umgang mit Datenströmen etc.
`java.lang`	*Basisklassen für die Programmiersprache* Java Standardmäßig eingebundenes Paket. Es enthält u. a. die Klasse `Object`, von der alle Klassen automatisch erben.
`java.math`	*Mathematische Hilfsmittel* Große Integer- und Realwerte, Formeln, Konstanten (π, e) etc.

[7] Die Version 5.0 wird teilweise auch als Version 1.5.0 bezeichnet (vgl. [Sun06c]).
[8] Link zur API-Spezifikation der JDK Version 5.0: `http://java.sun.com/j2se/1.5.0/docs/api/index.html`

`java.net`	*Netzwerkzugriffe* Stellt Klassen zum Senden und Empfangen von Daten über das Internet bereit.
`java.rmi`	*Remote Method Invocation* Aufbau und Verwaltung von Verbindungen zu anderen virtuellen Maschinen
`java.security`	*Sicherheitsmechanismen* Schlüsselverwaltung, Signaturen, Autorisierung, Protokolle etc.
`java.sql`	*Datenbankzugriff* Enthält Klassen und Interfaces zum Lesen und Schreiben von Daten in oder aus einer relationalen Datenbank.
`java.text`	*Text- bzw. Zeichenkettenbearbeitung* Stringformatierung, Regelinterpreter etc.
`java.util`	*Hilfsklassen* Collections, Kalender, Zeitzonen, Archivierung

Tabelle 3.1. Übersicht über wichtige Pakete des JDK

- `java.awt`: Dieses Paket enthält alle Komponenten, die für das Erstellen einer Benutzerschnittstelle oder zum Darstellen von graphischen Elementen oder Bildern benötigt werden, z. B. Buttons, Textfelder, Fenster oder verschiedene Layouts. Außerdem werden in `java.awt.event` Klassen und Interfaces definiert, die zur Behandlung von Ereignissen dienen, die durch AWT Komponenten ausgelöst werden.
- `java.lang`: In diesem Paket werden grundlegende Klassen für die Programmiersprache JAVA definiert. Eine der wichtigsten Klassen ist `Object`, von der alle anderen Klassen abgeleitet sind. In der abstrakten Klasse `System` sind Standard- Ein- und Ausgaben sowie Fehlermeldungen etc. definiert. In diesem Paket wird auch die Klasse `Exception` definiert, die dazu dient, bestimmte Probleme im Programm aufzuzeigen. Exceptions können durch die Anwendung abgefangen werden. Außerdem sind hier alle Standarddatentypen als Klassen, wie z. B. `Integer` oder `Boolean`, die Klasse `Math` für Berechnungen sowie `Threads`, die Ablaufsteuerung, etc. enthalten.
- `java.util`: Im Paket `java.util` sind eine Menge von nützlichen Klassen und Interfaces definiert, wie z. B. `Calendar`, `Date` etc. Auch

die Klassen `Hashtable` oder `Collection` sowie von ihr abgeleitete Klassen, z. B. die Klasse `Vector`, sind enthalten.

3.5 Werkzeuge

Zu einer Umgebung für die Softwareentwicklung gehören neben den Klassenbibliotheken des vorigen Abschnitts auch Werkzeuge, die das Erstellen von Programmen und deren Ausführung ermöglichen. Im JDK 5.0 von SUN MICROSYSTEMS sind folgende Elemente enthalten:

- *Standard JDK Tools and Utilities*
  - *Basic Tools*: Die Basiswerkzeuge dienen vor allem der Übersetzung von .java in .class Dateien (Compiler javac), dem Ausführen der Programme (Interpreter java) und der Fehlersuche (Debugger jdb).
  - *Security Tools*: Die Werkzeuge zum Sicherheits-Management gestatten das Festlegen von Richtlinien für Programme, die über das Netzwerk kommunizieren.
  - *Internationalization Tools*: Um Programme in mehreren Sprachen zu entwickeln, ist die Konvertierung der Zeichensätze notwendig (Im Englischen existiert z. B. kein „ä“).
  - *Remote Method Invocation (RMI) Tools*: Diese Werkzeuge helfen bei der Erstellung von Programmen, die über das Netzwerk miteinander kommunizieren.
  - *Java IDL and RMI-IIOP Tools*: Diese Werkzeuge werden benötigt um Anwendungen zu entwickeln, die über CORBA (ein Standard zur Kommunikation in Netzwerken) auf Datenbanken zugreifen.
  - *Java Deployment Tools*: Diese Werkzeuge werden in Verbindung mit der Entwicklung von Web-Anwendungen benötigt.
  - *Java Plug-in Tools*: Dieses Werkzeug gestattet die Entwicklung von Plug-ins für Produkte, die eine entsprechende Schnittstelle aufweisen.
  - *Java Web Start Tools*: Diese Werkzeuge werden in Verbindung mit JAVA WEB START benötigt.
- *Experimental JDK Tools and Utilities*
  - *Monitoring and Management Tools*: Diese noch experimentellen Werkzeuge dienen der Überwachung der Leistung und des Ressourcenverbrauchs des JAVA Interpreters (Java Virtual Machine (JVM)).

– *Troubleshooting Tools*: Diese noch experimentellen Werkzeuge dienen der erweiterten Fehlersuche.

Wie ersichtlich wird, ist im JDK kein eigenständiger Editor enthalten. Um Java-Programme zu entwickeln, kann man jedoch auf einen einfachen Texteditor zurückgreifen und die Basiswerkzeuge benutzen.

Alternativ existieren zahlreiche integrierte Entwicklungsumgebungen (Integrated Development Environment, IDE), welche die Arbeit erheblich erleichtern. Diese Umgebungen „verbergen“ den Zugriff auf die Basiswerkzeuge in der Regel hinter grafischen Oberflächen. Beispiele für kostenlos erhältliche IDEs bzw. Trial Versionen sind:

- das Opensource entwickelte Werkzeug *Eclipse* (vgl. [Ecl06]),
- der *JCreator* von Xinox in der Light Edition (vgl. [Xin06]),
- das *Sun Java Studio* von SUN Microsystems (vgl. [Sun06a]),
- *IntelliJ IDEA* von JetBrains in der Trial Version (vgl. [Jet06]),
- der *JBuilder* von Borland in der Enterprise Trial Version (vgl. [Bor06]).

3.6 Applet oder Standalone

Von Java werden grundsätzlich folgende Konstrukte unterstützt, die vor allem den unterschiedlichen Anforderungen der Umgebungen, in denen sie eingesetzt werden, entsprechen sollen:

- *Standalone:* Diese entsprechen am ehesten der „klassischen“ Definition eines Programmes. Zu ihrer Ausführung wird lediglich der Java-Interpreter benötigt. Dieser „sucht“ nach der Klassenmethode `public static void main(String[] args)` (vgl. Abschnitt 2.5.10 und Abschnitt 4.4) und führt diese aus. Standalones unterliegen keinen Sicherheitsbeschränkungen.
- *Applets:* Applets besitzen kein eigenes Hauptfenster und werden deshalb insbesondere in Webseiten eingebunden. Alternativ kann der Appletviewer verwendet werden. Der Zugriff auf das darunterliegende System ist stark eingeschränkt (z. B. kein direkter Dateizugriff).
- *Servlets:* Um die Funktionalität von Webservern zu erweitern, können Servlets eingebunden werden. Das Java Development Kit enthält bereits Beispiele der Anbindung an Server von Apache, Netscape oder Microsoft.
- *Appletcations:* Diese stellen eine Mischform aus Applets und Standalones dar. Sie entscheiden beim Start (im Browser oder direkt) über ihre Eigenschaften und entsprechende Beschränkungen. Werden als Applets mit `main`-Methode implementiert.

3.7 Übungsaufgaben

Aufgabe 6. Erstellen Sie mit einem Texteditor eine Klasse namens `ErsteKlasse`! Übersetzen Sie diese mit Hilfe des Werkzeuges `javac` und erzeugen Sie eine Dokumentation mit Hilfe von `javadoc`!

Aufgabe 7. Welche Aufgaben übernehmen die Klassen, die im Paket `javax.swing.border` enthalten sind? Wie sind Sie an Ihre Informationen gelangt?

Aufgabe 8. Welche Version des JDK ist momentan zum Download bei SUN MICROSYSTEMS verfügbar? Wie ist deren korrekte Bezeichnung?

4

JAVA-Syntax und -Semantik

„Mit der Bedeutung und der Schreibweise von Worten sollte man sehr vorsichtig sein. Um ein Beispiel zu nennen: Der habgierige Serif von Al-Ybi wurde einmal von einer Gottheit mit unzureichenden orthographischen Kenntnissen verflucht. Während der nächsten Tage verwandelten sich alle von ihm berührten Gegenstände in Glod. Zufälligerweise hieß so ein Zwerg, der Hunderte von Kilometern entfernt unter einem Berg lebte und zu seinem großen Verdruss feststellen musste, dass ihn etwas zum Königreich fortzerrte und dort gnadenlos vervielfältigte. Etwa zweitausend Glods später ließ der böse Zauberspruch nach. Bis heute gelten die Bewohner von Al-Ybi als ungewöhnlich klein und mürrisch.“

Terry Pratchetts
„Total verhext“

4.1 Anweisungsblock

Ein Anweisungsblock wird in JAVA durch zwei geschweifte Klammern eingeschlossen. Innerhalb eines solchen Blockes werden die einzelnen Anweisungen durch ein Semikolon getrennt. Die so gebildeten Blöcke werden wie eine Anweisung behandelt und können dementsprechend dazu genutzt werden, alternative Pfade im Programm (beispielsweise innerhalb von Schleifen oder nach Alternativen) mit mehreren Anweisungen zu entwerfen. Anweisungsblöcke selbst werden nicht durch ein Semikolon abgeschlossen. Folgende Abbildung ist demnach nicht ganz korrekt:

```
if (x < 0) {
  berechne(y);
  schreibe(y);
};
```

Zwischen der schließenden geschweiften Klammer und dem darauf folgenden Semikolon entsteht im obigen Beispiel eine *leere Anweisung*. Diese führt zu keiner größeren Konsequenz, außer einer Warnung des

Compilers, die man ignorieren kann. Anders hingegen sieht es in folgendem Beispiel aus:

```
if (x < 0); {
  berechne(y);
  schreibe(y);
}
```

Das Semikolon nach der IF-Anweisung führt dazu, dass im Falle von $x < 0$ eine Leeranweisung ausgeführt wird. Der anschließende Teil des Programmes (also die Berechnung und die anschließende Ausgabe) wird in jedem Fall durchgeführt, und dies dürfte nicht im Sinne des Programmierers gewesen sein. Richtig müssen die Anweisungen also lauten:

```
if (x < 0) {
  berechne(y);
  schreibe(y);
}
```

Anweisungsblöcke können auch ohne eine vorangestellte IF-Anweisung oder Schleife gebildet werden, jedoch haben diese dann keine Bedeutung für den Programmablauf, obwohl ein neuer Namensraum (vgl. Abschnitt 4.2) entsteht.

```
berechne(a);
{
  berechne(b);
  berechne(c);
  berechne(d);
}
schreibe(b, c, a, d);
```

Das Beispiel wird fehlerlos übersetzt, führt aber zum gleichen Ergebnis wie:

```
berechne(a);
berechne(b);
berechne(c);
berechne(d);
schreibe(b, c, a, d);
```

4.2 Bezeichner und deren Namensräume

Objekte, Klassen, Interfaces, Variablen, Methoden und Pakete besitzen in JAVA einen eindeutigen Namen, der *Bezeichner* genannt wird. Bei der Vielzahl von Klassen, die bereits vom JDK zur Verfügung gestellt werden, ist es leicht vorstellbar, dass einige Namen doppelt vergeben wurden. Gerade bei den so genannten Hilfsvariablen (welche z. B. Schleifendurchläufe mitzählen oder ganz allgemein temporär Daten aufnehmen) wird immer wieder ein kleines i oder ein x verwendet. Ein weiteres Beispiel ist die Klasse `List`, die sowohl im Paket `java.awt` als auch im Paket `java.util` definiert wurde, nebenbei mit recht unterschiedlichen Aufgaben.

Aus diesem Grund existieren in JAVA Namensräume, d. h. Bereiche, in denen ein Bezeichner benutzt werden kann. Namensräume werden durch Pakete, Klassen, Methoden und Anweisungsblöcke gebildet. Dabei gilt eine hierarchische Ordnung der Namensräume. In untergeordneten Namensräumen kann der Bezeichner erneut vergeben werden und überschreibt die vorherige Definition. Findet der Compiler die Definition eines Bezeichners nicht im aktuellen Namensraum, wird die Definition aus dem übergeordneten verwendet. Ein Beispiel:

```
int y = 0;
berechne(y);
if (y < 0) {
  berechne(x);
  schreibe(y);
}
```

In der ersten Zeile definieren wir eine Variable y und weisen ihr den Wert 0 zu. Nach der Berechnung `berechne(y)` wird mit der geschweiften Klammer ein neuer Anweisungsblock (und damit Namensraum) geöffnet. Bei der Verwendung von y in `schreibe(y)` wird auf die Definition von y aus dem übergeordneten Raum zurückgegriffen. Folgendes Beispiel wird nicht funktionieren:

```
int y = 0;
berechne(y);
if (y < 0) {
  int x = 10;
  berechne(x);
}
schreibe(x);
```

Die Anweisung `schreibe(x)` verwendet eine Variable x, die in einem untergeordneten Namensraum definiert wurde. Der Compiler wird also seine Arbeit mit dem Hinweis abbrechen, dass er x nicht kennt.

Ein Beispiel für das Überschreiben von Definitionen in untergeordneten Namensräumen liefern folgende Zeilen:

```
int y = 0;
berechne(y);
if (y < 0) {
  String y = "Neudefinition";
  berechne(x);
  schreibe(y);
}
schreibe(y);
```

Die Zeile `String y = ‘‘Neudefinition’’` erzeugt einen neuen Bezeichner y, der nicht mehr vom Typ `int` sondern `String` ist. Das anschließende `schreibe(y)` wird also einen String ausgeben. Die Anweisung `schreibe(y)` nach der geschweiften Klammer wird hingegen wieder auf unsere zuerst definierte Integer-Variable zugreifen.

Das in der Einführung erwähnte Beispiel der Klasse `List` führt hingegen zu keinem Überschreiben eines Bezeichners, da die Namensräume `java.awt` und `java.util` in keiner hierarchischen Beziehung zueinander stehen. Werden beide Namensräume zur Definition einer eigenen Klasse importiert (vgl. Abschnitt 4.3), muss bei der Verwendung der Klasse `List` angegeben werden, aus welchem Paket sie stammt (durch die Angabe des vollen Namens `java.awt.List` oder `java.util.List`).

Innerhalb von Methoden gilt, dass deren Parameter die Definition eines Bezeichners des Objekts überschreiben.

```
class Example {

  int x = 0;

  public void overWrite(Object x) {
    System.out.println("X ist:" + x);
  }

  public void normal() {
    System.out.println("X ist:" + x);
  }
}
```

Innerhalb der Methode `overWrite(Object x)` wird das Objekt x auf dem Bildschirm ausgegeben und innerhalb von `normal()` der mit `int x = 0` definierte Integer-Wert.

4.3 Klassendefinition

Mit dem Schlüsselwort `class` und einem anschließenden Bezeichner (Namen) wird in JAVA die Definition einer Klasse eingeleitet. Der erste Buchstabe des Bezeichners ist dabei groß zu schreiben. Innerhalb des darauf folgenden Anweisungsblocks (also in den geschweiften Klammern) werden die Variablen und Methoden der Klasse definiert. Die einfachste JAVA-Klasse ist also:

```
class EinfachsteKlasse {
}
```

Die Klasse im Beispiel ist bereits, ohne dass man dies beeinflussen kann, eine Subklasse der Klasse `java.lang.Object`, erbt also alle Eigenschaften und Methoden dieser, beispielsweise eine Methode `toString()`, mit der man ein Objekt dieser Klasse in Textform darstellen kann.

Jede Klasse wird in genau einer Datei definiert. Diese erhält die Endung `.java` und wird so vom JAVA-Compiler auch als potentieller Kandidat für die Übersetzung erkannt. Innerhalb dieser Datei kann vor der Definition der Klasse der *Import eines Namensraumes* erfolgen. Notwendig hierfür ist das Schlüsselwort `import`. Nach dieser Anweisung steht entweder der vollständige Name einer Klasse (also auch deren Paket) oder der Name ihres Paketes und ein anschließendes `.*`. Das folgende Beispiel importiert den kompletten Namensraum des Packages `java.awt` und eine Klasse `Vector` aus dem Paket `java.util`.

```
import java.awt.*;
import java.util.Vector;

class SchonKomplexereKlasse { }
```

Genauso wie die Klassen des JDK in verschiedenen Paketen liegen, lassen sich selbst entwickelte Klassen auch Paketen zuordnen. Spätestens bei einem Programm, das mehr als zehn Klassen enthält, wird man zu diesem Hilfsmittel greifen, um die Übersicht zu wahren:

```
package programmiersprache.beispiele;

import java.awt.*;
import java.util.Vector;

class SchoenEinsortierteKlasse {
}
```

Eine Klasse aus dem Paket `programmiermethodik` wird erst mit folgenden Zeilen auf diese Klasse zugreifen können:

```
package programmiermethodik;

import programmiersprache.beispiele.*;

class NochEineSchoenEinsortierteKlasse {
}
```

oder:

```
package programmiermethodik;

import programmiersprache.beispiele.SchoenEinsortierteKlasse;

class NochEineSchoenEinsortierteKlasse {
}
```

Um das Beispiel aus dem Abschnitt 2.5.1 weiterzuführen, wird mit den folgenden Abschnitten eine Klasse `Kuh` definiert, die eine bestimmte Menge Milch (gemessen in Litern) im Euter hat und eine Methode `gibMilch()` besitzt, um auf das Melken zu reagieren. Bis zu diesem Punkt kann als Zwischenergebnis schon einmal festgehalten werden:

```
package programmiersprache.bauernhof;

class Kuh {
}
```

Mit dem Schlüsselwort `implements` können Klassen ein Interface implementieren (vgl. Abschnitt 2.5.7). Eine Anwendung hierfür ist die

Verwendung von Listenern, die Gegenstand vom Abschnitt 5.4 sind. Ein Beispiel:

```
package programmiersprache.bauernhof;

class Kuh implements EinSpeziellesKuhInterface {
}
```

Zusätzlich können Klassen mit bestimmten Zugriffsrechten ausgestattet werden. Diese werden in Abschnitt 4.7 näher erläutert.

4.4 Methoden

Im objektorientierten Paradigma kommunizieren Objekte über Nachrichten miteinander. Das, was geschehen soll, wenn ein Objekt eine bestimmte Botschaft erhält, wird in Methoden beschrieben. Für die Definition von Methoden werden mindestens benötigt:

- der Name der Methode,
- der Typ des Rückgabewerts und
- die Typen und Namen der Parameter.

Der Name der Methode und die Liste der Parameter (als geordnete Folge der Typen und Namen der Parameter) werden als *Signatur* der Methode bezeichnet, da sie die Methode eindeutig identifizieren. Gleichnamige Methoden in einer Klasse werden in der Praxis sehr häufig verwendet, die Liste ihrer Parameter muss sich dabei jedoch unterscheiden. Der erste Buchstabe des Methodennamens ist klein zu schreiben.

Die Anweisungen, die innerhalb einer Methode ausgeführt werden sollen, werden mit einem Anweisungsblock (also geschweiften Klammern) eingeschlossen. Somit setzt sich eine Methode zusammen aus:

```
TypDesRueckgabewertes methodenName(TypDesParameters paramName) {
  anweisung1();
  anweisung2();
  ...
  return rueckgabeWert;
}
```

Müssen einer Methode mehrere Parameter übergeben werden, so werden diese durch Kommata voneinander getrennt. Liefert eine Methode keinen Rückgabewert, so ist hierfür das reservierte Wort `void`

einzusetzen. Die RETURN- Anweisung (vgl. Abschnitt 4.9.7) entfällt in diesem Fall. Besitzt eine Kuh also eine Methode `gibMilch()` und eine Methode für das Fressen von Heu, so müsste unsere Klasse `Kuh` folgendermaßen aussehen:

```
package programmiersprache.bauernhof;

class Kuh {
  int gibMilch() {
    return 1;
  }

  void frissHeu(int kiloHeu) {
  }
}
```

Die runden Klammern nach dem Namen der Methode müssen stets gesetzt werden, auch wenn keine Parameter übergeben werden, da JAVA nur durch die Klammern Methoden von Variablen unterscheiden kann.

Die Definition von Methoden kann nur innerhalb einer Klasse erfolgen, Methodendefinitionen innerhalb (des Rumpfes) einer anderen Methode sind nicht erlaubt. Zusätzlich können Methoden mit bestimmten Zugriffsrechten, wie z. B. `public` oder `private`, ausgestattet werden. Diese werden in Abschnitt 4.7 näher erläutert.

Wird einer Methode eine Objektreferenz übergeben, so wird unter dem Namen des Parameters eine neue Referenz erzeugt, die auf dasselbe Objekt verweist. Somit kann die Referenz innerhalb der Methode verändert werden, ohne dass dies Auswirkungen auf das Objekt hat, welches durch die Objektreferenz (nur indirekt) übergeben wurde.[1] Ein Beispiel:

```
public void justDoIt() {
  String einString = "Hallo!";
  manipuliere(einString);
  System.out.println(einString);
}

private void manipuliere(String zuManipulieren) {
  zuManipulieren = "Huhu!";
}
```

[1] Dieses Prinzip wird auch *call by value* genannt.

Nach dem Aufruf von `justDoIt()` wird auf dem Bildschirm „Hallo!“ stehen. Im Beispiel wird mit dem Aufruf der Methode `manipuliere(...)` die Referenz auf das String-Objekt „Hallo“ in die Variable `zuManipulieren` kopiert. Wird dieser Variable anschließend ein neuer Wert zugewiesen, in diesem Fall eine Referenz auf das String-Objekt „Huhu!“, hat dies keine Auswirkungen auf die Referenz `einString`.[2]

Beim Aufruf einer Methode kann ihr möglicher Rückgabewert ignoriert werden. Der zurückgegebene Wert wird zur weiteren Verarbeitung einfach nirgendwo abgelegt. Sinn macht dies, wenn die Methode Statusmeldungen (z. B. „Aktion war erfolgreich oder nicht“) liefert, diese jedoch für den weiteren Ablauf unerheblich sind. Beispielsweise sind folgende Zeilen alternativ möglich, wenn `vera` ein Objekt der oben definierten Klasse `Kuh` ist:

```
private void melkeVeraZweiMal() {
  /**
   * Rückgabewert der Methode gibMilch() wird in gemolkeneMilch abgelegt
   */
  int gemolkeneMilch = vera.gibMilch();

  /**
   * Rückgabewert der Methode gibMilch() wird ignoriert
   */
  vera.gibMilch();
}
```

Eine besondere Form der Methoden stellen die *statischen* oder auch *Klassenmethoden* dar. Bereits aus ihrem Namen lässt sich schlussfolgern, dass sie nicht zu den Objekten der Klasse, sondern zur Klasse selbst gehören. Eine Klassenmethode, die stets benötigt wird, um ein Java-Programm auszuführen, wurde bereits mit der Methode `public static void main(String[] args)` vorgestellt (vgl. S. 12). Sie wird benötigt, da zur Startzeit des Programms noch gar keine Objekte existieren und somit zwangsläufig die erste Nachricht an eine Klasse und nicht an ein Objekt geschickt werden muss. Innerhalb dieser Methode sollte also das erste Objekt erzeugt werden und diesem die Kontrolle über den weiteren Ablauf übergeben werden.

Um eine Methode als statisch zu definieren, wird das Schlüsselwort `static` vor dem Typ des Rückgabewerts eingefügt:

[2] Ein Objekt der Klasse `String` kann auch nicht mit einer seiner Methoden dazu bewegt werden, den Text, den es speichert, zu verändern.

```
public static void meineStatischeMethode() { }
```

Innerhalb einer statischen Methode können wiederum statische Methoden und statische Variablen der Klasse (d. h. Klassenmethoden und Klassenvariablen) verwendet werden. „Normale“ Methoden und Variablen können innerhalb einer statischen Methode nur unter expliziter Angabe des adressierten bzw. zugehörigen Objekts verwendet werden.

4.5 Variablen, Attribute und Referenzen

Das Verhalten der Objekte wird mit Methoden beschrieben, ihr Zustand mit Daten, die mit Hilfe von *Attributen* gespeichert sind. Attribute stellen eine besondere Form von *Variablen*[3] dar. Variablen, die innerhalb von Methoden definiert werden, sind keine Attribute der Objekte und können nur in der Methode verwendet werden, in der sie definiert wurden. Attribute werden im Gegensatz dazu innerhalb von Klassen und außerhalb von Methoden definiert, können aber innerhalb aller Methoden der Klasse verwendet werden.

Allgemein werden Variablen mit einem Namen versehen und haben einen bestimmten Typ. Dieser legt sowohl den Wertebereich (also alle für die Variable gültigen Werte) als auch die Operationen (z. B. Addition für Zahlen oder die Verkettung für Strings), die auf die Werte möglich sind, fest. Mögliche Standarddatentypen werden im Abschnitt 4.13 eingeführt. Die Definition setzt sich also zusammen aus:

```
TypDerVariable variable;
```

Mit diesem Schritt haben wir eine Variable definiert, jedoch noch keinen Wert zugewiesen. Diesen können wir mit folgenden Möglichkeiten setzen:

- Einer Variable können Konstanten zugewiesen werden:
  `variable = konstante;`
- Einer Variable kann der Wert einer anderen Variable gleichen Typs zugewiesen werden:
  `variable = andereVariable;`
- Einer Variable kann der Rückgabewert gleichen Typs einer Methode zugewiesen werden:
  `variable = methode();`

[3] allgemein: Dinge mit variablem Inhalt

Die Definition und die erste Wertzuweisung kann in einem Schritt erfolgen:

```
int meineVariable = 1;
```

oder auch:

```
int meineVariable = berechne();
```

Eine besondere Art der Variablen bilden die *Referenzen*. Der Wert einer Referenz ist stets ein Objekt (vgl. auch Abschnitt 2.5.9). Nur wenn ein Objekt eine Referenz auf ein anderes Objekt besitzt, ist es in der Lage, ihm eine Nachricht zu schicken. Anderenfalls wüsste es nicht, wer der Empfänger seiner Botschaft sein soll. Der Typ der Referenzen ist die Klasse der Objekte, auf die die Referenz verweisen kann:

```
Kuh kuhVera;
```

Oftmals wird die Definition und die erste Wertzuweisung in einem Schritt vollzogen. Da mit einem Konstruktor (vgl. Abschnitt 2.5.10) Objekte erzeugt werden, sind beispielsweise folgende Zeilen für die Erzeugung einer Kuh denkbar:

```
Kuh kuhAnjaVonDerAlm = new Kuh();
```

Da Referenzen stets Verweise auf Objekte sind, werden bei der Wertzuweisung einer Referenz anhand einer anderen nicht die Objekte kopiert, sondern nur die Verweise auf diese:

```
Kuh kuhAnjaVonDerAlm = new Kuh();
Kuh kuhVera = kuhAnjaVonDerAlm;
```

Die beiden Referenzen verweisen im Anschluss also nicht auf zwei Kühe, sondern auf ein und dieselbe. Füttert man die Kuh mit Hilfe der einen Referenz, so kann man sie entsprechend mit der anderen melken:

```
Kuh kuhAnjaVonDerAlm = new Kuh();
Kuh kuhVera = kuhAnjaVonDerAlm;
```

```
kuhAnjaVonDerAlm.frissHeu(10);
kuhVera.gibMilch();
```

4.6 Instanziierung von Objekten

Im vorangegangenen Abschnitt wurde bereits ein Objekt der Klasse `Kuh` erzeugt und anschließend eine Referenz auf dieses abgelegt. Dieser Vorgang wird als *Instanziierung* bezeichnet. Notwendig dafür ist das reservierte Wort `new` und der Konstruktor (vgl. Abschnitt 2.5.10) einer Klasse. Die Verwendung des Konstruktors kann je nach dessen Definition in der entsprechenden Klasse mit der Übergabe von Parametern (in Analogie zu Methoden, vgl. Abschnitt 4.4) verbunden werden. Beispielsweise werden Schalter (Klasse `Button` aus dem Paket `java.awt`) alternativ mit:

```
new Button();
```

oder:

```
new Button("Text auf dem Schalter");
```

erzeugt. Die Instanziierung muss nicht zwangsläufig mit dem Ablegen einer Referenz verbunden werden. Sinnvoll ist dies nur, wenn dem Objekt später Nachrichten gesendet werden sollen. Für diesen Fall ist notwendig, den Empfänger (also das entsprechende Objekt) zu kennen. Wollen wir den Text auf dem Schalter später noch einmal ändern, sind folgende Zeilen die Lösung:

```
Button meinButton;
meinButton = new Button("Text auf dem Schalter");
meinButton.setLabel("Neuer Text auf dem Schalter");
```

Da bereits der Konstruktor eine Referenz erzeugt, sind folgende Zeilen alternativ verwendbar, jedoch mit der Einschränkung, dass der Text nur einmal geändert werden kann:

```
(new Button("Text")).setLabel("Neuer Text");
```

Die Beispielklasse `Kuh` aus dem Abschnitt 4.4 soll im weiteren Verlauf um einen eigenen Konstruktor erweitert werden. Dieser hat die Aufgabe, den ihm übergebenen Namen im Attribut des Objekts abzulegen:

```
package programmiersprache.bauernhof;

class Kuh {

  String name;

  Kuh(String neuerName) {
    name = neuerName;
  }

  int gibMilch() {
    return 1;
  }

  void frissHeu(int kiloHeu) {
  }
}
```

Konstruktoren unterscheiden sich von Methoden dadurch, dass ihr Name dem der Klasse entspricht (Groß-/Kleinschreibung beachten!) und auf die Angabe des Typs des Rückgabewertes verzichtet werden kann, da dieser ebenfalls durch den Namen der Klasse abgebildet wird. Erzeugen können wir jetzt eine Kuh mit folgenden Zeilen:

```
Kuh kuhAnjaVonDerAlm = new Kuh("Anja von der Alm");
```

Das Objekt wird sich über seine gesamte Lebensdauer den Namen merken. Mit Hilfe einer entsprechenden Methode kann eine so erzeugte Kuh ihren Namen anderen Objekten mitteilen.

4.7 Zugriffskontrolle und Vererbung

Im Abschnitt 2.5.2 wurde erklärt, dass Objekte ihre Eigenschaften und ihr Verhalten vor anderen Objekten hinter einer Kapsel „verstecken“ können. Je nachdem, vor wem etwas verborgen werden soll, existieren in JAVA unterschiedliche *Zugriffsarten*. Diese können bei der Definition von Klassen, Methoden und Attributen verwendet werden.

Zugriffsart	**Bedeutung für Attribute und Methoden**	**Bedeutung für Klassen und Interfaces**
`public`	Andere Objekte besitzen ein uneingeschränktes Zugriffsrecht. Der Zugriff ist somit „öffentlich“.	Die Klasse/das Interface ist „öffentlich“, d. h. die Klasse kann in alle anderen Klassen importiert bzw. das Interface von jeder Klasse implementiert werden (vgl. Abschnitt 4.3).
`protected`	Nur Objekte der selben Klasse (und damit auch das Objekt selbst) sowie Objekte von Subklassen und von Klassen des selben Pakets können auf die so geschützten Attribute und Methoden zugreifen.	Nur die in Abschnitt 4.8 vorgestellten inneren Klassen können als `protected` deklariert werden.
`<ohne>`	Wird die Benennung der Zugriffsart weggelassen, können nur Objekte der selben Klasse sowie Objekte von Klassen des selben Pakets auf die so definierten Attribute und Methoden zugreifen.	Wird die Benennung der Zugriffsart weggelassen, kann die Klasse nur von Klassen des selben Paketes importiert bzw. das Interface nur von Klassen des selben Paketes implementiert werden.
`private`	Nur Objekte der selben Klasse können auf die so geschützten Attribute und Methoden zugreifen. Der Zugriff ist somit „privat“.	Die Klasse/das Interface kann nur von Klassen innerhalb des selben Paketes importiert werden.

Tabelle 4.1. Zugriffsarten und ihre Bedeutung

Die gleichzeitige Verwendung mehrerer Zugriffsarten ergibt keinen Sinn und ist deshalb nicht zulässig. Die Definition der Klasse `Kuh`:

```
package programmiersprache.bauernhof;

public class Kuh {

  private String name;

  Kuh(String neuerName) {
    name = neuerName;
  }

  public int gibMilch() {
    return 1;
  }

  public void frissHeu(int kiloHeu) {
  }

  private void schlafe() {
  }
}
```

führt also zu folgenden Sachverhalten:

- Der Ausdruck `public class Kuh { ... }` definiert eine Klasse `Kuh`, auf die beispielsweise auch eine Klasse aus dem „benachbarten“ Paket `programmiersprache.beispiele` zugreifen kann.
- Die Deklaration `private String name;` lässt nur die Kuh selbst auf ihren Namen zugreifen. Er wird also vor anderen Objekten gekapselt.
- Der Konstruktor `Kuh(String neuerName)` wurde mit keiner Zugriffsart versehen, folglich können sich nur Objekte aus dem Paket `programmiersprache.bauernhof` Kühe mit diesem Konstruktor erzeugen.
- Sowohl das Fressen von Heu (mit `public void frissHeu(int KiloHeu)`) als auch das Geben von Milch (mit `public int gibMilch()`) kann von „außen“ (also von Objekten anderer Klassen) angestoßen werden.
- Mit der Methode `private void schlafe()` kann nur die Kuh sich selbst oder eine andere Kuh zum schlafen auffordern.

Durch die Vererbung können Zugriffsrechte auf Methoden und Attribute verändert werden. Dabei gilt:

- Eine als `private` deklarierte Methode/Attribut kann als `public` oder `protected` Methode/Attribut überschrieben werden. Darüber hinaus kann das Zugriffsrecht durch das Weglassen der Zugriffsart modifiziert werden.
- Eine Methode/Attribut, deren/dessen Zugriffsart unbenannt blieb, kann als `public` oder `protected` Methode/Attribut überschrieben werden.
- Eine als `protected` deklarierte Methode/Attribut kann als `public` Methode/Attribut überschrieben werden.
- Eine als `public` deklarierte Methode/Attribut kann nur als `public` Methode/Attribut überschrieben werden.

Die Syntax für die Vererbung in JAVA besteht zum einen aus dem Schlüsselwort `extends` und zum anderen aus der Angabe der Klasse, von der geerbt wird:

```
package programmiersprache.bauernhof;

public class Cow extends Kuh {
}
```

Dieses Beispiel erzeugt eine Klasse `Cow`, die von der Klasse `Kuh` erbt. Private Methoden und Attribute werden nicht vererbt. Damit schränkt das Zugriffsrecht `private` die generelle Vererbung im objektorientierten Paradigma ein (vgl. Abschnitt 2.5.4). Für ein Objekt der Klasse `Cow` bedeutet dies, dass es sich selbst die Nachricht `schlafe()` nicht schicken kann, es sei denn, die entsprechende Methode wird in dieser Klasse neu definiert.

Um `Cow` an die speziellen Anforderungen einer englischen Kuh anzupassen, können einige der von `Kuh` geerbten Methoden erneut definiert, also überschrieben werden. Beispielsweise könnte aus dem übergebenen Heu in der Methode `frissHeu(int kiloHeu)` weniger Milch produziert werden, jedoch kann das Zugriffsrecht auf die Methode `frissHeu(int kiloHeu)` nicht auf `protected` oder `private` heruntergesetzt werden.

Soll explizit auf die geerbten Methoden oder Attribute innerhalb einer Klasse zugegriffen werden, so wird das Schlüsselwort `super` verwendet. Um in der Methode `frissHeu(int kiloHeu)` die Methode `frissHeu(int kiloHeu)` der Klasse `Kuh` aufzurufen, reicht es nicht ein `frissHeu(kiloHeu)` in die Methode einzufügen. Dies würde zu unerwünschter Rekursion führen (vgl. Abschnitt 6.3.1). Durch das Einfügen der Anweisung `super.frissHeu(kiloHeu)` wird die geerbte Methode aufgerufen. Ein Beispiel:

```
package programmiersprache.bauernhof;

public class Cow extends Kuh {

  public void frissHeu(int kiloHeu) {
    super.frissHeu(kiloHeu);
    produziereWenigMilch();
  }
}
```

4.8 Innere und anonyme Klassen

In den vorangegangenen Abschnitten wurde erläutert, dass eine Klasse pro JAVA-Datei angelegt werden kann. Die Definition mehrerer Klassen in einer Datei ist jedoch nur dann nicht erlaubt, wenn sie folgendermaßen erfolgt:

```
package programmiersprache.bauernhof;

public class Kuh {
}

public class Bauer {
}
```

Der Compiler wird die Übersetzung mit der Fehlermeldung ablehnen, dass er die Definition der Klasse `Bauer` in einer Datei mit dem Namen `Bauer.java` erwartet. Zulässig ist jedoch die Definition einer *inneren Klasse* innerhalb der Definition einer anderen Klasse:

```
package programmiersprache.bauernhof;

public class Kuh {
  public class Bauer {
  }
}
```

Innere Klassen können sowohl dazu dienen, den Code lesbarer zu machen als auch zur völligen Verwirrung beitragen. Eine sinnvolle Verwendung der inneren Klassen liegt bei „kleineren“ Aufgaben (z. B. bei der Umsetzung von Listenern, siehe hierfür Abschnitt 5.4) vor sowie bei der Notwendigkeit des Zugriffs auf als `private` definierte Methoden/Variablen der Klasse, die sie aufnehmen. Folgende Zeilen sind mit der inneren Lösung möglich, nicht jedoch, wenn die Klasse `TextFieldListener` „außerhalb“ definiert werden würde:

```
public class GUI {
  ...
  private void updateButtons() {
    button.setEnabled(false);
  }
  ...
  private class TextFieldListener implements TextListener {
    public void textValueChanged(TextEvent e) {
      updateButtons();
    }
  }
  ...
}
```

Innere Klassen können sowohl als `private` als auch als `protected` deklariert werden. Da innere Klassen jedoch von „außerhalb“ nicht instanziiert werden können, ist lediglich die Entscheidung „privat oder nicht privat“ von Bedeutung.

Zu einer *anonymen Klasse* wird eine innere Klasse dann, wenn sie nicht vollständig definiert wird. Im folgenden Beispiel wird der Methode `addActionListener()` eines `Button` ein Objekt übergeben, dessen Klasse erst mit diesem Schritt der Übergabe definiert wird. Anonym ist die Klasse deshalb, weil sie keinen Bezeichner besitzt und dementsprechend auch keine Zugriffsrechte wie bei „normalen“ inneren Klassen definiert.

```
Button abbruch = new Button(“Mache irgendwas”);
abbruch.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent e) {
      macheIrgendWas();
    }
  }
```

```
);
```

Diese anonyme Klasse – im Beispiel wird sie mit `new ActionListener() {...}` definiert – erbt ihre Eigenschaft von der Klasse `Object` und implementiert das Interface `ActionListener` (vgl. Abschnitt 5.4.1), wobei sich nur Letzteres aus dem verwendeten Konstruktor `new ActionListener()` ableiten lässt (Für die Bedeutung des Schlüsselwortes `implements` vgl. Abschnitt 4.3). Der entstehende Quellcode ist kürzer und für diese „kleineren“ Zwecke deshalb sicherlich effektiver, hat jedoch den Nachteil, dass er schwerer zu lesen ist und sich eigentlich mit der herkömmlichen Definition einer Klasse und der Syntax für die Erzeugung eines Objekts nur sehr schwer erklären lässt.

4.9 Anweisungen

4.9.1 IF-Anweisung

Bei der IF-Anweisung werden in Abhängigkeit von einer Bedingung Anweisungen (oder Anweisungsblöcke) ausgeführt. Die Bedingung selbst besteht aus einem logischen Ausdruck, der einen Boolean-Wert liefert (vgl. Abschnitt 4.13.2), und wird stets mit zwei Klammern eingeschlossen. Nimmt die Bedingung den Wert `true`[4] an, wird der Teil hinter der IF-Anweisung ausgeführt. Optional kann die Anweisung `else` verwendet werden, um eine Anweisung auszuführen, die im anderen Fall (Bedingung ist `false`[5]) ausgeführt wird:

```
if (x != 0) {
  y = y / x;
} else {
  System.out.println("Division durch 0!");
}
```

IF-Anweisungen können geschachtelt werden. Dies kann optional wiederum mit ELSE-Anweisungen verbunden werden, dabei ist die Schachtelung zu beachten:

```
if (x >= 0) {
  if (x > 0) {
```

[4] (engl.) wahr
[5] (engl.) falsch

```
    tuWas();
  }
} else {
  tuWasAnderes();
}
```

Die Methode `tuWasAnderes()` wird im Beispiel ausgeführt, wenn x kleiner 0 ist. Für den Fall, dass x nicht größer als 0 aber größer oder gleich 0 ist (demzufolge also 0 ist), wird keine Anweisung ausgeführt.

4.9.2 SWITCH-Anweisung

Besteht die Forderung, nicht nur zwei Fälle für die Ausführung von Alternativen zu unterscheiden, wird die SWITCH-Anweisung verwendet. Der Ausdruck, welcher zur Unterscheidung der einzelnen Fälle (cases) herangezogen wird, wird als *Selektor* bezeichnet. Dieser muss vom Typ `int`, `byte`, `short` oder `char` sein. Zusätzlich dürfen die Werte der einzelnen cases nicht doppelt auftreten:

```
switch (x) {
  case 0:
       System.out.println("X ist 0");

  case 1:
       System.out.println("X ist 1");

  case 2:
       System.out.println("X ist 2");

  default:
       System.out.println("X ist was anderes");
}
```

Führt man diese Zeilen für einen Wert von $x = 2$ aus, so wird auf dem Bildschirm Folgendes erscheinen:

```
X ist 2
X ist was anderes
```

Alle Anweisungen, deren `case` nach dem zutreffenden `case` liegen, werden ausgeführt. Dieses Verhalten ist in einigen Fällen sicher erwünscht. Damit unsere Beispielzeilen aber das Verhalten zeigen, wel-

ches man auf den ersten Blick vermutet hätte, muss man die SWITCH-Anweisung nach dem zutreffenden `case` abbrechen. Hierzu verwendet man die BREAK-Anweisung (siehe auch Abschnitt 4.9.6):

```
switch (x) {
  case 0:
       System.out.println("X ist 0");
       break;

  case 1:
       System.out.println("X ist 1");
       break;

  case 2:
       System.out.println("X ist 2");
       break;

  default:
       System.out.println("X ist was anderes");
}
```

Diese Zeilen führen mit der IF-Anweisung zu dem gleichen Ergebnis:

```
if (x == 0) {
  System.out.println("X ist 0");
} else {
  if (x == 1) {
    System.out.println("X ist 1");
  } else {
    if (x == 2) {
      System.out.println("X ist 2");
    } else {
      System.out.println("X ist was anderes");
    }
  }
}
```

Ist die Ausführung der selben Anweisung für mehrere alternative Fälle gewünscht, könnte das Beispiel so aussehen:

```
switch (x) {
  case 0:
```

```
  case 1:
      System.out.println("X ist 0 oder 1");
      break;

  case 2:
      System.out.println("X ist 2");
      break;

  default:
      System.out.println("X ist was anderes");
}
```

4.9.3 WHILE-Anweisung

Sollen in JAVA Anweisungen wiederholt werden, stehen 3 Alternativen zur Auswahl:

- WHILE-Anweisung: Diese Anweisung dient der Wiederholung einer Anweisung oder eines Anweisungsblockes, während eine bestimmte Bedingung erfüllt ist. Bereits vor dem ersten Durchlauf wird geprüft, ob die Bedingung erfüllt ist.
- DO-Anweisung: Ähnlich der WHILE-Anweisung wird eine Anweisung oder ein Anweisungsblock solange wiederholt, wie eine Bedingung erfüllt ist. Jedoch wird erst nach dem ersten Durchlauf geprüft, ob dieser noch einmal durchgeführt wird (vgl. Abschnitt 4.9.4).
- FOR-Anweisung: Die Grundfunktion der FOR-Anweisung ist ähnlich der WHILE-Anweisung. Zusätzlich kann jedoch eine Variable bei jedem Durchlauf verändert werden, womit relativ einfach eine Zählschleife mit einer festen Anzahl von Durchläufen erzeugt werden kann (vgl. Abschnitt 4.9.5).

Wird mit der WHILE-Anweisung nur eine Anweisung wiederholt, könnte diese wie folgt aussehen:

```
while (bedingung) {
  anweisung();
}
```

Für einen Anweisungsblock entsteht entsprechend:

```
while (bedingung) {
```

```
  anweisung1();
  anweisung2();
  ...
}
```

4.9.4 DO-Anweisung

Im Gegensatz zur WHILE-Anweisung, wird die Bedingung erst am Ende eines jeden Durchlaufs überprüft:

```
do {
  anweisung();
} while (bedingung);
```

oder für einen Anweisungsblock entsprechend:

```
do {
  anweisung1();
  anweisung2();
  ...
} while (bedingung);
```

4.9.5 FOR-Anweisung

Eine FOR-Anweisung kann wie die WHILE-Schleife Anweisungen beliebig wiederholen. Während eines jeden Durchlaufs kann eine Zählvariable manipuliert werden (z. B. kann ein Integer-Wert immer um 1 erhöht werden, um die Anzahl der Durchläufe zu zählen). Die FOR-Anweisung besteht aus zwei Teilen: dem Kopf und dem Rumpf. Der Kopf der FOR-Anweisung setzt sich zusammen aus:

- dem Zuweisen eines Startwertes für die Zählvariable,
- der Formulierung der Bedingung für einen weiteren Durchlauf (i. d. R. ist die Zählvariable Bestandteil der Bedingung) und
- der Anweisung, welche die Zählvariable in irgendeiner Art in ihrem Wert verändert.

Sollen die Zahlen von 0 bis 9 auf dem Bildschirm ausgegeben werden, kann folgende Schleife verwendet werden:

```
int i;
for (i = 0; i < 10; i = i + 1) {
  System.out.println("Zahl i = " + i);
}
```

Sowohl die Angabe eines Startwertes, einer Bedingung als auch einer Inkrementierungsanweisung ist optional. Werden alle drei weggelassen, wird eine Endlosschleife abgebildet:

```
for (;;) {
  anweisung();
}
```

entspricht also:

```
while (true) {
  anweisung();
}
```

Werden mehrere Zählvariablen mitgeführt, so erfolgt die Zuweisung ihrer Startwerte (und die Inkrementierung dieser) durch eine Trennung mittels Kommata:

```
int i, j;
for (i = 0, j = 1; (i <10) && (j < 20); i = i + 1, j = i/2 + 1) {
  System.out.println("Was auch immer: " + i + ", " + j);
}
```

Selbstverständlich kann (wie überall in JAVA) die erste Wertzuweisung (Startwert) der Zählvariablen mit deren Definition verbunden werden. Es entstehen schleifenlokale Variablen, welche nur innerhalb des Anweisungsblocks nach der FOR-Anweisung definiert sind:

```
for (int i = 0; i < 9; i = i + 1) {
  System.out.println("Zahl i = " + i);
}
```

Mit der Definition mehrerer Zählvariablen und deren erster Wertzuweisung kann der Compiler jedoch auch verwirrt werden, da dem Komma eine doppelte Bedeutung zukommt:

```
int j;
for (int i = 0, j = 1; (i < 9) && (j > 1); i = i + 1) {
  anweisung();
}
```

Der Compiler wird mit der Meldung: *„Variable 'j' is already defined in this method.“* abbrechen.

4.9.6 BREAK/CONTINUE-Anweisung

Mit der Anweisung `break` kann ein Anweisungsblock innerhalb einer SWITCH-, einer WHILE-, einer DO- oder FOR-Anweisung jederzeit verlassen werden. Die Endlosschleife aus dem vorigen Abschnitt kann also mit folgenden Zeilen doch noch verlassen werden:

```
for (;;) {
  anweisung();
  break;
}
```

und entspricht somit in ihrer Funktion der DO-Anweisung. Die Anweisung `continue` kann zu einem „vorzeitigen“ Neustart eines Schleifendurchlaufs verwendet werden. Das folgende Beispiel ist ebenfalls eine Endlosschleife, da der Wert von i nie um 1 erhöht wird:

```
int i = 0;
while (i < 10) {
  continue;
  i = i + 1;
}
```

4.9.7 RETURN-Anweisung

Methoden mit Rückgabewert müssen mindestens eine RETURN-Anweisung enthalten. Hinter einem `return` folgt der Wert, den die Methode zurückgeben soll. Das Beispiel definiert eine Methode, die zwei `int` Werte miteinander addiert und das Ergebnis übergibt:

```
public int addiere(int ersterSummand, int zweiterSummand) {
  return ersterSummand + zweiterSummand;
}
```

4.10 Kommentare

Kommentare werden in Programme eingefügt, um die Zusammenhänge und Abläufe im späteren Ergebnis verständlicher zu gestalten. An jeder Stelle einer Klassendefinition können diese eingefügt werden. Grundsätzlich werden

- ein- und
- mehrzeilige

Kommentare unterschieden. Einzeilige Kommentare dienen für die Dokumentation von einzelnen Programmzeilen:

```
// Dies ist ein einzeiliges Kommentar−Beispiel,
// das die folgende Schleife beschreibt
for (int i = 0; i < 9; i = i + 1) {
  System.out.println("Zahl i = " + i);
}
```

Für die mehrzeilige Variante stehen wiederum zwei Alternativen zur Auswahl:

```
/*
 * Dies ist ein mehrzeiliges Beispiel, bei dem der Kommentar
 * mit einem Slash und einem Stern eingeleitet wird. Beendet
 * wird er mit einem Stern und einem anschließenden Slash.
 */

/**
 * Dies ist ein mehrzeiliges Beispiel (zwei Sterne am Anfang), aus
 * dem das Tool javadoc HTML−Dokumentationen erzeugen kann.
 * Sieht dann genauso aus wie die JDK Dokumentation
 * (API−Spezifikation).
 */
```

Die Kommentarblöcke können nicht ineinander geschachtelt werden. Das erste Auftreten von */ beendet den Kommentar.

4.11 Namenskonventionen

JAVA ist sensibel für die Groß- und Kleinschreibung. Ein einmal vergebener Bezeichner muss in der gleichen Art wiederverwendet werden (`int buchstabe` ist nicht das Gleiche wie `int Buchstabe`). Die Umlaute des deutschen Zeichensatzes können, ohne Probleme für den Compiler zu verursachen, verwendet werden. Wenn Projekte auf unterschiedlichen Plattformen entwickelt werden (z. B. UNIX und Windows), empfiehlt sich jedoch die Umschreibung mit ue, oe etc., da die verwendeten Editoren für die Quelltexte die Sonderzeichen unterschiedlich in den entsprechenden Dateien ablegen.

Bezeichner können mit allen Buchstaben oder den Sonderzeichen $ und _ beginnen. Zahlen dürfen erst nach dem ersten dieser Zeichen verwendet werden.

Grundsätzlich sollten „sprechende Bezeichner“ verwendet werden. Eine Variable, die die Anzahl der Liter Milch in einem Eimer speichert, sollte also wie folgt definiert werden:

```
private int milchImEimer = 0;
```

Die kleinen, einzeln stehenden Buchstaben (z. B. i und j) sind für Schleifenvariablen reserviert. Als `final` und damit als „nicht mehr veränderbar“ deklarierte Konstanten werden in Großbuchstaben verfasst (vgl. auch Abschnitt 4.12):

```
private final int KEY_DOWN = 0;
```

Klassennamen beginnen mit Großbuchstaben, im weiteren Verlauf sollte für die bessere Lesbarkeit wie bei den Variablen und Methoden zwischen Groß- und Kleinschreibung gewechselt werden:

```
public class MeineErsteKlasse {
  public void melkeDieKuh() {
    ...
  }
}
```

4.12 Konstanten und Literale

Auf Grund der strengen Objektorientierung existieren in JAVA keine globalen (überall gültigen) Variablen. Somit gehören auch *Konstanten* (Variablen mit festem Wert - in dem Sinne also keine Variablen mehr) stets zu einer Klasse. Damit ihr Wert nicht verändert werden kann, werden sie als `final` deklariert.

Zur optischen Unterscheidung von „richtigen“ Variablen werden sie, wie bereits in Abschnitt 4.11 erwähnt, groß geschrieben und sofern sie aus mehreren Wörtern bestehen, werden diese durch Unterstriche voneinander getrennt.

```
private final String KUH_NAME = ‘‘Vera’’;
```

Literale sind feste Werte für die in JAVA verfügbaren Standarddatentypen (vgl. Abschnitt 4.13). Denkbar sind Zahlen oder Zeichen(-ketten) wie beispielsweise `true` oder `false`. Sie werden in dem jeweiligen Abschnitt zum entsprechenden Standarddatentyp in Abschnitt 4.13 erläutert.

Im Beispiel wird der Variable `euterIstVoll` vom Typ `boolean` das Literal `true` dieses JAVA-Standarddatentyps zugewiesen.

```
private boolean euterIstVoll;
euterIstVoll = true;
```

Das Schlüsselwort `null` steht für eine Konstante, die einer Objektreferenz eines beliebigen Typs zugewiesen werden kann. Streng genommen ist es daher kein Literal, arbeitet aber wie eine universelle Konstante für Referenzen, die eben auf kein Objekt, sondern auf nichts verweisen.

Im Beispiel verweist die Referenz `haensel` auf ein Objekt, die Referenz `gretel` auf nichts.

```
private Rindvieh haensel = new Rindvieh("Hänsel");
private Rindvieh gretel = null;
```

4.13 Standarddatentypen

4.13.1 Übersicht

Ein *Datenobjekt* ist daran erkennbar, dass es Speicher belegt. In JAVA kann es in drei verschiedenen Formen auftreten:

- als Objekt (im Sinne des objektorientierten Paradigmas),
- als Variable (die auch eine Referenz auf ein Objekt darstellen kann) und
- als Konstante.

Einfache Datenobjekte setzen sich aus einem *Namen*, einem *Wertebereich* und einem aktuellen *Wert* (bei Referenzen das aktuelle Objekt) zusammen.

Jedes Datenobjekt besitzt einen bestimmten *Datenobjekttyp*. Dieser legt u. a. den *Wertebereich*, d. h. die Menge der gültigen Werte, fest. Zusammen mit den auf diesen Wertebereich gültigen Operatoren bildet der Datenobjekttyp den *Datentyp*. Standarddatentypen werden von jeder höheren Programmiersprache zur Verfügung gestellt. In JAVA werden vier Arten unterschieden:

- Logischer Typ,
- Ganzzahltypen,
- Gleitpunkttypen und
- Char.

Eine Sonderstellung nehmen die Felder (Arrays) in JAVA ein. Diese besitzen sowohl Eigenschaften einfacher Datenobjekte als auch die Eigenschaften von „richtigen“ Objekten (vgl. Abschnitt 4.13.6).

4.13.2 Logischer Typ

Mit Variablen vom Typ `boolean` können genau zwei Zustände beschrieben werden: `true` und `false`. Der Wertebereich von `booelan` besteht aus nur diesen zwei Literalen. `Boolean` Werte sind sowohl ein Ergebnis als auch ein möglicher Bestandteil logischer Ausdrücke. Die Operatoren auf `boolean`-Werte sind folgender Tabelle 4.2 zu entnehmen:

Operator	Erklärung
=	*Zuweisungsoperator* z. B.: c = a (c enthält den gleichen Wert wie a)
==	*Überprüfen auf Gleichheit* z. B.: c = a == b (c ist true, wenn a gleich b)
!	*Negation* z. B.: c = !a (c ist true, wenn a false)
!=	*Überprüfen auf Ungleichheit* z. B.: c = a != b (c ist true, wenn a ungleich b)
&	*UND-Verknüpfung* z. B.: c = a & b (c ist true, wenn a und b true)
&&	*UND-Verknüpfung ohne vollständige Auswertung* z. B.: c = a && b (c ist true, wenn a und b true sind; Auswertung wird abgebrochen, wenn a bereits das Ergebnis bestimmt)
|	*ODER-Verknüpfung* z. B.: c = a | b (c ist true, wenn a oder b true ist)
||	*ODER-Verknüpfung ohne vollständige Auswertung* z. B.: c = a || b (c ist true, wenn a oder b true ist; Auswertung wird abgebrochen, wenn a bereits das Ergebnis bestimmt)
∧	*EXCLUSIV-ODER-Verknüpfung* z. B.: c = a ∧ b (c ist true, wenn a ungleich b)

Tabelle 4.2. Operatoren auf den Logischen Typ

4.13.3 Ganzzahltypen

In JAVA unterscheidet man vier unterschiedliche Ganzzahltypen. Ihr Wertebereich ist unterschiedlich, die Operatoren sind gleich. Der unterschiedliche Wertebereich entsteht aus der unterschiedlichen Belegung vom Speicher. Je nach dem wie viele Bits verwendet werden, können die Bereiche unterteilt werden in:

Ganzzahltyp	Wertebereich	Anzahl der Bits
`byte`	−128 bis 127	8
`short`	−32768 bis 32767	16
`int`	−2147483648 bis 2147483647	32
`long`	−9223372036854775808 bis 9223372036854775807	64

Tabelle 4.3. Wertebereiche des Ganzzahltyps

In gewissem Sinne sind auch `char`-Werte Ganzzahltypen, aufgrund einiger Besonderheiten werden sie jedoch im Abschnitt 4.13.5 separat behandelt. Die Operatoren auf Ganzzahltypen sind in Tabelle 4.4 dargestellt.

Operator	Erklärung
=	*Zuweisungsoperator* z. B.: c = a (c enthält den gleichen Wert wie a)
==, !=	*Überprüfen auf Gleichheit/Ungleichheit* z. B.: c = a != b (c ist true wenn a ungleich b ist)
<, >, <=, >=	*Überprüfen auf kleiner als, größer als, kleiner gleich, größer gleich* z. B.: c = a > b (c ist true, wenn a größer als b ist)
-	*Ändern des Vorzeichens* z. B.: c = - a (c hat den gleichen Wert wie a, nur mit umgekehrten Vorzeichen)
+, -, *, /	*Addition, Substraktion, Multiplikation, Division* z. B.: c = a + b (c ist die Summe aus a und b)
%	*Modulo* z. B.: c = a % b (c enthält den ganzzahligen Rest bei der Division a durch b)
++, - -	*Inkrementieren, Dekrementieren* z. B.: a++ (a wird um eine 1 erhöht, entspricht a = a + 1)

Tabelle 4.4. Operatoren auf den Ganzzahltyp

Der Zuweisungsoperator arbeitet „abwärtskompatibel“. Ein `byte`-Wert kann einem `int`-Wert zugewiesen werden, ein `int`-Wert jedoch nur einem `int`- oder `long`-Wert, usw. Einige der Operatoren lassen sich mit dem Zuweisungsoperator verknüpfen. Dadurch entsteht ein kürzerer Quelltext, der jedoch nicht immer besser zu lesen ist. Ein Beispiel:

```
a *= b;
c += a;
```

Überführt man diese Zeilen in „normale“ Zuweisungen ergibt sich:

```
b = b * a;
c = c + a;
```

Zusätzlich existieren noch zahlreiche Operatoren auf Bit-Ebene, die es beispielsweise erlauben, alle Bits in einem Integer-Wert in eine bestimmte Richtung zu verschieben. Zur näheren Beschreibung wird auf [Mid+03], S. 31 verwiesen.

Bei der Verwendung der ganzzahligen Literale können Variablen bereits während des Programmierens feste Werte zugewiesen werden, z. B.:

```
int x = 123;
long z = 34L;
```

Um explizit kenntlich zu machen, dass es sich bei einem Literal um einen `long`-Wert handelt, kann ein kleines oder großes L angefügt werden. Dies wird dann sogar vom Compiler gefordert, wenn die Zahl kleiner als -32768 oder größer als 32767 ist. Bei der Verwendung einer führenden 0 in Literalen ist Vorsicht geboten, weil der anschließende Wert oktal (Basis des Zahlensystems ist die 8 und nicht wie im dezimalen die 10) interpretiert wird:

```
int x = 010;
```

bedeutet also, dass dem x die Zahl 8 zugewiesen wird. Alternativ kann als Basis für das verwendete Zahlensystem auch die 16 (hexadezimales Zahlensystem) gewählt werden, in welchem die Zahl 13 bei-

spielsweise einem D entspricht. Kenntlich gemacht wird dies durch ein vorangestelltes 0x:

```
int x = 0xFF;
```

Die Zahl x ist im Anschluss mit dem Wert 255 belegt.

4.13.4 Gleitpunkttypen

Ähnlich wie bei den Ganzzahltypen werden auch bei den Gleitpunkttypen unterschiedliche Arten aufgrund ihres Wertebereichs unterschieden:

Gleitpunkttyp	Wertebereich	Anzahl der Bits
`float`	$\pm 1.4013 \cdot 10^{-45}$ bis $3.4028 \cdot 10^{38}$	32
`double`	$\pm 4.9407 \cdot 10^{-234}$ bis $1.7977 \cdot 10^{308}$	64

Tabelle 4.5. Wertebereiche des Gleitpunkttyps

Die Literale für Gleitpunkttypen werden mit einem großen oder kleinen 'F' für einen `float`-Wert bzw. mit einem großen oder kleinen 'D' für einen `double`-Wert versehen. Standardmäßig wird ein Wert der Form $x.xxx$ als `double` interpretiert, weshalb für `float`-Literale stets angegeben werden muss:

```
float f = 0.23f;
```

Bitweise Operationen sind für Gleitpunktzahlen nicht möglich, weshalb die Übersicht in Tabelle 4.6 im Gegensatz zu den Ganzzahltypen vollständig ist.

Mit dem Zuweisungsoperator können `float`-Werte einem `double`-Wert (aber nicht umgekehrt) zugewiesen werden.

Operator	Erklärung
=	*Zuweisungsoperator* z. B.: c = a (c enthält den gleichen Wert wie a)

==, !=	*Überprüfen auf Gleichheit/Ungleichheit* z. B.: c = a != b (c ist true wenn a ungleich b ist)
<, >, <=, >=	*Überprüfen auf kleiner als, größer als, kleiner gleich, größer gleich* z. B.: c = a > b (c ist true, wenn a größer als b ist)
-	*Ändern des Vorzeichens* z. B.: c = - a (c hat den gleichen Wert wie a mit umgekehrten Vorzeichen)
+, -, *, /	*Addition, Subtraktion, Multiplikation, Division* z. B.: c = a + b (c ist die Summe aus a und b)
%	*Modulo* z. B.: c = a % b (c enthält den Rest bei der Division von a durch den ganzzahligen Anteil von b; c = 0.23 % 1.34 liefert das etwas erklärungsbedürftige Ergebnis 0.23 für c)
++, - -	*Inkrementieren, Dekrementieren* z. B.: a++ (a wird um eine 1 erhöht, entspricht a = a + 1)

Tabelle 4.6. Operatoren auf den Gleitpunkttyp

4.13.5 Char

Character (`char`) dienen dem Ablegen einzelner Zeichen. Innerhalb des Unicode-Zeichensatzes sind 65536 Symbole möglich, dementsprechend kann einem `char`-Wert ein entsprechender `int`-Wert zugewiesen werden. Bei der Ausgabe auf dem Bildschirm wird dieser als ein String mit nur einem Zeichen behandelt. Folgenden Zeilen führen zu einem '*A*' in der JAVA-Konsole:

```
char a = 65;
System.out.println(a);
```

Da `char`-Werte einen Unterbereich der `int`-Werte (vgl. dazu Abschnitt 4.13.3) darstellen, sind alle Operationen dieses Typs anwendbar. Dies geht sogar soweit, dass `char`- und `int`-Werte mit den Operatoren verknüpft werden können. Ist einer der Operanden vom Typ `int`, so ist das Ergebnis ein `int`-Wert. Das Beispiel

```
char d = 'd' + 'f';
System.out.println(d);
```

führt zu der Bildschirmausgabe 'Ê', welches aus der Summe der Unicodes von 'd' und 'f' entsteht. Welchen Sinn folgende Zeilen ergeben, ist fraglich, compiliert werden sie:

```
int d = 'd' + 1;
System.out.println(d);
```

Ausgegeben wird '101'. Sinnvoller hingegen ist die Verwendung der `char`-Literale. Möglich sind neben den einzelnen Zeichen in Hochkommas Escape-Sequenzen und Unicode-Sequenzen. Letztere haben folgendes Format:

Unicode	**Ausgabe**
'u0065'	ergibt ein kleines e
'u002F'	ergibt ein ?

Tabelle 4.7. Beispiel für Unicodes

Für Unicodes, die keinem Zeichen im verwendeten Zeichensatz entsprechen, wird der Unicode ausgegeben. Somit führt:

```
char z = '\u001f';
System.out.println(z);
```

zur Ausgabe von '\u001f'. Einige der möglichen Escape-Sequenzen für `char`-Werte sind beispielsweise:

Sequenz	**Bedeutung**
\b	Rueckschritt (Delete)
\t	Tabulator
\n	Zeilenumbruch (Enter)
\'	Hochkomma

Tabelle 4.8. Beispiel für Escape-Sequenzen

4.13.6 Felder

Felder bestehen stets aus einer Menge von eindeutigen Schlüsseln (keiner darf doppelt auftreten), denen ein bestimmter Wert zugewiesen wird. Dargestellt werden kann dieser Zusammenhang in Form einer Tabelle, wie beispielsweise in Tabelle 4.9.

Schlüssel	**Wert**
0	10
1	20
2	3
3	5
4	17
5	0

Tabelle 4.9. Tabellarische Darstellung eines Feldes

Die Syntax von JAVA schränkt diese Felder auf vier Arten ein:

- Die Schlüssel sind natürliche Zahlen.
- Das erste Element hat den Schlüssel 0.
- Die Schlüssel haben einen Abstand von 1, größere Abstände sind nicht möglich.
- Als Wertebereich der Elemente ist nur jeweils ein Datentyp möglich, also entweder `int`, `float` etc. oder eine Klasse.

Felder können auf zwei Arten angelegt werden, wobei eine Änderung der Größe des Feldes später nicht mehr möglich ist. Zum einen können die Elemente bei der Initialisierung aufgeführt werden:

```
int[] array = {10, 20, 3, 5, 17, 0};
```

Und zum anderen kann das Feld mit einem Konstruktor und der Größenangabe initialisiert werden:

```
int[] array = new int[6];
```

In jedem Fall liegt in der Variable `array` nur die *Referenz* auf ein Feld. Ähnlich wie bei den Referenzen auf ein Objekt, kopieren folgende Zeilen nicht das Array mit all seinen Elementen, sondern nur die Referenz auf dieses:

```
int[] array = {10, 20, 3, 5, 17, 0};
int[] arrayKopie = array;
```

Soll nicht nur eine neue Referenz auf das Feld angelegt werden, sondern mit allen Einträgen, die es beinhaltet, kopiert werden, so ist die Methode `System.arraycopy(quelle, index_q, ziel, index_z, count)` zu nutzen. Es müssen also auch die Anfangspositionen und die Anzahl der zu kopierenden Elemente angegeben werden (in diesem Fall jeweils eine 0 und die Anzahl 6):

```
int[] array = {10, 20, 3, 5, 17, 0};
int[] arrayKopie = new int[6];
System.arraycopy(array, 0, arrayKopie, 0, 6);
```

Um auf einzelne Elemente eines Feldes zuzugreifen, ist die Angabe des jeweiligen Schlüssels notwendig. Liegt dieser außerhalb des Gültigkeitsbereiches (z. B. $i < 0$ oder $i > 5$), so kommt es zu einer Ausnahme (Exception) der Klasse `ArrayIndexOutOfBoundsException` und somit zu einer Fehlermeldung des Systems zur Laufzeit. Der Lese- und Schreibzugriff stellt sich wie folgt dar:

```
int[] array = {
   10, 20, 3, 5, 17, 0
 };
 int x = array[2];
 array[5] = 1;
```

Im ersten Schritt wird die Variable x mit dem Wert 3 initialisiert (Zählung beginnt bei 0). Im zweiten Schritt wird dem 6. Wert anstelle der 0 eine 1 zugewiesen.

Neben der Möglichkeit Standarddatentypen wie `int` oder `float` abzulegen, können Arrays auch aus Referenzen auf Objekte definiert werden. In den folgenden Zeilen wird ein Array für drei Dialoge angelegt:

```
Dialog[] dialoge = new Dialog[3];
dialoge[0] = new Dialog(null, ‘‘Erster Dialog’’);
dialoge[1] = new Dialog(null, ‘‘Zweiter Dialog’’);
dialoge[2] = new Dialog(null, ‘‘Dritter Dialog’’);
```

Felder können mehrere Ebenen der Schachtelung besitzen. Beispielsweise soll ein Feld `kundenKartei` einen Index von A bis Z (entspricht den Zahlen 0 bis 23) enthalten. Unter den Indizes liegt jeweils wieder ein String-Feld, welches die wirklichen Namen abspeichert:

```
String[][] kundenKartei = {
  {‘‘Albert’’, ‘‘Ahmeln’’, ‘‘Ahrend’’}, ... , {‘‘Scheuber’’, ‘‘Spilles’’}, ... ,
  {‘‘Veljanov’’}, ... , {‘‘Zuchard’’}
};
```

Anders als z. B. in der Programmiersprache Pascal, handelt es sich hierbei nicht um Dimensionen, da die einzelnen Ebenen der Verschachtelung nicht die gleiche Größe besitzen müssen.

Die Größe eines Feldes lässt sich mittels `arrayName.length` ermitteln. Folgendes Beispiel definiert ein Array aus Kundennamen und gibt sie auf dem Bildschirm aus:

```
String[] kundennamen = {‘‘Scheuber’’, ‘‘Spilles’’, ‘‘Veljanov’’};
  for (int i = 0; i < kundennamen.length; i++) {
    System.out.println("Name an Position’’ + i + ‘‘: ‘‘ + kundennamen[i]);
}
```

Bereits in der Einleitung zu diesem Abschnitt wurde auf die Rolle der Felder (Arrays) als Objekte und als einfache Datenobjekte hingewiesen. Für ihre Eigenschaft als Objekt spricht:

- Arrays können mit einem Konstruktor erzeugt werden.
- Die Überprüfung `array instanceof Object` liefert `true`. Arrays sind also Instanzen der Klasse `Object`.
- Arrays können somit Methoden als `Object`-Parameter übergeben werden.
- Es können mehrere Referenzen auf ein Feld existieren.

Dagegen spricht:

- Arrays können nicht in Form einer Klassendefinition beschrieben werden.

- Von Arrays können keine Klassen abgeleitet werden. Es können also keine Methoden oder zusätzliche Variablen definiert werden.
- `new` ist in diesem Sinne also kein Konstruktor-Aufruf, sondern nur „zufällig“ das gleiche Schlüsselwort für die Erzeugung von Feldern.

5

Gestaltung grafischer Oberflächen

„Bill Gates wäre in Deutschland schon deshalb gescheitert, weil nach der Baunutzungsordnung in einer Garage keine Fenster drin sein dürfen."

Jürgen Rüttgers
dt. Jurist u. Politiker, geb. 1951

5.1 Motivation

Zur Kommunikation mit dem Benutzer müssen Programme eine Schnittstelle auf dem Bildschirm, eine so genannte *Benutzungsoberfläche*, zur Verfügung stellen. Zwei unterschiedliche Modi sind für die Darstellung möglich:

1. *Textorientiert:* In diesem Modus besteht der Bildschirm aus einer Menge von Zeichen. In der Regel sind hier 80x24 Buchstaben, Zahlen und Sonderzeichen gleichzeitig darstellbar. Der Textmodus hat den Vorteil, dass er nahezu auf jedem Rechner in irgendeiner Form verfügbar ist.
2. *Grafische Oberfläche (engl. Graphical User Interface - GUI):* Ist dieser Modus selektiert, können auf dem Bildschirm einzelne Bildpunkte angesteuert werden, d. h. neben den bereits im Textmodus möglichen Zeichen können Elemente wie Linien, Kreise usw. dargestellt werden. Somit ist dieses Verfahren das wesentlich flexiblere und gerade im Umgang mit „Heimanwendern" von Vorteil, da Interaktionselemente wie Schalter, Fenster oder auch Icons verwendet werden können.

Vom JDK werden beide Modi unterstützt. Zum einen ist es möglich, in einer *Console* Aus- und Eingaben durchzuführen, und zum anderen können so genannte *GUI-Elemente* verwendet werden, um mit dem Benutzer zu interagieren.

Für Ersteres steht die Klasse `System` mit den Klassenvariablen `System.out` zur Ausgabe sowie `System.in` zum Lesen von Tastatureingaben zur Verfügung. Die wichtigsten Methoden von `out` lau-

ten `print(String s)` und `println(String s)`. Die Klassenvariable `in` liest mit `read(byte[] b)` ein Array aus Bytes.

Mit folgenden Zeilen lassen sich maximal 100 Zeichen von der Tastatur lesen und anschließend ausgeben:

```
byte[] toRead = new byte[100];
System.out.print("Bitte geben Sie einen Text ein: ");
try {
  System.in.read(toRead);
} catch (Exception e) {
  System.out.println("Fehler beim Lesen!");
}
System.out.println(new String(toRead));
```

Neben dem Umstand, dass gerade durch den Umweg über das Byte-Array `toRead` das Verfahren nicht sehr komfortabel wirkt, sind die Möglichkeiten in der Console sehr eingeschränkt. Weder komplexe Interaktionen (z. B. mit einer Maus) noch einfache Möglichkeiten der Gestaltung (z. B. kein Löschen des Bildschirms möglich) sind mittels der Console umsetzbar. Dementsprechend gering wäre die Akzeptanz des Benutzers.

Im Gegensatz dazu sind grafische Oberflächen und für diese entwickelte Fenstersysteme geeignet für:

- eine optimale Nutzung eines limitierten Bildschirmbereiches,
- mehrere Sichten (z. B. in Dialogen) des Nutzers auf ein Problem,
- parallele Nutzung der Sichten (z. B. können mehrere Programme in unterschiedlichen Fenstern laufen und sind jederzeit „erreichbar“) und
- eine bequeme Steuerung durch den Benutzer (z. B. durch Mauseingaben).

Vom JDK werden Komponenten innerhalb des *Abstract Window Toolkit (AWT)* im Paket `java.awt` zur Verfügung gestellt, damit nicht für jedes neu zu entwickelnde Programm aus der einfachen Möglichkeit einzelne Punkte darzustellen und der Anforderung komplexe Dialogelemente einzubinden, ein enorm hoher Aufwand bei der Programmierung entsteht. Demzufolge werden sich die folgenden Abschnitte dieses Kapitels mit dem Aufbau und Begriffssystem des AWT, den Möglichkeiten der Gestaltung und Interaktion beschäftigen.

5.2 Aufbau und Anwendung des AWT

Das *Abstract Window Toolkit (AWT)* ist, wie bereits erwähnt, eines der Pakete im JDK (vgl. Abschnitt 3.4). Es liefert, basierend auf den Klassen `Component` und `MenuComponent`, zahlreiche Klassen zur Darstellung von Informationen in Fenstern und zur Eingabe durch den Benutzer über die Tastatur (z. B. in Eingabefelder) oder Maus (z. B. Schalter).

Alternativ existiert seit dem JDK 1.2 das Paket `javax.swing`, welches die Elemente „schöner“ darstellt und komplexere Strukturen, wie Tabellen oder vorgefertigte Dialoge, zur Verfügung stellt. Auf dieses Paket wird an dieser Stelle jedoch nicht weiter eingegangen, da die Beziehungen zwischen den dortigen Klassen auf den Beziehungen im AWT basieren und diese erweitern. Einen guten Einstiegspunkt zum Umgang mit den Swing-Klassen bietet [Sun05].

Eine Auswahl wichtiger Klassen des AWT stellt Abbildung 5.1 dar (zur Notation des Diagramms vgl. Abschnitt 6.2.1).

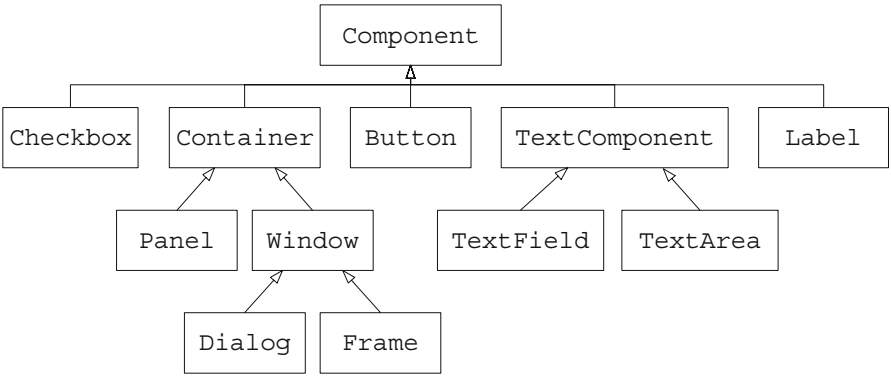


Abb. 5.1. Ausgewählte Klassen des AWT und ihre Vererbungsbeziehungen

Die Basisklasse `Component` stellt ihren Nachfahren grundlegende Methoden und ein einfaches „Standardverhalten“ zur Verfügung. Ein Überblick wird mit der Tabelle 5.1 gegeben.

Die meisten Methoden davon werden in den erbenden Klassen überschrieben, damit diese sowohl ihr spezielles Verhalten bei Benutzereingaben als auch ihre spezielle Form der Darstellung umsetzen können.

Methode	Verhalten
`setVisible(boolean b)`	*Verbergen oder Zeigen einer Komponente* Diese Methode verbirgt ein Objekt der Klasse Component, wenn als Argument `false` übergeben wird und zeigt sie an, wenn `true` als Parameter verwendet wird (z. B. schließt `meinFenster.setVisible(false)` das Fenster `meinFenster`).
`paint(Graphics g)`	*Darstellung auf dem Bildschirm* Diese Methode wird von allen erbenden Klassen überschrieben, um ihr typisches Aussehen zu erlangen (z. B. sind Schalter rechteckig und haben einen Text in der Mitte).
`setBounds(Rectangle r)`	*Verändern der Größe und Position* Jedes Element kann in seiner Größe und Position innerhalb eines Containers (siehe weiter unten in diesem Abschnitt) verändert werden. In der Regel übernehmen Layoutmanager (vgl. Abschnitt 5.3) diese Aufgabe.
`processEvent(AWTEvent e)`	*Reaktion auf Benutzereingaben* Jedes Element, das von `Component` erbt, kann bereits mit dem Benutzer mittels Objekten der Klasse `Event` interagieren. Auch hier gilt, dass jede Klasse ihr typisches Verhalten durch Überschreiben der Methode (vgl. Abschnitt 2.5.6) hinzufügt. Wie Reaktionen auf diese Aktionen definiert werden, wird im Abschnitt 5.4 erläutert.

Tabelle 5.1. Methoden der Klasse `Component`

Die Klasse `Container` ist ein direkter Nachfahre der Klasse `Component`. Container-Objekte (wie z. B. `java.awt.Frame`) können Objekte der Klasse `Component` oder Objekte von Klassen, die von `Component` erben (wie z. B. `java.awt.Button`) „in sich aufnehmen“ und verwalten. Wenn ein solcher Container unsichtbar wird, so „versteckt“ er

auch alle seine eingefügten Objekte. Wird er wieder sichtbar, so werden auch alle eingefügten Objekte wieder dargestellt. Da Container selbst auch von `Component` erben, lassen sich diese problemlos ineinander verschachteln.

Abbildung 5.2 stellt den Aufbau eines Dialogs (als von `Container` erbendes Beispiel) dar.

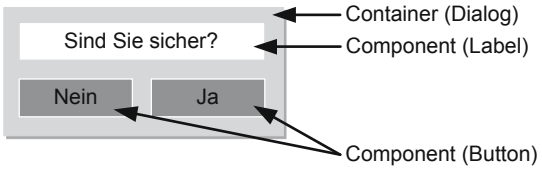


Abb. 5.2. Beispiel für einen Container

Methode	**Verhalten**
`add(Component comp)`	*Einfügen einer Komponente*
`remove(Component comp)`	*Entfernen einer Komponente*
`setLayout(LayoutManager mgr)`	*LayoutManager setzen* Layoutmanager werden im Abschnitt 5.3 näher vorgestellt.

Tabelle 5.2. Methoden der Klasse `Container`

Zu einer näheren Beschreibung aller Klassen des AWT wird auf die API-Spezifikation des JDK verwiesen, die beispielsweise bei SUN Microsystems unter der URL `http://java.sun.com/j2se/1.5.0/docs/api/index.html`, für die z. Zt. aktuelle Version 5.0 abrufbar ist.

5.3 Layouts

Elemente einer grafischen Oberfläche besitzen i. d. R. auf jedem System (z. B. Solaris, Linux oder Windows) eine andere Form der Darstellung (z. B. andere Schriftarten und Größen). Für den Programmierer entsteht

ein hoher Arbeitsaufwand, wenn er jede Komponente einer Oberfläche exakt mit Position und Größe und für jedes Betriebssystem definieren müsste.

Die Aufgabe der Anordnung und der Definition der Größen von Komponenten[1] kann von *Layoutmanagern*, d. h. Objekten von Klassen, die das Interface `LayoutManager` bzw. `LayoutManager2` implementieren, übernommen werden (vgl. Abbildung 5.3; zur Notation des Diagramms vgl. Abschnitt 6.2.1).

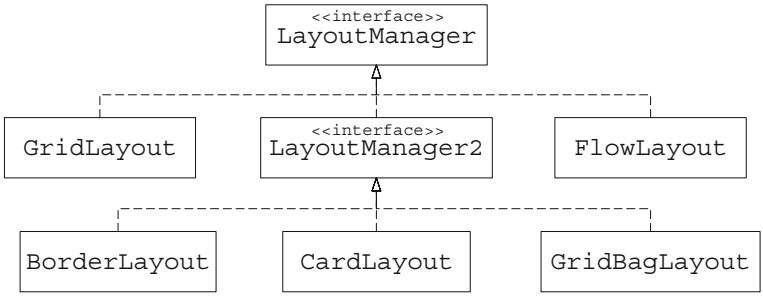


Abb. 5.3. Ausgewählte Layoutmanager des AWT

Jedes Objekt der Klasse `Container` und jedes Objekt von einer ihrer Subklassen (vgl. Abbildung 5.1 auf Seite 81) kann ein solches Objekt besitzen (wird mit der Methode `setLayout(LayoutManager mgr)` gesetzt). Wird kein Layoutmanager gesetzt, ist standardmäßig das `BorderLayout` (vgl. Abschnitt 5.3.3) zugewiesen.

5.3.1 Layoutmanager im AWT

Folgende Zeilen erzeugen ein Fenster `fensterchen`, weisen diesem einen Layoutmanager mittels `setLayout()` zu und fügen die Elemente `komponente1` und `komponente2` hinzu. Mit der Methode `pack()` wird das `fensterchen` dazu aufgefordert, seine untergeordneten Komponenten anzuordnen. Dieses überträgt die Aufgabe anschließend an seinen Layoutmanager.

Jeder Layoutmanager wird anhand der bevorzugten Größen der Dialogelemente diese anordnen und gegebenenfalls in ihrer Größe anpassen. Zu diesem Zweck besitzen alle Objekte, deren Klassen direkt

[1] Das sind Objekte der Klasse `Component` und Objekte von ihren Subklassen (vgl. Abbildung 5.1 auf Seite 81).

oder indirekt von `Component` erben (also z. B. Textfelder), die Methode `getPreferredSize()`, welche die von der Komponente gewünschte Größe liefert.

```
// Anlegen des Fensters
Frame fensterchen = new Frame();
// LayoutManager setzen – Diese Zeile muss angepasst werden!
fensterchen.setLayout(new LayoutManager());
// Komponenten hinzufügen
fensterchen.add(komponente1); fensterchen.add(komponente2);
...
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

In den folgenden Abschnitten werden die Layoutmanager aus Abbildung 5.3 vorgestellt.

5.3.2 FlowLayout

Ein spezieller Vertreter der Layoutmanager ist die Klasse `FlowLayout`. Ihre Objekte ordnen Dialogelemente „fließend“ an, d. h. sie werden von links nach rechts im Container eingeordnet.

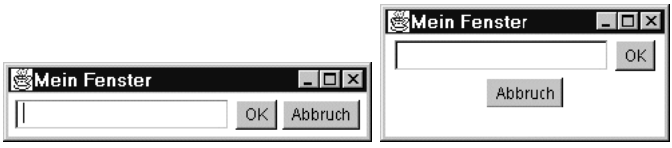


Abb. 5.4. Beispiel für ein FlowLayout und Verhalten bei Größenänderung

```
// Anlegen des Fensters, Titel setzen und Layout festlegen
Frame fensterchen = new Frame();
fensterchen.setTitle("Mein Fenster");
fensterchen.setLayout(new FlowLayout());
// Komponenten einfügen
fensterchen.add(new TextField(20));
fensterchen.add(new Button("OK"));
fensterchen.add(new Button("Abbruch"));
```

```
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

Nachfolgend werden die Elemente rechtsbündig angeordnet, so dass sie einen vertikalen Abstand von 10 Pixel und einen horizontalen Abstand von 50 Pixel haben:

Abb. 5.5. Beispiel für ein FlowLayout mit anderen Parametern

```
// Anlegen des Fensters, Titel setzen und Layout festlegen
Frame fensterchen = new Frame();
fensterchen.setTitle("Mein Fenster");
fensterchen.setLayout(new
FlowLayout(FlowLayout.RIGHT,50, 10));
// Komponenten einfügen
fensterchen.add(new TextField(20));
fensterchen.add(new Button("OK"));
fensterchen.add(new Button("Abbruch"));
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

5.3.3 BorderLayout

Grundlage der Klasse `BorderLayout` ist die Anordnung nach Himmelsrichtungen, wobei von einer Komponente als `CENTER` in der Mitte ausgegangen wird und in den Richtungen `NORTH`, `SOUTH`, `EAST` und `WEST` jeweils eine weitere Komponente eingefügt werden kann.

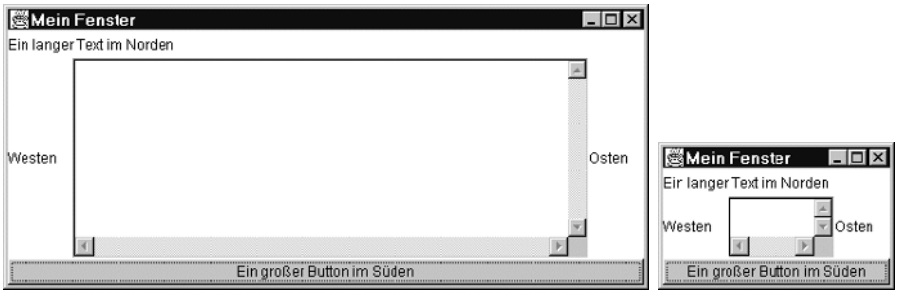


Abb. 5.6. Beispiel für ein BorderLayout und Verhalten bei Größenänderung

```
// Anlegen des Fensters, Titel setzen und Layout festlegen
Frame fensterchen = new Frame();
fensterchen.setTitle("Mein Fenster");
fensterchen.setLayout(new BorderLayout());
// Komponenten einfügen an bestimmter Stelle
fensterchen.add(BorderLayout.NORTH,
                new Label("Ein langer Text im Norden"));
fensterchen.add(BorderLayout.WEST, new Label("Westen"));
fensterchen.add(BorderLayout.EAST, new Label("Osten"));
fensterchen.add(BorderLayout.SOUTH,
                new Button("Ein grosser Button im Sueden"));
fensterchen.add(BorderLayout.CENTER, new TextArea());
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

5.3.4 GridLayout

Ein `GridLayout` ordnet Dialogelemente tabellenförmig an. Dabei ist die Anzahl der Spalten und Zeilen dem Konstruktor zu übergeben. Die später hinzugefügten Elemente werden von links nach rechts angeordnet. Ist eine Zeile „voll“, wird in der nächsten die Arbeit fortgesetzt.

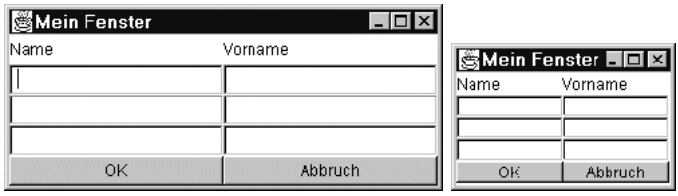


Abb. 5.7. Beispiel für ein GridLayout und Verhalten bei Größenänderung

```
// Anlegen des Fensters, Titel setzen und Layout festlegen
Frame fensterchen = new Frame();
fensterchen.setTitle("Mein Fenster");
fensterchen.setLayout(new GridLayout(5, 2));
// Komponenten einfügen. Reihenfolge bestimmt die Stelle
fensterchen.add(new Label("Name"));
fensterchen.add(new Label("Vorname"));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new Button("OK"));
fensterchen.add(new Button("Abbruch"));
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

Im Anschluss werden auch die Elemente des Gridlayouts rechtsbündig angeordnet, so dass sie einen vertikalen Abstand von 10 Pixel und einen horizontalen Abstand von 50 Pixel haben:

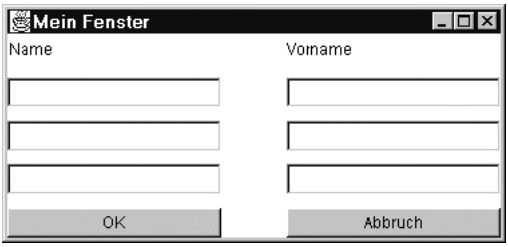


Abb. 5.8. Beispiel für ein GridLayout mit anderen Parametern

```
// Anlegen des Fensters, Titel setzen und Layout festlegen
Frame fensterchen = new Frame();
fensterchen.setTitle("Mein Fenster");
fensterchen.setLayout(new GridLayout(5, 2, 50 , 10));
// Komponenten einfügen. Reihenfolge bestimmt die Stelle
fensterchen.add(new Label("Name"));
fensterchen.add(new Label("Vorname"));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new TextField(20));
fensterchen.add(new Button("OK"));
fensterchen.add(new Button("Abbruch"));
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

5.3.5 GridBagLayout

Ein `GridBagLayout` stellt die Erweiterung zum `GridLayout` dar. Dieses ermöglicht innerhalb der Tabelle die Gewichtung von Zellen (z. B. kann eine Komponente damit stets doppelt so breit wie ihr Nachbar dargestellt werden) und eine spalten- oder zeilenweise Zusammenfassung von Zellen (z. B. kann ein Schalter drei senkrechte Zellen „überspannen“, während daneben drei Objekte der Klasse `Label` untereinander stehen). Konfiguriert wird die Gestalt der einzelnen Zellen über jeweils ein Objekt der Klasse `GridBagConstraint`.

Somit ist diese Klasse auch wesentlich komplizierter anzuwenden. Bereits das in der Dokumentation des JDK angegebene Beispiel, welches 10 Buttons in einen Frame einfügt, besteht aus ca. 50 Zeilen Quellcode.

Abb. 5.9. Beispiel für ein GridBagLayout

5.3.6 CardLayout

Das `CardLayout` dient dem Einfügen von Komponenten in einen übergeordneten Container. Somit ist es am sinnvollsten durch eine Kombination der Panels und ihrer Layouts einsetzbar. Jedes einzelne Panel entspricht einer Karte, wobei immer nur die oberste sichtbar ist.

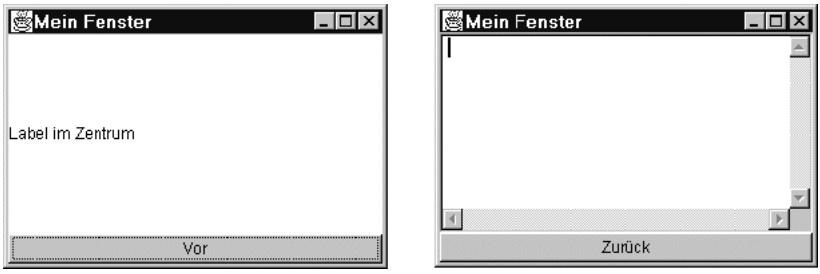


Abb. 5.10. Beispiel für ein CardLayout

Das „Umschalten“ dieser Karten erfolgt nicht automatisch beim Klicken auf den Schalter „Vor“ oder „Zurück“. Vielmehr müssen beide Buttons einen `ActionListener` besitzen, der diese Aufgabe übernimmt (vgl. dazu Abschnitt 5.4.1). Ein mit folgenden Zeilen erzeugtes Fenster:

```
// Layout erzeugen und einfügen
CardLayout cl = new CardLayout();
fensterchen.setLayout(new CardLayout());
...
```

```
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

kann seine „Karten“ mit den Zeilen:

```
cl.show(fensterchen, "Erstes Panel");
cl.next(fensterchen);
cl.last(fensterchen);
```

weiterschalten.

5.3.7 Schachtelung von Layouts

Die Nutzung eines einzigen Layouts ist bei der Gestaltung grafischer Oberflächen häufig unzureichend[2], daher lassen sich Layouts verschachteln. Hierfür werden Objekte, deren Klassen von `Container` erben, „ineinander“ eingefügt. Beispielsweise kann ein Panel, welches zwei Schalter (Buttons) enthält, in einen Frame gelegt werden, wie es in Abbildung 5.11 dargestellt wird.

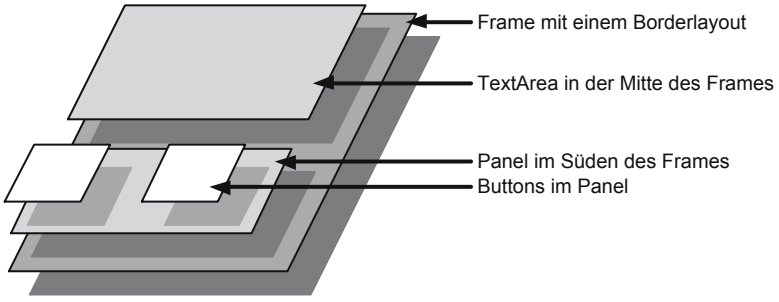


Abb. 5.11. Schachtelung von Containern

Die Klasse `Panel` ist der „einfachste“ Nachfahre der Klasse `Container`. Ihr einziger Zweck besteht darin, andere Container oder Komponenten aufzunehmen. Im Folgenden werden also mittels eines

[2] Beispielsweise kann bei Verwendung eines `BorderLayouts` nur eine Komponente in jeder Himmelsrichtung platziert werden (vgl. Abschnitt 5.3.3).

Panels `p` die Buttons in den unteren Rand des Fensters `fensterchen` eingefügt.

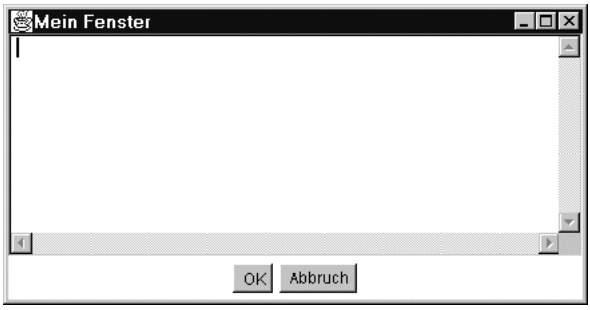


Abb. 5.12. Beispiel für eine Schachtelung von Containern

```
// Layout setzen
fensterchen.setLayout(new BorderLayout());
// Textfeld in der Mitte einfügen
fensterchen.add("Center", new TextArea());
// Panel mit zwei Buttons im Süden einfügen
Panel p = new Panel();
p.setLayout(new FlowLayout());
p.add(new Button("OK"));
p.add(new Button("Abbruch"));
fensterchen.add("South", p);
// optimale Groesse ermitteln und setzen
fensterchen.pack();
// Fenster anzeigen
fensterchen.setVisible(true);
```

5.4 Ereignisbehandlung

Ereignisse, die vom Benutzer ausgelöst werden, werden als Objekte innerhalb eines Programmes versandt. Als Klassen dieser Objekte kommen z. B. folgende in Frage:

- `ActionEvent`: Dieses wird versendet bei Aktionen. Eine solche wird beispielsweise bei Schaltern durch Klicken oder bei Textfeldern durch Drücken der `ENTER`-Taste ausgelöst.

- `WindowEvent`:`WindowEvent` Ein `WindowEvent` wird von Fenstern verschickt, deren Status sich ändert. Dieses wird beispielsweise durch Mini- oder Maximieren bzw. Schließen ausgelöst.
- `TextEvent`:`TextEvent` Solche Ereignisse werden von Objekten der Klasse `TextComponent` (oder deren Nachfahren) verschickt, wenn sich der Text durch Benutzeraktionen verändert hat.

Alle Komponenten, die Benutzer-Nachrichten verschicken, erledigen dies anhand einer Liste von *Listenern* (dt. Zuhörer). Die Listener sind also Objekte, die an den Nachrichten der Bedienelemente „interessiert“ sind und entsprechend informiert werden möchten. Dazu ein Beispiel:

> Ein Fenster enthält einen Button, auf dem „Schließen“ steht. Wenn dieser Button angeklickt wird, soll sich das Fenster schließen und anschließend das Programm beendet werden.

Wie ein Fenster erzeugt und ein Button eingefügt wird, wurde bereits in den vorherigen Kapiteln erläutert. Das Problem der Reaktion auf die ausgelösten Ereignisse ist Gegenstand der folgenden Abschnitte.

5.4.1 ActionListener

Listener sind Objekte, deren Klassen ein bestimmtes Interface implementieren (vgl. Abschnitt 2.5.7). Sie haben die Aufgabe, die vom Dialogelement ausgesendete Nachricht in ein bestimmtes Verhalten des Programms umzuwandeln. Ein Objekt, dessen Klasse das Interface `ActionListener` implementiert, erledigt dies für Nachrichten der Klasse `ActionEvent`, die z. B. von Objekten der Klasse `TextField` ausgesendet werden, wenn der Benutzer die Eingabe mit `ENTER` beendet.

Eine Klasse, die dieses Interface implementiert, muss eine Methode mit der Signatur `public void actionPerformed (ActionEvent e)` besitzen. Damit sieht eine entsprechende Klasse im einfachsten Fall folgendermaßen aus:

```
import java.awt.event.*;
import java.awt.*;

public class MeinErsterListener implements ActionListener {

  public void actionPerformed(ActionEvent e) {
    // An dieser Stelle sollte gehandelt werden
    System.out.println("Ich bin am machen!");
  }
}
```

Von welcher Klasse `MeinErsterListener` *erbt*, ist für das Problem unerheblich. So ist es beispielsweise denkbar, dass ein Fenster, welches den Button enthält, sich auch für dessen Nachrichten interessiert.

Ein weiterer wichtiger Punkt ist das *Registrieren* des Listeners beim Sender der Nachrichten, damit dieser „weiß“, wer sich für „seine“ Ereignisse interessiert und diese entsprechend adressieren kann. Basierend auf der oben im Beispiel implementierten Klasse `MeinErsterListener`, wird ein Objekt dieser Klasse folgendermaßen erzeugt und bei einem Objekt der Klasse `Button` registriert:

```
Button b = new Button("Mach was");
MeinErsterListener meinErsterListener = new MeinErsterListener();
b.addActionListener(meinErsterListener);
```

Anschließend wird bei jedem Klick mit der Maus auf den Button „Mach was“ die Methode `public void actionPerformed(ActionEvent e)` des vorher erzeugten Objekts `meinErsterListener` ausgeführt.

5.4.2 WindowListener

Ein Objekt, dessen Klasse das Interface `WindowListener` implementiert, kann auf Statusänderungen von Fenstern reagieren. Beim Minimieren senden diese beispielsweise eine Nachricht vom Typ `WindowEvent.WINDOW_ICONIFIED`. Das Schließen führt zum Versenden von `WindowEvent.WINDOW_CLOSED`. Dementsprechend existieren im Gegensatz zum `ActionListener` mehrere Methoden, die implementiert werden müssen. Das folgende Beispiel enthält sie:

```
import java.awt.*;
import java.awt.event.*;

public class MeinWindowListener implements WindowListener {

  public void windowActivated(WindowEvent e) { }

  public void windowClosed(WindowEvent e) { }

  public void windowClosing(WindowEvent e) { }

  public void windowDeactivated(WindowEvent e) { }
```

```
  public void windowDeiconified(WindowEvent e) { }

  public void windowIconified(WindowEvent e) { }

  public void windowOpened(WindowEvent e) { }
}
```

Analog zum Objekt `meinErsterListener` aus dem vorigen Abschnitt erzeugen die folgenden Zeilen einen WindowListener und registrieren ihn bei einem Fenster.

```
Frame f = new Frame("Mach was Fenster");
MeinWindowListener meinWindowListener = new MeinWindowListener();
b.addActionListener(meinWindowListener);
```

Aufgrund der Tatsache, dass `WindowListener` ein Interface ist, welches alle Klassen implementieren können, kann auch ein Fenster *selbst* auf die von ihm erzeugten Ereignisse reagieren. Dafür wird neben der Umsetzung des Interfaces eine entsprechende Zeile `addWindowListener(this)` im Quelltext benötigt:

```
import java.awt.*;
import java.awt.event.*;

public class Fenster extends Frame implements WindowListener {
  /**
   * Da von Frame geerbt wird und Frame das Interface Serializable
   * implementiert, muss eine Klassenvariable 'serialVersionUID'
   * definiert werden. Wird diese serialVersionUID nicht explizit
   * definiert, wird seit dem JDK 5.0 eine Warnung ausgegeben.
   * Um diese zu vermeiden, wird ein default-Wert vergeben.
   */
  private static final long serialVersionUID = 1L;

  public void windowActivated(WindowEvent e) { }

  public void windowClosed(WindowEvent e) { }

  /**
   * Beim Schließen des Fensters das Programm beenden
   */
  public void windowClosing(WindowEvent e) {
    System.exit(0);
```

```
  }

  public void windowDeactivated(WindowEvent e) { }

  public void windowDeiconified(WindowEvent e) { }

  public void windowIconified(WindowEvent e) { }

  public void windowOpened(WindowEvent e) { }

  public Fenster() {
    super();
    setLayout(new FlowLayout());
    setTitle("Mein Fenster");
    add(new TextField(20));
    add(new Button("OK"));
    add(new Button("Abbruch"));

    addWindowListener(this);

    pack();
  }

  public static void main(String[] args) {
    Fenster meinFenster = new Fenster();
    meinFenster.setVisible(true);
  }
}
```

Ein Objekt der Klasse `Fenster` wird bei seinem Schließen das Programm beenden.

5.4.3 Adapter

Klassen, deren einzige Aufgabe darin besteht, ein Interface zu implementieren, werden als *Adapter* bezeichnet. In diesem Sinne sind sowohl die Klasse `MeinErsterListener` als auch die Klasse `MeinWindowListener` aus den vorigen Abschnitten Adapter. Im Paket `java.awt.event` steht bereits für jedes Interface auch ein entsprechender Adapter zur Verfügung, für das Interface `WindowListener` beispielsweise der `WindowAdapter`.

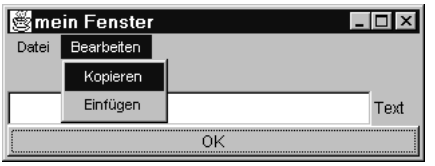
Die Adapter setzen zwar die Schnittstellen um, füllen die einzelnen Methoden jedoch nicht mit „Leben“. Um sie zu verwenden, müssen also

Klassen geschaffen werden, die vom jeweiligen Adapter erben. Im Zusammenhang mit der Verwendung einer inneren Klasse (vgl. Abschnitt 4.8) stellt sich die Lösung für das „Fensterproblem“ aus dem vorigen Abschnitt alternativ wie folgt dar:

```
import java.awt.*;
import java.awt.event.*;

public class Fenster extends Frame {
  /**
   * Da von Frame geerbt wird und Frame das Interface Serializable
   * implementiert, muss eine Klassenvariable 'serialVersionUID'
   * definiert werden. Wird diese serialVersionUID nicht explizit
   * definiert, wird seit dem JDK 5.0 eine Warnung ausgegeben.
   * Um diese zu vermeiden, wird ein default– Wert vergeben.
   */
  private static final long serialVersionUID = 1L;

  public Fenster() {
    super();
    setLayout(new FlowLayout());
    setTitle("Mein Fenster");
    add(new TextField(20));
    add(new Button("OK"));
    add(new Button("Abbruch"));

    addWindowListener(new InnererAdapter());

    pack();
    setVisible(true);
  }

  public static void main(String[] args) {
    Fenster meinFenster = new Fenster();
  }

  private class InnererAdapter extends WindowAdapter {
    /** Beim Schließen des Fensters das Programm beenden */
    public void windowClosing(WindowEvent e) {
      System.exit(0);
    }
  }
}
```

5.5 Übungsaufgaben

Aufgabe 9. Identifizieren Sie alle Objekte in folgender Abbildung und geben Sie die korrespondierende Klasse aus dem AWT an!



Aufgabe 10. Worin liegt der Vorteil in der Verwendung von Adaptern gegenüber der „direkten“ Implementierung eines Interfaces?

Aufgabe 11. Mit welchem Interface werden Mouse-Ereignisse empfangen? Entwerfen Sie ein Programm, das eine Ausgabe in der Konsole erzeugt, wenn der Mauszeiger bewegt wird!

6

Methodik

„Ist dies schon Tollheit, hat es doch Methode.“

Hamlet in William Shakespeares
„Hamlet“

6.1 Modellorientiertes Problemlösen

In den vergangenen Teilen wurden die Grundlagen der Objektorientierung erläutert und die Programmiersprache JAVA vorgestellt. Bei der Entwicklung von Anwendungen (oder im weitesten Sinne von Informationssystemen) ist die besprochene Implementierung jedoch erst der letzte Schritt, dem allgemein die Phasen der Analyse des zu lösenden Problems und der Entwurf einer Problemlösung vorausgehen.[1]

Man bedient sich bei der Entwicklung von Informationssystemen des modellorientierten Problemlösens. Dabei wird bei der Analyse des zu lösenden Problems der Ausgangszustand (Ist-Zustand) aus dem Problemraum mittels Abstraktion in ein Ist-Modell und damit in den Modellraum überführt. Dieses Ist-Modell wird im Modellraum in den Entwurf der Problemlösung transformiert. Anschließend wird dieser Entwurf durch Konkretisierung in den Endzustand im Problemraum überführt (vgl. [Geh+04]).

Während der Systementwicklung überführt die Phase der Systemanalyse ein gestelltes Problem in den Modellraum. Mittels des Systementwurfs wird dieses transformiert und durch die Implementierung in ein Informationssystem als Problemlösung überführt.

In jeder dieser Phasen können verschiedene Wiederverwendungsansätze zum Einsatz gebracht werden. Einen Überblick über in diesem Abschnitt verwendete Ansätze vermittelt die Abbildung 6.1. Alle Ansätze haben das Ziel, etwas einmal Entworfenes oder sogar Implementiertes für andere Probleme erneut zu verwenden (vgl. [Diet02]).

[1] Unter einem *Problem* wird dabei i. d. R. die Diskrepanz zwischen einem wahrgenommenen Ist-Zustand und einem gewünschten Soll-Zustand verstanden (vgl. [Dres96], S. 6).

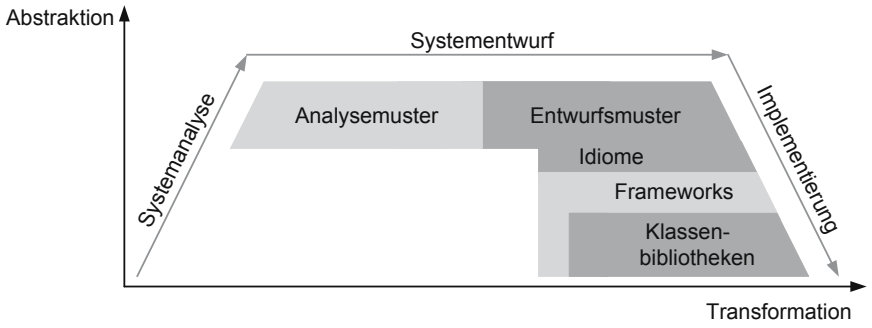


Abb. 6.1. Einordnung der Wiederverwendungsansätze

Analysemuster werden nach FOWLER definiert als „ . . . an idea that has been useful in one practical context and will probably be useful in others.“ ([Fowl97], S. 8).

In die Gruppe der Entwurfsmuster lässt sich die Definition von GAMMA einordnen: „The design patterns . . . are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context„ ([Gam+95], S. 3). GAMMA verzichtet dabei nicht auf eine Beschreibung der Überführung der Entwurfsmuster in Quellcode, jedoch sind die Muster in ihrer Struktur (z. B. durch zahlreiche abstrakte Klassen und Methoden) als Ausgangspunkt für diesen Teil des Buches eher ungeeignet.

Da BUSCHMANN in seiner Kategorisierung von Mustern den Begriff nicht nur im Modellraum ansiedelt, sondern vielmehr auch die Anwendung bei der Implementierung (also die Überführung in den Problemraum) betrachtet, basieren die folgenden Abschnitte auf seiner Definition und Kategorisierung. Mit den Idiomen stellt BUSCHMANN sogar Muster vor, welche an konkrete Programmiersprachen gebunden sind.

Als Vertreter für Klassenbibliotheken wurde im Teil 3 bereits das JDK vorgestellt. Frameworks werden von FAYAD und SCHMIDT ausführlich vorgestellt und klassifiziert (vgl. [FaSc97]).

Um die Analyse und den Entwurf zu dokumentieren und die Ergebnisse in die Implementierung zu überführen, ist eine Form der Beschreibung notwendig, welche die einzelnen Elemente der Informationssysteme genau darstellen kann. Zum einen sind dies Aspekte der Struktur solcher Systeme (statische Sicht) und zum anderen Aspekte des Verhaltens des Systems (dynamische Sicht).

Ähnlich wie bei der Kategorisierung von Programmiersprachen (vgl. Abschnitt 2.2.2) existieren auch für die Analyse und den Entwurf ver-

schiedene Formen von visuellen Beschreibungssprachen (Modellierungssprachen). Es liegt nahe, im Rahmen des objektorientierten Paradigmas auch objektorientierte Modellierungssprachen für die Analyse- und die Entwurfsphase von Informationssystemen zu verwenden.

Aus diesem Grund wird als ein Vertreter dieser Klasse die *Unified Modeling Language (UML)* in Abschnitt 6.2 in relevanten Auszügen kurz erläutert. Sie liegt sämtlichen konzeptionellen Ausführungen in diesem Buch zugrunde. In den darauf folgenden Abschnitten werden zwei weitere Instrumente der Wiederverwendung in der Softwareentwicklung vorgestellt. Abschnitt 6.3 beschäftigt sich mit Algorithmen und Abschnitt 6.4 mit den bereits erwähnten Entwurfsmustern.

6.2 Unified Modeling Language

Die Unified Modeling Language (UML) dient primär der Erstellung von objektorientierten Analyse- und Entwurfsmodellen im Rahmen der Entwicklung von betrieblichen Informationssytemen. In der UML existieren verschiedene Arten von Diagrammen, die jeweils unterschiedliche Aspekte der Informationssysteme beschreiben können.

Für die Darstellung von statischen Aspekten wird an dieser Stelle das *Klassendiagramm* (Abschnitt 6.2.1) und zur Abbildung von dynamischen Aspekten das *Sequenzdiagramm* (Abschnitt 6.2.2) sowie eine, für die Zielgruppe des Buches, leicht modifizierte Form des *Aktivitätsdiagramms* (Abschnitt 6.2.3) vorgestellt. Die Beschreibungen fokussieren dabei jeweils nur auf die für den Zweck des Buches relevanten Teile der drei Diagrammarten. Die vollständige Spezifikation der UML in der Version 2.0 ist bei der *Object Management Group (OMG)* abrufbar (vgl. [OMG05]).

6.2.1 Klassendiagramm

Eine Klasse wird in ausführlicher Form als ein in drei Abschnitte unterteiltes Rechteck dargestellt, das im oberen Feld den Namen der Klasse, im mittleren ihre Attribute und im unteren Feld ihre Methoden enthält. Zusätzlich kann der Name des Paketes angegeben werden, zu dem die Klasse gehört. Verkürzt kann eine Klasse nur mit einem einfachen Rechteck repräsentiert werden, welches den Klassennamen und ggf. das Paket enthält. Ein Klassenname muss stets mit einem Großbuchstaben beginnen.

Für die in Abschnitt 4.7 kennengelernten Zugriffsrechte auf Attribute und Methoden, werden im Klassendiagramm Abkürzungen verwendet, sie sind in Tabelle 6.1 dargestellt.

UML-Notation	**Zugriffsart**
+	`public`
#	`protected`
∼	(`package`), entspricht in JAVA dem Weglassen der Zugriffsart
–	`private`

Tabelle 6.1. UML-Notation der JAVA-Zugriffsarten

Beispiel:

Die Klasse `Kuh` gehört zum Paket `Bauernhof`. Sie besitzt die Attribute `alter`, `name` und eine `kleeMenge`, die sie verdrücken kann. Ihre Methoden sind neben dem Konstruktor `Kuh()`, `frissKlee(int menge)`, `produziereMilch()` und `gibMilch()`.

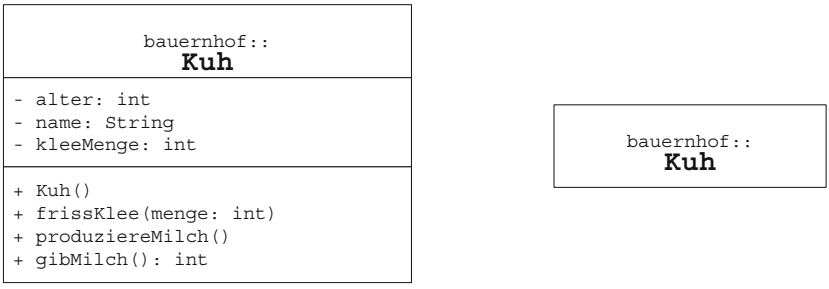


Abb. 6.2. Klasse `Kuh` in ausführlicher und verkürzter Form

Im JAVA-Quellcode wird diese Klasse wie folgt beschrieben:

```
package bauernhof;
```

```
public class Kuh {

  private int alter;
  private String name;
  private int kleeMenge;

  public Kuh() { }

  public void frissKlee(int menge) { }

  public void produziereMilch() { }

  public int gibMilch() { }

}
```

Vererbungsbeziehung

Eine *Vererbungsbeziehung* zwischen zwei Klassen wird durch einen nicht ausgefüllten Pfeil von der Unter- zur Oberklasse dargestellt. Im Beispiel aus Abbildung 6.3 erben `DeutscheKuh` und `BritischeKuh` ihre Eigenschaften von `Kuh`.

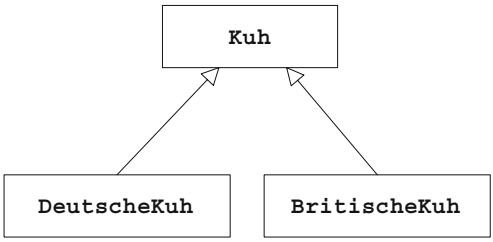


Abb. 6.3. Vererbung im Klassendiagramm

Im Quellcode stellt sich dies wie folgt dar:

```
public class DeutscheKuh extends Kuh{

  public DeutscheKuh() { }

}
```

```
public class BritischeKuh extends Kuh {

  public BritischeKuh() { }

}
```

Interface

Um ein Interface darzustellen, wird dieses ebenfalls als Klasse modelliert, erhält jedoch zusätzlich das Schlüsselwort `«interface»`, welches über dem Klassennamen und dem Paket ausgezeichnet wird.

Zur Implementierung eines Interfaces nutzen wir die so genannte *Interface-Realisierungsabhängigkeit*. Ähnlich der Vererbungsbeziehung wird diese mit einem nicht ausgefüllten Pfeil dargestellt, die Linie ist jedoch gestrichelt. Der Pfeil führt aus der Klasse heraus und mündet in dem Interface, welches implementiert wird. Im Beispiel der Abbildung 6.4 wird das Interface `Ackertier` von der Klasse `BritischeKuh` implementiert.

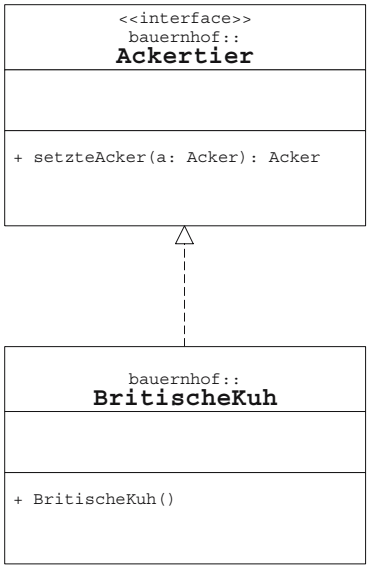


Abb. 6.4. Darstellung eines Interface

Im Quellcode stellt sich dies wie folgt dar:

```
public interface Ackertier {

  public Acker setzeAcker(Acker a);

}
```

```
public class BritischeKuh extends Kuh implements Ackertier {

  public BritischeKuh() { }

  /**
   * Implementierte Methode aus dem Interface
   */
  public Acker setzeAcker(Acker a) {
  // Hier folgen Anweisungen ...
  }
}
```

Assoziationsbeziehung

Besitzen Objekte Referenzen aufeinander (z. B. besitzt ein Objekt der Klasse `Kuh` eine Referenz auf ein Objekt der Klasse `Bein`), so spricht man von einer *Assoziationsbeziehung*. Diese bildet man ab, indem man eine Linie zwischen den beteiligten Klassen zieht.

Mit der *Multiplizität* wird angegeben, wie viele Objekte der Klasse am Ende der Beziehung mit einem Objekt der Klasse am Anfang der Beziehung assoziiert sein können (und umgekehrt, wenn die andere Seite beschriftet wird). Hier kann entweder eine genaue Zahl oder ein Minimum und Maximum angegeben werden. Liegt das Minimum bei 0, kann, muss aber keine Assoziation vorliegen.

Um die Beziehung näher zu beschreiben, kann eine Assoziation benannt werden. Ein kleines gleichseitiges schwarzes Dreieck legt dabei die Leserichtung des *Assoziationsnamens* fest. Der Assoziationsname erleichtert die Nachvollziehbarkeit von Multiplizitäten. Ferner kann die Assoziation auch noch mit *Rollennamen* versehen werden, zur Identifikation „... der spezifischen Funktionen der an der Beziehung beteiligten Objekte.“ ([Hit+05], S. 64)

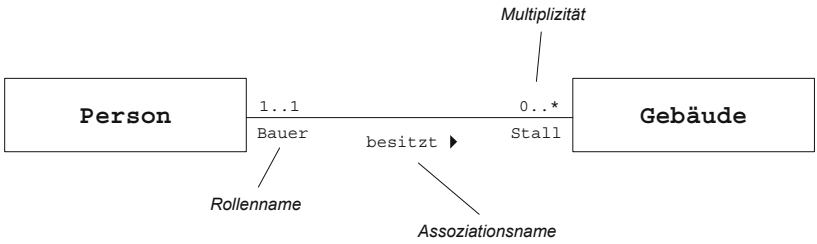


Abb. 6.5. Assoziation im Klassendiagramm

Als Beispiel zur Umsetzung einer Assoziation in JAVA ziehen wir wieder die Klasse `Kuh` heran. Diese besitzt die folgende Assoziation mit der Klasse `Bein` (vgl. Abbildung 6.6): Eine Kuh hat genau vier Beine, ein Bein gehört zu genau einer Kuh.

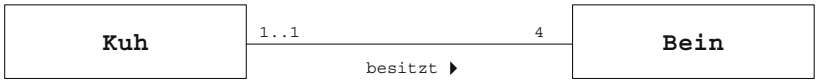


Abb. 6.6. Implementierung einer Assoziation in JAVA

In JAVA würde man daraufhin in der Klasse `Kuh` bspw. ein Array `beine` mit der Länge 4 definieren, also ein Feld, welches die vier Beine aufnehmen kann. Und um andersherum jedem Bein eindeutig eine Kuh zuordnen zu können, erhält die Klasse `Bein` ein entsprechendes Attribut und ihr Konstruktor einen entsprechenden Parameter.

```
public class Kuh {

  private Bein[] beine;

  public Kuh() {
    beine = new Bein[4];
    beine[0] = new Bein(this); // Bein vorne links
    beine[1] = new Bein(this); // Bein vorne rechts
    beine[2] = new Bein(this); // Bein hinten links
    beine[3] = new Bein(this); // Bein hinten rechts
  }
}
```

```
public class Bein {

  private Kuh besitzer;

  public Bein(Kuh besitzer) {
    this.besitzer = besitzer;
  }
}
```

6.2.2 Sequenzdiagramm

Beim Sequenzdiagramm wird der zeitliche Ablauf des Nachrichtenaustauschs zwischen Objekten deutlich gemacht. Ein Objekt wird durch seinen Namen und eine senkrechte gestrichelte Lebenslinie dargestellt. Die Zeitdarstellung verläuft dabei von oben nach unten. Der senkrechte Balken auf der Lebenslinie symbolisiert, dass das Objekt gerade aktiv ist. Eine Nachricht wird durch einen Pfeil, ihren Namen und eventuelle Parameter in Klammern gekennzeichnet. Ein gestrichelter Pfeil mit offener Pfeilspitze stellt eine Antwort auf die Nachricht dar, deren Modellierung ist optional. Dabei kann der Name der aufrufenden Nachricht sowie, durch einen Doppelpunkt getrennt, links von dem Namen, ein etwaiger Rückgabewert und durch ein Gleichheitszeichen getrennt, rechts von dem Namen, das Attribut dem dieser Wert zugeordnet ist, mit an die Antwort angetragen werden.

Im Beispiel sendet das Objekt der Klasse `Bauer` die Nachricht `gibMilch()` an die Kuh `lolitaLandliebe`. Das Objekt der Klasse `Kuh` reagiert, indem es eine Nachricht an sich selbst sendet und damit die Methode `frissKlee()` startet. Diese ruft anschließend `produziereMilch()` auf. Ist diese beendet, wird wieder zu `frissKlee()` und dann zu `gibMilch()` und schließlich zur Ausgangssituation zurückgesprungen, wobei als Antwort das Attribut `literMilch` mit einem Wert von `5` zurückgeben wird.

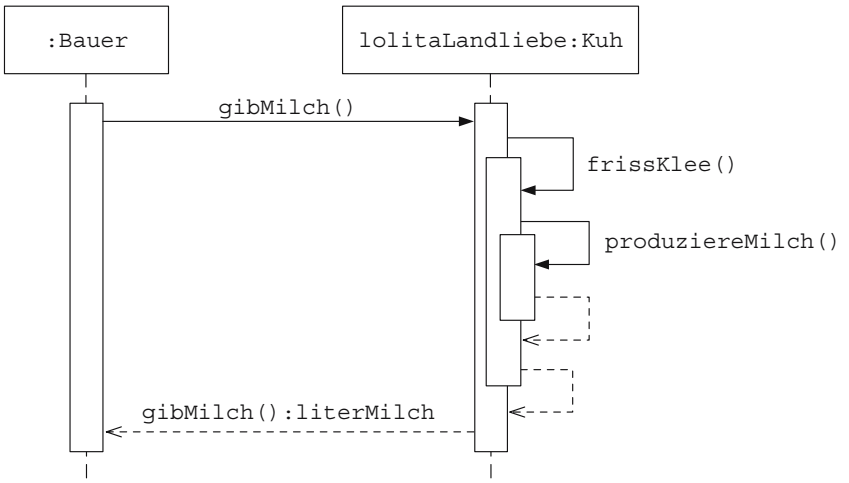


Abb. 6.7. Beispiel für ein Sequenzdiagramm

Im JAVA-Code kann man diesen Ablauf wie folgt in zwei Klassen abbilden. Zuerst der Inhalt von `Bauer`:

```
public class Bauer {

  private Kuh lolitaLandliebe;

  public Bauer() { }

  public void beispiel(){
    lolitaLandliebe.gibMilch();
  }
}
```

Der Inhalt der Klasse `Kuh`:

```
public class Kuh {

  private int literMilch;

  public void frissKlee(){
    produziereMilch();
  }
```

```
  public void produziereMilch(){
    literMilch = 5;
  }

  public int gibMilch(){
    frissKlee();
    return literMilch;
  }
}
```

6.2.3 Aktivitätsdiagramm

Aktivitätsdiagramme stellen analog zu Sequenzdiagrammen Abläufe dar.[2] Während das Sequenzdiagramm jedoch die Modellierung von Interaktionen hervorhebt, d. h. die Art und Weise, wie Nachrichten über die Zeit hinweg zwischen verschiedenen Interaktionspartnern in einem bestimmten Kontext ausgetauscht werden, um eine bestimmte Aufgabe zu erfüllen (vgl. [Hit+05], S. 251), liegt der Schwerpunkt des Aktivitätsdiagramms auf der Darstellung von prozeduralen Verarbeitungsaspekten (vgl. [Hit+05], S. 186).

Abgebildet werden Aktivitäten[3] (als abgerundetes Rechteck), die mit ihren nachfolgenden Aktivitäten über sog. Transitionen (Pfeile) verbunden werden. An diese können Bedingungen angetragen werden, die erfüllt sein müssen, damit in die durch den Pfeil angegebene Richtung „weiterabgearbeitet“ wird. Der Beginn einer Aktivitätsfolge wird durch einen ausgefüllten Kreis und das Ende durch einen ausgefüllten Kreis mit zusätzlichem Rahmen dargestellt.

[2] Das hier vorgestellte Aktivitätdiagramm wurde von den Autoren leicht modifiziert. Sie erhoffen sich davon, bezogen auf die Zielgruppe des Buches, ein verbessertes Verständnis für diese Diagrammart.

[3] Das können allgemeine Aktivitäten sein (z. B. „Morgens aufstehen“) oder konkrete Anweisungen einer Programmiersprache (z. B. `x = x + 1`).

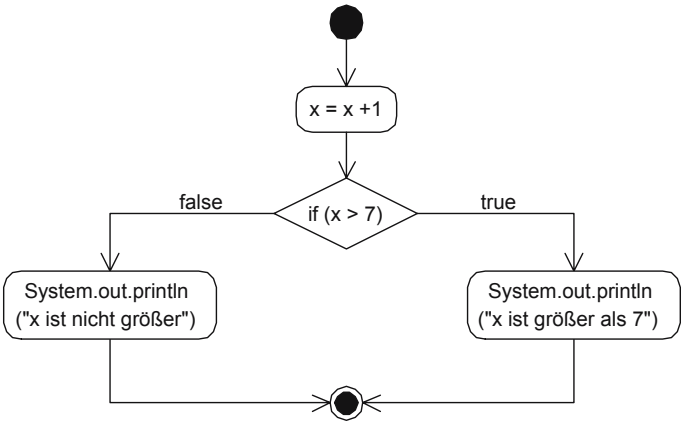


Abb. 6.8. Beispiel für ein Aktivitätsdiagramm

Im JAVA-Quellcode würde derselbe Ablauf folgendermaßen beschrieben sein:

```
int x = x + 1;

if (x > 7) {
  System.out.println("x ist größer als 7");
} else {
  System.out.println("x ist nicht größer");
}
```

Die Abläufe innerhalb einzelner Methoden (bspw. auch innerhalb eines Konstruktors) lassen sich nach den Ablaufstrukturtypen

- Sequenz,
- Alternative,
- Wiederholung und
- Parallele Verarbeitung

klassifizieren. Deren Modellierung mittels Aktivitätsdiagrammen wird im Folgenden vorgestellt. Zusätzlich erfolgt jeweils eine Überführung des so erstellten Entwurfs in die Programmiersprache JAVA.

Sequenz

Bei der Sequenz sind die Anweisungen in einer Methode `F` nacheinander auszuführen.

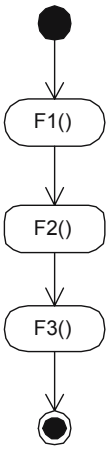


Abb. 6.9. Darstellung einer Sequenz

```
private F() {
  F1();
  F2();
  F3();
}
```

Zwei Alternativen

Im weiteren Programmverlauf wird entweder die Anweisung `F1` oder die Anweisung `F2` ausgeführt. Notwendig für die Entscheidung ist die Auswertung der Bedingung `B`, die entweder `true` (wahr) oder `false` (falsch) liefert.

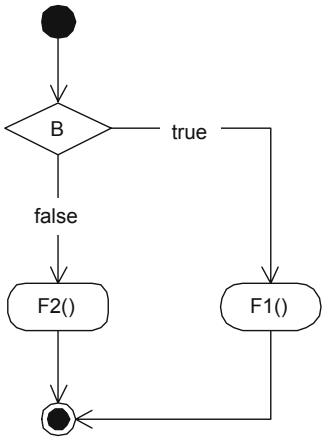


Abb. 6.10. Darstellung von 2 Alternativen

```
private F() {
  if (B) {
    F1();
  } else {
    F2();
  }
}
```

N+1 Alternativen

Im weiteren Programmverlauf wird entweder die Anweisung `F1`, `F2` oder `FN` ausgeführt. Nimmt die Bedingung `B` den Wert `W1` an, wird `F1` ausgeführt. Nimmt die Bedingung `B` den Wert `W2` an, wird `F2` ausgeführt. In allen anderen Fällen wird die Anweisung `FN` ausgeführt.

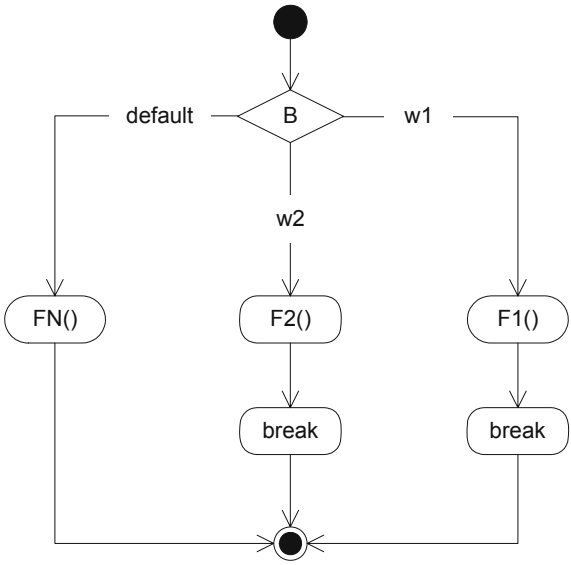


Abb. 6.11. Darstellung von N+1 Alternativen

```
private F() {
  switch (B) {
    case W1: F1();
             break;
    case W2: F2();
             break;
    default: FN();
  }
}
```

Wiederholung mit konstanter Anzahl

Die Methode `F` besteht in diesem Fall aus einer Wiederholung der Anweisung `F2`. Die Anzahl der Wiederholungen wird mit der Zahl `n` festgelegt.

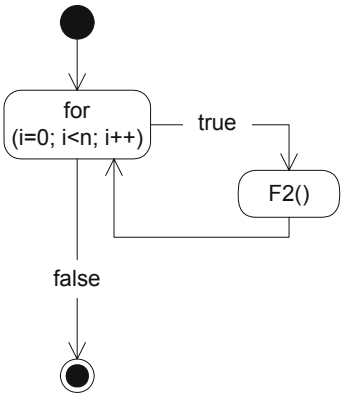


Abb. 6.12. Darstellung einer Wiederholung mit konstanter Anzahl

```
private F() {
  for (i = 0; i < n; i++) {
    F2();
  }
}
```

Wiederholung mit variabler Anzahl und Vortest

Die Anzahl der Wiederholungen der Anweisung `F2` steht nicht fest. Die Auswertung der Bedingung `B` führt bei einem `true` zu einem weiteren Durchlauf. `F1` wird erst ausgeführt, wenn die Bedingung nicht mehr erfüllt ist.

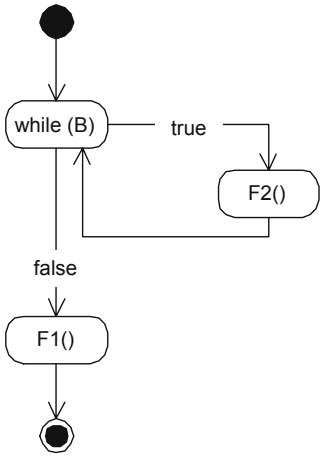


Abb. 6.13. Darstellung einer Wiederholung mit variabler Anzahl und Vortest

```
private F() {
  while (B) {
    F2();
  }
  F1();
}
```

Wiederholung mit variabler Anzahl und Nachtest

Die Anzahl der Wiederholungen der Anweisung `F2` steht nicht fest. Die Auswertung der Bedingung `B` führt bei einem `true` zu einem weiteren Durchlauf (`B` bestimmt also die Bedingung für den Abbruch). `F2` wird aber auf jeden Fall einmal ausgeführt. `F1` wird hier ebenfalls erst ausgeführt, wenn die Bedingung nicht mehr erfüllt ist.

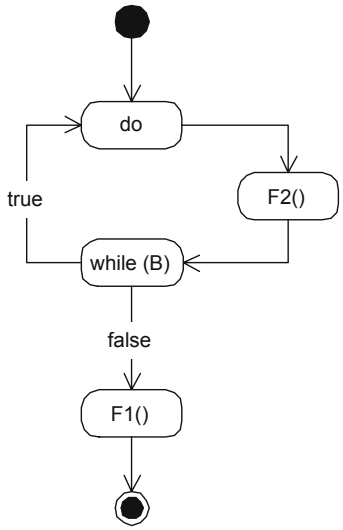


Abb. 6.14. Darstellung einer Wiederholung mit variabler Anzahl und Nachtest

```
private F() {
  do {
    F2();
  } while (B);
  F1();
}
```

Parallele Verarbeitung

Die Anweisungen `F1` bis `FN` können parallel ausgeführt werden. Der Start aller Anweisungen erfolgt mit der Ausführung der Methode `F`.

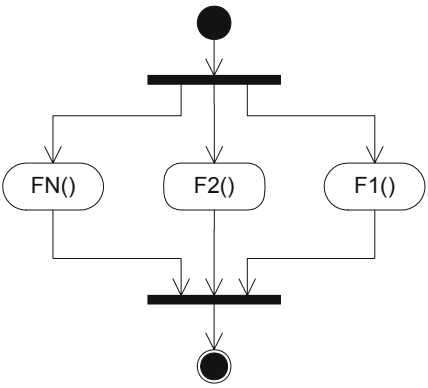


Abb. 6.15. Darstellung einer parallelen Verarbeitung

Die Darstellung mittels JAVA erfordert die Verwendung von Threads, die im Rahmen des Buches nicht behandelt wurden. Aus diesem Grund wird an dieser Stelle auf eine Detaillierung verzichtet. Ein Überblick dazu ist bspw. bei SCHADER und SCHMIDT-THIEME zu finden (vgl. [ScSc03], S. 409ff).

6.3 Algorithmen

Die Algorithmen sind Bestandteil der elementaren Operationen auf strukturierte Daten, zu denen

- Initialisieren der Datenstruktur,
- Suchen eines Datensatzes,
- Einfügen und Löschen eines Datensatzes und
- Sortieren der Datensätze

gehören.

Der Aufbau der einzelnen Algorithmen wird wesentlich von der verwendeten Datenstruktur beeinflusst. Sie werden, je nachdem ob eine lineare Liste (vgl. Abschnitt 6.3.4), eine Baumstruktur (vgl. Abschnitt 6.3.5) oder einfache Felder (vgl. 4.13.6) vorliegen, anders gestaltet sein. In den Abschnitten 6.3.2 und 6.3.3 wird mit den elementaren Such- und Sortieralgorithmen die Suche in einfachen Feldern behandelt.

Während sich die Muster aus dem Abschnitt 6.4 im Allgemeinen mit Architekturproblemen beschäftigen, liegt das Anwendungsgebiet für Algorithmen im Bereich der Rechenprobleme.

Die Beschreibung von Datenstrukturen in den Abschnitten 6.3.4 und 6.3.5 nehmen eine Zwischenstellung ein, wobei der Schwerpunkt der Beschreibung auf dem Problem der Organistation der Strukturen (also Initialisieren, Suchen, Sortieren, Einfügen und Löschen von Daten) liegt.

Algorithmen und Muster existieren nicht unabhängig voneinander, sondern sind durch eine zweiseitige Beziehung miteinander verbunden. Zum einen erfordert die Anwendung eines Musters die Dienste der beteiligten Komponenten, welche unter Umständen sehr komplex sein können. Zur Komplexitätsreduktion können vorgefertigte Datenstrukturen und Algorithmen verwendet werden. Muster können aber auch die Instanziierung von Algorithmen und Datenstrukturen unterstützen, indem diese in Form eines Musters dokumentiert werden.

6.3.1 Rekursive Algorithmen

Das einfachste Beispiel für einen rekursiven Algorithmus bildet die Berechnung der Fakultät einer natürlichen Zahl n. Diese Berechnung basiert auf folgenden Definitionen:

- $0! = 1$
- $n! = n * (n-1)!$

In JAVA ergibt sich daraus die Methode:

```
public int fakultaet(int x) {
  if (x == 0) {
    return 1;
  } else {
    return x * fakultaet(x - 1);
  }
}
```

Allgemein besteht eine rekursive Methode m aus einer Grundanweisung S und einer Abbruchbedingung B. Ist B erfüllt (z. B. $x == 0$), kommt es zu keinem weiteren Aufruf von S (in diesem Fall also $fakultaet(x-1)$). Eine Rekursion ist *direkt*, wenn m direkt m aufruft. *Indirekt* ist die Rekursion, wenn m eine Methode n aufruft, welche wiederum m ausführt. Grundsätzlich gilt, dass sich jeder rekursive Algorithmus in einen nicht rekursiven überführen lässt. Angewendet werden sollte er vor allem, wenn das Problem (wie in unserem Beispiel die Fakultät) rekursiv definiert ist.

6.3.2 Suchalgorithmen

Eine der grundlegenden Bestandteile von Informationssystemen ist die Suche in einer Menge gespeicherter Informationen. Diese werden als Datensätze mit einem bestimmten Schlüssel zur eindeutigen Adressierung im Speicher abgelegt. Im einfachsten Fall entstehen Tabellen, die mindestens zwei Spalten besitzen, zum einen den Suchschlüssel und zum anderen das Objekt, das gespeichert wurde.

Für die Suche existieren zahlreiche Algorithmen, die in vielen Programmen verwendet werden. Aus diesem Grund werden diese als elementare Suchalgorithmen bezeichnet.

Ausgangspunkt bildet eine Datenstruktur `DataType[] a`, wobei `DataType` ein Standarddatentyp oder eine Klasse sein kann. Ziel der Suche ist das Auffinden eines Elements x in a durch die Ermittlung von dessen Schlüssel in a.

Die Beschreibung der Algorithmen basiert auf der Formulierung von

- Pre-Conditions (gelten vor einem Programmblock),
- Post-Conditions (gelten nach einem Programmblock) und
- Invarianten (gelten für Wiederholungsstrukturen).

Lineares Suchen

Bei der linearen Suche existieren keine zusätzlichen Informationen über das zu untersuchende Feld. Während der Ausführung wird das Array sequentiell durchlaufen bis x gefunden wurde oder das Ende des Arrays erreicht ist. Beim Durchlauf vergrößert sich somit sukzessive der Bereich, in dem sich x mit Sicherheit nicht befindet.

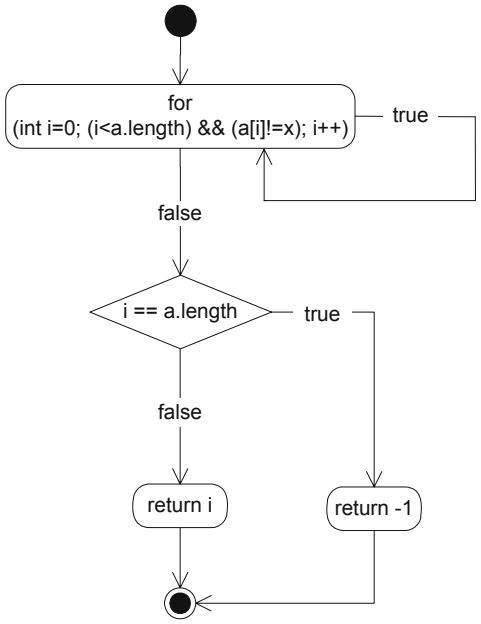


Abb. 6.16. Lineares Suchen

Aus dem Aktivitätsdiagramm ergibt sich folgende Umsetzung in JAVA:

```
private int lineareSuche(int[] a, int x) {
  int i;
  for (i = 0; (i < a.length) && (a[i] != x); i++);
  if (i == a.length) {
    return −1;
  } else {
    return i;
  }
}
```

Binäres Suchen

Anders als bei der linearen Suche geht die binäre Suche von einem aufsteigend sortierten Feld aus. Dieser Zustand kann durch die Anwendung einer der Sortieralgorithmen aus Abschnitt 6.3.3 erzielt werden. Bei jedem Schritt der Suche wird zuerst eine möglichst große Anzahl

von Elementen des Feldes für die weitere Suche ausgeschlossen. Hierfür wird eine linke (l) und eine rechte Grenze (r) definiert und in diesem Bereich die Suche erneut angestoßen. Das Element an der Position in der Mitte zwischen beiden Grenzen (m) wird mit dem zu suchenden Wert verglichen. Ist es kleiner, bildet $m + 1$ die neue linke Grenze, ist es größer, bildet $m - 1$ die rechte Grenze, und ist das Element der gesuchte Begriff, dann war der Algorithmus erfolgreich. Besitzen l und r die gleichen Werte, war die Suche nicht erfolgreich. Abbildung 6.17 demonstriert den Ablauf, wobei das zu suchende Element grau dargestellt wird.

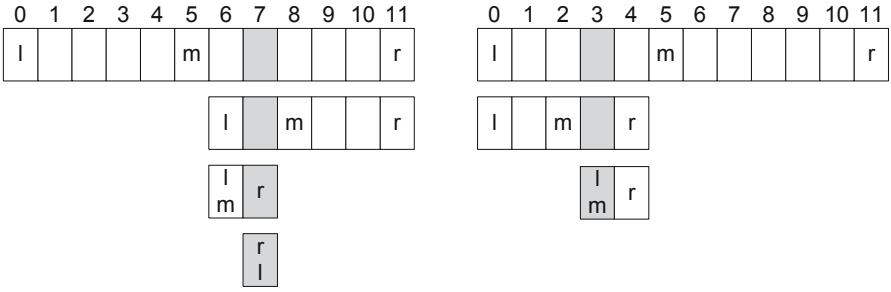


Abb. 6.17. Beispiele für die binäre Suche

In JAVA ergeben sich daraus folgende Zeilen:

```
private int binaereSuche(int[] a, int x) {
  int l = 0;
  int r = a.length −1;
  int m = 0;
  do {
    m = (l + r) / 2;
    if (a[m] < x){
        l = m + 1;
    } else{
        r = m − 1;
    }
  } while ((l <= r) && (a[m] != x));
  if (a[m] == x) {
    return m;
  } else {
    return −1;
  }
```

```
}
```

6.3.3 Sortieralgorithmen

Die Sortieralgorithmen in diesem Abschnitt sortieren ein Array a, gefüllt mit `int` Werten $i_0, i_1 \ldots i_{n-1}$ entsprechend einer Ordnungsfunktion f. Beispielsweise kann diese vorgeben, dass ein Wert i desto weiter vorn im Array a steht, je kleiner er ist.

Die Definition eines solchen Arrays kann in JAVA wie folgt vorgenommen werden:

```
int[] a = {12, 13, 45, 0, 9};
```

Die Effizienz der Algorithmen wird anhand von zwei Kriterien gemessen:

- M: Anzahl der Umstellungen (moves); Gibt an, wie oft ein Element in a an eine andere Position gelegt wird.
- C: Anzahl der Schlüsselvergleiche (compare); Gibt an, wie oft zwei Elemente miteinander verglichen werden.

Welcher Algorithmus der effektivste ist, kann anhand des Vergleiches der beiden Werte M und C bestimmt werden. Nicht immer ist jedoch eine genaue Berechnung möglich, ohne ein konkretes Beispiel zu wählen. Im JDK 5.0 gilt ungefähr, dass 4 Umstellungen in etwa genauso lange dauern wie 3 Vergleiche. Grundlage für die Berechnung bilden folgende Zeilen:

```
// Array und zwei Positionen definieren
int[] a = {10, 20, 30};
int k = 2; int l = 1;

// Uhrzeit merken
long now = System.currentTimeMillis();

// Zuweisungen ausfuehren
for (int i = 0; i < 99999999; i++) {
  a[k] = a[l];
}

System.out.println("Dauer : " + (System.currentTimeMillis() - now));
```

```
now = System.currentTimeMillis();

/*
 * Vergleiche ausfuehren - nach dem oberen Test gilt a[k] = a[l],
 * deshalb wird hier nur die Dauer des Vergleichs gemessen,
 * da keine Zuweisungen erfolgen.
 */
 for (int i = 0; i < 99999999; i++) {
     if (a[k] > a[l]){
         a[k] = a[l];
     }
}

System.out.println("Dauer : " + (System.currentTimeMillis() - now));
```

Mögliche Verfahren zum Sortieren, die in den folgenden Abschnitten näher erläutert werden, sind:

- Sortieren durch direktes Einfügen,
- Sortieren durch direktes Auswählen,
- Sortieren durch direktes Austauschen und
- der rekursive Algorithmus Quick-Sort.

Sortieren durch direktes Einfügen

Den wohl einfachsten Ansatz bildet das *Sortieren durch direktes Einfügen*. Eine Variable i durchläuft die Positionen im Array a. In dem Bereich von i bis zum Ende des Arrays wird das kleinste Element x gesucht und an die Stelle `a[i]` gelegt. Die Suche erfolgt mit der Laufvariable j, die entsprechend von i bis `a.length - 1` laufen muss.

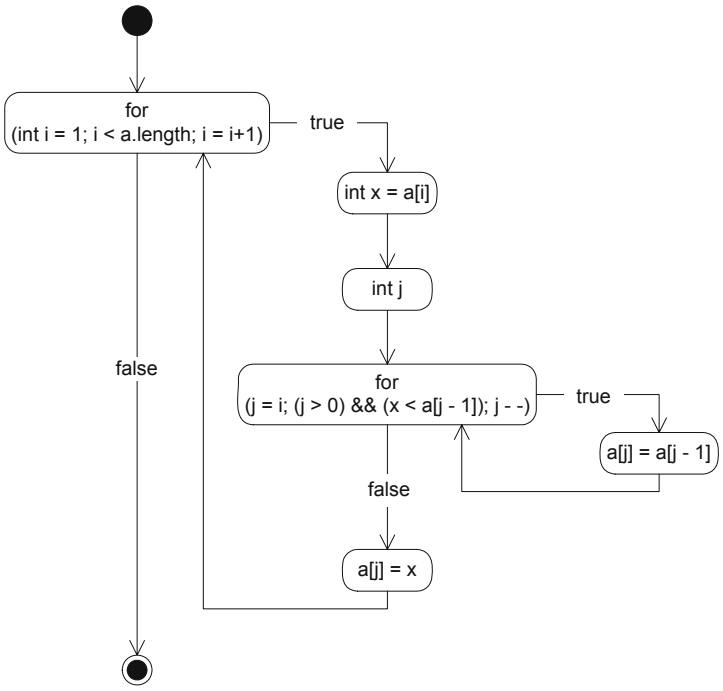


Abb. 6.18. Sortieren durch direktes Einfügen

Anhand des Aktivitätsdiagramms kann folgende Methode für das direkte Einfügen entworfen werden:

```
private void direktesEinfuegen(int[] a) {
  // i läuft von links nach rechts
  for (int i = 1; i < a.length; i++) {

    int x = a[i];
    int j;

    // direktes Einfuegen der Werte in
    // die vorderen, wenn x kleiner ist
    for (j = i; (j > 0) && (x < a[j−1]); j−−) {
      a[j] = a[j−1];
    }

    a[j] = x;
  }
}
```

Da in diesem Beipiel bei jedem Schleifendurchlauf mit `(j > 0)` geprüft wird, ob der Anfang des Arrays a (also die Position 0) bereits erreicht ist, kann der Algorithmus weiter optimiert werden. Die zusammengesetzte Bedingung wird durch das Einfügen einer Marke verhindert. Man stellt sicher, dass im ersten Element des Arrays stets ein so kleiner Wert enthalten ist, dass der Durchlauf abgebrochen wird, ohne dass die Untergrenze des Arrays unterschritten wird. Hierfür muss das der Methode übergebene Array aber im ersten Feld einen „unwichtigen Wert" als Platzhalter (Dummy) enthalten, da dieser überschrieben wird.

```
private void direktesEinfuegen(int[] a) {
  // i läuft von links nach rechts
  for (int i = 1; i < a.length; i++) {

    int x = a[i];
    a[0] = x;
    int j;

    // direktes Einfuegen der Werte in
    // die vorderen, wenn x kleiner ist
    for (j = i; (x < a[j−1]); j−−) {
      a[j] = a[j−1];
    }
    a[j] = x;
  }
}
```

Sortieren durch direktes Auswählen

Leicht abgewandelt arbeitet das *Sortieren durch direktes Auswählen.* In einer Schleife wird die Position des jeweils kleinsten Elements in k abgelegt. Diese wird mit dem Element in a_0 getauscht. Anschließend wird im verbleibenden Feld $a_1 \ldots a_{n-1}$ das kleinste Element gesucht und mit a_1 getauscht usw.

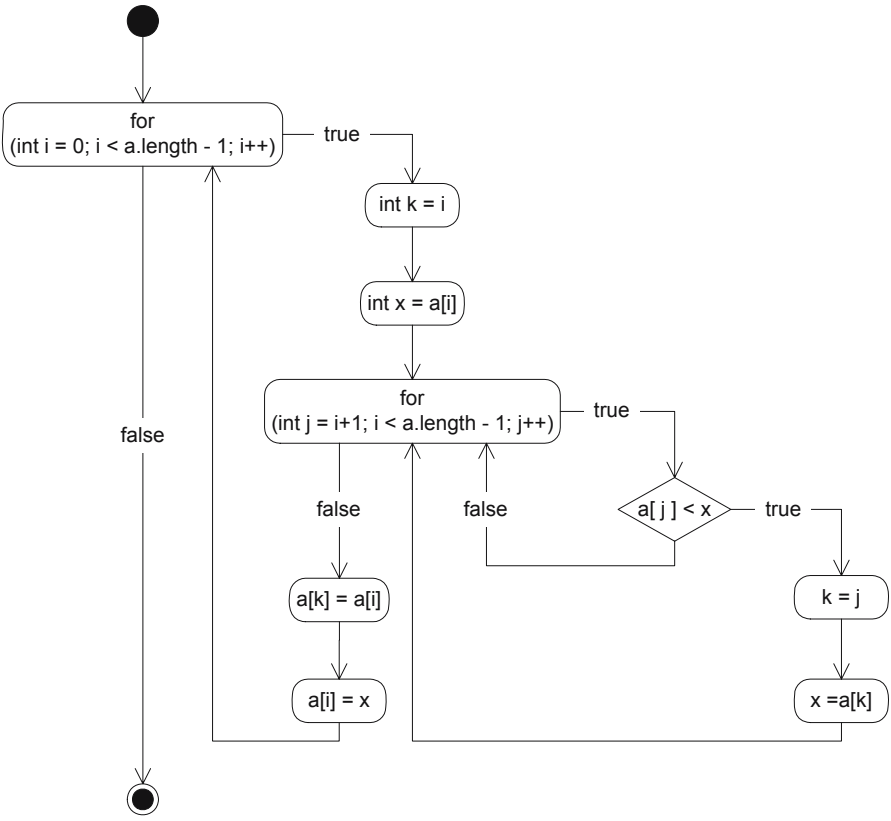


Abb. 6.19. Sortieren durch direktes Auswählen

Sortieren durch direktes Austauschen (Bubble-Sort)

Der Sortieralgorithmus *Bubble-Sort* vergleicht das jeweils letzte Element im Array (im folgenden als Blase bzw. Bubble bezeichnet) mit seinem Vorgänger. Ist dieser größer, wird die Blase mit dem nächsten Vorgänger verglichen. Ist ein Element kleiner als die Blase, wird es selbst zur Blase. Mit diesem Prinzip steigen die kleinen Elemente im Array nach „oben“.

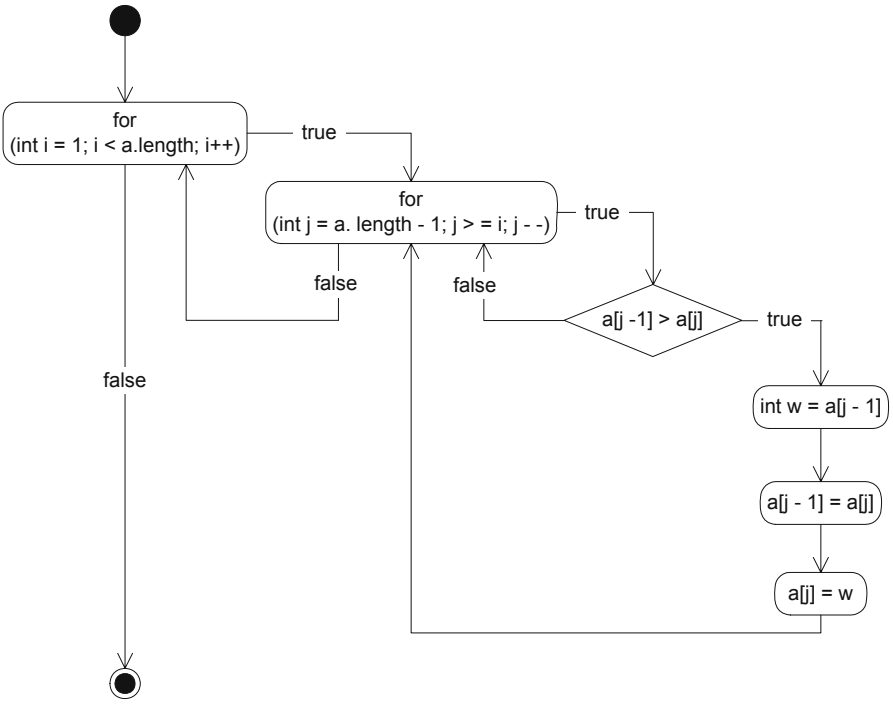


Abb. 6.20. Sortieren durch direktes Austauschen

Die entsprechende Methode in JAVA besitzt folgenden Aufbau:

```
private void bubbleSort(int[] a) {
  // i laeufft von links nach rechts
  for (int i = 1; i < a.length; i++) {
    // j läuft von rechts bis i
    for (int j = a.length − 1; j >= i; j−−) {
      // Steigt ein Blaeschen?
      if (a[j−1] > a[j]) {
        int w = a[j−1];
        a[j−1] = a[j];
        a[j] = w;
      }
    }
  }
}
```

Quick-Sort

Die wohl größte Praxisrelevanz besitzt der Algorithmus *Quick-Sort*. Bereits 1960 wurde er von HOARE entwickelt und der Öffentlichkeit vorgestellt. Quick-Sort bietet sehr gute Durchlaufzeiten unter unterschiedlichsten Bedingungen und kann deshalb mit Recht als Mehrzweckverfahren bezeichnet werden.

Nachteil ist vor allem die Implementierung als rekursiver Ablauf. Ein einfacher Fehler kann unbemerkt bleiben und zu unerwarteten Resultaten bei ausgewählten Problemstellungen führen, obwohl er bei anderen funktioniert.

Grundsätzlich wird das zu sortierende Feld stets in zwei Teile zerlegt. Beide Teile werden nach dem selben Prinzip immer weiter unterteilt. Die Effektivität besteht vor allem in der Identifikation bereits sortierter Bereiche. Unsortierte Elemente werden über die entsprechenden Grenzen „geworfen“ und bei späteren Durchläufen weiterbearbeitet.

```
private void quickSort(int[] a, int l, int r) {

  // i und j werden die Grenzen fuer den bereits
  // sortierten Bereich. Erst einmal sind es die alten
  // Grenzen. Spaeter werden wir diese einengen
  int i = l;
  int j = r;

  // Die Mitte des zu sortierenden Bereiches
  int x = a[(l + r) / 2];

  // Solange die Grenzen sich nicht ueberschneiden,
  // neue Grenzen setzen und unsortierte Elemente
  // ueber diese Grenzen werfen
  do {

    // linke Grenze solange hochzaehlen, bis
    // ein unsortierter Bereich kommt
    while (a[i] < x){
        i++;
    }

    // rechte Grenze solange runterzaehlen, bis
    // ein unsortierter Bereich kommt
    while (x < a[j]){
        j--;
    }
```

```
    // Tauschen der Werte an den Grenzen, wenn
    // diese auseinanderliegen oder sich
    // beruehren
    if (i <= j) {
      // Austausch von a[i] und a[j]
      // (Werte ueber die Grenzen werfen
      // zum spaeteren sortieren)
      int w = a[i];
      a[i] = a[j];
      a[j] = w;
      // Weitersetzen der Grenzen
      i++;
      j--;
    }
  } while (i <= j);

  // Damit sind alle Werte zwischen i und j sortiert
  // Die Werte jenseits der mit i und j
  // definierten neuen Grenzen sortieren
  if (l < j) quickSort(a, l, j);
  if (i < r) quickSort(a, i, r);
}
```

6.3.4 Lineare Listen

Lineare Listen dienen der Speicherung von Daten in sequentieller Form. Damit ist der Zugriff auf ein bestimmtes Element stets mit einem Durchlauf zu seiner Position in der Liste verbunden (ähnlich wie bei einem Bandlaufwerk).

Ein Knoten in einer linearen Liste enthält die zu speichernden Daten und einen Verweis auf den nächste Knoten. Nur das letzte Element einer solchen Liste verweist auf ein leeres Element, also auf `null`.

```
public class Node {

  private Node next = null;
  private Object daten;

  public Node (int key, Object daten) {
    this.daten = daten;
    this.key = key;
```

```
  }

  public void setNext(Node next) {
    this.next = next;
  }

  public Node getNext() {
    return next;
  }
}
```

Das Objekt, welches die Liste verwaltet, muss stets eine Referenz auf die Wurzel `root` speichern. Geht diese Referenz verloren, so ist kein Zugriff mehr auf die Elemente der Liste möglich.

Die Operatoren auf eine lineare Liste werden am Beispiel einer Klasse `NodeList` demonstriert:

```
public class NodeList {

  Node root;

  public void append(Node newNode) {
    if (root != null) {
      Node aNode = root;
      while (aNode.getNext() != null) aNode = aNode.getNext();
      aNode.setNext(newNode);
    } else root = newNode;
  }

  public void insert(Node newNode) {
    newNode.setNext(root)
    root = newNode;
  }

  public void insertAfter(Node afterNode, Node newNode) {
    if (afterNode != null) {
      newNode.setNext(afterNode.getNext());
      afterNode.setNext(newNode);
    }
  }

  public void delete(Node aNode) {
    if (root != null) {
      if (root == aNode) {
```

```
          root = root.getNext();
        } else {
          Node tempNode = root;

          while ((tempNode.getNext() != null) &&
          (tempNode.getNext() != aNode)) {
            tempNode = tempNode.getNext();
          }

          if (tempNode.getNext() != null) {
            tempNode.setNext(aNode.getNext());
          }
        }
      }
    }
}
```

6.3.5 Baumstrukturen

Ein *Baum* ist eine nichtleere Menge von Knoten und Kanten. Eine Kante verbindet stets zwei Knoten. Jeder Knoten, bis auf den Wurzelknoten, besitzt einen Vorgänger und eine beliebige Anzahl nachfolgender Knoten. Zusätzlich speichert er Informationen, wie beispielsweise einen Namen. Ein Pfad beschreibt eine Liste von Knoten, die über Kanten miteinander verbunden sind. Zwischen zwei Knoten in einem Baum existiert genau ein Pfad.

Als Blatt wird ein Knoten ohne Nachfolger bezeichnet. Ein Teilbaum besteht aus einem Knoten und allen seinen untergeordneten Knoten. Die Stufe (oder auch Ebene) eines Knotens im Baum ist die Anzahl der Kanten auf dem Pfad zur Wurzel. Die Höhe des Baumes ist die höchste Stufe, die ein Knoten im Baum besitzt. Die Pfadlänge eines Baumes berechnet sich aus der Summe aller Stufen der Knoten.

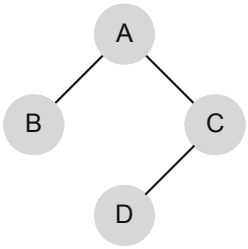


Abb. 6.21. Beispiel für einen Baum

In dem Beispiel aus Abbildung 6.21 gelten stellvertretend folgende Sachverhalte:

- A ist der Wurzelknoten
- D liegt auf der Stufe 2, A auf der Stufe 0
- Der Vorgänger von C ist A
- Die Nachfolger von A sind B und C
- Der Pfad von B zu D besteht aus: B, A, C, D
- Die Pfadlänge des Baumes beträgt 4

Der Grad eines Baumes beschreibt die maximale Anzahl der Nachfolger eines Knotens. Wird dieser Grad auf 2 beschränkt, handelt es sich um einen Binärbaum. In binären Bäumen ist die Definition von äußeren Knoten (oder auch Pseudoknoten, da sie keine weiteren Informationen speichern) zweckmäßig. Ein äußerer Knoten hat keinen Nachfolger und ein innerer Knoten genau zwei. Der Nachfolger eines inneren Knotens kann sowohl ein innerer als auch ein äußerer Knoten sein.

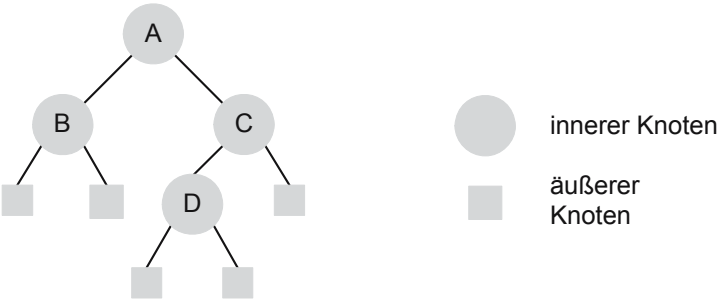


Abb. 6.22. Äußere und innere Knoten

Zur Realisierung eines binären Baumes definiert man sich zwei Klassen. Als erstes eine Klasse `Node` zur Abbildung eines Knotens:

```
public class Node extends Object {

  private Node leftChild = null;
  private Node rightChild = null;
  private Node parent = null;

  private Object userObject = null;

  public Node (Object newUserObject) {
    userObject = newUserObject;
  }

  public Object getUserObject() {
    return userObject;
  }

  public Node getLeftChild() {
    return leftChild;
  }

  public Node getRightChild() {
    return rightChild;
  }

  public void setLeftChild(Node newNode) {
    leftChild = newNode;
    if (leftChild != null) leftChild.setParent(this);
  }

  public void setRightChild(Node newNode) {
    rightChild = newNode;
    if (rightChild != null) rightChild.setParent(this);
  }

  public Node getParent() {
    return parent;
  }

  public void setParent(Node newNode) {
    parent = newNode;
  }
```

```
  public String getKey() {
    if (userObject != null) return userObject.toString();
    else return "no object";
  }
}
```

Um nach einem bestimmten Schlüssel zu suchen, wird der binäre Baum zu einem *Binären Suchbaum* erweitert. Die Eigenschaften eines Knotens t_i im binären Suchbaum sind:

- alle Schlüssel im linken Teilbaum sind kleiner als der Schlüssel von t_i
- alle Schlüssel im rechten Teilbaum sind größer als der Schlüssel von t_i

Um die Anforderungen der Suche nach einem Knoten zu realisieren, wird die Klasse `Node` um die rekursive Methode `findNode(String key)` ergänzt. Somit wird die Suche im Baum als Aufgabe stets dem Wurzelknoten übergeben. Dieser kann die Suche entweder an seinen linken oder rechten Unterknoten delegieren oder die Suche beenden. Dies geschieht, wenn der Schlüssel gefunden wurde oder kein entsprechender Unterknoten vorhanden ist. Die Knoten, an welche die Aufgabe delegiert wurde, verfahren nach demselben Prinzip.

```
public Node findNode(String key) {
  int vergleich = key.compareTo(getKey());
  Node nextNode = null;

  if (vergleich < 0) nextNode = getLeftChild();
  else if (vergleich > 0) nextNode = getRightChild();
  else return this;

  if (nextNode != null) return nextNode.findNode(key);
  else return null;
}
```

Um einen Knoten einzufügen, muss im ersten Schritt ein Knoten gefunden werden, an welchem dieser angehangen werden soll. Somit wird diese Aufgabe ebenfalls an einen Knoten (und damit zuerst einmal wie bei der Suche nach dem Wurzelknoten) übergeben. Damit im Ergebnis die Eigenschaft des binären Suchbaums erhalten bleibt, wird der Schlüssel des einzufügenden Knotens mit dem Schlüssel des Knotens

`insertAfter` verglichen. Ist er kleiner, wird `insertAfter` auf den linken und anderenfalls auf den rechten Knoten gesetzt. Wurde im Baum die entsprechende Stelle gefunden, wird der Knoten eingefügt.

```
public void insertNode(Node aNode) {

  if (getParent() != null) {

    Node insertAfter = this;
    Node vorgaenger = null;

    do {
      vorgaenger = insertAfter;

      if (aNode.getKey().compareTo(insertAfter.getKey()) < 0) {
        insertAfter = insertAfter.getLeftChild();
      } else {
        insertAfter = insertAfter.getRightChild();
      }

    } while (insertAfter != null);

    insertAfter = vorgaenger;

    if (aNode.getKey().compareTo(insertAfter.getKey()) < 0) {
      insertAfter.setLeftChild(aNode);
    } else {
      insertAfter.setRightChild(aNode);
    }

  } else {
    parent.insertNode(aNode);
  }
}
```

Das Löschen eines Knotens ist ein etwas komplexerer Vorgang, da hierbei eventuell untergeordnete Knoten nach „oben“ wandern. Es gilt in der Struktur den Knoten durch denjenigen zu ersetzen, der den nächst größeren Schlüssel im angehängten Teilbaum besitzt. Besitzt ein Knoten keine Nachfolger, entfällt dieser Schritt selbstverständlich. In jedem anderen Fall gilt es jedoch folgende Schritte abzuarbeiten:

- Ermitteln des „linkesten“ Unterknotens des rechten Teilbaums

- „Aushängen“ des Knotens, indem dessen Nachfolger „über“ ihm angehangen werden
- „Einhängen“ des Knotens durch Anfügen des rechten Teilbaums des zu löschenden Knotens

```
public void delete() {
  // Die Wurzel darf nicht geloescht werden
  if (getParent() != null) {
    // Den Vorgaenger dieses Knotens ermitteln
    Node parent = getParent();

    // Das neue Child des Parents initialisieren
    Node newChildOfParent = this;

    // Mit den folgenden Schritten wird der neue Nachfolger festgelegt,
    // wenn rechts nichts mehr haengt, ist es einfach
    if (getRightChild() == null) {
      newChildOfParent = newChildOfParent.getLeftChild();
    } else {
      if (getRightChild().getLeftChild() == null) {
        newChildOfParent = newChildOfParent.getRightChild();
        newChildOfParent.setLeftChild(getLeftChild());
      } else {
        Node leftestChildOfRightChild = getRightChild();
        while (leftestChildOfRightChild.getLeftChild().getLeftChild()
                  != null) {
          leftestChildOfRightChild =
                  leftestChildOfRightChild.getLeftChild();
        }

        newChildOfParent = leftestChildOfRightChild.getLeftChild();

        leftestChildOfRightChild.setLeftChild(
          newChildOfParent.getRightChild()
        );

        newChildOfParent.setLeftChild(getLeftChild());
        newChildOfParent.setRightChild(getRightChild());
      }
    }

    if (getKey().compareTo(parent.getKey()) < 0) {
      parent.setLeftChild(newChildOfParent);
    } else {
```

```
        parent.setRightChild(newChildOfParent);
      }
    }
}
```

Um die Knoten in einem Baum verwalten zu können, wird die Klasse `BinaerBaum` verwendet. Diese erbt von der Klasse `java.awt.Panel`, um den Baum mit der Methode `printTree(Node aNode)` auch auf dem Bildschirm darstellen zu können.

```
public class BinaerBaum extends Panel {

  private Node rootNode = null;

  public BinaerBaum () {
    super();
    setLayout(null);

    rootNode = new Node("\u0000");

    insertNode(new Node("Her"));
    ...
  }

  public void printTree(Node aNode) {
    if (aNode != null) {
      printTree(aNode.getLeftChild());
      aNode.print();
      printTree(aNode.getRightChild());
    }
  }

  public Node findNode(String key) {
    rootNode().findNode(key);
  }

  public void deleteNode (Node aNode) {
    aNode.delete();
  }
}
```

Durch häufiges Einfügen und Löschen mittels der oben beschriebenen Verfahren wird der Baum schnell „unausgeglichen“, da ein ausgeglichener Baum folgende Eigenschaften besitzt:

- gleichmäßige Verteilung der Knoten auf linken und rechten Teilbaum
- daraus folgt eine minimale Höhe des Baumes

Die Definition lautet: Ein Binärbaum heißt perfekt ausgeglichen, wenn sich für jeden Knoten die Anzahl der Knoten in seinem linken und rechten Teilbaum nicht unterscheidet (vgl. [Sedg03], S. 593f).

6.4 Entwurfsmuster

6.4.1 Einführung

Programmierer tendieren dazu, Quellcode aus anderen Anwendungsfällen zu imitieren. Gemeint ist damit nicht das Kopieren und Einfügen, sondern ein Zusammenspiel aus dem Erkennen und Anpassen von Lösungen erfahrener Programmierer auf den eigenen Problembereich (vgl. [Pree94], S. 61). Dazu ist es notwendig, zunächst von den konkreten Problemlösungen zu abstrahieren und die gemeinsamen Faktoren, also die Essenz der Lösung, zu extrahieren. Dieser Prozess führt zur Entstehung von *Mustern*: „These problem - solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions.“ ([Johs94]).

Dieses Denken in Problemlösungen ist weit verbreitet in der Architektur, Volkswirtschaft und auch der Softwareentwicklung (vgl. [Bus$^+$96], S. 3). Damit lässt sich insbesondere im Bereich der Softwareentwicklung auf die Erfahrungen fähiger Softwareingenieure aufbauen. Weiterhin ist es mit Hilfe von Mustern möglich, das existierende Wissen und die Erfahrungen in der Softwareentwicklung effizient zu dokumentieren und verfügbar zu machen und gute Designlösungen einer breiten Öffentlichkeit zugänglich zu machen.

Neben diesen qualitativen Aussagen ist eine Definition für den eindeutigen Gebrauch des Begriffs des Musters unerlässlich. Sehr allgemein definiert der Architekt Christopher Alexander in seinem Buch „The timeless Way of Building“ Muster als eine dreiteilige Regel, welche eine Beziehung zwischen einem bestimmten Kontext, einem Problem und einer Lösung ausdrückt (vgl. [Alex79], S. 247).

Diese Definition ist aber für die Softwareentwicklung unzureichend und wurde daher durch verschiedene Autoren erweitert und spezifiziert. Buschmann formuliert deshalb: „Every Pattern deals with a specific,

recurring problem in the design and implementation of a software system.“ ([Bus+96], S. 3ff). Neben der Anpassung der allgemeinen Definition von Mustern auf den Bereich der Softwareentwicklung enthält diese Definition bereits den wichtigen Aspekt der Mehrfachverwendung.

Aus ökonomischen Gründen lohnt die Erstellung eines Musters nur, wenn es mehrfach eingesetzt und somit die Analyse- und Designphase erheblich verkürzt werden kann. Um dies zu erreichen, muss man von konkreten Problemlösungen abstrahieren und die Essenz der Lösungen als Muster extrahieren.

Im weiteren Verlauf konkretisiert Buschmann: „A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.“ ([Bus+96], S. 8). Neben der bereits diskutierten Wiederverwendbarkeit finden wir in der Definition die bereits bei Alexander angesprochenen Elemente: *Kontext*, *Problem* und *Lösung*. Dabei wird zusätzlich vorgeschrieben, in welcher Art die Lösungsstruktur zu dokumentieren ist.

Pree definiert Entwurfsmuster als eine Menge von Regeln, die beschreiben, wie bestimmte Aufgaben im Bereich der Softwareentwicklung durchzuführen sind (vgl. [Pree94], S. 61). Damit fallen z. B. sämtliche Sortieralgorithmen ohne weitere Ergänzungen oder Erweiterungen sofort in den Bereich der Entwurfsmuster. Obwohl dieser Ansatz sehr pragmatisch ist, vernachlässigt er den Aspekt der Mehrfachverwendbarkeit, der die Anwendbarkeit der Regeln unter Umständen stark einschränken kann sowie die Einbeziehung des Kontextes und des Problems, womit insbesondere das Auffinden und die Verwendbarkeitsprüfung in großen Mengen von Mustern kompliziert wird. Die von [Pree94] in die Definition mit aufgenommenen Algorithmen sind Gegenstand des Abschnitts 6.3.

Grundlage dieses Abschnitts wird deshalb eine abgewandelte Form der Definition von Buschmann bilden. Muster sind demnach erprobte Lösungsstrukturen für ein bestimmtes wiederkehrendes Problem in einem bestimmten Kontext. Dazu enthält das Lösungsschema mindestens:

- Komponenten der Struktur,
- deren Beziehungen untereinander und
- Regeln für die Zusammenarbeit der Komponenten.

In diesem Kontext können Klassen, deren Instanzen, Pakete oder beispielsweise Methoden Komponenten sein.

6.4.2 Ziele und Anforderungen an Muster

Die Abbildung 6.1 weist bereits darauf hin, dass Muster insbesondere während des Designs und der Implementierung eingesetzt werden, wobei sie vereinzelt auch eine Unterstützung für die Analyse und den Test der Systeme bieten. Entsprechend sind auch deren Ziele zu differenzieren.

Ein wichtiges Ziel während der gesamten Systementwicklung ist die Bereitstellung eines einheitlichen Sprachraumes. Durch die in den Beschreibungen der Muster verwendeten Begriffe werden Missverständnisse zwischen den Projektmitarbeitern vermieden und somit die Kommunikation unter den Beteiligten effektiviert. Durch die einheitliche Bedeutung der Begriffe werden langwierige Erläuterungen zum Verständnis der gewählten Lösungen überflüssig. Des Weiteren erlauben Muster die Komplexitätsreduktion großer, heterogener Systeme durch den Einsatz als „mental - building blocks“. Dabei werden die großen Architekturen zunächst aus groben Architekturmustern zusammengesetzt. Diese benötigen zu ihrer Implementierung wiederum kleinere Muster. Durch die rekursive Anwendung der Muster wird die Komplexität der Systemumsetzung schrittweise reduziert, ohne den Überblick über die Gesamtanwendung zu verlieren.

Muster dienen häufig als Wissensspeicher für existierende, erprobte Designlösungen. In Verbindung mit ihrer Veröffentlichung ist durch die mehrfache Anwendung der Muster durch verschiedene Entwickler sichergestellt, dass Fehler im Muster schnell erkannt und die vorgegebenen Lösungsstrukturen optimiert werden. Durch die Vielzahl der Entwickler ist es aber auch wahrscheinlich, dass bei Verfügbarkeit neuer Technologien oder Paradigmen gänzlich neue Muster mit wesentlich verbesserten Problemlösungsstrukturen entwickelt werden. Damit ist im Allgemeinen eine hohe Qualität der öffentlich verfügbaren Muster sichergestellt.

Eine wichtige Anforderung, insbesondere im Zusammenhang mit dem Aufkommen der objektorientierten Programmiersprachen, ist die Wiederverwendbarkeit des Codes. Wiederverwendbarkeit erfordert aber oft Abstraktionen und Strukturen, die sich im Rahmen der Systemanalyse nicht oder nur schwer erkennen lassen. Die notwendige Flexibilität wird nur durch mehrere Iterationen von Analyse und Design erreicht. Muster können den Iterationsbedarf entscheidend verringern, da sie bereits erprobte Strukturen repräsentieren. Dabei ist nicht unbedingt eine bessere als die eigene Lösung garantiert, zumindest wird jedoch ein Bewertungsmaßstab bei der Evaluierung vorgegeben.

Durch den Einsatz von Mustern im Systemdesign lässt sich aber auch die notwendige Dokumentation vereinfachen. Die Angabe der ver-

wendeten Muster vereinfacht insbesondere die Dokumentation komplexer Systeme. Damit wird es auch für andere Entwickler möglich, die Architektur und Konzeption des Systems für Weiterentwicklungen mit vertretbarem Aufwand zu verstehen.

Muster können ebenfalls als Zusatz für existierende Methoden verstanden werden, die z. B. aufzeigen, wie einfache Elemente (Objekte, Vererbung und Polymorphie) zu verwenden sind. Es lassen sich aber auch die Gründe für ein bestimmtes Design ablegen, so dass die Entwicklung zunächst nachvollziehbar und anschließend auf ein Problem anwendbar wird.

Ein weiteres Ziel ist die Unterstützung der Umformung des Analysemodells in eine konkrete Implementierung unter Sicherstellung einer möglichst hohen Wiederverwendbarkeit des Codes. Für die vollständige Unterstützung einer Methode werden aber z. B. zusätzlich Analysepatterns, User-Interface-Patterns oder Performance-Tuning-Patterns benötigt.

In der Vermittlung von Programmiersprachen liegt ein weiteres Einsatzgebiet der Muster. Anfängern werden in JAVA bestimmte Muster für oft vorkommende Programmierprobleme vorgegeben, um häufig vorkommende Fehler zu vermeiden und das Erlernen der Syntax zu vereinfachen.

Fasst man die wesentlichen Ziele zusammen, so dienen Muster der:

- Bereitstellung eines einheitlichen Sprachraumes,
- Wiederverwendbarkeit des erstellten Programmcodes,
- Vereinfachung der Dokumentation von Informationssystemen,
- Umformung des Analysemodells in eine konkrete Implementierung und der
- Vermittlung von Programmiersprachen.

Um die gestellten Ziele zu erreichen und insbesondere eine breite Verwendbarkeit der Muster zu gewährleisten, sind gewisse Mindestanforderungen an den Inhalt der Musterdokumentationen zu stellen. Aufgrund der unterschiedlichen Schwerpunkte bei der Definition von Mustern finden sich in der Literatur auch unterschiedliche Varianten des Aufbaus von Musterbeschreibungen. BUSCHMANN fordert beispielsweise für die Beschreibung eines Musters (vgl. [Bus$^+$96]):

- *Problembeschreibung*: Diese beinhaltet die Essenz des Problems, die zu beachtenden Regeln (constraints), die Lösungsanforderungen und gegebenenfalls wünschenswerte Eigenschaften der Lösung. Zusätzlich sollte die Problembeschreibung eine Menge von Zwängen (forces)

enthalten, die das Problem aus mehreren Sichten darstellen und somit das Verständnis der Details erleichtern.
- *Anwendungskontext*: Der Anwendungskontext enthält eine meist allgemein gehaltene Beschreibung der Situationen, in denen das zu lösende Problem auftritt. Eine detaillierte Darstellung des Kontextes ist zumeist schwierig, ein pragmatischer Ansatz könnte deshalb die Auflistung aller Situationen fordern, in denen das Problem bis jetzt auftrat (vgl. [Bus+96], S. 9).
- *Lösungsstruktur*: Die Strukturbeschreibung enthält sowohl statische als auch dynamische Aspekte.

6.4.3 Muster-Kategorien

Um die im Abschnitt 6.4.2 geforderten Eigenschaften von Muster-Kategorien umzusetzen, kann es unter Umständen sinnvoll sein, zugunsten einer pragmatischen Kategorisierung auf eine akademisch vollständige zu verzichten. Eine erste solche Kategorisierung schlägt beispielsweise [Bus+96] vor. Danach werden Muster zunächst nach:

- Analysemuster (bzw. Architekturmuster),
- Entwurfsmuster und
- Idiome (programmiersprachenspezifisches Muster)

unterteilt (vgl. [Bus+96] S. 12ff).

Analysemuster (bzw. Architekturmuster) adressieren dabei insbesondere die Gesamtstrukturen von Systemen, wie z. B. das Smalltalk Model-View-Controller-Konzept (MVC). Es wird insbesondere für interaktive Anwendungen verwandt. In derartigen Anwendungen muss es möglich sein, die Benutzeroberfläche ohne großen Aufwand ändern zu können, ohne dabei an der eigentlichen Programmlogik Veränderungen vornehmen zu müssen. Aus diesen Gründen werden derartige Systeme dreigeteilt, wobei alle Teile einen abgeschlossenen Bereich beinhalten. Das Modell enthält die eigentliche Programmlogik, während die Views bzw. Sichten für die Darstellung des Systems auf den Bildschirmen verantwortlich sind. Die Controller wiederum sind für die Entgegennahme der Benutzereingaben sowie für die Aktualisierung der Views verantwortlich. Es ist zu beachten, dass lediglich die Aufteilung festgelegt wird, nicht wie z. B. die Aktualisierung der Views durch die Controller realisiert werden soll.

Diese Einzelheiten werden durch die zweite Kategorie, die *Entwurfsmuster* beschrieben. Diese Muster fokussieren zwar im Allgemeinen

auf Untersysteme, sind aber trotzdem noch unabhängig von der verwendeten Programmiersprache. Dabei sind Entwurfsmuster insbesondere „... descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.“ ([Gam+95], S. 3) Ein im Abschnitt 6.4.5 zitiertes Beispiel ist das Observer-Muster (vgl. [Bus+96], S. 13f). Durch seine Anwendung wird das oben beschriebene Teilproblem zwischen den Komponenten eines MVC-Systems gelöst. Mit Hilfe der Architektur- und Entwurfsmuster ist eine Unterstützung des Analyse- und Designprozesses zwar gegeben, jedoch obliegt die konkrete Umsetzung in die Zielsprache noch immer dem Programmierer.

Idiome (deutsch: Spracheigentümlichkeiten) unterstützen dies, indem die Implementierung bestimmter Aspekte von Komponenten oder deren Beziehungen untereinander in einer konkreten Programmiersprache vorgeschlagen wird. Somit ist es möglich, dass Idiome für C++ keinen Sinn in Java machen und umgekehrt. Ein typisches Beispiel für ein solches Idiom ist das *Counted-Body-Pattern*, wie es in [Bus+96] auf S. 15 beschrieben wird.

Obwohl mit dieser Kategorisierung bereits eine Strukturierung der Muster erreicht wurde, ist die Einteilung insbesondere im Bereich der Entwurfsmuster für das schnelle Auffinden für ein spezifisches Problem noch unzureichend. Gamma et al. unterteilen daher Entwurfsmuster weiter mittels

- *Zweck (Purpose)* und
- *Fokus (Scope)*

der Muster (vgl. [Gam+95], S. 9ff). Dabei unterscheiden die Autoren Muster bezüglich des Zwecks nach:

- *creational*: betreffen den Prozess der Objekterzeugung
- *structural*: betreffen die Komposition von Klassen und Objekten
- *behavioural*: charakterisieren die Art, in der Klassen und Objekte miteinander kommunizieren und Verantwortlichkeiten verteilen

Im Bereich des Fokus unterscheiden die Autoren Muster nach:

- *Klassen (class)*: Beziehungen zwischen Klassen und ihren Subklassen mittels Vererbung zur Compilierzeit und
- *Objekten (objects)*: dynamische Beziehungen zwischen Objekten

Damit ergibt sich folgendes Kategorisierungsschema, welches in Auszügen wiedergegeben wird:

Scope	**Purpose**		
	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter
Object	Abstract Factory Prototype	Adapter (object) Composite Bridge	Chain of Responsibility Strategy Visitor

Tabelle 6.2. Kategorisierung von Mustern (vgl. [Gam+95])

Weitere Kategorisierungsmöglichkeiten nach [Gam+95] sind eine Ordnung nach der gemeinsamen Verwendung, nach der alternativen Verwendung, nach Mustern, die identische Implementierungen erzeugen und nach Mustern, welche sich gegenseitig referenzieren.

Kritik an einer derartigen Unterteilung nach Zweck und Fokus der Muster wird insbesondere von [Bus+96] geübt. Die Autoren bemängeln die vage Unterscheidung zwischen strukturellen und verhaltensorientierten Mustern. Damit ist eine effektive Unterstützung des Entwicklers beim Auffinden geeigneter Muster nicht gewährleistet. Sie schlagen stattdessen eine Ersetzung der Zweckkategorien durch Problemkategorien vor. Außerdem werden in der vorgestellten Kategorisierung neben Architektur- und Entwurfsmustern auch Idiome betrachtet.

Die von den Autoren angebotenen Problemkategorien orientieren sich dabei an konkreten Designproblemen, lassen sich aber durchaus um neue Bereiche erweitern.

	Architectural Patterns	**Design Patterns**	**Idioms**
From Mud to Structure	Pipes and Filters		
Distributed Systems	Broker, Client-Server		
Interactive Systems	MVC		
Adaptable Systems	Microkernel		

Structural Decomposition		Whole-Part	
Organization of Work		Master-Slave	
Access Control		Proxy	
Management		View Handler	
Communication		Publisher-Subscriber	
Resource Handling			Counted Pointer

Tabelle 6.3. Kategorisierung von Mustern (vgl. [Bus+96])

6.4.4 Anwendung von Mustern

Praktische Anleitungen zur Überführung eines Musters in eine konkrete Lösung finden sich sowohl bei [Gam+95] auf den S. 29ff als auch bei [Bus+96] auf den S. 373ff. Während [Bus+96] einen Grobablauf für den Prozess der Systementwicklung vorgibt, beschäftigt sich [Gam+95] konkret mit der Umsetzung der Muster in Programmcode. Im folgenden Teil werden beide Ansätze kombiniert, um eine ganzheitliche Unterstützung des Systementwicklers zu erreichen. Dabei ist zu beachten, dass durch die Verwendung von Mustern die objektorientierten Analyse- und Designmethoden nicht ersetzt, sondern ergänzt werden.

1. Zunächst ist der gesamte Softwareentwicklungsprozess mit Hilfe der bekannten objekt-orientierten Methoden, wie z. B. der Object Modeling Technique (vgl. [Rumb91]) oder der Object Oriented Analysis (vgl. [CoYo91]), zu spezifizieren. Nach Durchführung der Spezifikation liegt die Systemarchitektur im Groben vor, so dass sich mit Hilfe der geforderten Systemeigenschaften und der Analyseergebnisse ein passendes Architekturmuster auswählen lässt. Im Anschluss daran erfolgt die Umsetzung des Musters, wobei es erforderlich sein kann, weitere Muster rekursiv einzubeziehen.
2. Falls sich für bestimmte Probleme bei der Umsetzung oder bereits bei der Architekturentscheidung keine Muster finden lassen, sind weitere Quellen zu Rate zu ziehen. Finden sich auch dort keine passenden Muster, sind die allgemeinen Analyse- und Designrichtlinien als Hilfestellung bei der Lösung zu konsultieren.

3. Im Rahmen der Auswahl und Umsetzung der Muster ist zunächst einmal die Beschreibung des Musters und der verwendeten Komponenten, deren Struktur und Zusammenarbeit zu studieren und auf ihre Anwendbarkeit im konkreten Problemfall zu untersuchen. Falls das gewählte Muster geeignet ist, sind die Designelemente mit Namen zu versehen, die im Kontext der Anwendung eine konkrete Bedeutung haben. Anschließend sind die zur Umsetzung erforderlichen Klassen zu definieren sowie die Operationen innerhalb des Musters mit geeigneten Namen zu versehen. Zum Abschluss ist die Logik für die Operationen zu implementieren. An diesem Punkt kann es notwendig sein, weitere Pattern für Unterprobleme bei der Implementierung der Operation zu verwenden.

Aus dem beschriebenen Vorgehen für die Anwendung von Mustern lassen sich drei Forderungen an Musterbibliotheken ableiten:

1. Es müssen ausreichend viele Muster zur Verfügung stehen: Der Entwickler wird damit in die Lage versetzt, für zahlreiche Probleme alternative Lösungsstrukturen zu finden.
2. Ein schneller und präziser Zugriff auf die Wissensbasis ist wünschenswert: Es werden Musterkategorien benötigt, welche für den Programmierer leicht zu erlernen sind.
3. Die Wissensbasis muss so konzipiert sein, dass neue Muster aufgenommen und alte entfernt werden können. Außerdem muss das Kategorisierungsschema an die neuen Muster anpassbar sein.

Hat der Programmierer anwendbare Muster gefunden, so muss er in der Lage sein, die Alternativen miteinander zu vergleichen, auch wenn diese aus verschiedenen Wissensbasen stammen. Eine unabdingbare Forderung dafür ist wiederum eine möglichst einheitliche, umfassende und leicht erfassbare Dokumentation bzw. Beschreibung der Muster.

Ausgehend von den Anforderungen an die Dokumentation von Mustern wird im Folgenden ein Kategorisierungsschema von Buschmann vorgestellt und diskutiert.

6.4.5 Beschreibung von Mustern

Muster müssen in einer möglichst einheitlichen Form beschrieben werden. Dem Nutzer müssen die notwendigen Informationen in ausreichendem Umfang und in einer leicht erfassbaren und verständlichen Darstellung zur Verfügung gestellt werden. Um diese Anforderungen an die Musterbeschreibung zu erfüllen, existieren verschiedene Vorschläge.

Einen ersten Ansatz liefert [Pree94] auf den S. 63ff. Der Autor beschreibt jedoch lediglich die Notationen für Muster, wobei sowohl informale Textbeschreibungen, formale Textnotationen (Programmiersprache oder andere Formalismen) als auch graphische Beschreibungen zugelassen sind.

Um die im Abschnitt 6.4.2 definierten Anforderungen zu erfüllen, sollten die Beschreibungen auf dem Schema *Kontext-Problem-Lösung* (vgl. [Bus$^+$96], S. 19ff und [Gam$^+$95], S. 6ff) basieren. Für das Auffinden und Vorselektieren von Mustern ist jedoch dieses Schema nicht ausreichend. Zusätzlich werden folgende Elemente gefordert:

1. Name
2. Einführendes Beispiel
3. Beschreibung des Musters (Kontext, Problem, Lösung)
4. Hinweise zur Implementierung
5. Konsequenzen der Anwendung

Für eine schnelle Identifikation muss das Muster zwingend mit einem *kurzen, prägnanten Namen* bezeichnet werden. Dieser ist entscheidend, da er Bestandteil des Designvokabulars wird, wenn z. B. komplexe Systeme in Form von zusammengesetzten Mustern beschrieben werden. Des Weiteren sollte die Beschreibung aber auch bekannte Synonyme für das Muster enthalten, um Missverständnissen vorzubeugen.

Anschließend wird in einem *einführenden Beispiel* eine konkrete Problemsituation und die Lösung derselben dargelegt. Dieses Beispiel dient nicht nur dem Verständnis und als Bezugspunkt für die sich anschließende abstrakte Beschreibung, sondern zeigt auch die zu beachtenden Zwänge bei der Lösung in praxi. An das Beispiel können sich Hinweise anschließen, die den Entwickler bei der Prüfung der Anwendbarkeit des Musters für das vorliegende Problem unterstützen.

An diesen Block der Vorbetrachtungen schließt sich die *eigentliche Beschreibung des Musters* in Form des oben definierten Schemas an. Dabei sollten die textuellen Beschreibungen, insbesondere bei der Darstellung der Struktur und des Verhaltens der Designelemente durch graphische Beschreibungsmittel, wie z. B. der UML, ergänzt werden.

Es folgen *Hinweise zur Implementierung* des Musters und eventueller Beispielcode. Dabei ist dieser Teil nicht als feste Vorgabe, sondern lediglich als Vorschlag bzw. Warnung vor bekannten Fallen bei der Umsetzung des Musters zu betrachten. Abgeschlossen wird dieser Teil mit einer kurzen Angabe von bekannten Varianten der Lösung.

Der letzte Teil beschäftigt sich mit der Darstellung der *Konsequenzen der Anwendung* des Musters. Dabei werden nicht nur die erreichten

Ergebnisse, sondern auch die dabei eingegangenen Verpflichtungen sowie eventuell resultierende Nachteile dargestellt. Dies ist wichtig, um einen Vergleich zwischen alternativen Mustern durchführen zu können. Des Weiteren sollten auch bekannte Anwendungen aus verschiedenen Domänen dargestellt werden. Abschließend werden andere durch das Pattern referenzierte Muster sowie weitere ähnliche Muster aufgelistet.

6.4.6 Beispiel 1 – Das Observer-Muster

Name

Name: *Observer*
Synonyme: *Beobachter*, *Publisher-Observer*, *Publisher-Subscriber*, *Publizieren-Abonnieren*, *Abhängigkeit*, *Subjekt-Beobachter*.

Einführendes Beispiel

Ein Objekt verwaltet die Daten einer Wahlprognose, d. h. wie viel Prozent der Bevölkerung Partei A, B oder C wählen würden. Diese Daten können unterschiedlich dargestellt werden, bspw. als Tabelle, als Säulen- oder Kreisdiagramm. Kommen neue Auszählungsergebnisse hinzu, wird automatisch jede der Darstellungen aktualisiert.

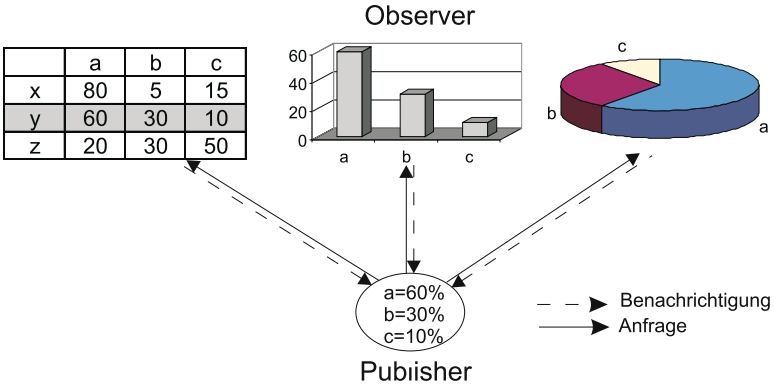


Abb. 6.23. Beispielanwendung des Observer-Musters

Beschreibung des Musters

Kontext

Das Muster wird in Systemen angewandt, in denen Objekte die Daten anderer Objekte verwenden. Dabei kommt es vor, dass die Quellkompo-

nente ihren inneren Zustand ändert und damit die Daten der Zielkomponenten veraltet sind. Deshalb muss es einen Mechanismus geben, der sicherstellt, dass die Veränderungen der Quellkomponente den Zielkomponenten mitgeteilt werden (Publisher-Subscriber) oder die Zielkomponenten sich selbständig über die Quelle informieren (Observer). Dabei ist zu beachten, dass zur Design- bzw. Implementierungszeit die Anzahl und Art der Zielkomponenten noch nicht bekannt sind. Weiterhin sollte lediglich eine lose Kopplung erreicht werden, um die Quellkomponente möglichst unabhängig von den Zielkomponenten entwickeln zu können.

Kurz bedeutet dies: Das Observer-Muster wird dazu angewandt, Datenänderungen in einer Komponente (Publisher) den von diesen Daten abhängigen Komponenten (Observer) mitzuteilen.

Problem

- Es müssen eine oder mehrere Komponenten über die Zustandsänderung einer Komponente informiert werden.
- Die Anzahl und Identität der abhängigen Komponenten ist anfangs nicht bekannt und kann sich im Laufe der Zeit auch ändern.
- Das explizite Auswählen neuer Informationen durch die abhängigen Komponenten ist nicht möglich.
- Quell- und Zielkomponenten sollten möglichst locker gekoppelt sein.

Lösung

Eine Komponente übernimmt die Rolle des Publishers (genannt „Subjekt“ in [Gam$^+$95]). Dieser besitzt eine Liste von Subscribern (Observer), die bei einer Änderung seines inneren Zustandes informiert werden. Möchte sich ein Objekt in dieser „Abonnentenliste“ ein- oder austragen, so nutzt es das Subscribe-Interface des Publishers. Generell kann man zwischen dem Push- und dem Pull-Modell unterscheiden. Beim Push-Modell sendet der Publisher alle geänderten Daten. So können die Subscriber nicht wählen, ob oder wann sie die Informationen erhalten möchten. Das andere Extrem ist das Pull-Modell, bei dem der Publisher die Subscriber nur darüber informiert, dass sich etwas geändert hat. Die Subscriber müssen nun selbst herausfinden, welche Daten betroffen sind. Zwischen diesen Extremen gibt es zahlreiche Variationen, von denen hier folgende stellvertretend aufgeführt werden:

Gatekeeper – Das Publizieren kann auch aus zwei Prozessen bestehen: Ein Prozess erzeugt die Nachrichten und ein anderer, der „Gatekeeper“, informiert die Subscriber, wenn ein Ereignis eingetreten ist, für das sie registriert sind.

Event Channel – Diese Variante ist besonders für verteilte Systeme geeignet. Es kann mehrere Publisher geben, die jeweils die Identität ihrer Subscriber nicht kennen. Analog dazu wissen die Subscriber nur, dass Datenänderungen aufgetreten sind, aber nicht bei welchem Publisher. Quell- und Zielkomponenten sind also streng entkoppelt. Soll eine Nachricht publiziert werden, so sendet sie der Publisher an den Event Channel, der für ihn als Subscriber erscheint. Die Subscriber sehen den Event Channel wiederum als Publisher, der sie über Datenänderungen informiert.

Der Event Channel kann zusätzlich noch mit einem Puffer ausgestattet sein, so dass er die Benachrichtigung der Subscriber zu einem von ihm gewählten Zeitpunkt durchführen kann. Außerdem ist es möglich, mehrere Event Channel zu verketten.

Producer-Consumer – Es existiert genau ein Produzent, der einen einzigen Konsumenten mit Informationen versorgt. Oft wird ein Puffer zwischen Quell- und Zielkomponente gesetzt und beide damit streng voneinander gelöst. Der Produzent schreibt in den Puffer und der Konsument liest daraus. Die einzige Synchronisation zwischen beiden besteht darin, dass der Produzent angehalten wird, wenn der Puffer voll ist, der Konsument hingegen bei leerem Puffer.

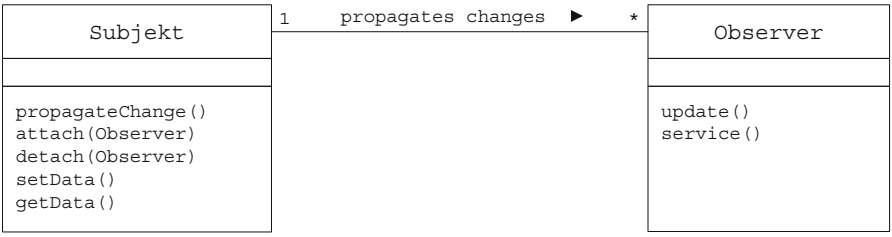


Abb. 6.24. Klassenstrukturmodell des Publisher-Observer-Musters

Hinweise zur Implementierung

Damit ein Objekt, welches als Observer bei einem anderen registriert wurde, auch bei Bedarf aus dem Speicher entfernt werden kann, darf man das Deregistrieren nicht vergessen.

Wenn ein Observer mehrere Publisher besitzt, ist es sinnvoll, bei jeder Nachricht jeweils den Namen des Publishers als Parameter mitzusenden.

Welche Komponente löst das Update aus? Eine Möglichkeit ist, dass der Publisher bei jeder Änderung seines Zustandes die Subscriber informiert. Bei häufigen Datenänderungen kann es dabei allerdings zu einer Reihe von unnötigen Zwischenupdates kommen. Dies lässt sich vermeiden, wenn die Observer in regelmäßigen Abständen ein Update aufrufen. Ein Nachteil an dieser Variante ist, dass die Observer zusätzliche Verantwortung bekommen und damit häufiger Fehler auftreten können.

Wird ein Publisher entfernt, müssen auch alle Referenzen auf ihn gelöscht werden. Das kann realisiert werden, indem der Publisher seinen Observern mitteilt, wenn er gelöscht wird.

Es ist darauf zu achten, dass die Daten des Publishers in sich konsistent sind, bevor die Observer benachrichtigt werden.

Man kann die Effizienz der Updates verbessern, indem man das Observer-Interface so erweitert, dass sich die Observer nur für bestimmte Ereignisse registrieren lassen können.

Bei komplexen Beziehungen zwischen Publishern und Observern kann es sinnvoll sein, einen Change Manager einzuführen, der diese kennt. Wenn mehrere Publisher in Folge geändert werden, sorgt er dafür, dass die Observer nur einmal informiert werden, nachdem alle Änderungen erfolgt sind.

Konsequenzen der Anwendung

Konsequenzen aus der Anwendung sind:

- Publisher und Observer können unabhängig voneinander wiederverwendet werden.
- Observer können neu hinzugefügt oder entfernt werden, ohne das Subjekt (Publisher) oder andere Observer zu ändern.
- Die abstrakte Kopplung zwischen Publisher und Observer wird durch die Benachrichtigung erreicht. Publisher und Observer gehören verschiedenen Schichten der Benutzerhierarchie an, ohne dabei Zyklen zu erzeugen.

6.4.7 Beispiel 2 – Das MVC-Muster

Name

Name: *Model-View-Controller-Muster (MVC)*

Einführendes Beispiel

Ein Informationssystem soll die Ergebnisse einer Wahl verwalten und graphisch darstellen. Der Nutzer bedient das System über ein graphisches User-Interface. Er kann Daten mittels einer dafür vorgesehenen Tabelle aktualisieren. Diese Daten sollen auf verschiedene Art und Weise repräsentiert werden, z. B. durch ein Kreis- oder Balkendiagramm. Die unterschiedlichen Repräsentationen müssen die Datenänderungen sofort widerspiegeln. Außerdem soll es möglich sein, ohne größeren Aufwand eine neue Darstellung einzufügen, z. B. eine Grafik, die die Platzverteilung im Bundestag abbildet. Zusätzlich soll das System auf verschiedene Plattformen mit unterschiedlichen „look and feel“ Standards übertragbar sein.

Beschreibung des Musters

Kontext

Das MVC-Muster wird bei interaktiven Anwendungen angewandt, die ein flexibles User-Interface besitzen. Das *Model-View-Controller-Muster (MVC)* teilt dabei eine interaktive Anwendung in drei Teile: Das *Modell* beinhaltet die eigentliche Funktionalität und die Daten des Programmes. Die *Views* dienen der graphischen Darstellung der Informationen. *Controller* verarbeiten die Eingaben des Benutzers. *Views* und *Controller* zusammen bilden das User-Interface. Damit die Daten des *Modells* und des User-Interfaces konsistent bleiben, ist ein Mechanismus zum Bekanntgeben von Änderungen erforderlich.

Problem

Wird eine Anwendung geändert, so muss meist auch das User-Interface angepasst werden. Wenn zum Beispiel neue Funktionen hinzukommen, müssen die Menüs so abgeändert werden, dass man diese auch erreicht. Es kann auch notwendig sein, die Oberfläche an die Bedürfnisse unterschiedlicher Benutzer anzupassen. Eine Schreibkraft benötigt zum Beispiel Formulare zum Eintragen der Daten mit der Tastatur. Ein Manager dagegen benutzt verschiedene Buttons und Icons, um dieselbe Anwendung zu bedienen. Eine Anwendung sollte auch auf unterschiedliche Plattformen übertragbar sein, die verschiedene „look and feel“ Standards haben. Auch bei einer Aufrüstung der Anwendung durch ein neues Release verändert sich der Programmcode.

Es ist sehr kostspielig, eine Anwendung mit dieser Flexibilität zu entwickeln, wenn die Programmlogik und das User-Interface eng miteinander verbunden sind. Außerdem ist dann die Fehleranfälligkeit relativ hoch.

Deshalb besteht eine andere Möglichkeit darin, für jede Implementierung des User-Interface ein eigenes Programm zu entwickeln. Folgende Anforderungen sollte die Lösung erfüllen:

- Es muss möglich sein, Informationen in verschiedenen Fenstern unterschiedlich darzustellen.
- Die Anzeige und das Verhalten der Anwendung sollten Datenänderungen sofort widerspiegeln.
- Änderungen am User-Interface müssen einfach vorzunehmen sein, wenn möglich sogar während der Laufzeit.
- Die Unterstützung unterschiedlicher „look and feel“ Standards darf keine Codeänderungen im funktionalen Kern der Anwendung bedingen.

Lösung

Das MVC-Muster wurde zuerst in der Smalltalk-80 Programmierumgebung angewandt. MVC unterteilt eine interaktive Anwendung in drei Teile: *processing*, *output* und *input*.

- Das *Modell* kapselt die Kernfunktionalität und die Daten der Anwendung. Es ist unabhängig von der graphischen Darstellung der Informationen oder dem Verhalten bei der Eingabe von Daten.
- *Views* zeigen die Informationen, die sie vom Modell erhalten, dem Nutzer an. Es kann mehrere *Views* für ein *Modell* geben.

Jede *View* ist einem *Controller* zugeordnet, der den Input verarbeitet. Wird zum Beispiel mit der Maus ein Button angeklickt, verarbeitet der *Controller* dieses Ereignis und fordert daraufhin beim *Modell* die Ausführung einer bestimmten Funktion an.

Die Trennung des *Modells* von den *Views* und den *Controllern* ermöglicht es, dass verschiedene Sichten auf ein Modell existieren. Ändert eine *View* das *Modell* durch ihren *Controller*, dann müssen alle anderen *Views* dies möglichst schnell widerspiegeln. Um das zu realisieren, informiert das *Modell* jedesmal all seine *Views*, wenn sich sein Zustand geändert hat. Dieser Mechanismus zum Bekanntgeben von Änderungen wird unter Abschnitt 6.4.6 näher beschrieben.

Eine Variante des MVC-Musters ist die *Document-View*. Hier wird die Trennung zwischen *View* und *Controller* aufgelockert. So sind z. B.

bei einigen Plattformen die Ereignisverarbeitung und die Darstellung der Fenster sehr eng miteinander verbunden (vgl. [Bus+96]). Man kann die Aufgaben von *View* und *Controller* in einer Komponente zusammenfassen, gibt damit aber auch die Austauschbarkeit der *Controller* auf. In einer Dokument-View-Architektur entspricht das Dokument dem *Modell* im MVC-Muster. Es implementiert gleichfalls einen Benachrichtigungsmechanismus. Die View-Komponente in der Dokument-View-Architektur vereinigt die Aufgaben von *View* und *Controller* im MVC und stellt die Benutzerschnittstelle zur Verfügung. Wie auch beim MVC ermöglicht die lose Kopplung zwischen Dokument- und View-Komponenten, dass gleichzeitig verschiedene *Views* für ein Dokument existieren. Die Dokument-View-Architektur ist z. B. in die C++ Umgebung integriert, die Foundation Class Bibliothek für die Entwicklung von Windows Applikationen.

Struktur der Designelemente: Das *Modell* enthält die Kernfunktionalität und die Daten der Anwendung. Es stellt Methoden zur programmspezifischen Verarbeitung von Informationen zur Verfügung, die von *Controllern* im Namen des Nutzers aufgerufen werden. Außerdem liefert es Methoden, mit denen die abhängigen *Views* bei einer Zustandsänderung des *Modells* benachrichtigt werden. Um dies zu realisieren, besitzt das *Modell* eine Liste von abhängigen Komponenten. Alle *Views* sowie bestimmte *Controller* lassen sich in dieser Liste registrieren. Eine Datenänderung des *Modells* löst eine Benachrichtigung aller eingetragenen Komponenten aus. Dieser Mechanismus stellt die einzige Verbindung zwischen dem *Modell* und den *Views* und *Controllern* dar. Zusammenfassend übernimmt das *Modell* folgende Aufgaben:

- Es liefert die Kernfunktionalität der Anwendung,
- registriert abhängige *View* und *Controller* und
- benachrichtigt abhängige Komponenten über Datenänderungen.

Views stellen die Information für den Benutzer graphisch dar, wobei unterschiedliche *Views* die Informationen auf verschiedene Arten präsentieren. Jede *View* besitzt eine Methode `update()`. Wird diese aufgerufen, dann fordert die *View* die aktuellen Daten vom *Modell* und stellt diese auf dem Bildschirm dar. Während der Initialisierung werden alle *Views* mit dem *Modell* verknüpft und bei dem Benachrichtigungsmechanismus registriert. Jede *View* erzeugt ihren eigenen *Controller*. Oft bieten die *Views* Funktionen an, mit denen die Anzeige durch den *Controller* manipuliert werden kann, ohne dass sich im *Modell* etwas ändert. Ein Beispiel dafür ist das Scrollen von Bildschirmseiten. Zusammenfassend übernimmt die *View* folgende Aufgaben:

- Erzeugen und Initialisieren des zugehörigen *Controllers*,
- Anzeigen von Informationen für den Nutzer,
- Implementierung von `update()` und
- Lesen der Daten aus dem *Modell*.

Controller verarbeiten Nutzereingaben als Ereignisse. Auf welchem Wege ein *Controller* diese Events erhält, hängt von der Plattform der Benutzerschnittstelle ab. Der Einfachheit halber gehen wir davon aus, dass jeder *Controller* eine Methode `handleEvent()` zur Ereignisbehandlung implementiert, die für jedes relevante Ereignis aufgerufen wird. Dieses wird verarbeitet und als Methodenaufruf an das *Modell* oder die zugehörige *View* weitergeleitet. Wenn das Verhalten des *Controllers* vom Zustand des *Modells* abhängt, registriert er sich selbst im Benachrichtigungsmechanismus des *Modells* und implementiert die Methode `update()`. Das ist zum Beispiel notwendig, wenn eine Änderung im *Modell* einen Menüpunkt ein- und ausschaltet. Zusammenfassend übernimmt der *Controller* folgende Aufgaben:

- Entgegennahme der Nutzereingaben als Ereignisse
- Umwandeln der Ereignisse in Service-Abfragen an das *Modell* oder die *View*
- Implementieren der Methode `update()`, wenn nötig

In unserem Beispiel besitzt das *Modell* die gesamten Stimmzahlen aller Parteien und erlaubt den *Views*, diese abzufragen. Außerdem definiert es Methoden zur Datenmanipulation für die *Controller*.

Wir definieren verschiedene *Views*: ein Balkendiagramm, ein Kreisdiagramm und eine Tabelle. Diese Diagrammdarstellungen nutzen *Controller*, die das *Modell* nicht beeinflussen. Die Tabelle hingegen ist mit einem *Controller* zur Dateneingabe verbunden.

Das MVC-Muster kann angewandt werden, um ein System interaktiver Anwendungen zu entwickeln, wie z. B. in der Smalltalk-80 Umgebung. Solch ein System bietet vorgefertigte View- und Controller-Subklassen für häufig verwendete Elemente wie z. B. Menüs, Buttons oder Listen.

Verhalten des MVC-Musters zur Laufzeit: Die folgenden beiden Szenarios beschreiben das Verhalten des MVC-Musters. Der Einfachheit halber werden hier nur ein *Controller* und eine *View* abgebildet.

Szenario I zeigt, wie Nutzereingaben, die Änderungen im Modell verursachen, den Benachrichtigungsmechanismus auslösen:

1. Der Controller nimmt Nutzereingaben als Ereignisse durch seine Methode `handleEvent()` entgegen. Er interpretiert die Ereignisse und aktiviert eine Methode `service()` des *Modells*.

2. Das *Modell* führt den angeforderten Service aus und verändert dadurch seine internen Daten.
3. Das *Model* benachrichtigt alle *Views* und *Controller*, indem es deren Methoden `update()` aufruft.
4. Jede *View* fordert die geänderten Daten vom *Modell* an und erneuert seine Anzeige.
5. Jeder registrierte *Controller* holt die Daten vom *Modell*, um bestimmte Funktionen für den Nutzer zu aktivieren oder deaktivieren.
6. Der *Controller* beendet seine Methode `handleEvent()`.

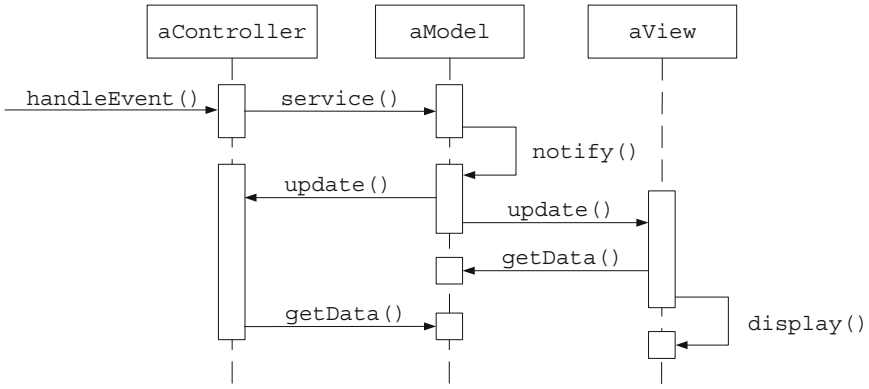


Abb. 6.25. Sequenzdiagramm Szenario I

Szenario II zeigt, wie *Modell*, *View* und *Controller* initialisiert werden. Dieser Code befindet sich meist außerhalb der MVC Komponenten, z. B. in einer Methode `public static void main(String[] args)` (vgl. Abschnitte 2.5.10 und 4.4). Die Initialisierung der einzelnen *Views* und *Controller* verläuft auf ähnliche Art und Weise. Folgende Schritte werden ausgeführt:

1. Die Instanz des *Modells* wird erzeugt. Anschließend initialisiert diese ihre internen Datenstrukturen.
2. Eine *View* wird gebildet. Diese erhält als Parameter bei der Initialisierung eine Referenz auf das zugehörige *Modell*.
3. Die *View* ruft die Methode `attach()` des *Modells* auf und registriert sich damit beim Benachrichtigungsmechanismus des *Modells*.

4. Die *View* fährt mit ihrer Initialisierung fort, indem sie ihren *Controller* erzeugt. An die Initialisierungsprozedur des *Controllers* wird die Referenz auf sich selbst und das Modell weitergegeben.
5. Auch der *Controller* registriert sich damit beim Benachrichtigungsmechanismus des *Modells*, indem er die `attach()` Methode aufruft.
6. Nach der Initialisierung beginnt die Anwendung, Prozesse zu verarbeiten.

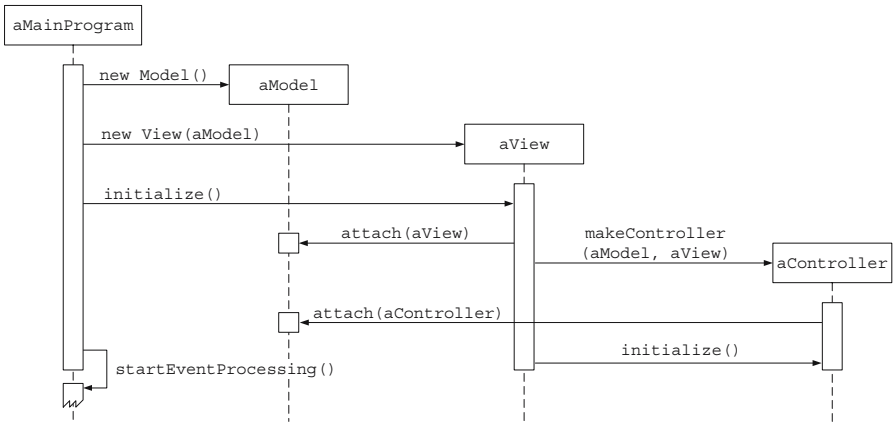


Abb. 6.26. Sequenzdiagramm Szenario II

Hinweise zur Implementierung

Laut Buschmann sind die folgenden Schritte erforderlich, um eine MVC basierte Anwendung zu entwickeln:

1. Trennen der Kernfunktionalität von Funktionen zur Kommunikation zwischen Mensch und Computer
2. Implementieren eines Benachrichtigungsmechanismus
3. Entwurf und Implementierung der *Views*
4. Entwurf und Implementierung der *Controller*
5. Entwurf und Implementierung der *View-Controller*-Beziehung
6. Initialisierung des MVC
7. Dynamische Erzeugung von *Views*
8. Austauschbare *Controller* implementieren
9. Infrastruktur für hierarchische *Views* und *Controller* schaffen
10. Weiteres Entkoppeln von Abhängigkeiten

Im Schritt 1 wird der Bereich der Anwendung analysiert und die Kernfunktionalität von den erforderlichen Datenein- und Ausgabefunktionen getrennt. Daraufhin wird das *Modell* entworfen und die für den Kern der Anwendung benötigten Daten und Funktionen werden gekapselt. Für die Anzeige müssen Funktionen für den Zugriff auf die entsprechenden Daten bereitgestellt werden. Nachdem entschieden ist, welche Funktionen des *Modells* für den Nutzer via *Controller* zugänglich sein sollen, muss dem *Modell* eine entsprechende Schnittstelle hinzugefügt werden.

Das *Modell* im Beispiel speichert jeweils in einer Liste die Namen der Parteien und die zugehörigen Stimmzahlen. Über die Methoden `changeStimmzahl()` und `resetStimmzahlen()` können die Daten geändert, bzw. alle Stimmzahlen auf den Wert 0 gesetzt werden.

```
public class Model {

  private List stimmzahlen;
  private List parteien;

  public Model(List parteinamen) {
  }

  /**
   * dient dem Datenzugriff durch Controller
   */
  public void resetStimmzahlen() {
  }

  public void changeStimmzahl(String partei, Integer stimmzahl) {
  }

  //... wird noch fortgesetzt
```

Als nächstes wird im Schritt 2 das Benachrichtigungsschema implementiert. Dazu kann das *Observer-Muster* genutzt werden, welches unter Abschnitt 6.4.6 näher beschrieben wird. Dem *Modell* wird die Rolle des Publishers zugeordnet. Es wird um einen Registrierungsvektor `registry` erweitert, der die Referenzen auf alle angemeldeten Observer enthält. Um diese Registry verwalten zu können, ist es notwendig, den *Views* und *Controllern* Methoden bereitzustellen, um sich beim Benachrichtigungsmechanismus an- und abzumelden. Nach jeder Änderung des Zustands des *Modells* werden die Observer benachrichtigt. Die Me-

thoden `resetStimmzahlen()` und `changeStimmzahl()` rufen `notify()` des *Modells* auf. Hier wird die Aktualisierung aller Observer mittels `update()` veranlasst.

```
public class Model {

// Fortsetzung...

  private Vector registry;

  /**
   * zur An– bzw. Abmeldung eines Observers
   */
  public void attachRegistry(Observer obs) {
  }

  public void detachRegistry(Observer obs) {
  }

  /**
   * alle Observer benachrichtigen
   */
  private void notifyObserver() {
  }
}
```

```
public interface Observer {

  public void update();

}
```

Im Schritt 3 wird die Erscheinung jeder einzelnen *View* entworfen. Die `display()` Methode, die dazu dient, die *View* auf dem Bildschirm anzuzeigen, wird spezifiziert und implementiert. In erster Linie benötigt diese Methode die Daten des *Modells*. Der Rest der Methode ist abhängig von der Plattform der Benutzerschnittstelle. So werden dann zum Beispiel Methoden zum Zeichnen von Linien oder zur Darstellung von Text aufgerufen.

Außerdem muss die `update()` Methode implementiert werden, um Datenänderungen im *Modell* widerspiegeln zu können. Die einfachste

Variante dafür ist, die `display()` Methode aufzurufen. Sie fordert alle für die Darstellung benötigten Daten an und stellt diese dar. Dieses Vorgehen kann allerdings schnell ineffizient werden, wenn die *View* sehr komplex ist und Aktualisierungen in relativ kurzen Zeitabständen notwendig sind. In diesem Fall gibt es zahlreiche Strategien zur Optimierung. Eine davon ist, der `update()` Methode zusätzliche Parameter zur Verfügung zu stellen. Die *View* kann dann entscheiden, ob eine Auffrischung notwendig ist. Eine andere Lösung besteht darin, das Neuzeichnen zwar zu vermerken, jedoch nicht auszuführen, wenn angenommen werden kann, dass noch weitere Änderungen folgen. Wenn schließlich keine weiteren Ereignisse bevorstehen, wird die `display()` Methode ausgeführt. Damit kann eine Reihe von unnötigen Aktualisierungen vermieden werden.

Zusätzlich zu den `update()` und `display()` Methoden benötigt jede *View* eine Initialisierungsprozedur `initialize()`. Durch diese wird die *View* beim Benachrichtigungsmechanismus angemeldet und die Verbindung zum *Controller* aufgebaut wie im Schritt 5. Nachdem der *Controller* initialisiert ist, stellt sich die *View* selbst auf dem Bildschirm dar. Durch die Plattform oder den *Controller* könnte noch weitere Funktionalität gefordert werden, wie z. B. das Verkleinern des Darstellungsfensters.

Für alle später im System verwendeten *Views* wird nun eine Basisklasse `View` definiert. Diese wird von `Frame` abgeleitet und implementiert das Interface `Observer`. Sie enthält als Variablen Referenzen auf ihr *Modell* und ihren *Controller*. Die Verbindung zum *Modell* wird gleich zu Beginn im Konstruktor hergestellt, indem die *View* beim Benachrichtigungmechanismus angemeldet wird. Der *Controller* wird in der Methode `initialize()` erzeugt.

```
public class View extends Frame implements Observer{

  private Model myModel;
  private Controller myController;

  /**
   * Konstruktor. Hier wird die View als Observer
   * beim zugehörigen Modell angemeldet.
   */
  public View(Model aModel) {
    myModel=aModel;
    myModel.attachRegistry(this);
  }
```

```
  /**
   * Abfrage aller notwendigen Daten und Anzeige im Fenster
   */
  public void display() {
  }

  /**
   * aus dem Interface Observer: Aktualisieren der Daten
   */
  public void update() {
  }

  /**
   * Initialisierung --> Erzeugen des Controllers
   */
  public void initialize() {
  }
}
```

Im nun folgenden Schritt 4 wird die Klasse `em Controller` entworfen und implementiert. Für jede *View* sollte die Reaktion des Systems auf Aktivitäten des Nutzers spezifiziert werden. Es wird unterstellt, dass die darunterliegende Plattform die Handlungen des Nutzers als Ereignisse übergibt. Ein *Controller* erhält und interpretiert diese Ereignisse mittels einer speziellen Methode `handleEvent()`. Bei nicht trivialen *Controllern* hängt diese Interpretation vom Zustand des *Modells* ab. Bei der Initialisierung wird ein *Controller* an sein *Modell* und seine *View* gebunden und ermöglicht die Verarbeitung von Ereignissen. Wie dies im Einzelnen erreicht wird, hängt von der Plattform ab.

Wie auch die Klasse `View` implementiert die Klasse `Controller` das Interface `Observer`. Die Methode `update()` wird neu definiert. Da die *Views* in unserem Beispiel nur zur Anzeige der Daten dienen, benötigen sie keine spezielle Methode zur Ereignisbehandlung. Deshalb wird nur die leere Methode `handleEvent()` definiert. Diese kann später von einer Subklasse von `View` überschrieben werden, z. B. von einer Tabellendarstellung, die Nutzereingaben akzeptiert und daraufhin die Daten des *Modells* ändert.

```
public class Controller implements Observer{

  private Model myModel;
```

```
  private View myView;

  /**
   * Konstruktor. Hier wird der Controller
   * beim zugehörigen Modell angemeldet.
   */
  public Controller (Model aModel, View aView) {
    myModel = aModel;
    myView = aView;
    myModel.attachRegistry(this);
  }

  /**
   * aus dem Interface Observer: Aktualisieren der Daten
   */
  public void update() {
  }

  /**
   * Methode zur Ereignisbehandlung
   */
  public void handleEvent(Event e) {
  }
}
```

Nun wird im Schritt 5 die Beziehung zwischen *View* und *Controller* entworfen und implementiert. Eine *View* erzeugt ihren zugehörigen *Controller* gewöhnlich bei ihrer Initialisierung. In einer übergeordneten View-Klasse wird eine Methode `makeController()` definiert. Jede *View*, die einen *Controller* benötigt, der von dem in der Superklasse festgelegten abweicht, definiert die `makeController()` Methode neu. In unserem Beispiel erzeugt die Subklasse `TableView` einen eigenen `TableController`, der auch Nutzereingaben verarbeiten kann.

```
public class View extends Frame implements Observer{

  /**
   * Initialisierung --> Erzeugen des Controllers
   */
  public void initialize() {
    myController = makeController();
  }
```

```
  public Controller makeController () {
    return new Controller(myModel,this);
  }
}
```

```
public class TableController extends Controller implements
Observer{

  public TableController (Model aModel, TableView aTableView) {
    super(aModel, aTableView);
  }

  public void handleEvent(Event e) {
    // Ereignisse interpretieren und ggf. Daten des Modells
    // aktualisieren.
  }
}
```

```
public class TableView extends View {

  private Model myModel;
  private Controller myTableController;

  public TableView(Model aModel) {
    super(aModel);
  }

  /**
   * Hier wird festgelegt, dass die View
   * in Form einer Tabelle dargestellt wird.
   */
  public void display() {
  }

  /**
   * Hier wird ein spezieller Controller erzeugt, der
   * auch Nutzereingaben akzeptieren soll
   */
  public Controller makeController() {
    return new TableController (myModel,this);
  }
```

```
}
```

Als Nächstes wird im Schritt 6 das MVC initialisiert, beginnend mit dem *Modell*. Danach werden die *Views* initialisiert und erzeugt. Nach der Initialisierung wird die Ereignisverarbeitung gestartet. Weil das *Modell* selbst unabhängig von speziellen Views und Controllern bleiben soll, befindet sich der Quellcode zur Initialisierung des MVC-Musters außerhalb des *Modells*.

Die nun folgenden Schritte 7 bis 10 behandeln zusätzliche Themen, die sich mit der Schaffung von mehr Freiheiten beim Entwickeln von komplexeren Systemen oder sehr flexiblen Anwendungen beschäftigen:

Wenn die Anwendung das dynamische Öffnen und Schließen von Views erlaubt, kann es im Schritt 7 sehr hilfreich sein, eine Komponente zur Verwaltung der geöffneten *Views* einzusetzen. Diese Komponente könnte auch dafür verantwortlich sein, die Anwendung zu beenden, wenn die letzte *View* geschlossen wurde. Für die Implementierung dieser View-Verwaltungs-Komponente kann das *View-Handler-Pattern* (vgl. [Bus+96], S. 291) genutzt werden.

Es ist leicht möglich, eine *View* mit verschiedenen *Controllern* zu nutzen (Schritt 8), da die Funktionalität zur Darstellung der *Views* von Funktionen zu deren Kontrolle getrennt ist. So können unterschiedliche Betriebsarten, z. B. für Experten oder für weniger erfahrene Nutzer, realisiert werden. Außerdem können verschiedene Ein- und Ausgabegeräte in eine Anwendung integriert werden.

Bei größeren Anwendungssystemen kann es sinnvoll sein, eine hierarchische Struktur für *Views* und *Controller* zu erstellen, wie hier im Schritt 9 beschrieben. Ein auf MVC basierendes System enthält wiederverwendbare *View*- und *Controller*-Klassen. Meist sind dies Klassen für sehr häufig verwendete Elemente, wie Buttons, Textfelder oder Listen. Um hierarchisch aufgebaute *Views* zu erstellen, kann das von Gamma beschriebene *Composite-Pattern* genutzt werden.

Wenn verschiedene *Views* gleichzeitig aktiv sind, kann es auch vorkommen, dass mehrere *Controller* zugleich an Ereignissen interessiert sind. Mit dem *Chain-of-Responsibility-Pattern* beschreibt Gamma, wie man die Reihenfolge bestimmen kann, in der die Ereignisse an die `handleEvent()` Methoden der einzelnen *Controller* gesendet werden. So kann ein *Controller* z. B. ein Ereignis unbearbeitet an den *Controller* der übergeordneten *View* weiterleiten.

Legt man besonderen Wert darauf, dass eine Anwendung auf verschiedenen Plattformen lauffähig ist, können die Abhängigkeiten zwischen *View* und *Controller* noch weiter gelöst werden (Schritt 10). Es

ist sehr kostspielig, ein System mit komplexen *View*- und *Controller*-Klassen zu entwickeln. Sinnvoller ist es, diese Klassen plattformunabhängig zu machen. Mit dem *Bridge-Pattern* stellt GAMMA eine Möglichkeit vor, dies zu erreichen. Views nutzen eine Klasse namens `Display` als eine abstrakte Klasse für Fenster und *Controller* eine Klasse `Sensor`.

Die Klasse `Display` definiert Methoden, um ein Fenster zu erzeugen, Linien und Text darzustellen, das Aussehen des Maus-Cursors zu ändern, usw. Die abstrakte Klasse `Sensor` legt plattformunabhängige Ereignisse fest. Jede ihrer Subklassen wandelt systemspezifische Ereignisse in plattformunabhängige Ereignisse um. Für jede unterstützte Plattform wird eine spezielle Subklasse von `Sensor` und `Display` gebildet.

Der Entwurf der abstrakten Klassen `Display` und `Sensor` ist nicht trivial, denn davon hängt sowohl die Effizienz des sich ergebenden Codes als auch die Effizienz ab, mit der die konkreten Klassen auf unterschiedlichen Plattformen implementiert werden können. Eine Variante ist, `Display` und `Sensor` mit einer nur sehr geringen Funktionalität auszustatten, die alle Plattformen direkt bereitstellen. Eine andere Möglichkeit besteht darin, dass die Klassen `Display` und `Sensor` einen höheren Grad an Abstraktion anbieten. Solche Klassen sind schwieriger zu erzeugen, verwenden aber mehr Quellcode von der Plattform der Benutzerschnittstelle. Folglich ist das Aussehen der so erzeugten Anwendungen mehr der jeweiligen Plattform angepasst, während die erste Variante zu Anwendungen führt, die auf allen Plattformen gleich erscheinen.

Konsequenzen der Anwendung

Vorteile

Die Anwendung des MVC-Musters bringt zahlreiche Vorteile. Es sind *Views* für ein und dasselbe *Modell* möglich. Durch die strikte Trennung des *Modells* und der Komponenten der Benutzerschnittstelle können zur Laufzeit gleichzeitig mehrere *Views* geöffnet sein. Die *Views* können dynamisch geöffnet und geschlossen werden.

Die *Views* können synchronisiert werden. Der Benachrichtigungsmechanismus des *Modells* informiert alle angemeldeten Observer über Datenänderungen. Dadurch werden alle abhängigen *Views* und *Controller* synchronisiert.

Durch die Entkopplung des *Modells* im MVC-Muster ist es möglich, die *Views* und *Controller* beliebig auszutauschen. Objekte der Benutzerschnittstelle können sogar während der Laufzeit gewechselt werden.

Durch die Unabhängigkeit des *Modells* von der Benutzerschnittstelle kann das Anwendungssystem auf verschiedene Plattformen übertragen werden, ohne dass dies Auswirkungen auf die Kernfunktionalität hat. Es müssen lediglich die entsprechenden *View*- und *Controller*-Klassen bereitgestellt werden.

Es ist möglich, mit der MVC Architektur komplexe Anwendungssysteme aufzubauen, wie in den Schritten 7-10 gezeigt wurde.

Nachteile

Auf der anderen Seite resultieren jedoch auch Nachteile aus der Anwendung des MVC-Musters. Die strikte Anwendung der MVC Architektur erhöht die Komplexität der Anwendung. Gamma führt an, dass getrennte *Modell*, *View* und *Controller* bei einfachen Textfeldern oder Menüs nur mehr Komplexität verursachen, ohne viel zusätzliche Flexibilität zu bringen.

Es kann zu einer übermäßig großen Anzahl von Aktualisierungen kommen, wenn sich Daten im *Modell* ändern. Um unnötige Benachrichtigungen zu vermeiden, sollte eine Unterscheidung gemacht werden, welche Ereignisse für die einzelnen *Views* interessant sind. So braucht z. B. ein Fenster kein Update, wenn es auf ein Icon minimiert ist, sondern erst dann, wenn es wieder seine normale Größe annimmt.

Da *View* und *Controller* eng zusammenhängende Komponenten sind, behindert dies ihre individuelle Wiederverwendung. So wird nie ein *Modell* ohne *Controller* verwendet oder umgekehrt. Eine Ausnahme bilden „read-only Views“ (vgl. [Bus+96], S.142), die einen *Controller* besitzen, welcher sämtliche Eingaben ignoriert.

Die noch relativ enge Kopplung zwischen *Views* und *Controller*n und dem *Modell* verringert die Flexibilität. Da *Views* und *Controller* direkt auf die Funktionen des *Modells* zugreifen, ändert sich der Code dieser Komponenten bei einer Änderung des Interface des *Modells*. Dieser Umstellungsaufwand steigt mit der Anzahl der vom *Modell* abhängigen *Views* und *Controller*. Man kann dieses Problem lösen, indem man auf indirektem Weg auf das *Modell* zugreift. Eine Möglichkeit, dies zu erreichen, beschreibt das *Command-Processor-Pattern* (vgl. [Bus+96], S. 277).

Da die *Views* vom Interface des *Modells* abhängig sind, kann es vorkommen, dass mehrere Abfragen notwendig sind, um alle für die Anzeige notwendigen Daten zu erhalten. Wenn Updates häufig durchgeführt werden, verschlechtert diese Ineffizienz beim Datenzugriff die Performance der Anwendung. Um dies zu vermindern, kann man die

geänderten Daten innerhalb der *View* puffern und damit mehrere aufeinanderfolgende Updates zusammenfassen.

Alle Abhängigkeiten von der jeweiligen User-Interface-Plattform werden in den *View*- und *Controller*-Komponenten gekapselt. Bei einem Wechsel der Plattform ist es also unvermeidbar, die *Views* und *Controller* zu ändern. Da beide Komponenten sowohl plattformabhängigen als auch -unabhängigen Code beinhalten, ist es bei komplexen Anwendungen oder Anwendungssystemen sinnvoll, plattformabhängige Teile zu kapseln. Dies ermöglicht eine einfachere Umstellung auf eine andere User-Interface-Plattform.

Bei der Verwendung von modernen Entwicklungswerkzeugen ist es oft schwierig, die MVC Architektur anzuwenden. Diese Werkzeuge definieren oft ihre eigenen Kontrollabläufe und behandeln einige Ereignisse intern, wie z. B. das Scrollen einer Liste oder das Anzeigen eines Popup-Menüs. Der Aufwand, diese vorgefertigten Komponenten in eine MVC Architektur zu integrieren, ist relativ hoch. Außerdem interpretieren Plattformen höheren Niveaus oft schon Ereignisse und bieten Methoden für alle möglichen Nutzeraktionen an. Somit wird die *Controller*-Funktionalität schon von dem Entwicklungswerkzeug mitgeliefert und eine zusätzliche Komponente dafür ist nicht mehr notwendig.

6.4.8 Ausblick

Die vergangenen Abschnitte haben sich mit dem Inhalt, der Verwendung und der Beschreibung von Mustern beschäftigt. Durch die Vorstellung der Kategorien könnte der Eindruck entstehen, dass bereits für die meisten Probleme ausreichend Muster zur Verfügung stehen und lediglich die Umsetzung der Muster noch Schwierigkeiten bereiten würde.

Einzig für den Bereich der Entwicklung graphischer Benutzeroberflächen existiert eine Vielzahl von Mustern. In anderen Bereichen, wie z. B. der Entwicklung von Sicherungs- oder Transaktionssystemen, besteht jedoch noch ein hoher Bedarf an Mustern. Deshalb wird die Entwicklung neuer Muster („Pattern Mining“) auch in Zukunft einen hohen Stellenwert einnehmen. Ein mögliches Vorgehen dabei wird von Buschmann et al. vorgeschlagen (vgl. [Bus+96], S. 376ff).

Neben der Entwicklung der Entwurfsmuster erwarten Buschmann et al. aber auch einen verstärkten Einsatz von Organisationsmustern, d. h. Muster für die Organisation einer Anforderungsanalyse.

Zudem existiert, unabhängig von entwickelten domänenunabhängigen Mustern, zunehmend ein Bedürfnis nach Verwendung von domänenspezifischen Mustern. Diese werden zur Dokumentation und Speicherung von Wissen in Unternehmen verwandt. So hat z. B. AT & T die

Vorgehensweise bei der Umstellung von Vermittlungsanlagen in Form von Mustern dokumentiert. Da diese Muster jedoch zumeist das gesamte Wissen einer Firma repräsentieren, haben die Firmen kein Interesse an einer breiten Nutzung der Muster durch die Öffentlichkeit.

Eine weitere Entwicklung, insbesondere im akademischen Bereich, stellen die Bestrebungen zur Formalisierung von Pattern dar. Diese Ansätze gehen dabei auf Richard Helm zurück, der 1990 Frameworks und ihre Komponenten formal beschrieb. Die dabei entstehenden formalen Kontrakte beschrieb der Autor wie folgt: „behavioural composition represents groups of interdependent objects cooperating to accomplish tasks ... Patterns of communication within a behavioural composition are often repeated throughout the system with different participating objects. Contracts are a construct for the explicit specification of behavioural compositions“ ([Pree94], S. 88).

Die Vorteile einer formalen Beschreibung liegen insbesondere in der Vereinfachung der Dokumentation und Archivierung der Muster. Außerdem lassen sich komplexe Kompositionen aus grundlegenden Kontrakten durch Verfeinerungen und die Einbeziehung von anderen Kontrakten aufbauen.

Die Formalisierung führt jedoch auch zu einer Einschränkung der Anwendbarkeit. Zunächst kann es in manchen Situationen sehr schwierig sein, Muster durch die verfügbaren Elemente ausreichend zu beschreiben. Dies könnte durch die Einführung neuer Elemente, welche aber dann die gewählte Notation komplizieren, gelöst werden. Außerdem wird durch die formal exakte Beschreibung eine Exaktheit der Spezifikation vorgegeben, die eventuelle Mängel verdeckt. Des Weiteren wird die Abstraktionsstufe nicht mehr durch das Problem und die Lösungsstruktur vorgegeben, sondern orientiert sich zu stark an der der Notation zugrunde gelegten objektorientierten Sprache.

Neben diesen Nachteilen, die zum großen Teil die Beschreibung der Muster betreffen, leidet aber auch die Anwendbarkeit durch den Entwickler. Ein Kriterium bei der Beschreibung von Mustern war die leichte Verständlichkeit für den potentiellen Anwender. Dies ist durch eine Formalisierung nicht notwendigerweise gegeben. Falls dem Anwender die Notation unbekannt ist, muss er sich zunächst in diese Notation einarbeiten, was im Gegenzug die Akzeptanz des Pattern Systems stark vermindert. Durch die Formalisierung erfolgt gleichzeitig eine Spezialisierung. Damit wird die Einsetzbarkeit dahingehend eingeschränkt, sodass sich formalisierte Beschreibungen sehr schwer auf informal beschriebene Probleme anwenden lassen. Mit der Spezialisierung ist gleichzeitig auch eine Detaillierung der Pattern verbunden, womit aber wiederum eine

Variantenbildung erschwert, wenn nicht gar unmöglich wird. Wie aber bereits in den vorangegangenen Abschnitten diskutiert wurde, stellt die Variabilität einen wichtigen Bestandteil des Nutzens von Mustern dar.

Zur Verbesserung der Anwendung von Mustern werden zur Zeit diverse Methoden und Werkzeuge entwickelt. Aber auch die Entwicklung von musterbasierten Analyse- und Designmethoden in der Softwareentwicklung ist in der Diskussion. Dabei wird aber die Sinnhaftigkeit derartiger Entwicklungen durch erfahrene Anwender von Mustern in Frage gestellt (vgl. [Bus+96] S. 425). Diese vertreten den Standpunkt, dass eine Anwendung von Pattern nur sinnvoll ist, wenn der Nutzer das Muster vorher vollständig verstanden hat. Dieses Verständnis kann aber nicht durch Werkzeuge oder Methoden erzeugt, sondern lediglich unterstützt werden. Aus diesem Grunde wird durch die Autoren die Entwicklung eines leistungsfähigen Pattern Browsers oder einer WWW-basierten Suchmaschine gegenüber patternbasierten Entwicklungsumgebungen favorisiert.

6.5 Übungsaufgaben

Aufgabe 12. Beschreiben Sie den Vorgang beim Löschen des Knotens E unter Verwendung des Algorithmus aus dem Abschnitt 6.3.5!

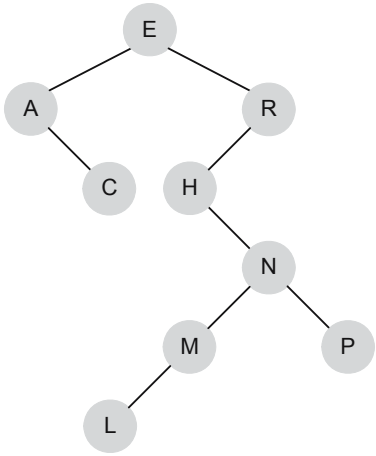


Abb. 6.27. Löschen des Knotens E

Aufgabe 13. Erläutern Sie den Begriff der Höhe eines Baumes!

Aufgabe 14. Die untenstehende Methode berechnet die FIBONACCI-Zahlen der n-ten Generation. Entwerfen Sie ein entsprechendes Aktivitätsdiagramm!

```
public int fibonacci (int n) {
  if (n<1) {
    if (n==0) {
      return 1;
    } else {
      return 0;
    }
  } else {
    return (fib(n−1)+fib(n−2));
  }
}
```

Aufgabe 15. Entwerfen Sie für den Algorithmus Quick Sort ein Aktivitätsdiagramm!

Aufgabe 16. Nach welchen Kriterien würden Sie die Effizienz von Such- und Sortieralgorithmen bewerten?

Aufgabe 17. Welches Suchverfahren würden Sie in den folgenden Fällen anwenden:

1. Eine Liste mit Zufallszahlen wurde erzeugt. Nun soll überprüft werden, ob eine bestimmte Zahl enthalten ist oder nicht.
2. In einem elektronischen Telefonbuch von Schwedt/Oder wird nach dem Eintrag von Erwin Schwabe gesucht.
3. In einem Feld, welches einige leere Stellen enthält, soll die erste freie Position ermittelt werden.

Aufgabe 18. Ein Feld enthält die Zahlen von 1 bis 100 in absteigender Reihenfolge. Schreiben Sie ein Programm, welches das Feld mittels direktem Einfügen, direktem Auswählen, Bubblesort und Quicksort aufsteigend sortiert. Richten Sie Zählvariablen ein, um die jeweils nötigen Vergleichs- und Austauschschritte zu zählen und vergleichen Sie die Werte der einzelnen Methoden!

Aufgabe 19. Der stark expandierende Hof des Bauern Obermoser verwendet seit einiger Zeit ein JAVA-Programm zur Verwaltung seiner Kühe, welches vor einiger Zeit eine Aushilfskraft programmiert hat. Die Klasse `KuhDaten` enthält Felder für das Kaufdatum, den Namen, das

Alter und die Menge der täglich abgegebenen Milch. In der bisherigen Programmversion wurden die Neuzugänge auf Obermosers Hof einfach in das Feld `kühe` eingetragen, ohne auf eine Sortierung zu achten. Da jedoch mittlerweile über 600 Kühe in den Ställen und auf den Weiden stehen, dauert die Suche nach einer speziellen Kuh recht lange.

Schreiben Sie eine möglichst effiziente Sortierroutine, welche die vorhandenen Datenbestände nach den Namen der Kühe ordnet. Es kann davon ausgegangen werden, dass in dem Array `kühe` alle Elemente belegt sind. Die Klasse `KuhDaten` verfügt über eine Methode `getName()`, welche den Namen der Kuh als String zurückgibt.

Aufgabe 20. Nach welchem Prinzip arbeitet die Klasse `Hashtable` aus dem Paket `java.util`?

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Alex79] Alexander, C.: *The timeles Way of Building.* Oxford University Press, 1979

[Boe+78] Boehm, B. W.; Brown, J. R.; Kaspar, H.; Lipow, M.; McLeod, G.; Merritt, M.: *Characteristics of Software Quality.* TRW Series of Software Technology, Amsterdam: North-Holland, 1978

[Bor06] Borland Software Corporation: *JBuilder.* `http://www.borland.com/products/downloads/download_jbuilder.html`, 2006

[Bra+99] Brandt, P.; Dettmer, D.; Dietrich, R.-A.; Schön, G.: *Sprachwissenschaft: Ein roter Faden für das Studium.* Böhlau Studienbücher: Grundlagen des Studiums, Köln: Böhlau Verlag, 1999

[Bus+96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: *A System of Patterns.* Chichester et al.: John Wiley & Sons, 1996

[Byou98] Byous, J.: *Java Technology: The early Years.* `http://java.sun.com/features/1998/05/birthday.html`, 1998

[CoYo91] Coad, P.; Yourdon, E.: *Object Oriented Analysis.* New York et al.: Prentice Hall, 1991

[Diet02] Dietzsch, A.: *Systematische Wiederverwendung in der Software-Entwicklung.* Wiesbaden: Deutscher Universitätsverlag (duv), 2002

[DIN05] DIN: *DIN EN ISO 9000:2005-12: Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 9000:2005); Dreisprachige Fassung EN ISO 9000:2005.* Deutsches Institut für Normung e.V., 2005

[Dres96] Dresbach, S.: *Modeling by Construction: Entwurf einer Allgemeinen Entwurfsmethodologie für betriebswirtschaftliche Entscheidungen.* Berichte aus der Betriebswirtschaft, Aachen: Shaker Verlag, 1996

[Ecl06] Eclipse Foundation Inc.: *Eclipse.* `http://www.eclipse.org/downloads/index.php`, 2006

[Ecm99] Ecma International - European association for standardizing information and communication systems: *Standard ECMA-262: ECMAScript Language Specification (ISO/IEC 16262).* 3.

Edition. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, 1999

[Eco02] Eco, U.: *Semiotik: Entwurf einer Theorie der Zeichen.* 9. Auflage, Wilhelm-Fink-Verlag, 2002

[Essw93] Esswein, W.: *Das Rollenmodell der Organisation: Die Brücksichtigung aufbauorganisatorischer Regelungen in Unternehmensmodellen.* In: Wirtschaftsinformatik, 35 (1993) 6, S. 551–561

[FaSc97] Fayad, M.; Schmidt, D. C.: *Object-Oriented Application Frameworks.* In: Communications of the ACM, (1997)

[FeSi01] Ferstl, O. K.; Sinz, E. J.: *Einführung in die Wirtschaftsinformatik.* Band 1, 4. Auflage, München et al.: Oldenbourg, 2001

[Fowl97] Fowler, M.: *Analyse patterns: reusable object models.* Menlo Park et al.: Addison-Wesley, 1997

[Gam+95] Gamma, E.; Helm, R.; R.Johnson; Vlissides, J.: *Design Patterns - Elements of Reusable Object - Oriented Software.* Reading et al.: Addison-Wesley, 1995

[Geh+04] Gehlert, A.; Goldmann, J.; Karow, M.; Reuner, A.; Hebeda, A.; Kalmar, T.: *Konzeptuelle Modellierung, Teil 1: Theoretische Grundlagen.* In: Wirtschaftswissenschaftliches Studium, 33 (2004) 12, S. 742–745

[GeWi05] Gebert, T.; Wiese, R.: *Model Driven Architecture in der Praxis.* In: JavaSPEKTRUM, Sonderheft CeBit 2005 (2005), S. 29–31

[Gosl98] Gosling, J.: *A Brief History of the Green Project.* http://today.java.net/jag/old/green/, 1998

[Hit+05] Hitz, M.; Kappel, G.; Kapsammer, E.; Retschitzegger, W.: *UML@Work: Objektorientierte Modellierung mit UML2.* 3. Auflage, dpunkt.verlag, 2005

[Jet06] JetBrains: *IntelliJ IDEA.* http://www.jetbrains.com/idea/download/, 2006

[Johs94] Johson, R.: *JCreator.* In: SIGS Publications, (1994)

[Mid+03] Middendorf, S.; Singer, R.; Heid, J.: *Java: Programmierhandbuch und Referenz für die Java-2-Plattform.* 3. Auflage, Berlin et al.: dpunkt, 2003

[Oest01] Oestereich, B.: *Objektorientierte Softwareentwicklung: Analyse und Design mit der Unified Modeling Language.* 5. Auflage, München, Wien: Oldenbourg, 2001

[OMG05] OMG (Object Management Group): *Unified Modeling Langugage: Superstructure.* Version 2.0. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf, 2005

[Pree94] Pree, W.: *Design Pattern for Object-Oriented Software Development.* New York et al.: Addison-Wesley, 1994

[Rüdi03] Rüdiger, R.: *Quantenprogrammiersprachen.* In: Informatik Spektrum, 26 (2003) Heft 6, S. 406–409

[Rumb91] Rumbaugh, J.: *Object-oriented Modeling and Design.* New York et al.: Prentice Hall, 1991

[Schü98] Schütte, R.: *Grundsätze ordnungsgemäßer Referenzmodellierung: Konstruktion konfigurations- und anpassungsorientierter Modelle.* Neue betriebswirtschaftliche Forschung 233, Wiesbaden: Gabler, 1998

[Schn81] Schneider, H. J.: *Problemorientierte Programmiersprache.* Leitfäden der angewandten Informatik, Stuttgart: B. G. Teubner, 1981

[ScSc03] Schader, M.; Schmidt-Thieme, L.: *Java: Eine Einführung.* 4. Auflage, Springer, 2003

[Sedg03] Sedgwick, R.: *Algorithmen in Java.* Teil 1-4, 3. Auflage, München: Pearson Education, 2003

[SiWe97] Siyan, K. S.; Weaver, J. L.: *Inside Java.* New Riders Publishing, 1997

[Sun04] Sun Microsystems: *JDK 5.0 Documentation.* `http://java.sun.com/j2se/1.5.0/docs/index.html`, 2004

[Sun05] Sun Microsystems: *Creating a GUI with JFC/Swing.* `http://java.sun.com/docs/books/tutorial/uiswing/learn/index.html`, 2005

[Sun06a] Sun Microsystems: *Sun Java Studio Enterprise.* `http://developers.sun.com/prodtech/javatools/jsenterprise/index.html`, 2006

[Sun06b] Sun Microsystems: *Java 2 Platform Standard Edition 5.0: Compatibility with Previous Releases.* `http://java.sun.com/j2se/1.5.0/compatibility.html`, 2006

[Sun06c] Sun Microsystems: *Version 1.5.0 or 5.0?* `http://java.sun.com/j2se/1.5.0/docs/relnotes/version-5.0.html`, 2006

[WaHa97] Wagner, K. H.; Hackmack, S.: *Grundkurs Sprachwissenschaft.* Universität Bremen, Fachbereich 10: Sprach- und Literaturwissenschaften, Forschungsbericht, `http://www.fb10.uni-bremen.de/linguistik/khwagner/Grundkurs1/grund.pdf`, 1997

[Wall90] Wallmüller, E.: *Software-Qualitätssicherung in der Praxis.* München et al.: Carl Hanser Verlag, 1990

[Xin06] Xinox Software: *JCreator.* `http://www.jcreator.com/download.htm`, 2006

[Zema92] Zemanek, H.; Zeidler, G. (Hrsg.): *Das geistige Umfeld der Informationstechnik.* Edition SEL-Stiftung, Berlin et al.: Springer-Verlag, 1992

Sachverzeichnis

W

Z

Druck und Bindung: Strauss GmbH, Mörlenbach