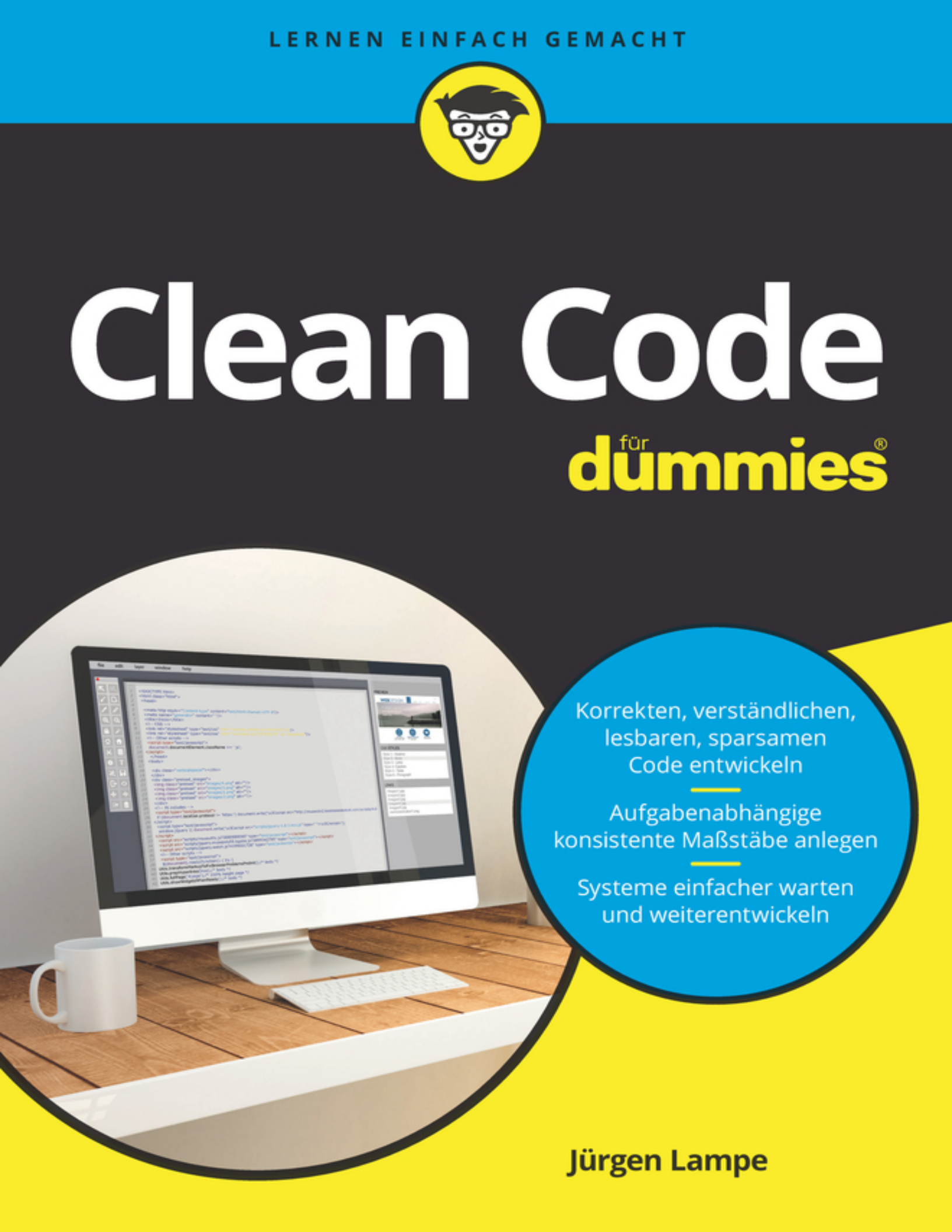
LERNEN EINFACH GEMACHT



Clean Code

für dummies®

Korrekten, verständlichen, lesbaren, sparsamen Code entwickeln

Aufgabenabhängige konsistente Maßstäbe anlegen

Systeme einfacher warten und weiterentwickeln

Jürgen Lampe

Clean Code für Dummies

Schummelseite

DAS CLEAN-CODE-PRINZIP

Software ist in unserem Alltag unverzichtbar geworden. Software ist aber vor allem der Code, der Funktionen definiert und aus dem lauffähige Systeme erzeugt werden. Clean Code ist eine wichtige Methode, um durch lesbaren Code zu langfristig pflegbarer Software zu kommen – wenn der Anspruch umfassend verstanden wird:

- ✔ **Software ist Code, nicht nur – aber ohne Code keine Software.**
- ✔ **Code ist wichtig als ultimative Beschreibung dessen, was die Software macht.**
- ✔ **Jedes Projekt braucht ein angepasstes Code-Qualitätsziel.**
- ✔ **Guter Code ist das Ergebnis sorgfältig abgewogener Entscheidungen.**
- ✔ **Sauberer Code ist gut lesbar und verständlich und damit nachvollziehbar.**
- ✔ **Unsauberer Code verrottet leicht und bedroht langfristig die Wartbarkeit.**

PROGRAMMIEREN IST MEHR ALS HANDWERKSKUNST

Handwerkliche Fähigkeiten sind für eine professionelle Softwareentwicklung unverzichtbar, reichen allein aber noch nicht aus:

- ✔ **Solide handwerkliche Fertigkeit ist Voraussetzung für Professionalität.**
- ✔ **Code ist formalisiertes Wissen.**

Um guten Code schreiben zu können, brauchen Sie ein mentales Modell der Wirklichkeit.

- ✔ **Programmieren erfordert damit teilweise wissenschaftliche Arbeit, nämlich das Aufstellen eines schlüssigen Modells für einen Teil der Anwendungsdomäne.**

GRUNDPRINZIPIEN FÜR SAUBEREN CODE

Wichtige Regeln und Grundsätze für Clean Code:

- ✔ **Gute Namen**
  - Namen sind der einzige Weg, Programmelementen eine erfassbare Bedeutung zuzuordnen. Sie verbinden das mentale Modell mit der Implementierung.
  - Wählen Sie treffende Namen sorgfältig und konsistent!
- ✔ **Formatierung**
  - Macht die Struktur sichtbar und unterstützt so die Lesbarkeit
- ✔ **Kommentare**
  - Fast immer überflüssig und kein Mittel, unverständlichen Code besser zu machen
  - Wenn unumgänglich, dann kurz und präzise. Nicht für das *Wie* (das zeigt der Code), sondern für das *Warum* verwenden.
- ✔ **Methoden**
  - Eine Methode muss auf genau einer Abstraktionsebene angesiedelt sein und genau eine Funktion/Aufgabe erledigen
  - So wenige Parameter wie möglich verwenden
- ✔ **Schnittstellen**
  - Schmal und kohärent definieren
  - Fremdcode durch Adapter sichtbar abgrenzen
- ✔ **Objekte und Datenstrukturen**
  - Objekte und Datenstrukturen sind unterschiedliche Abstraktionen mit jeweils eigenen Anwendungsfeldern.
  - Objekte haben ein Verhalten und verbergen ihre Daten. Das erleichtert das Hinzufügen neuer Objektarten mit neuem Verhalten. Es ist jedoch schwierig, existierendes Verhalten zu ändern.

- Datensätze haben kein Verhalten und zeigen ihre Daten. Das macht es einfach, neue Verarbeitungen zu realisieren, aber aufwendig, die Datenstruktur zu verändern.

✔ **Exceptions**

- Strukturieren Sie Ihre Anwendung in try-catch-Blöcke so, dass am Ende immer wieder ein konsistenter Zustand erreicht wird
- Exceptions so zur Fehlerbehandlung einsetzen, dass sie den Code vereinfachen
- Verwenden Sie nur unchecked Exceptions (gegebenenfalls einpacken)

WICHTIGE REGELN

Einige bewährte Regeln für das Schreiben sauberen Codes

✔ **Wiederholungen vermeiden** – *Don't repeat yourself*

✔ **Nur liefern, was verlangt wird** – *You ain't gonna need it*

✔ **Anliegen sortieren und nach Gruppen getrennt erledigen** – *Separation of concerns*

✔ **Nur eine Verantwortung pro Klasse** – *Single responsibility principle*

✔ **Offen für Erweiterungen und abgeschlossen gegenüber Änderungen** – *Open closed principle*

✔ **Klasse soll durch Unterklassen ersetzbar sein** – *Liskov substitution principle*

✔ **Schmale und kohärente Schnittstellen** – *Interface segregation principle*

✔ **Umkehr der Abhängigkeiten** – *Dependency inversion principle*

✔ **Einfach halten** – *Keep it simple*

CLEAN CODE LEBEN

Mit dem Schreiben von sauberen Programmzeilen ist es nicht getan. Der gesamte Softwareentwicklungsprozess und die Kultur der Softwareentwicklung müssen sich ändern:

- ✔ **Code Reviews zur gegenseitigen Weiterbildung**
- ✔ **Fehler als Quelle neuer Kenntnisse sehen und analysieren**
- ✔ **Ohne gute Testabdeckung kein sauberer Code**
- ✔ **Refactoring ist der zentrale Weg zu gutem und langlebigem Code**

Jürgen Lampe

Clean Code

für dummies®

Fachkorrektur von Maud Schlich

WILEY

WILEY-VCH Verlag GmbH & Co. KGaA

Clean Code für Dummies

Bibliografische Information der Deutschen Nationalbibliothek

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

Print ISBN: 978-3-527-71634-0
ePub ISBN: 978-3-527-82392-5

Coverfoto: © georgejmclittle – stock.adobe.com
Korrektur: Claudia Lötschert, Neuss

Über den Autor

Jürgen Lampe ist promovierter Mathematiker. Nach dem Studium arbeitete er mehrere Jahre als Entwickler für Prozessrechner. Die dabei zwingende Notwendigkeit, mit sehr begrenzten technischen Mitteln für den Kunden einen messbaren ökonomischen Nutzen zu erreichen, hat ihn dauerhaft geprägt. Natürlich blieb ihm auch das zeitraubende und mühselige Suchen und Beheben von Programmierfehlern nicht erspart, sodass nahezu zwangsläufig der Wunsch reifte, Exaktheit und Effektivität der mathematischen Verfahren auf die Softwareentwicklung zu übertragen. Insbesondere die teilweise bereits formalisierten Fachsprachen schienen dafür ein guter Ausgangspunkt zu sein. Über Jahre beschäftigte er sich dann mit der Definition von Fachprogrammiersprachen und deren Implementierung. Diese Forschungen zeigten ihm unter anderem die Wichtigkeit der Anwendungsmodellierung für eine saubere Softwareentwicklung.

Mit diesen Voraussetzungen war der Weg zu Clean Code einfach eine gern genommene logische Fortsetzung. Sie bestärkte ihn auch in der Ansicht, dass große und komplexe Systeme nicht aus dem Stand, sondern nur in einer evolutionären Folge von kleinen Veränderungen – Irrtümer eingeschlossen – gebaut und lebensfähig erhalten werden können.

Inhaltsverzeichnis

Illustrationsverzeichnis

Einleitung

Software ist wichtig und wird immer wichtiger. Vielleicht haben Sie angesichts dieser Tatsache auch, so wie ich, das flaue Gefühl, dass die Qualität der Programme und die Kosten ihrer Entwicklung viel zu oft nicht den Erfordernissen entsprechen.

Es ist offensichtlich, dass die Erstellung von Software in vielerlei Richtungen entscheidend verbessert werden muss. Wenn das nicht gelingt, drohen Ausfälle mit gewaltigem Schadenspotenzial.

Berichte über – oft nach einem Update – nicht oder nur noch eingeschränkt funktionierende Bank- oder Buchungssysteme vermitteln einen kleinen Vorgeschmack darauf, was in Zukunft möglicherweise alles passieren kann.

Je mehr Software im Einsatz ist, desto bedeutender wird deren konsequente Verbesserung und Weiterentwicklung. Bereits heute entsteht ein großer Teil der Kosten nicht durch Neuentwicklungen, sondern durch die notwendigen Anpassungen.

Clean Code ist ein wichtiges Konzept, um Programme so zu schreiben, dass sie möglichst wenig Fehler enthalten und über lange Zeit stabil weiterentwickelt werden können. Jeder engagierte Entwickler sollte deshalb zumindest die wichtigsten Grundsätze kennen.

Über dieses Buch

Dieses Buch soll Ihnen die Gedankenwelt des Clean-Code-Konzepts nahebringen.

Das ist nicht mit der Darstellung einer Reihe von Regeln getan. Infolgedessen kann ich Ihnen auch nicht versprechen, dass Sie nach dem Studium dieses Buchs ein perfekter oder auch nur guter Clean-Code-Programmierer seien werden.

Ich kann Ihnen jedoch versprechen, dass Sie nach dem Lesen eine andere Sicht auf die Softwareentwicklung haben werden. Sie

werden zumindest einiges besser verstehen und verständlicheren Code schreiben können. Und das ganz unabhängig davon, ob Sie das komplette Buch intensiv studiert oder nur Teile davon überflogen haben.

Softwareentwicklung kann gut mit der Expedition zu einem Ziel in schlecht erschlossenem Gelände verglichen werden. Je nachdem wie gut die Gegebenheiten bereits bekannt sind, lässt sich der erforderliche Aufwand mehr oder weniger genau vorhersagen. Je weniger Sie wissen, desto sorgfältiger muss die Unternehmung vorbereitet werden.

Mit diesem Buch will ich Ihnen helfen, sich zielstrebig auf Softwareprojekte einzustellen. Deshalb diskutiere ich nützliche handwerkliche Fertigkeiten und Verfahren ausführlich.

Daneben geht es mir aber auch um die mentale Vorbereitung. Anforderungen können sich stark unterscheiden und Schwierigkeiten beispielsweise durch den Umfang an Funktionen, Leistungsanforderungen oder ein unzureichendes Verständnis der Anwendungsdomäne verursacht werden.

Jeder dieser Punkte erfordert ein anderes Herangehen. Diese Differenzierung wird viel zu häufig unterlassen.

Schließlich noch ein dritter Punkt: Sie lernen (hoffentlich) jeden Tag etwas dazu. Ich zeige Ihnen Wege, wie Sie das Gelernte nutzen können, um darauf aufbauend jeden Tag besser für Ihr Projekt zu arbeiten.

Mit dieser umfassenden Sicht hoffe ich, Ihnen auch dann zu helfen, wenn Sie die vorgestellten Techniken nicht oder nur stark eingeschränkt einsetzen können.

Und nicht zuletzt wünsche ich mir, dass dieses Buch auch dem einen oder anderen, der nicht selbst Code schreibt, aber eng mit IT-Entwicklungen verbunden ist, hilft, Softwareentwickler besser zu verstehen und ihnen den Freiraum zu gewähren, den sie für gute Ergebnisse brauchen.

Konventionen in diesem Buch

Wenn es konkret wird, verwende ich Java. Diese Programmiersprache ist sehr vielen Entwicklern zumindest bekannt und wird seit Jahren in zahlreichen Projekten eingesetzt. Allein aufgrund des vorhandenen Codebestands wird Java noch lange wichtig sein.

Allerdings ist Clean Code nicht an eine bestimmte Sprache gebunden. Fast alle Beispiele lassen sich ohne großen Aufwand in andere Sprachen übertragen. Und wahrscheinlich lernen Sie bei Ihren entsprechenden Versuchen sogar mehr, als wenn ich dieses Buch durch die Wiederholung von Beispielen in verschiedenen Programmiersprachen aufgebläht hätte.

Damit Sie sich gut zurechtfinden, erkläre ich Ihnen kurz die verwendeten Schriftarten und Hervorhebungen.

- In einem Buch über Code finden Sie erwartungsgemäß Quellcode-Listings. Zur Darstellung wird wie allgemein üblich eine Festbreiten-Schriftart verwendet:

```
public class DasIstEinBeispiel extends Nothing {}
```

- Wenn im Text auf konkrete in Java definierte Namen Bezug genommen wird, werden diese ebenfalls in der Schriftart für Code dargestellt, zum Beispiel, wenn es um die Basisklasse `java.lang.Object` geht.
- Neue Begriffe werden in der Regel *kursiv* gesetzt.

  Manchmal wird die Kursivschreibung auch zur Hervorhebung benutzt.

Was Sie nicht lesen müssen

Selbstverständlich bin ich der Meinung, dass es sich lohnt, das gesamte Buch zu lesen. Aber natürlich müssen Sie das nicht.

Die Texte in Kästen sind Hintergrundinformationen, die Sie überspringen können.

Was für Sie wichtig ist und was nicht, hängt stark davon, warum Sie sich für dieses Thema interessieren und was Sie dazu bereits wissen.

Wenn Ihnen das Thema Clean Code noch relativ neu ist und Sie überhaupt erst einmal verstehen wollen, worum es dabei geht, sollten Sie mit den Kapiteln 1, 2 und 4 starten und dann mit einem Ihnen interessant erscheinenden Thema aus Teil III fortfahren.

Wenn Sie Clean Code bereits kennen, können Sie eigentlich sofort mit Teil III beginnen. Trotzdem empfehle ich Ihnen, unabhängig davon auch die Kapitel 6 und 7 zu lesen.

Für alle, die keine oder wenig Programmiererfahrung haben, sind die Kapitel 1, 2, 6, 7 und 21 geeignet, um das Verständnis für den Softwareentwicklungsprozess zu erhöhen.

Der Aufbau des Buchs ist so gewählt, dass es von Anfang bis Ende einer Linie folgt. Trotzdem ist es so geschrieben, dass Sie grundsätzlich jedes Kapitel für sich allein lesen können.

Wenn einzelne Gesichtspunkte in einem anderen Kapitel detailliert behandelt werden, finden Sie entsprechende Verweise – denen Sie folgen können oder nicht.

Wenn Sie etwas schon wissen oder es Ihnen unwichtig erscheint, überspringen Sie es – oder lesen Sie es später. Wie ich Ihnen überhaupt empfehlen möchte, es nicht beim einmaligen Durchlesen zu lassen. Ich bin ziemlich sicher, dass Sie im Lichte Ihrer gewonnenen Erfahrungen bei jedem erneuten Lesen einige Dinge besser oder erst richtig verstehen werden, die Ihnen zuvor entgangen sind.

Törichte Annahmen über die Leser

In diesem Buch dreht sich alles um Softwareentwicklung und Code. Daher sollte mindestens einer der folgenden Punkte auf Sie zutreffen.

- ✔ **Sie können programmieren.**
  Für das vollständige Verständnis ist eine gewisse Vertrautheit mit einer im Idealfall objektorientierten Programmiersprache nötig.
- ✔ **Sie wirken an IT-Projekten als Entwickler mit.**
  Praktisch erleben können Sie den Nutzen von Clean Code vor allem bei größeren und länger laufenden Softwareprojekten.
- ✔ **Sie wollen besseren Code produzieren.**
  Diese Motivation wird Ihnen ganz bestimmt helfen, schneller und gründlicher zu lernen.
- ✔ **Sie verfügen über Ausdauer und Stehvermögen.**
  Um sauberen Code zu schreiben, werden Sie möglicherweise lieb gewonnene Wege verlassen müssen. Eine solche Umstellung Ihrer Arbeitsweise ist nicht von heute auf morgen bewerkstelligt, sondern braucht Zeit und Training.
- ✔ **Sie sind eng mit der Entwicklung von Software verbunden.**
  Wenn Sie als Projektleiter, Product Owner oder in einer ähnlichen Rolle eng mit Entwicklern zusammenarbeiten, gehören Sie nicht unmittelbar zur angepeilten Zielgruppe. Sie können jedoch viel über deren Methoden und Probleme lernen und dadurch das gegenseitige Verständnis erheblich fördern. Abschnitte, die sich unmittelbar auf den Code beziehen, können Sie natürlich überspringen.
- ✔ **Sie sind bereits ein perfekter Clean-Coder.**
  Dann brauchen Sie dieses Buch eigentlich nicht. Lesen Sie es bitte trotzdem kritisch und machen Sie mich auf Fehler und Vergessenes aufmerksam. Oder – noch besser – verfassen Sie gleich »Cleaner Clean Code für Dummies«.

Wie dieses Buch aufgebaut ist

Dieses Buch besteht aus fünf Teilen.

Teil I: Das Clean-Code-Prinzip

In diesem Teil erfahren Sie die Grundgedanken des Clean-Code-Prinzips. Warum ist Code als solcher so wichtig? Was ist überhaupt guter und sauberer Code? Daneben geht es um Fragen des Abwägens zwischen konkurrierenden Zielstellungen.

Teil II: An Herausforderungen wachsen

In Teil II lernen Sie wesentliche Herausforderungen kennen, die Sie als Softwareentwickler bewältigen müssen. Der Schwerpunkt liegt dabei auf jenen Problemen, die vor dem Beginn des Codeschreibens gelöst sein sollten.

Diese Differenzierung der Problembereiche hilft Ihnen, keine unrealistischen Erwartungen bezüglich des Effekts von sauberer Programmierung zu hegen.

Teil III: Sauberen Code schreiben

Teil III zeigt Ihnen die wichtigsten Regeln zum Schreiben von sauberem Code. Dabei erfahren Sie jeweils auch, unter welchen Bedingungen eine Regel verwendet werden sollte und wann es besser ist, sie nicht anzuwenden.

Sie lernen zudem, wie Sie mögliche Fehler klassifizieren und dementsprechend sauber behandeln können.

Schließlich geht es auch noch um neuere Sprachbestandteile, und Sie sehen, wie diese ohne Beeinträchtigung der Codequalität sinnvoll eingegliedert werden können.

Teil IV: Wege zum Ziel

In Teil IV steht die praktische Umsetzung des Clean-Code-Konzepts im Mittelpunkt. Sie erhalten Tipps, wie es Ihnen

gelingen kann, das Schreiben von sauberem Code in den Softwareentwicklungsprozess zu integrieren.

Teil V: Der Top-Ten-Teil

Im letzten Teil des Buchs habe ich noch ein paar nützliche Hinweise für Sie versammelt:

- Ratschläge, die Sie auf dem Weg zum Clean-Code-Entwickler vor Fehlern bewahren können und Ihnen hoffentlich helfen, das Ziel zu erreichen.
- Eine Liste von Webseiten, die Ihnen detailliertere Informationen zu den aufgeworfenen Themen und praktischen Fragen geben.

Symbole, die in diesem Buch verwendet werden

Die Glühbirne markiert einen Tipp. Falls Sie nur Wissen erwerben wollen, können Sie solche Hinweise überspringen. Wenn es Ihnen aber auch um die praktische Umsetzung geht, sollten Sie diese Tipps nicht nur einmal sorgfältig lesen.

An dieser Stelle finden Sie Hinweise, die Ihnen helfen sollen, bei der praktischen Anwendung des Dargestellten Fehler zu vermeiden.

Daher können Sie diese Texte überspringen, wenn Sie nur das Thema verstehen wollen.

Hier schränke ich die Bedeutung einzelner Wörter durch eine genauere Abgrenzung ein, um im folgenden Text Missverständnisse zu vermeiden. Solche Definitionen sollten Sie besser nicht ignorieren.

Mit dem Wegweiser kennzeichne ich zusätzliche Erläuterungen beispielweise durch Analogien aus anderen Bereichen oder kurze Beispiele. Für das Verständnis sind diese Ausführungen zwar nicht zwingend erforderlich, aber hoffentlich hilfreich.

Wenn Sie wenig Zeit haben oder das Kapitel bereits zum zweiten Mal lesen, können Sie diese Texte überspringen.

Umfangreichere Beispiele, die unabhängig vom umgebenden Text verständlich sind, werden auf diese Weise gekennzeichnet.

Sie können diese Teile beim ersten Lesen gern überspringen und eventuell später genauer studieren.

Wie es weitergeht

Ganz gleich, aus welchen Gründen Sie sich für Clean Code interessieren. Fangen Sie an zu lesen – das muss nicht auf Seite 1 sein – und versuchen Sie Ihre Erkenntnisse sinnvoll anzuwenden.

Bevor Sie nun loslegen, möchte Ihnen aber noch einen Rat geben. Den können Sie jetzt gern überspringen, aber lesen Sie ihn bitte auf jeden Fall noch, bevor Sie eventuell aufgeben:

Es gibt keinen Express auf dem Weg zum guten Clean-Code-Entwickler. Viel eher ist ein Fußmarsch vonnöten. Wie lang und wie beschwerlich dieser Marsch ist, hängt vom Ausgangspunkt und den spezifischen Schwierigkeiten ab, die Sie überwinden müssen.

Und sollte Ihnen unterwegs einmal alles zu lange dauern und der Mut sinken, dann kann Ihnen diese jüdische Weisheit vielleicht neue Energie und Zuversicht geben: »Wer langsam und besonnen geht, doch oft zuerst am Ziele steht.«

In diesem Sinne – auf denn!

Teil I

Das Clean-Code-Prinzip

IN DIESEM TEIL …

- ✔ Erfahren Sie, warum Code wichtig ist
- ✔ Erfahren Sie, dass Clean Code mehr ist als eine Sammlung von Regeln
- ✔ Lernen Sie wichtige Voraussetzungen für das Schreiben sauberen Codes kennen
- ✔ Werden Sie vor leicht zu übersehenden Fallstricken gewarnt

Kapitel 1

Software ist Code

IN DIESEM KAPITEL

Was wird von Software erwartet?

Die Softwarekrise und ihre Ursachen

Code ist wichtig

In diesem Kapitel werden, ausgehend von der Bedeutung, die Software im Alltag spielt, Probleme erläutert, die bei der Softwareentwicklung immer wieder auftreten. Danach wird die zentrale Stellung des Programmcodes im Entwicklungsprozess begründet.

Erwartungen an Software

Software ist aus dem täglichen Leben nicht mehr wegzudenken. Sie findet sich nicht nur in Computern und Smartphones, sondern ersetzt zunehmend sogar einen Teil der mechanischen Bauteile, beispielsweise in Türschlössern und Nähmaschinen. Vor allem die aufwendig herzustellenden Steuerungskonstruktionen weichen elektronisch angesteuerten einfachen Schaltelementen.

Den meisten Benutzern ist es dabei völlig gleichgültig, was Software ist. Sie erwarten, dass das Gerät oder das Programm funktioniert.

Allerdings ist niemand total überrascht, wenn das einmal nicht der Fall ist. Überlegen Sie kurz, wie oft Sie selbst schon schulterzuckend einen Softwarefehler hinnehmen mussten. Meist ist das zum Glück nur ärgerlich.

Es scheint unmöglich zu sein, weitgehend fehlerfreie Software zu produzieren.

Der Vergleich mit anderen komplexen technischen Apparaten wirft allerdings die Frage auf, wieso es bisher nicht gelungen ist, Verbesserungen in vergleichbarem Maße zu erreichen wie etwa bei der Sicherheit von Flugzeugen. Darauf gibt es keine einfache Antwort.

Die beschriebene Problematik ist fast so alt wie der Computer. Bereits 1972 erwuchs aus der unbefriedigenden Situation der Begriff *Softwarekrise*, und 1993 wurde auf einer Tagung gefragt: »Kann eine Krise 20 Jahre dauern?«

Trotz unleugbarer Fortschritte muss man eingestehen, dass die Krisensymptome heute noch genauso sichtbar sind wie damals. Geändert hat sich die Wahrnehmung. Das, was zunächst als Krise erschien, wird mittlerweile als Dauerzustand hingenommen.

Das heißt allerdings nicht, dass Sie als Entwickler diesen Zustand akzeptieren müssen. Ich hoffe, Sie lesen dieses Buch gerade deshalb, um daran etwas zu ändern. Denn Veränderung tut not, und zwar immer dringender. Je mehr Software in unser Leben eingreift, desto schwerwiegender werden ganz zwangsläufig die Auswirkungen von Fehlern sein.

Softwarekrise

In den Anfangsjahren der Computernutzung entfiel der Löwenanteil der Kosten auf die Hardware. Programmiert wurden vor allem gut aufbereitete und verstandene mathematische Algorithmen. Bezeichnenderweise war die erste Programmiersprache im heutigen Sinn das 1957 erschienene FORTRAN, kurz für FORmula TRANslator. Über Methoden zur Softwareentwicklung dachte niemand intensiver nach.

Der technische Fortschritt führte jedoch schnell zu sinkenden Hardwarekosten und in der Folge zu einer stärkeren Verbreitung von Computern. Die Aufgaben wurden anspruchsvoller und komplizierter. Dadurch stiegen die Kosten der Programmierung und begannen Mitte der 1960er-Jahre diejenigen der Hardware zu übersteigen.

Gleichzeitig kam es zu ersten spektakulären Folgen von Programmierfehlern wie 1962 dem Verlust der Venussonde Mariner-1. Es wurde unübersehbar,

dass mangelhafte Software gewaltige Kosten verursachen kann. So wie bisher konnte es nicht weitergehen.

Die Entwicklung war in eine Krise, die Softwarekrise, geraten. Auf der Suche nach einem Ausweg entstand das *Softwareengineering*. Dieser Begriff tauchte zum ersten Mal 1968 im Namen einer NATO-Tagung auf.

Trotz entscheidender Verbesserungen kann die Softwarekrise bis heute nicht als tatsächlich überwunden gesehen werden.

Probleme haben Ursachen

Wenn man auf Probleme stößt, ist es erfahrungsgemäß nützlich, deren Ursachen aufzuklären. In den meisten Fällen gibt es dafür nicht nur einen Grund. Für die Softwarekrise lassen sich zwei wichtige Ursachengruppen finden.

Allgemeine Ursachen

Für die oft mangelhafte Qualität von Software gibt es ebenso wie für das ganze oder teilweise Scheitern von Entwicklungsprojekten verschiedene Gründe, unter anderem:

- ✔ Ungenügende Vorbereitung, insbesondere unvollständige Aufgabenbeschreibung und Abgrenzung
- ✔ Fehler in der Projektorganisation und der Projektleitung
- ✔ Unverständnis oder Unkenntnis im Management bezüglich der Besonderheiten der Softwareentwicklung
- ✔ Unzureichende Mittel (Zeit und Geld)
- ✔ Vernachlässigung der Qualitätssicherung (Test)
- ✔ Schwierigkeiten bei der Beherrschung der Komplexität der entstehenden Software

Diese Liste ist bei Weitem nicht vollständig. Sie soll Ihnen nur zeigen, dass es zwar viele aus Sicht der Entwicklung externe Ursachen gibt, aber zumindest die letzten beiden Punkte fallen in die Verantwortung der Entwickler.

Tests sind unverzichtbarer Teil jeder professionellen Softwareentwicklung. Sie können damit zwar nicht die vollständige Korrektheit Ihres Programms beweisen – ein Test ist immer nur eine Stichprobe –, aber Sie dokumentieren, was nachweisbar funktioniert, und verringern die Wahrscheinlichkeit, dass Fehler übersehen werden.

Dieser Hinweis steht seiner Wichtigkeit wegen bereits hier und damit ganz am Anfang. Nehmen Sie ihn bitte ernst und vergessen Sie nicht: Auch Tests sind Code, und für diesen Testcode sind die gleichen Qualitätsanforderungen zu erfüllen wie für den zu testenden Code.

Hardwareentwicklung

Einen sehr großen Einfluss auf die Softwarekrise hat die Entwicklung der Hardware ausgeübt. Durch die rapide Leistungssteigerung der Prozessoren und die parallel dazu gewachsenen verfügbaren Speichervolumina entstand ein stetiger Druck, noch größere Aufgaben in Angriff zu nehmen.

Die Softwarehersteller hatten nie Zeit, um in Ruhe Luft zu holen und ihre Produkte zu konsolidieren. Stets stand schon eine neue, noch leistungsfähigere Hardwaregeneration vor der Tür, die Anpassungen und Weiterentwicklungen verlangte. Die Programme wurden immer umfangreicher und komplexer, ohne dass mehr Zeit für die Entwicklung verfügbar war.

Alle im Entwicklungsprozess erreichten Verbesserungen wurden sofort dafür eingesetzt, in diesem Rennen Schritt zu halten. Die Softwarequalität geriet dabei oft ins Hintertreffen. Schon in den 1980er-Jahren entstand dazu das wahlweise Niklaus Wirth oder Martin Reiser zugeschriebene Bonmot: Software wird schneller langsamer, als die Hardware schneller wird.

Mittlerweile steigt die reine Prozessorgeschwindigkeit kaum noch, dafür werden die Strukturen mit Mehrkern- und Multiprozessoren komplizierter. Diese neuen Möglichkeiten können nur durch nebenläufige Programme voll ausgenutzt werden, was die

Software noch einmal um Größenordnungen komplexer macht. Die Lage hat sich also kaum verändert.

Nichts ohne Code

Bisher habe ich ständig von Software gesprochen, ohne diesen Begriff genauer zu erläutern. Das hat seinen Grund. Es ist schwer, eine Definition zu finden, die sowohl ausreichend konkret als auch umfassend genug ist. Hier werde ich mich deshalb auf einen wesentlichen Aspekt beschränken.

Software ist in gewisser Weise ein indirektes Produkt. Sie entsteht aus formalen Beschreibungen, die im Allgemeinen *Programmcode* oder kurz *Code* genannt werden. Ihre Wirksamkeit oder Nutzbarkeit ist an bestimmte technische Geräte – die Hardware – und weitere Software wie Betriebssysteme und Compiler gebunden.

Wenn Sie von dieser Umgebung, die gemeinhin vorgegeben ist, absehen, lassen sich alle wichtigen Eigenschaften einer Software auf ihren Code zurückführen. Um eine Software zu schreiben oder zu erweitern, müssen Sie Code schreiben. Das Gleiche gilt, wenn Sie einen Fehler beheben wollen.

Das Produkt jedes Softwareentwicklungsprozesses ist Code. Es gibt gemeinhin keine genauere Beschreibung dessen, was eine Software leistet, als den Programmcode.

Die Schlussfolgerung lautet daher: Software besteht im Wesentlichen aus Code. Und daraus folgt unmittelbar: Code ist wichtig. Bessere Software lässt sich nur durch besseren Code produzieren.

Es wird Sie daher nicht überraschen, dass in vielen Unternehmen schon heute ein erheblicher Teil der Vermögenswerte aus Code besteht. Dessen wirtschaftliche Bedeutung zeigt sich nicht zuletzt in den mit viel Einsatz geführten Auseinandersetzungen um Rechtsverstöße und Lizenzbedingungen.

Da immer mehr Funktionen durch Software realisiert werden, ist absehbar, dass die Wichtigkeit von Code für unser Leben weiter zunehmen wird.

Code existiert oft wesentlich länger als die Hardware, für die er ursprünglich entwickelt wurde. Beispielsweise verwenden manche Forschungseinrichtungen noch heute mathematische Fortran-Bibliotheken, die bereits mehrere Jahrzehnte alt sind. In vielen großen Programmsystemen finden sich Codeteile, die aus weit zurückliegenden Versionen überlebt haben.

Voraussetzung eines solch langen Lebens ist lediglich, dass neuere Versionen der Compiler abwärtskompatibel sind, das heißt, dass sie den alten Quelltext weiterhin akzeptieren und fehlerfrei verarbeiten können. Wenn ein genügend großer Codebestand vorhanden ist, wird das von den Compilerproduzenten gewöhnlich gewährleistet, um die Kunden schneller auf neue Versionen locken zu können.

Ganz nebenbei: Die Angst, dass die für eine solche langfristige Sicherung der Codebasis notwenige kritische Masse an Programmcode nicht zustande kommt, ist eines der größten Hindernisse bei der Verbreitung neuer Programmiersprachen.

Das Wichtigste in Kürze

- ✔ Software ist zu einem unverzichtbaren Bestandteil des Alltagslebens geworden.
- ✔ Unzureichende Softwarequalität und fehlgeschlagene Entwicklungsprojekte sind seit Langem ein Dauerthema.
- ✔ Trotz aller Bemühungen hat die Softwareentwicklung mit der rasanten Verbesserung der Hardware nur mühsam mithalten können.

- ✔ Die schnell fortschreitende Digitalisierung verlangt erhebliche Verbesserungen der Softwarequalität.
- ✔ Software ist vor allem der Code, der die von Ihnen gewünschte Funktion definiert und aus dem lauffähige Systeme erzeugt werden.
- ✔ Code ist wichtig!

Kapitel 2

Dimensionen von Codequalität

IN DIESEM KAPITEL

Abgrenzung des Begriffs Qualität

Die unterschiedlichen Erwartungen bezüglich der Codequalität

Warum Sie ein Qualitätsziel brauchen

In diesem Kapitel werden die grundlegenden Eigenschaften des Qualitätsbegriffs dargestellt. Sie lernen einige der möglichen Ausprägungen von Qualitätserwartungen an Code kennen und erfahren, dass jedes Projekt sein individuell angepasstes Codequalitätsziel braucht.

Was bedeutet Qualität?

Bevor ich genauer auf die Codequalität eingehe, erscheint es mir angebracht, den Qualitätsbegriff, den ich in diesem Buch verwende, klarzustellen. Das Schlüsselwort *Qualität* wird immer öfter unzulässig weit interpretiert. Das führt dazu, dass seine Bedeutung verschwimmt und in die Beliebigkeit abzurutschen droht.

Eigenschaften des Qualitätsbegriffs

Klare Gedanken erfordern eine klare Sprache. Um der Gefahr von Missverständnissen vorzubeugen, beginne ich mit folgender Definition:

Qualität ist zunächst lediglich die Summe aller Eigenschaften und Merkmale eines Objekts – ohne jede Bewertung. Dem allgemeinen Sprachgebrauch folgend wird hier vorrangig die Güte der Eigenschaften und Merkmale als Qualität bezeichnet.

Wenn man den auf die Güte bezogenen Qualitätsbegriff betrachtet, ergeben sich die beiden nachfolgenden wichtigen Eigenschaften:

- **Qualität ist prinzipiell nicht absolut messbar.**

  Gemeint ist damit, Qualität lässt sich nicht als Zahlenwert ausdrücken, sonst wäre sie *Quantität*. Trotzdem gibt es Qualitätsindikatoren, die messbar sind.

- **Die Bewertung von Qualität ist immer auf eine Erwartung bezogen.**

  Ob etwas hohe Qualität hat, hängt vom Betrachter ab. Sie können beispielsweise ein Stück Code positiv bewerten, weil es hervorragend zu lesen ist, während jemand anderes es negativ einschätzt, weil es die erwartete Funktion nicht erfüllt.

Sie können sich die genannten prinzipiellen Eigenschaften gut vergegenwärtigen, wenn Sie das Wort »Qualität« durch »Schönheit« ersetzen. Letztere ist ebenfalls eine (spezielle) Qualität im Sinne der obigen Definition.

Es ist klar, dass man Schönheit nicht absolut messen kann, und der Volksmund sagt nicht zu Unrecht mit Bezug auf die Bewertung: Die Schönheit liegt im Auge des Betrachters.

Obwohl Qualität nicht messbar ist, lassen sich Objekte hinsichtlich ihrer Qualitätsbewertung vergleichen. Sie dürfen dabei nur nicht vergessen, dass dieser Vergleich stets auf eine bestimmte Erwartung bezogen ist.

Daneben müssen die verglichenen Objekte hinsichtlich der Erwartung ausreichend ähnlich sein, wenn Sie zu sinnvollen Ergebnissen kommen wollen. Äpfel mag man mit Birnen vergleichen, wenn es um den Geschmack oder andere Qualitäten geht, die sich auf das Objekt »Frucht« beziehen – ansonsten besser nicht.

Qualitätsindikatoren

Getreu dem Motto »Was man nicht messen kann, gibt es nicht« wird immer wieder versucht, Qualität zu messen. Leider scheinen viele Manager dann auch zu glauben, dass ihnen das wirklich gelingt – eine riskante Selbsttäuschung.

Alle sogenannten *Qualitätsmaße* sind bestenfalls *Qualitätsindikatoren*. Es wird versucht, messbare Eigenschaften zu finden, die in hohem Maße mit den jeweiligen Qualitätserwartungen korreliert sind.

Das Problem mit diesen Korrelationen besteht aber darin, dass sie keinem zwingenden Zusammenhang entspringen. Es ist stets möglich, den Indikator zu manipulieren, ohne dass sich an der betrachteten Qualität etwas ändert.

Betrachten Sie Qualitätsindikatoren mit Vorsicht. Zurückhaltend angewandt, können Sie Ihnen wichtige Erkenntnisse liefern.

Vermeiden Sie jedoch unbedingt eine Verabsolutierung. Wenn Sie einen Indikator zum Maßstab erklären, wird er über kurz oder lang wertlos, und Sie berauben sich dadurch einer nützlichen Informationsquelle.

Qualitätsbewertung mittels Indikatoren

Ein eindrucksvolles Beispiel für das unauflösbare Dilemma der Qualitätsbewertung durch Indikatoren zeigt sich bei der Relevanzermittlung in Suchmaschinen. Relevanz ist ebenfalls eine nicht messbare Qualität, die anhand verschiedenster Indikatoren ermittelt wird.

Die sogenannte Suchmaschinen-Optimierung (SEO) ist damit nichts anderes als ein gezielter Versuch zur Indikatormanipulation. Um die Relevanzermittlung trotzdem einigermaßen sinnvoll aufrechterhalten zu können, sind die Betreiber der Suchmaschinen gezwungen, ihre Indikatoren ständig neu zu justieren.

Ganz Ähnliches gilt für die Bonitätsermittlung durch Auskunfteien. Auch dabei muss aus Indikatoren auf eine Qualität geschlossen werden.

In beiden Fällen würde eine vollständige Offenlegung der Bewertungsverfahren diese letztlich wertlos machen. Dass das keine rein theoretische Konsequenz ist, haben einige aufsehenerregende Vorfälle im Bereich der wissenschaftlichen Veröffentlichungen gezeigt.

Durch Bildung von »Zitier-Kartellen« ist es wissenschaftlich unbedeutenden Personen gelungen, allein durch intensives gegenseitiges Zitieren mit ihren Artikeln an die Spitze der als relevant eingestuften Veröffentlichungen zu gelangen.

Die Dimensionen von Codequalität

Was ich hier *Dimensionen* von Codequalität nenne, sind die Erwartungen der Qualitätsbewertung aus dem vorangegangenen Abschnitt. Der Ausdruck erscheint mir deshalb angebracht, weil es sich um ganz unterschiedliche Erwartungen handelt, die im konkreten Fall auch unterschiedlich ausgeprägt sein können, sodass die konkrete Qualitätserwartung einem Punkt im mehrdimensionalen Raum entspricht.

Die Unabhängigkeit der Erwartungen voneinander bedeutet nicht, dass die Realisierungen ebenfalls unabhängig sind – ganz im Gegenteil. Kollisionen zwischen stark ausgeprägten Erwartungen sind eher unvermeidlich.

Das heißt letztlich, nicht alles, was Sie gern gleichzeitig hätten, ist auch möglich. Oder abstrakter gesagt: Nicht jeder Punkt im Qualitätsraum ist realisierbar.

Korrektheit des Codes

Auf diese Dimension wird kein sinnvoller Code verzichten. Trotzdem können die Erwartungen recht unterschiedlich sein, wie ich Ihnen an ein paar Beispielen zeigen werde:

- ✔ **Prototyp oder Demonstrationsprogramm**

  In der Regel steht die schnelle und kostengünstige Herstellung im Fokus. Abstriche an der Funktion und einzelne Fehler werden toleriert.

- ✔ **Programm für beschränkten Einsatz**

  Das sind oft Programme für Datenmigration oder zeitlich beschränkte Funktionalitäten. Es ist manchmal kostengünstiger, die Folgen von einzelnen Fehlern nachträglich von Hand zu beseitigen, als ein Programm zu korrigieren.

- ✔ **Programm für langfristigen Einsatz**

  Da die Anwendungsfälle begrenzt und bekannt sind, können Fehler bei nicht oder nur selten genutzten Funktionen akzeptabel sein.

- ✔ **Softwareprodukt**

  Es wird ein weitgehend fehlerfreies Verhalten über alle Anwendungsfunktionen erwartet.

- ✔ **Sicherheitsrelevantes Programm**

  Sehr hohe Anforderungen an die Korrektheit werden mit großem Aufwand realisiert.

Lesbarkeit und Wartbarkeit

Diese beiden Dimensionen sind nicht identisch, aber stark miteinander verknüpft. Auch wenn Sie das auf den ersten Blick vermuten könnten, gute Wartbarkeit setzt eine gute Lesbarkeit des Codes nicht zwingend voraus. Gute Wartbarkeit kann durch aktuelle und umfassende Dokumentation ebenso erreicht werden.

Für Assemblercode lässt sie sich beispielsweise anders praktisch gar nicht erreichen. Bei höheren Programmiersprachen ist allerdings der Zwang, die Dokumentation auf dem aktuellen Stand zu halten, nicht ganz so stark. Deshalb spielt die Lesbarkeit eine wichtigere Rolle.

Da sich das Verhältnis von Lesbarkeit zu Wartbarkeit wie ein roter Faden durch dieses Buch zieht, werde ich es hier nicht weiter ausführen. In den folgenden Kapiteln werden Sie ausführliche Erläuterungen finden.

Leistungseigenschaften

Damit ist in erster Linie die Laufzeiteffizienz des Codes gemeint. In vielen Anwendungsbereichen ist es inzwischen zwar kostengünstiger, mehr oder schnellere Hardware bereitzustellen, als den Code zu optimieren, aber es gibt auch einen großen Sektor, in dem Antwortzeiten und damit die Verarbeitungsgeschwindigkeit eine Rolle spielen. In dem Ausmaß, in dem computergesteuerte Geräte den Alltag durchdringen, wird der Bedarf an solcher echtzeitfähigen Software zunehmen.

Leistung ist eine Dimension, die bei der Umsetzung häufig in Konkurrenz zu anderen Erwartungen wie Lesbarkeit oder Modularisierung tritt.

Weitere Dimensionen

Je nach Projektziel gibt es zusätzliche Dimensionen. Als Anregung möchte ich Ihnen kurz noch einige weitere präsentieren:

- **Speicherbedarf**

  Das kann sich sowohl auf den Umfang des Codes als auch auf den während der Laufzeit für die Daten erforderlichen Platz beziehen. Da Speicher nicht teuer ist, wird dieses Kriterium vor allem dann wichtig, wenn die Software in vielen Tausenden (echten oder virtuellen) Instanzen läuft. Dazu gehören ebenfalls die vielen »smarten« Geräte, die meist nur über eine beschränkte Ausstattung verfügen.

- **Stromverbrauch**

  Diese Dimension weist Beziehungen zur Leistung auf, ist aber ähnlich wie der Speicherbedarf vor allem bei einer großen Anzahl von Instanzen und langen Laufzeiten wichtig.

  Außerdem ist niedriger Stromverbrauch bei mobilen Geräten von entscheidender Bedeutung, weil dadurch die Laufzeit, die ohne Nachladen möglich ist, verlängert werden kann.

- **Komplexität des Codes**

  Bei dieser Dimension wird zur Abwechslung eine möglichst niedrige Ausprägung angestrebt.

- **Modularität**

  Auch diese Dimension hat Beziehungen zu anderen Dimensionen. Da Modularität kein Allheilmittel ist und möglicherweise andere Qualitäten negativ beeinflussen kann, sollten Sie sie aber als eigene Dimension betrachten.

Das Qualitätsziel festlegen

Angesichts der vielen Sichtweisen auf den Begriff der Qualität rate ich Ihnen, sich zu Beginn eines Entwicklungsprojekts rechtzeitig Gedanken über Ihr spezifisches Qualitätsziel zu machen. Wenn Sie das unterlassen, besteht die Gefahr, dass Sie im Laufe der Entwicklung die für Ihr Projekt relevanten Ziele aus den Augen verlieren oder gar auf abweichende, das heißt weniger wichtige Zielstellungen, hinarbeiten.

Bei der Definition müssen Sie als Erstes die lang- und kurzfristigen Interessen des Auftraggebers erkunden. Zusammen mit der Analyse der Aufgabenstellung erhalten Sie eine Basis, auf der Sie die zu erfüllenden Erwartungen, deren Ausprägungsgrad und die Wichtigkeit für den Projekterfolg ermitteln können.

Da in diesen Prozess viele subjektive Einschätzungen einfließen, ist es wichtig, dass sich alle Teammitglieder daran beteiligen. Am Ende muss das Qualitätsziel von allen akzeptiert und mitgetragen werden.

Nehmen Sie die gemeinsame Bestimmung eines Qualitätsziels ernst und planen Sie genügend Zeit für die notwendigen Diskussionen ein. Nur ein von allen und von Anfang an akzeptiertes Ziel kann gemeinsam erreicht werden.

Sorgen Sie dafür, dass das vorab definiert Qualitätsziel im Laufe des Projekts nicht dem Vergessen anheimfällt. Andererseits dürfen Sie es natürlich nicht als Dogma sehen. Ebenso wie alle anderen Entwicklungsentscheidungen muss das Ziel von Zeit zu Zeit und den gewonnenen Erfahrungen entsprechend angepasst werden.

Beispiel: Der euklidische Algorithmus

Abschließend werde ich Ihnen die wichtigsten Überlegungen an einem (sehr) kleinen Beispiel illustrieren. Es handelt sich um Euklids Algorithmus zur Berechnung des größten gemeinsamen Teilers (ggT) zweier Zahlen. Listing 2.1 zeigt zwei mögliche Implementierungen.

```
public static int ggT01(int m, int n) {
    int r;
    while (n != 0) {
        r= m % n;
        m= n;
        n= r;
    }
    return m;
}
//-----------------------------------
public static int ggT02(int m, int n) {
    if (n == 0) {
        return m;
```

```
    } else {
        return ggT02(n, m % n);
    }
}
```

Listing 2.1: Zwei Implementierungen des euklidischen Algorithmus

Im Unterschied zu fast allen anderen Beispielen in diesem Buch sehen Sie hier Abkürzungen und Namen, die nur aus einem Buchstaben bestehen. Bei der Implementierung von allbekannten mathematischen Formeln halte ich das für vertretbar, weil es der Praxis im Anwendungsgebiet, also der Mathematik, entspricht.

Euklidischer Algorithmus

Der *euklidische Algorithmus* ist ein bereits um 300 vor Christus angegebenes Verfahren, um zu zwei Zahlen *m* und *n* (*m* > *n*) den größten gemeinsamen Teiler, also die größte Zahl *g* zu bestimmen, für die bei den Divisionen *m* / *g* und *n* / *g* kein Rest bleibt. In einer Kette von Divisionen, beginnend mit *m* / *n*, wird der Divisor (*n*) im jeweils nächsten Schritt durch den Rest (*m* **mod** *n*) geteilt, bis kein Rest mehr bleibt. Der letzte Divisor ist das gesuchte Ergebnis.

Beispiel: 2613 / 2010 = 1 R 603, 2010 / 603 = 3 R 201, 603 / 201 = 3 R 0, Ergebnis 201.

Überlegen Sie selbst, ob der Code verständlicher wird, wenn Sie `m`, `n` und `r` durch `zahl1`, `zahl2` und `rest` ersetzen sowie den Methodennamen ausschreiben. Wahrscheinlich wird Ihre Entscheidung davon abhängen, wie vertraut Ihnen der mathematische Formalismus ist. Insofern zeigt Ihnen dieses Beispiel, gewissermaßen nebenbei, auch noch die Relativität von Qualitätserwartungen.

Nun werden die beiden Methoden `ggT01` und `ggT02` in Bezug auf einige der vorgestellten Qualitätsdimensionen verglichen. Da es hier nur um die prinzipielle Darstellung geht, berücksichtige ich nicht, dass es in der Realität möglicherweise noch Optimierungen durch den Compiler oder das Laufzeitsystem geben könnte.

- ✔ **Korrektheit**: Beide Implementationen sind korrekt.

- ✔ **Lesbarkeit**: Hier ist die Methode `ggT02` leicht im Vorteil, weil sie kürzer ist und sich stärker an der in der Mathematik üblichen rekursiven Beschreibung orientiert.
- ✔ **Leistung**: Da die Methode `ggT01` für die Berechnung nur drei lokale Variablen und keine zusätzlichen Methodenaufrufe benötigt, ist von ihr die bessere Leistung zu erwarten.
- ✔ **Speicherbedarf**: Auch bezüglich dieser Dimension bietet die Methode `ggT01` Vorteile, weil `ggT02` bei jedem erneuten Aufruf zusätzlichen Speicherplatz im Stack belegt.

Das Wichtigste in Kürze

- ✔ Qualität ist grundsätzlich nicht messbar.
- ✔ Qualitätsindikatoren sind eine wichtige Informationsquelle. Sie verlieren jedoch ihren Wert, wenn sie als »Qualitätsmaß« benutzt werden.
- ✔ Die Bewertung von Qualität ist stets auf bestimmte Erwartungen bezogen.
- ✔ Erwartungen an die Qualität von Code lassen sich in verschiedene Dimensionen wie Lesbarkeit, Leistung und Ressourcenbedarf zerlegen.
- ✔ Für jedes Projekt muss ein angepasstes Qualitätsziel vereinbart werden, das für die einzelnen Dimensionen den jeweils angestrebten Grad und das zugemessene Gewicht widerspiegelt.

Kapitel 3

Alles unter einen Hut – gute Kompromisse finden

IN DIESEM KAPITEL

Die hohe Kunst des Kompromisses – Warum Abwägen so wichtig ist

Alternativen finden und bewerten

Ausgewogene Entscheidungen treffen

In diesem Kapitel erfahren Sie, warum Abwägen eine unverzichtbare Fähigkeit für Entwickler ist. Sie lernen dazu einige wichtige Grundsätze kennen, die es Ihnen erlauben, besser mit Situationen umzugehen, in denen Sie vor mehr oder weniger schwierigen Entscheidungen stehen.

Warum gute Entscheidungen wichtig sind

Wenn Ihnen ein Produkt besonders gut gefällt, liegt das in der Regel nicht an einer ganz besonders herausragenden positiven Eigenschaft, sondern an der Ausgewogenheit der Vor- und Nachteile. Wobei Sie zu Recht erwarten, dass die Vorteile insgesamt überwiegen.

Das ist bei Code nicht anders. Guter Code zeichnet sich nicht dadurch aus, dass bestimmte Regeln mit letzter Konsequenz eingehalten werden. Vielmehr müssen die erwarteten guten Eigenschaften in einem angemessenen Verhältnis zu den

unvermeidlichen Schattenseiten stehen. Das zu erreichen, ist nicht leicht.

Weil ich aber fest davon überzeugt bin, dass die maßvolle Anwendung von Regeln mindestens ebenso wichtig ist, wie es die Regeln selbst sind, behandle ich dieses Thema hier ausführlicher.

Es kommt drauf an …

Entweder haben Sie es selbst schon erlebt oder können es leicht nachprüfen: Wenn Sie im Internet nach einer Antwort auf Fragen der Art »Ist es besser, ein `switch` oder eine Kette von `if` zu verwenden?« oder »Sollte man jetzt schon auf Version x wechseln?« suchen, werden Sie unter den nützlichen Einträgen fast immer eine Variante des Es-kommt-drauf-an finden, gefolgt von einigen Hinweisen, welche Faktoren Sie beim Abwägen Ihrer Entscheidung berücksichtigen sollten.

Dieses Austarieren unterschiedlicher und teilweise sich widersprechender Erwartungen ist ein bestimmendes Merkmal jeder ingenieurtechnischen Tätigkeit. Und das gilt ohne Einschränkungen für das Schreiben von Code.

Guter Code ist das Produkt einer Unmenge von Abwägungen, bei denen hinreichend oft die richtige Entscheidung getroffen wurde. Dabei ist es zunächst uninteressant, ob Abwägungen überhaupt bewusst wahrgenommen werden und wie die Entscheidungen letztlich zustande kommen. Wichtig ist, dass diese Abwägungs- und Entscheidungsprozesse einen großen Einfluss auf die Qualität des Codes haben.

Angesichts der weit über die Programmierung hinausgehenden Bedeutung ist es überraschend, dass Sie kaum Bücher oder andere Anleitungen finden werden, die sich mit dieser *Kunst des Abwägens* befassen. Ohne eine gewisse Versiertheit in dieser Kunst nützen Ihnen jedoch die besten Regeln nicht viel. Deshalb werde ich jetzt versuchen, Ihnen einige Tipps zu geben, die möglicherweise nützlicher sind als die eine oder andere formale Regel.

Sie haben vielleicht schon von der bei Konstrukteuren geläufigen Regel »so gut wie nötig – so schlecht wie möglich« gehört. Das ist der Prototyp einer Forderung, bei der es ums Abwägen geht.

Ingenieure haben es allerdings oft leichter als Softwareentwickler, weil sich hinter dem »schlecht« fast immer nur ein »kostengünstig« verbirgt, während das »gut« durch den Auftrag und Normen recht genau vorgegeben ist. Wirklich frei abwägen können Sie eher solche Dinge wie die Eleganz der Konstruktion.

Widersprüche überall

Sobald Sie ein etwas komplexeres Problem bearbeiten, werden Sie unvermittelt vor Entscheidungen stehen, für die es einander widersprechende Empfehlungen oder Regeln gibt oder in denen Sie sich gegenseitig ausschließende Anforderungen erfüllen sollen. Dann ist guter Rat teuer.

Im Alltag lassen sich verschiedene Strategien beobachten, wie Menschen in einer solchen Situation reagieren:

- **Aussitzen**

  Die Entscheidung wird so lange offengelassen, bis die Umstände oder jemand anderes eigenes Handeln überflüssig machen.

- **Ausweichen**

  Es wird versucht, eine Lösung zu finden, die die Konfliktsituation umgeht.

- **Nachahmen**

  Ohne Berücksichtigung der konkreten Bedingungen wird die Option gewählt, die man selbst oder ein anderer bei ähnlichen Wahlmöglichkeiten schon einmal genommen hat.

- **Abwägen und entscheiden**

Es wird, unter Bewertung der Vor- und Nachteile, eine der Situation entsprechend optimale Entscheidung getroffen.

Die ersten drei Alternativen können in speziellen Fällen angemessen sein. Problematisch ist jedoch, dass sie häufig unbewusst gewählt werden. Manches, was sich später als vermeintlich ungünstige oder falsche Entscheidung herausstellt, ist in Wirklichkeit eine nicht bewusst getroffene und daher im Endeffekt zufällige Entscheidung gewesen.

Softwareentwicklung ist im Allgemeinen eine äußerst komplexe Angelegenheit, und es gibt daher eine große Anzahl von Fragen, die durch das Ausbalancieren widersprüchlicher Forderungen gelöst werden müssen. Als Beispiel sei hier auf die in Kapitel 2 *Dimensionen von Codequalität* beschriebene Bestimmung eines projektbezogenen Qualitätsziels hingewiesen.

Auch die im Teil III dargestellten Programmierregeln lassen sich nicht immer widerspruchsfrei anwenden. Der produktive Umgang mit Widersprüchen und den daraus folgenden Konflikten ist eine weithin unterschätzte Kernkompetenz guter Entwickler.

Konflikte akzeptieren

Obwohl es völlig klar ist, dass Software nicht ohne Konflikte produziert werden kann, wird diese Tatsache viel zu oft verdrängt. Am ehesten lassen sich noch Veröffentlichungen zum ewigen Grundkonflikt zwischen Anforderungen und akzeptierten Kosten finden.

Es ist indes immens wichtig, dass Sie die vielen größeren und kleineren Widersprüche sehen und bewusst einer Lösung zuführen. Dadurch entsteht zwar Aufwand, aber Konflikte verschwinden nicht, wenn Sie sie ignorieren. Viele Probleme in Softwareprojekten lassen sich auf verspätete oder unterbliebene Entscheidungen zurückführen.

Akzeptieren Sie, dass es sich widersprechende Anforderungen gibt. Treffen Sie eine Entscheidung. Es ist meist besser, eine ungünstige Entscheidung zu treffen als gar keine.

Falls sich eine Entscheidung später als falsch herausstellt, korrigieren Sie diese, soweit das möglich ist, und versuchen Sie, die negativen Auswirkungen zu minimieren.

Einige Ratschläge dazu, wie Sie zu einer ausgewogenen Entscheidung kommen, werde ich Ihnen im nächsten Abschnitt geben.

Verzögerungskosten

In einem Projekt bei einer großen Reederei konnten die IT-Entwicklungsprozesse erheblich verbessert werden, indem zu jeder anstehenden Entscheidung Verzögerungskosten ermittelt wurden. Das heißt, jedem Tag, um den eine Entscheidung herausgezögert wird, ist ein konkreter Geldbetrag zugeordnet, der dadurch verloren geht, dass die betroffenen Projektergebnisse noch nicht nutzbringend eingesetzt werden können.

Der Verweis auf bezifferbare Verluste ist ziemlich effektiv, wenn es darum geht, Managern Entscheidungen abzuringen, die aus deren Sicht eher technisch und daher unbeliebt sind.

Entscheidungen systematisch treffen

In diesem Abschnitt gebe ich Ihnen einige Hinweise, wie Sie erfolgreich zu sinnvoll abgewogenen Entscheidungen kommen können. Verstehen Sie das bitte vor allem als Denkanstoß – keinesfalls als stur zu befolgendes Rezept.

Vielleicht haben Sie ja bereits eine gute eigene Strategie entwickelt. Dann können Sie diesen Abschnitt überspringen oder einfach nur sehen, ob Sie nicht noch eine Anregung finden. Auf

die Dauer entwickelt ohnehin jeder Mensch seine eigenen Verfahren, um Konflikte zu lösen. Dabei steht das Wort *Konflikt* hier für jede (Auswahl-)Situation, die eine Entscheidung erfordert.

Konflikte erkennen

Selbst wenn Ihnen diese Aussage trivial erscheinen mag, es ist wichtig, Konflikte überhaupt zu erkennen. Besonders wenn Sie auf einem Gebiet bereits etwas Übung haben, kann es leicht passieren, dass Sie durch gewohnheitsmäßiges Handeln Konflikte einfach übersehen.

Sie verzichten dabei auf Wahlmöglichkeiten, indem Sie eingeübte Muster kopieren. Oft werden Sie damit gar nicht schlecht fahren, insbesondere wenn sich an den sonstigen Rahmenbedingungen wenig geändert hat. Trotzdem sollten Sie sich dessen bewusst sein. Denn von Zeit zu Zeit oder spätestens bei einem völlig neuen Projekt ist es angebracht, solche Gewohnheiten zu überprüfen.

Bei allen Schritten zur Konfliktlösung ist es nützlich, wenn Sie sie nicht im »stillen Kämmerlein« allein ausführen. Versuchen Sie, wann immer möglich, mit jemanden darüber zu sprechen. Allein der Versuch, ein Problem verständlich zu erklären, führt nicht selten schon zu neuen Erkenntnissen.

Alternativen sammeln

Nachdem Sie einen Konflikt gefunden haben, folgt als nächster Schritt die Zusammenstellung der Optionen, also der möglichen Entscheidungsergebnisse. Versuchen Sie, dabei eine möglichst vollständige Übersicht zu erhalten. Manchmal sind gerade die Alternativen, an die zunächst niemand gedacht hatte, die besten.

Dazu müssen Sie das Problem genauer beschreiben und abgrenzen. Zuerst benötigen Sie eine Liste der Faktoren, für die Sie mit Ihrer Entscheidung eine Festlegung treffen müssen. Solche Faktoren können beispielsweise die Architektur, die

Programmiersprache oder auch die Anzahl oder das Kenntnisniveau der Entwickler sein.

Überprüfen Sie dann, ob jeder dieser Faktoren das Ergebnis wirklich beeinflusst. Wenn das nicht so ist, können Sie den betreffenden Faktor wieder aus der Liste streichen, weil nichts zu entscheiden ist.

Lassen Sie sich dabei nicht durch vermeintlich »sinnlose« Varianten, zum Beispiel den völligen Verzicht auf ein Projekt, irritieren. Es geht um eine Art des *Brainstormings*, mit dem Ziel, keine potenzielle Lösung zu übersehen. Bewertung und Auswahl folgen in den nächsten Schritten.

Kriterien finden

Um die gesammelten Alternativen bewerten zu können, brauchen Sie Kriterien. Das können die bereits erwähnten Code-Qualitätskriterien sein. Fast immer spielen Kosten eine Rolle, und sehr häufig hat auch die verfügbare Zeit einen wichtigen Einfluss. Weitere mögliche Kriterien:

- ✔ Vorhandene Kenntnisse und Erfahrungen der Entwickler
- ✔ Erwartungen des Auftraggebers
- ✔ Verfügbare Hard- und Software
- ✔ Vorgaben für den Betrieb der Anwendung

Die Reihe ließe sich fortsetzen. Innerhalb eines Projekts wird ein großer Teil der Kriterien immer gleich sein. Wenn Sie sich also einmal gründlich Gedanken gemacht haben, werden Sie die Ergebnisse mehrfach nutzen können.

Für die infrage stehende Entscheidung müssen Sie nun diejenigen Kriterien finden, die überhaupt einen spürbaren Einfluss haben. Aus diesen werden dann die wichtigsten ausgewählt.

Die Gewichtung der Kriterien ist nicht immer ganz einfach. Denken Sie an die Wartbarkeit des Codes. Wie wichtig diese

Eigenschaft ist, hängt nicht unerheblich davon ab, wie lange der Code produktiv sein wird. Um das abzuschätzen, brauchen Sie eine Prognose, die naturgemäß mit mehr oder weniger großen Unsicherheiten behaftet ist. Das macht das Ganze nochmals unübersichtlicher. Denn nun müssen Sie zusätzlich die Wahrscheinlichkeit für das Eintreten der Prognose berücksichtigen. Wenn die Unsicherheit groß ist, sollten Sie zumindest zwei Szenarien betrachten: den wahrscheinlichsten und den ungünstigsten Fall.

Wenn eine Prognose Einfluss auf die Wichtigkeit von mehreren Kriterien hat, ist es wichtig, dass Sie die gemeinsame Bewertung jeweils getrennt für jedes prognostizierte Szenario vornehmen und nicht das Gewicht eines Kriteriums für den schlechtesten Fall und gleichzeitig das eines anderen für den wahrscheinlichsten Fall ansetzen.

Wahlmöglichkeiten bewerten

So vorbereitet, sollte Sie eigentlich nichts mehr an einer zügigen Entscheidung hindern. Manchmal ist es tatsächlich so einfach. Sobald mehrere Kriterien gleichzeitig berücksichtigt werden sollen, erfüllt aber gewöhnlich keine der gefundenen Alternativen alle ausgewählten Bedingungen hinreichend gut.

Dann sind Sie gefordert, wirklich abzuwägen. Eine recht brauchbare Heuristik für diesen Fall ist das sogenannten *Paretoprinzip*, auch als *80/20-Regel* bekannt. Es besagt, dass sich 80 % eines vollständigen Ergebnisses bereits mit 20 % des gesamten Aufwands erreichen lassen.

Es gibt zwar keinerlei Beweis für die Richtigkeit dieser Behauptung, aber die praktische Erfahrung bestätigt zumindest, dass häufig ein sehr kleiner Teil der Anforderungen überproportionalen Aufwand verursacht. Sehr oft sind das besondere Konstellationen, die nur schwierig bewältigt werden können und die dann vielleicht im späteren Betrieb nur selten oder gar nicht vorkommen.

Diese Erfahrung können Sie sich zunutze machen und überprüfen, ob es hilft, ein Kriterium nur teilweise anzuwenden. Es wäre beispielsweise denkbar, dass Sie sich entschließen, eine gute Wartbarkeit des Codes nur für die wichtigen Teile oder die, die voraussichtlich häufige Anpassungen erfordern, anzustreben, weil andernfalls eine Termin- oder Kostengrenze nicht einzuhalten ist.

Eine Möglichkeit, zu hohen Kostendruck und daraus resultierende schlechte Codequalität zu vermeiden, die viel zu selten benutzt wird, besteht darin, zu überprüfen, ob es nicht wirtschaftlicher ist, bestimmte Sonderfälle gar nicht durch ein Programm zu behandeln, sondern auszusteuern.

Eine manuelle Nachbearbeitung verursacht zwar ebenfalls zusätzliche Kosten, aber bei überschaubaren Mengen sind diese oftmals sehr viel niedriger als das, was die Implementierung und der Test der programmtechnischen Lösung gekostet hätten. Zumal andernfalls die zusätzliche Komplexität über die gesamte Lebensdauer erhalten bleibt und höhere Wartungskosten zur Folge hat.

Wenn Ihnen das Paretoprinzip nicht zusagt oder kein befriedigendes Ergebnis liefert, können Sie ein aufwendigeres Bewertungsverfahren anwenden, das ich Ihnen im folgenden Beispiel demonstrieren möchte.

Angenommen, Sie haben vier mögliche Varianten V1–V4 gefunden und es sollen zwei Kriterien K1, zum Beispiel die Wartbarkeit, und K2, zum Beispiel die Verarbeitungsgeschwindigkeit, erfüllt werden. Im ersten Schritt wird den Kriterien eine relative Wichtigkeit (W) zugewiesen, das sei ein Wert zwischen 1 (wenig wichtig) und 5 (sehr wichtig). In gleicher Weise werden jeder Variante Erfüllungswerte (E) zugeordnet, die angeben, in wieweit sie ein Kriterium erfüllt (1 – minimal bis 5 – vollständig). Beispielsweise erfülle die Variante V1 die Kriterien K1 und K2

jeweils mit dem Wert 3 (mittelmäßig). Zusammen mit weiteren fiktiven Werten lässt sich diese Tabelle aufbauen:

Kriterium	*Wichtigkeit W*	*V1*			*V2*			*V3*			*V4*		
		E	E * W		E	E * W		E	E * W		E	E * W	
K1	4	3	12		2	8		4	16		5	20	
K2	2	3	6		5	10		3	6		2	4	
Summe			18			18			22			**24**	

Für jede Variante wird der Erfüllungsgrad mit dem Gewicht des Kriteriums multipliziert und das Ergebnis in die Spalte »E * W« geschrieben. Diese Produktwerte werden spaltenweise (pro Variante) summiert. Die Variante mit der größten Summe, hier V4, stellt dann den besten Kompromiss dar.

Falls eine feinere Abstufung sinnvoll ist, können Sie natürlich auch eine Bewertungsskala mit mehr als fünf Stufen verwenden. Es ist ebenfalls nicht notwendig, dass die beiden Bewertungsskalen gleich viele Stufen haben.

Entscheiden

Nachdem Sie so alles gründlich erwogen und vorbereitet haben, ist es Zeit, endlich zu einem Ergebnis zu kommen. Möglicherweise ist das für Sie inzwischen schon klar und Sie sind fertig.

Bisweilen passiert es allerdings, dass Sie trotz der aufwendigen Bewertung immer noch kein klares Bild haben. Dann können Sie versuchen, weitere Kriterien einzubeziehen, um auf diese Weise doch noch einen erkennbaren Unterschied zu finden.

Oder aber, Sie wählen willkürlich eine der (fast) gleichwertigen Varianten aus. Treffen Sie die Auswahl wirklich zufällig und ohne weitere Überlegungen. Im Mittel werden Sie damit ein gutes Ergebnis erreichen.

Schwierige Entscheidungen kann man als Spiel gegen die Komplexität sehen, wobei gilt, dass Ihr Gegner, die Komplexität, zufällig und nicht zielgerichtet agiert. Die Spieltheorie beweist, dass sich bei einem Spiel dieses Typs Ihre Gewinnchancen erhöhen, wenn Sie ebenfalls in beschränktem Maße zufällig handeln.

Mit Augenmaß

Vielleicht fragen Sie sich jetzt: Mit derartigem Aufwand soll ich jede Entscheidung abwägen? Die Antwort lautet glücklicherweise nein, auf keinen Fall. Was ich Ihnen gezeigt habe, ist eine Folge von Schritten, die Sie zu ausgewogenen Entscheidungen führt.

Im Alltag werden Sie gar nicht die Zeit haben, alles so gründlich zu bedenken, und das ist aus den folgenden Gründen auch nicht notwendig:

- ✔ Auch beim Abwägen dürfen Sie das Kosten-Nutzen-Verhältnis nicht unbeachtet lassen. Der Aufwand des Abwägens muss geringer sein als die Kosten einer schlechten Entscheidung. Andernfalls ist die zufällige Auswahl eine bessere Alternative.
- ✔ Wenn Sie den vorgeschlagenen Weg mehrmals durchexerziert haben, werden Sie sehen, dass der Aufwand mit jedem Mal zurückgeht und Sie bald die wichtigsten Schritte quasi automatisch durchführen.
- ✔ Akzeptieren Sie eine ausreichend gute Entscheidung. Es muss nicht immer die beste Wahl getroffen werden.
- ✔ Je mehr Übung Sie mit Entscheidungen in einem Gebiet haben, umso eher können Sie sich auf Ihr *Bauchgefühl* verlassen.

Bauchgefühl

Psychologische Untersuchungen legen nahe, dass intuitive Entscheidungen (Bauchentscheidungen) vor allem in komplexen Situationen besser sind als rein rationale. Das intuitive System ist schnell, unbewusst, basiert eher auf Qualitäten und scheint mit deutlich komplexeren Aufgaben umgehen zu können als das rationale. Allerdings muss ersteres erst trainiert werden.

Experten für einen Entscheidungsbereich waren bei Versuchen durchaus in der Lage, intuitiv signifikant bessere Entscheidungen zu treffen als durch vernünftige Überlegung. Erstaunlicherweise verschlechterte sich das Ergebnis, wenn die Bauchentscheidung anschließend gründlich überdacht wurde.

Für gut Bauchentscheidungen wird empfohlen:

- ✔ die möglichen Alternativen zu sammeln und aufzuschreiben,
- ✔ diese Liste nochmals durchzugehen und auf sich wirken zu lassen,
- ✔ die Alternative, die ein »gutes Gefühl« gibt, auszuwählen.

Das Wichtigste in Kürze

- ✔ Guter Code beruht auf einer Unmenge gut abgewogener Entscheidungen.
- ✔ Effektives Abwägen ist eine völlig zu Unrecht wenig gewürdigte Kernkompetenz guter Entwickler.
- ✔ Eingeschliffene Gewohnheiten behindern manchmal das Erkennen von Auswahlsituationen und beschränken dadurch unbewusst die Freiheit der Entscheidung.
- ✔ Vor jedem Abwägen muss geklärt werden, ob der Nutzen einer guten Auswahl den damit verbundenen Aufwand rechtfertigt. Entscheiden Sie zufällig, wenn das nicht der Fall ist.
- ✔ Versuchen Sie – mit angemessenem Aufwand – eine möglichst vollständige Übersicht der zur Auswahl stehenden Alternativen zu erstellen.
- ✔ Bestimmen Sie die Wichtigkeit der zu erfüllenden Kriterien. Konzentrieren Sie sich auf diejenigen, die den größten Einfluss auf das Gesamtergebnis haben.

- ✔ Bewerten Sie die gefundenen Alternativen im Licht der wichtigsten Kriterien.
- ✔ Verschwenden Sie keine Zeit mit der Suche nach einer perfekten Lösung. Es reicht eine Entscheidung zu finden, die gut genug ist.
- ✔ Mit etwas Erfahrung und nach guter Vorbereitung vermittelt die Intuition (Bauchgefühl) oft bessere Ergebnisse als rationales Analysieren.

Kapitel 4

Die Eigenschaften sauberen Codes

IN DIESEM KAPITEL

- Warum gibt es Clean Code?
- Was bedeutet Clean Code konkret?
- Was sind die kennzeichnenden Eigenschaften sauberen Codes?

In diesem Kapitel lernen Sie die Motivation kennen, aus der heraus das Clean-Code-Konzept entwickelt wurde. Anschließend erfahren Sie, durch welche Eigenschaften sauberer Code hilft, Software langfristig am Leben zu erhalten.

Des Clean Codes Kern

Es ist nicht nur schwierig, gute Programme zu schreiben. Als viel schwieriger hat sich herausgestellt, Programme weiterzuentwickeln. Mit dem Clean-Code-Konzept wird versucht, Erstellung und Weiterentwicklung von Software besser als bisher unter einen Hut zu bekommen.

Je leistungsfähiger die Hardware geworden ist, desto umfangreicher wurde auch die verwendete Software. Dementsprechend sind die investierten Geldbeträge gewachsen. Software, die so wertvoll ist, kann man nicht einfach ersetzen, wenn sich Anforderungen ändern. Ganz abgesehen davon, dass das eine viel zu lange Entwicklungszeit erfordern würde.

Deshalb ist im Laufe der Zeit die Weiterentwicklung oder Wartung von Software ständig wichtiger geworden. Gleichzeitig ist zu beobachten, dass die Kosten der Weiterentwicklung mit jeder weiteren Veränderung steigen.

Parallel wächst das Risiko, dass neue Versionen mit unerkannten Fehlern ausgeliefert werden. Trotz steigender Aufwendungen sinkt also die Qualität. Die Software degeneriert. Der daraus folgende fragile Zustand vieler Anwendungssysteme ist vielerorts zu einem wirtschaftlich bedeutenden Hemmschuh geworden.

Der Umstand, dass Software immer länger im produktiven Einsatz bleibt und dabei häufig inakzeptabel hohe Wartungskosten anfallen, hat zu einer Verschiebung des Blicks auf den Software-Lebensprozess geführt.

Im Mittelpunkt steht nicht mehr die Erstellung von Software, sondern die Gewährleistung der langfristigen Wartbarkeit. Auf den Code bezogen heißt das, er muss so gestaltet sein, dass die vorhandenen Strukturen nicht schon nach wenigen Änderungen in einem Wust von Komplexität verschwinden.

Mit Clean Code verhält es sich wie mit einem Kunstwerk, über dessen künstlerischen Wert einige Übereinstimmung herrscht, ohne dass man diesen Wert an genau definierten Eigenschaften festmachen kann.

Natürlich wird es immer unterschiedliche Meinungen geben. Aber ich bin sicher, Sie werden schnell erkennen können, ob ein bestimmter Code klar und sauber oder unverständlich ist.

Code als Ziel

Bemühungen, bessere Programme zu schreiben, gibt es schon fast so lange, wie es Computer gibt. Im Laufe der Jahre sind viele gute Ideen zusammengetragen worden. Unzählige Programmiersprachen wurden kreiert, jede mit dem Versprechen, die Softwareentwicklung, zumindest in einem Teilbereich, deutlich zu erleichtern.

Daneben entstanden Entwicklungskonzepte wie die strukturierte und die objektorientierte Programmierung. Bei allen Unterschieden haben diese Ansätze eine Gemeinsamkeit: Es geht um Software und deren Eigenschaften. Der Code ist dabei nur Mittel zum Zweck, gewissermaßen die Blaupause des Produkts »Software«.

Und genau an diesem Punkt setzt das *Clean-Code-Konzept* an. Im Unterschied zu fast allen älteren Konzepten erklärt es den Code zum eigentlichen Produkt des Entwicklungsprozesses. Das ist ein einschneidender Perspektivwechsel, aus dem sich bemerkenswerte Konsequenzen ergeben.

Bevor ich auf diese eingehe, noch eine kurze Bemerkung zur Berechtigung dieser neuen Perspektive. In Kapitel 1 *Software ist Code* zeige ich, wie angemessen, wenn nicht sogar zwingend dieser Wechsel der Betrachtung ist. Code ist dabei nicht auf die bekannten Programmiersprachen beschränkt, sondern umfasst alles, was sich ohne Eingriff eines Menschen in definierter Weise in ein lauffähiges Programm übersetzen lässt.

Aus dem Ansatz, den Programmcode zum Ziel der Entwicklung zu erklären, ergeben sich Folgerungen:

- ✔ Clean Code ist keine Methode, sondern ein Zielbild.
- ✔ Clean Code ist nicht an eine Programmiersprache oder eine Programmiermethode gebunden, wobei die Verwendung vor allem im agilen Kontext verbreitet ist.
- ✔ Alle Regeln und Empfehlungen sind nur Hilfsmittel, wobei selbst deren strikteste Befolgung nicht garantiert, dass das angestrebte Ergebnis erreicht wird.
- ✔ Es erfordert Kenntnisse und Übung, Clean Code zu schreiben.

Clean Code

Der Name dieses Konzepts geht auf das 2008 erschienene Buch mit dem Titel »Clean Code – A Handbook of Agile Craftsmanship« von Robert C. Martin

zurück. Darin werden erstmals Ziele und Grundsätze umfassend und eingängig dargestellt.

Ähnliche Ansätze hatte es bereits früher gegeben, beispielsweise Donald Knuths »Literate Programming« von 1984, aber durch die Konzentration auf bestimmte, oft wenig verbreitete Werkzeuge und Methoden gelangten sie nicht zu größerer Bedeutung.

Der allgemeinere, auf den Code fokussierende Ansatz von Clean Code vermeidet diesen Fehler und erreichte daher einen erheblich höheren Bekanntheitsgrad. Der Preis dafür ist allerdings, dass die Umsetzung schwieriger ist und hohe Anforderungen an die Entwickler stellt. In der Praxis sind deshalb leider noch nicht so große Fortschritte zu beobachten, wie sie aufgrund der Bekanntheit vielleicht zu erwarten und ohne jeden Zweifel zu begrüßen wären.

Professionalität

Ein zweiter wichtiger Aspekt des Clean-Code-Konzepts, der sich beinahe zwangsläufig aus dem ersten ergibt, ist die Betonung der Professionalität. Weil Code wichtig ist, muss er entsprechend sorgfältig und gekonnt geschrieben werden. Das Erfüllen der funktionalen Anforderungen allein genügt nicht. Und es sollte auch nicht vom Zufall oder Ihrer jeweiligen Tagesform abhängen, wie sauber Ihnen der Code gerät.

Zuverlässig und auf Dauer hohe Qualität zu liefern, wie es das Clean-Code-Konzept fordert, setzt Professionalität voraus, das heißt, Sie müssen über einen Vorrat an erprobten und bewährten Verfahren und Werkzeugen verfügen, um die anstehenden Aufgaben bewältigen zu können. Nur aus diesem Grund sind einige Regeln so eng mit dem Schreiben sauberen Codes verknüpft, dass bisweilen der Eindruck entsteht, diese Regeln wären bereits das Wesen von Clean Code.

Handwerkliches Geschick

Gemeinhin wird auch in deutschen Texten oft das englische Wort *Craftsmanship* verwendet. Tatsächlich gibt es dafür keine gute Entsprechung. Denn gemeint ist damit nicht nur die handwerkliche Fertigkeit im Umgang mit Code, sondern in gleichem Maße auch die Handwerkerehre, die darin besteht, für das Ergebnis der Arbeit geradezustehen.

Professionalität bedeutet eben auch Sorgfalt bis ins Detail. Alles was Sie schreiben, muss sinnvoll und wohl durchdacht sein. Überlassen Sie nichts dem Zufall. Nicht ohne Grund ist das *handwerkliche Geschick* beziehungsweise die *Handwerkskunst* ein wichtiger Teil des Clean-Code-Konzepts.

Das Ergebnis Ihrer Bemühungen, der produzierte Code, soll so aussehen, dass Sie sich dafür nicht schämen müssen. Für jeden Entwickler muss es eine Frage der Ehre sein, stets das Beste zu liefern, was unter den gegebenen Umständen möglich ist.

Die Einschränkung »unter den gegebenen Umständen« sollten Sie wirklich ernst nehmen. Professionalität drückt sich vor allem auch darin aus, dass Sie das jeweils richtige Maß finden und danach handeln.

Es geht immer weiter

Der dritte Aspekt schließlich betrifft die Dauerhaftigkeit. Sie werden mit Clean Code nie fertig sein. Software »lebt«, das heißt, sie muss angepasst, erweitert oder bezüglich der Leistungsfähigkeit verbessert werden. Jede Weiterentwicklung ist eine neue Herausforderung.

Je nachdem wie umfangreich die Veränderung ist, müssen frühere Entscheidungen überprüft und korrigiert werden. Sehr oft werden Sie bei kleineren Korrekturen Kompromisse eingehen müssen, weil Ihnen zur eigentlich notwendigen größeren Überarbeitung Zeit und Mittel fehlen.

Außerdem lernen Sie täglich dazu. Wenn Sie ein Codestück nach längerer Zeit erneut bearbeiten, werden Sie es mit ganz anderen Augen betrachten und neue Verbesserungsmöglichkeiten sehen. Auch diese werden Sie oft nicht alle sofort umsetzen können.

Das Clean-Code-Konzept trägt solchen praktischen Schwierigkeiten Rechnung, indem es die *kontinuierliche Verbesserung* zu einem Grundprinzip erklärt. Weder

organisatorische Probleme, weil Sie beispielsweise die Verantwortlichen nicht von der Notwendigkeit einer Überarbeitung überzeugen können, noch praktische Schwierigkeiten, wenn Sie noch keine befriedigende Lösung finden können, dürfen Sie davon abhalten, sich stets um das bestmögliche Ergebnis zu bemühen.


Nutzen Sie alle Möglichkeiten zur Verbesserung und beherzigen Sie die sogenannte Pfadfinderregel: Verlassen Sie den bearbeiteten Code immer etwas sauberer, als Sie ihn vorgefunden haben!

Aber übertreiben Sie nicht. Überlegen Sie stets, welcher Aufwand gerechtfertigt werden kann. Vergessen Sie dabei nie: Clean Code ist kein Selbstzweck.

Code als Kommunikationsmittel zwischen Menschen

Code ist Text, geschrieben in einer formalen Sprache, meist einer sogenannten Programmiersprache. Die Hauptfunktion von Sprachen ist es, Kommunikation zu ermöglichen. Code ist dabei sehr lange vorrangig als eine Möglichkeit zur Kommunikation mit dem Computer gesehen worden.

Es ist ein Verdienst des Clean-Code-Konzepts, dass der Blick nun stärker auf die Rolle von Code als Kommunikationsmittel zwischen Entwicklern, also Menschen, gerichtet wird. Aus diesem Grund lassen sich die charakteristischen Eigenschaften sauberen Codes in erster Linie aus den kognitiven Fähigkeiten und Beschränkungen des Menschen herleiten.

Die nachfolgenden Gesichtspunkte lassen sich nicht sauber voneinander trennen, sind jedoch nicht identisch.

Lesbarkeit

Die Lesbarkeit von Code ist deshalb so wichtig, weil Code wesentlich öfter gelesen als geschrieben wird. Schließlich unterbrechen Sie selbst häufig das Schreiben, um einen Blick auf bereits geschriebenen Code zu werfen, also zu lesen.

Lesen ist eine komplexe Tätigkeit, für die Sie kognitive Ressourcen aufwenden müssen, um aus dem optischen Eindruck die relevanten Strukturen, beispielsweise Namen, Ausdrücke oder Anweisungen, ermitteln zu können. Ihr Ziel muss deshalb sein, diesen Aufwand möglichst gering zu halten. Einige wichtige Methoden, die Ihnen helfen werden, dieses Ziel zu erreichen, finden Sie in Teil III *Sauberen Code schreiben*.

Ein ganz allgemeiner Grundsatz gilt für jede Art von Kommunikation: Störungen müssen minimiert werden!

Störungsfreie Kommunikation ist praktisch nicht möglich. Die Techniker bezeichnen diejenigen unvermeidbaren Signale, die zwischen den Kommunikationspartnern übertragen werden, ohne der eigentlichen Kommunikation zu dienen, als *Rauschen*. Dieses Rauschen belastet den Kommunikationskanal und verursacht unerwünschten Aufwand.

Bezogen auf den Code ist alles, was die Lesbarkeit behindert, Rauschen und daher möglichst zu vermeiden. Beispiele für solche störenden Elemente im Code sind überflüssige oder veraltete Kommentare, auskommentierter Code oder dauerhaft nicht benötigter Code.

Wenn derartige Störungen nicht vermeidbar sind, etwa Lizenzbedingungen, sollten Sie diese wenigstens optisch so abgetrennt anordnen, dass der normale Lesefluss nicht beeinträchtigt wird.

Formal und für den Computer sind Programme eine eindimensionale Folge von Zeichen. Für uns Menschen ist dagegen die zweidimensionale Darstellung extrem hilfreich, weil sie sonst verborgen bleibende Informationen sofort sichtbar macht. Allein aus (korrekten) Einrückungen können Sie beispielsweise schon die hierarchische Struktur erkennen, ohne

dass dazu die einzelnen Anweisungen wirklich gelesen werden müssen. Deshalb sollten Sie die äußere Form ganz bewusst in die Gestaltung einbeziehen.

Verständlichkeit

Verständlich ist Code, wenn er dem Leser das zugrunde liegende Modell vermitteln kann, oder anders ausgedrückt, wenn Sie beim Lesen sofort begreifen, was wie und warum gemacht wird.

Diese Anforderung ist schwerer zu erfüllen als die nach guter Lesbarkeit, weil dabei die individuellen Voraussetzungen, also unter anderem wie gut sich die Leser im betroffenen Anwendungsbereich auskennen, eine große Rolle spielen.

Unabhängig davon gibt es allgemeingültige Möglichkeiten, den Code verständlicher zu gestalten:

- ✔ Verwenden Sie gut gewählte und aussagekräftige Bezeichnungen.
- ✔ Seien Sie konsistent, ganz gleich, ob es sich um Namen, Klassen, Methoden oder die Formatierung handelt.
- ✔ Machen Sie das zugrunde liegende mentale Modell so weit wie möglich explizit erkennbar, indem Sie Klassen und Methoden analog strukturieren.

Ihr wichtigstes Ziel, das manchmal recht schwer zu erreichen sein kann, muss immer sein, die Komplexität nicht weiter zu erhöhen. Die meisten Algorithmen, die Sie programmieren, sind nicht trivial und daher nicht leicht zu verstehen. (Sonst hätte schließlich niemand Interesse an Ihrem Code.) Machen Sie es nicht noch komplizierter.

Als Beispiel für den Unterschied zwischen Lesbarkeit und Verständlichkeit können Sie an einen Artikel über ein Ihnen völlig unvertrautes Thema denken. Wenn er gut geschrieben ist, werden Sie ihn problemlos – vielleicht sogar mit Vergnügen – lesen können, ohne den Sinn wirklich zu verstehen. Den umgekehrten Fall, verständlich (oder eher verstehbar), aber schlecht lesbar, finden Sie manchmal bei gerade noch verwendbaren maschinell übersetzten Gebrauchsanleitungen.

Eleganz

Guter sauberer Code sollte schließlich eine gewisse Eleganz aufweisen. Zugegeben, diese Eigenschaft lässt sich noch schwerer fassen als die beiden vorangegangenen. Sie wird auch ganz bewusst erst nach diesen aufgeführt.

Doch ebenso wie ein geübter Handwerker seine Freude daran hat, eine Arbeit ganz besonders gut und effektiv zu erledigen, sollten Sie als Entwickler zumindest versuchen, Ihren Code so zu schreiben, dass der Anblick einfach Spaß macht. Niemand kann erwarten, dass Ihnen das jedes Mal oder sehr oft gelingt, aber es ist den Versuch wert.

Die Eleganz kann sich dabei auf ganz unterschiedliche Formen beziehen:

- ✔ Eine besonders kurze und übersichtliche Codesequenz
- ✔ Eine besonders effiziente Formulierung, die bei der Abarbeitung wenig Speicher oder Rechenzeit erfordert
- ✔ Eine besonders allgemeine Lösung, die mehrere speziellere Methoden ersetzen kann

Denken Sie daran: Eleganz ist nur ein untergeordnetes Ziel. Treiben Sie nicht allzu viel Aufwand dafür und seien Sie sich der Gefahr bewusst, dass übertrieben eleganter Code schwerer verständlich sein kann.

Gute Wartbarkeit

Die enormen Kosten großer Softwareprojekte führen dazu, dass die Weiterentwicklung existierender Software im Vergleich zu einer reinen Neuentwicklung immer wichtiger wird. Deshalb zielt das Clean-Code-Konzept vor allem auch darauf ab, die Wartbarkeit von Code zu verbessern.

Leichter durch Verständlichkeit

Die bereits angesprochenen Aspekte Lesbarkeit und Verständlichkeit erleichtern die Analyse bestehenden Codes. Das ist nicht nur bei späteren Änderungen nützlich, sondern kann Ihnen schon im Entwicklungsprozess helfen, wenn Sie zwischenzeitlich an einer anderen Funktion gearbeitet haben und nun wieder in den alten Kontext zurückkehren wollen. Außerdem erleichtert es die Einarbeitung neuer Entwickler.

Klar erkennbare Intentionen erhöhen die Wahrscheinlichkeit, dass später notwendig werdende Veränderungen innerhalb der vorhandenen Grundstrukturen bleiben. In Code, der über längere Zeit und von unterschiedlichen Entwicklern bearbeitet wurde, findet man häufig mehrere Methoden, die (fast) das Gleiche tun, oder Kommentare der Art »Ich verstehe das nicht und weiß nicht, ob das noch gebraucht wird, deshalb schreibe ich es neu.« Sauberer Code soll helfen, das zu vermeiden.

Kann Software gewartet werden?

Wörtlich genommen gibt es keine Softwarewartung, denn unter Wartung werden Maßnahmen zur Verhinderung oder Verminderung von Verschleißerscheinungen verstanden. Software verschleißt jedoch nicht.

Mitunter werden deshalb die richtigeren Begriffe *Weiterentwickelbarkeit* oder *Evolvierbarkeit* verwendet. Diese haben sich aber bisher nicht allgemein durchsetzen können. Ich verwende daher trotz der genannten Bedenken den etablierten Begriff Wartung.

Nicht ohne Test

Einen ganz wichtigen Teil des Clean-Code-Konzepts habe ich bisher noch gar nicht erwähnt: Tests. Es ist schön, wenn Code leicht lesbar, verständlich und vielleicht sogar elegant ist, aber das allein reicht nicht. Er muss vor allem korrekt sein.

Tests können die Fehlerfreiheit eines Programms zwar nicht beweisen, aber sie sind die beste verfügbare Methode, um zu prüfen, ob ein Programm so wie gedacht funktioniert. Tests sind für jede Art der Softwareentwicklung wichtig.

Weil Sie Clean Code nur selten im ersten Anlauf schreiben können und daher der normale Entwicklungsprozess eine Folge von kleinen und größeren Überarbeitungen ist, sind hier Tests, die am besten automatisiert nach jeder Änderung laufen, besonders nützlich. Ohne ausreichende Testabdeckung wird es Ihnen kaum gelingen, Codequalität und Korrektheit gleichermaßen sicherzustellen.

Das Testen ist für Clean Code so wichtig, dass an vielen Stellen die *testgetriebene Entwicklung* oder *Test Driven Development (TDD)* fast wie ein unverzichtbarer Bestandteil behandelt wird, was sie aber nicht ist.

Clean Code ist, wie gesagt, ein Zielbild, während TDD eine spezielle kleinschrittige Entwicklungsmethode ist, bei der die Tests stets vor den zu testenden Komponenten geschrieben werden.

Mit den Tests haben Sie bereits eine stabile Basis für die Weiterentwicklung des Codes gelegt. Die vorhandenen Tests geben Ihnen – bei fehlerfreiem Durchlauf – nach jeder

Modifikation des Codes die weitgehende Sicherheit, keine wichtige Funktionalität zerstört zu haben.

Selbst wenn diese Sicherheit nie vollständig sein kann, ist ihre psychologische Wirkung positiv. Sie arbeiten einfach freier und besser, wenn Sie nicht ständig mit der Angst leben, dass Ihre Arbeit – unbeabsichtigt – irgendwo eine übersehene Fehlfunktion verursacht.

Ein häufig anzutreffender Grund für die Degeneration von Code im Zuge der Wartung ist nämlich der folgende: Der beauftragte Entwickler möchte selbstverständlich Fehler vermeiden und versucht daher, so wenig wie möglich am Code zu ändern. Häufig fehlt ihm einfach die Zeit, sich ausreichend in die vorhandenen Strukturen einzudenken.

Änderungen erfolgen dann eher außerhalb der implementierten Konzepte, gewissermaßen als »Anbauten«. Dadurch wird die ursprüngliche Struktur schnell und effektiv zerstört. Besser wäre es, wenn er danach strebte, seine Korrekturen möglichst stimmig, und das heißt sehr oft durch stärkere Eingriffe, in das Vorhandene einzufügen. Das wird er jedoch nur machen, wenn das damit verbundene Risiko durch genügend viele Tests akzeptabel bleibt.

Zu guter Letzt

Ich betone es noch einmal: Clean Code ist ein Ziel, ein Ideal. Sauberer Code sollte wie ein gutes Buch sein: leicht zu lesen und gut zu verstehen. Aber wie bei einem Buch wird es immer verschiedene Meinungen geben, was gut und was besser ist. Und es wird Themen geben, die sich flüssig und eingängig beschreiben lassen, und andere, die bei allen Mühen sperrig und anspruchsvoll bleiben. Das ist so.

Ebenso sollten Sie akzeptieren, dass nicht jedes Werk ein die Jahrhunderte überdauerndes Kunstwerk werden muss. Die benötigte Zeit ist ein wichtiger Faktor. Aber wenn Sie es sich angewöhnt haben, nach den Clean-Code-Idealen zu streben, werden auch Ihre »kleineren Werke« besser werden.

Zum Schluss noch eine kleine, nicht zu ernst gemeinte Warnung: Guter Code stellt alles so klar und einfach dar, dass der Betrachter den darin steckenden Aufwand wahrscheinlich gar nicht wahrnimmt: »Das kann doch gar nicht so schwer gewesen sein.« Also vergessen Sie nicht, diejenigen, die Ihre Arbeit möglicherweise beurteilen, an Ihren Mühen, besseren Code zu schreiben, teilhaben zu lassen: Klappern gehört zum Handwerk!

Das Wichtigste in Kürze

- ✔ Wenn Code weiterentwickelt werden muss, was häufig der Fall ist, führt das sehr oft zu seiner Degeneration. In der Folge steigen die Wartungskosten stark an, bis im schlimmsten Fall keine Weiterentwicklung mehr möglich ist.
- ✔ Das Clean-Code-Konzept stellt den Code in den Mittelpunkt, um eine möglichst langfristige Weiterentwickelbarkeit zu erreichen.
- ✔ Clean Code ist keine Methode, sondern ein Ziel.
- ✔ Das Schreiben von Code muss auf solider Professionalität beruhen.
- ✔ Clean Code lässt sich kaum im ersten Anlauf schreiben. Die kontinuierliche Verbesserung des Codes ist daher ein wesentlicher Aspekt.
- ✔ Code ist vor allem auch Kommunikationsmittel zwischen Menschen. Deshalb sind Lesbarkeit, Verständlichkeit und Eleganz wichtige Eigenschaften.
- ✔ Um Clean Code zu erreichen und zu erhalten, sind umfassende Tests unabdingbar.
- ✔ Clean Code ist kein Selbstzweck. Aufwand und Nutzen müssen immer wieder neu abgewogen werden.

Kapitel 5

In der Praxis: Stolpersteine

IN DIESEM KAPITEL

Die Kosten von Clean Code

Clean Code sauber ändern

Hinderliche Rahmenbedingungen

Sie kennen jetzt die Grundzüge des Clean-Code-Konzepts, die Motivation dahinter und die erhofften Vorteile. Das Ziel ist somit klar. Auf dem Weg dorthin liegen allerdings Stolpersteine und, wenn Sie Pech haben, stoßen Sie auch auf größere Hindernisse. Damit diese Schwierigkeiten Sie nicht völlig unvermittelt treffen, lernen Sie in diesem Kapitel einige davon kennen und erhalten Hinweise, wie Sie gekonnt damit umgehen können.

Clean Code ist schwer

Es hat keinen Zweck, die Tatsache zu verleugnen: Clean Code ist schwer. Früher oder später hätten Sie das ohnehin erkannt und sich dann vielleicht enttäuscht abgewandt.

Ich glaube, dass einer der Gründe für die nur verhaltene Verbreitung von Clean Code darin liegt, dass einige seiner Verfechter versuchen, die Schwierigkeiten zu bagatellisieren. Mit dem Lernen und Einhalten einiger Regeln ist es jedoch nicht getan.

Lassen Sie sich davon bitte nicht entmutigen. Denken Sie daran, dass es viele Dinge im Leben gibt, bei denen absolute Meisterschaft nur schwer zu erlangen ist. Mit etwas Übung

erreichen die meisten von uns aber sehr wohl ein akzeptables Niveau.

Ganz ohne Anstrengungen werden Sie dem Clean-Code-Ideal allerdings nicht nahekommen können. Mit den zu bewältigenden Herausforderungen befasse ich mich ausführlich in Teil II.

Reden wir über die Kosten

Sauberen Code zu schreiben, ist – zumindest auf den ersten Blick – aufwendig und damit teuer. Wahrscheinlich müssen Sie oft und lange überlegen und die notwendigen Überarbeitungen brauchen ebenfalls ihre Zeit. Das gilt besonders am Anfang, wenn Sie noch nicht so geübt sind.

Aus Sicht der Geschäftsführung entstehen nur Mehrkosten ohne erkennbaren Nutzen. Selbst für Sie als Entwickler ist der Vorteil zunächst gering. Vielleicht empfinden Sie eine größere Befriedigung bei Ihrer Arbeit, viel mehr nicht.

Kurz- und mittelfristige Vorteile

Über die Zeit werden sich jedoch einige Vorteile gegenüber herkömmlichen Projekten zeigen:

- ✔ **Die Fehlerrate ist niedriger**, weil Sie bereits beim Entwickeln viele Tests schreiben und beim Überarbeiten der vorliegende Code, um die Sauberkeit zu erhalten, genauer inspiziert werden muss.
- ✔ **Die Entwicklungsgeschwindigkeit nimmt zu**. Je umfangreicher der Code wird, desto öfter müssen Sie in der Regel an Stellen nachsehen, die bereits vor längerer Zeit programmiert wurden. Durch die bessere Lesbarkeit benötigen Sie weniger Zeit, um älteren Code zu verstehen und die benötigten Informationen zu finden.
- ✔ **Die Motivation der Entwickler verbessert sich.** Es macht einfach mehr Spaß, mit sauberem Code zu arbeiten und ein ordentliches Ergebnis abzuliefern.

Gerade der letzte Faktor kann gar nicht hoch genug eingeschätzt werden. Motivierte Entwickler arbeiten schneller und besser. Außerdem verringert sich die Fluktuation. Der Austausch von Entwicklern ist wegen des damit verbundenen Verlusts von Wissen und durch die notwendige Einarbeitung eines Ersatzes nicht nur teuer, sondern verzögert unweigerlich das Projekt. Die daraus schon bei einem einzigen Wechsel resultierenden Kosten dürften meistens höher sein als diejenigen, die das Bemühen um sauberen Code verursacht.

Das Problem besteht darin, Managern diesen Zusammenhang bewusst zu machen. Viele neigen leider dazu, Fluktuation als unbeeinflussbare Heimsuchung hinzunehmen, und übersehen dabei ihre objektiv vorhandenen Steuerungsmöglichkeiten.

Langfristige Vorteile

Richtig rentiert sich Clean Code aber vor allem auf lange Sicht. Wenn die Software, die Sie entwickeln, in größeren Abständen und vielleicht sogar von unterschiedlichen Entwicklern bearbeitet werden muss, kommen die Vorteile zum Tragen.

Aber Vorsicht, die Sauberkeit des Codes muss verteidigt werden. Wenn niemand darauf achtet, dass die Clean-Code-Prinzipien bei jeder Erweiterung und jeder Fehlerbehebung genauso gewissenhaft befolgt werden wie beim Erstellen, wird sich die Degeneration des Codes nur verzögern.

Bei den ersten Veränderungen kommt Ihnen der ursprünglich investierte Aufwand noch zugute, aber sobald Sie bestimmte Codebereiche wiederholt bearbeiten müssen, werden Sie mit den gleichen negativen Effekten zu kämpfen haben, die sich in Projekten einstellen, bei denen von vornherein weniger achtsam mit dem Code umgegangen wurde.

Um Clean Code »clean« zu erhalten, müssen Sie auch in der Wartungsphase der Software noch zusätzlichen Aufwand erbringen. Der Grund dafür lässt sich leicht erklären: Sie müssen sich die ursprünglichen Strukturen und Begriffe erneut erarbeiten. Nur so lässt sich Code organisch weiterentwickeln.

Das heißt, zunächst wird auch die Weiterentwicklung noch (geringfügig) höhere Kosten verursachen. Einsparungen ergeben sich erst langfristig dadurch, dass die Aufwände für die Pflege weitgehend konstant bleiben.

Im Gegensatz dazu steigen die Kosten bei der Weiterentwicklung schlechten Codes schnell an. Abbildung 5.1 stellt den Verlauf der normierten Änderungskosten über die Software-Lebensdauer schematisch dar. In der Anfangsphase bis zum Zeitpunkt *B* verursacht sauberer Code die erwähnten höheren Kosten. Diese Investition beginnt sich danach zu rentieren und hat sich bei *G* amortisiert. Danach beginnt die Gewinnzone.

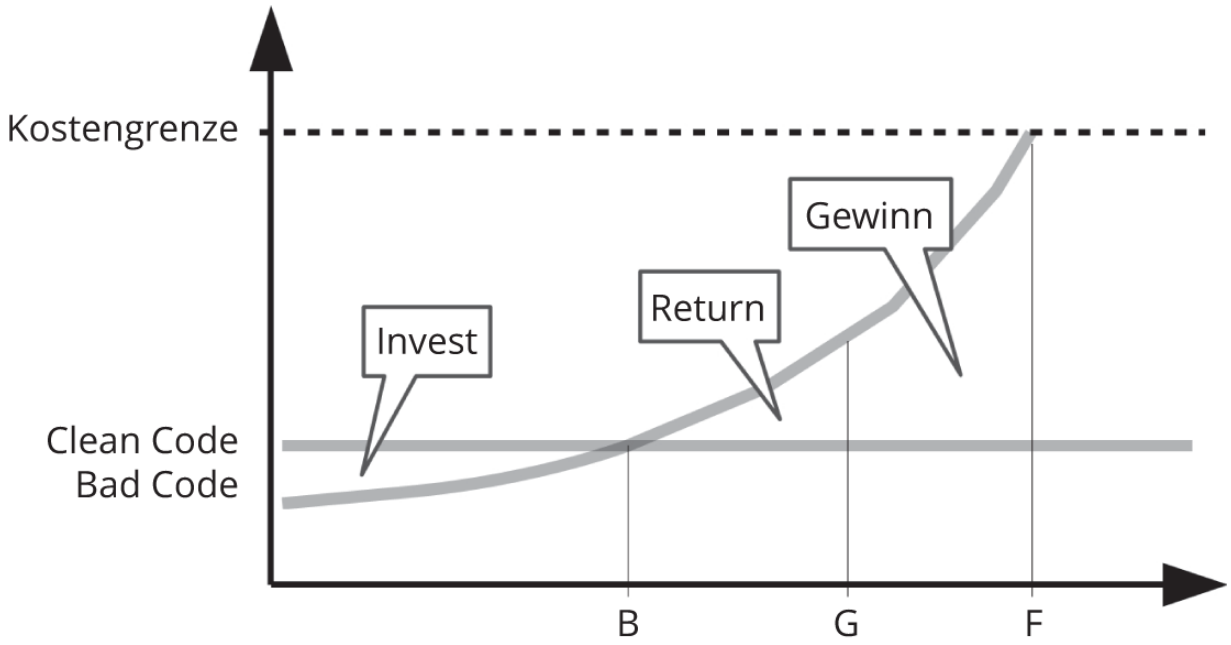


Abbildung 5.1: Normierte Änderungskosten

Außerdem wird noch ein weiterer Gesichtspunkt veranschaulicht, nämlich die Existenz einer Kostengrenze für Änderungen. Zum Zeitpunkt *F* ist es wirtschaftlich nicht mehr vertretbar, den schlechten Code am Leben zu erhalten; er muss ersetzt werden.

In der Praxis ist der Wert dieser Kostengrenze vielgestaltig: In jedem Fall stellen die Kosten einer Neuentwicklung eine absolute Grenze für die akzeptablen Änderungskosten dar. Die Kostengrenze kann aber auch durch den Preis eines alternativen

Produkts bestimmt sein. Überdies können Kostenvorgaben des Managements oder der Markteintritt von Wettbewerbern zu recht abrupten Änderungen führen.

Nicht vergessen sollten Sie auch, dass beispielsweise die Übergabe der Pflege von Clean Code an ein anderes Team, zum Beispiel durch Outsourcing, die Kosten nur vorübergehend erhöht. Entsprechende Qualifikation vorausgesetzt, profitiert das neue Team ebenso wie vorher bereits neue Mitarbeiter von der leichteren Einarbeitung und wird dann bald effektiv arbeiten können.

Die wichtigsten Vorteile allerdings sind mangels einer Vergleichsbasis kaum zu quantifizieren:

- ✔ Schnellere Realisierung neuer Features.
- ✔ Bessere Qualität der ausgelieferten Versionen.
- ✔ Weniger Aufwand für Fehlersuche und Fehlerbehebung.
- ✔ Ein aus den vorgenannten Eigenschaften folgendes hohes Ansehen bei den Kunden.

Bewertung

Das Clean-Code-Konzept hat die langfristige Weiterentwickelbarkeit von Software zum Ziel. Deshalb ist es nicht verwunderlich, dass sich Vorteile eher über einen längeren Zeitraum ergeben. Für eine Bewertung besteht deshalb die größte Schwierigkeit darin, den passenden Zeithorizont zu finden. Je weiter ein Projekt in die Zukunft reicht, umso stärker wird es vermutlich von sauberem Code profitieren.

Denken Sie daran, dass ein großer Teil der Softwareprojekte länger läuft als zunächst geplant. Mit langer Entwicklungszeit steigt gleichzeitig die Wahrscheinlichkeit, dass sich Änderungswünsche oder sogar -notwendigkeiten bereits vor der Fertigstellung ergeben.

Ein anderer Faktor, der Clean Code wirtschaftlich angeraten macht, ist die Anzahl der Entwickler. Je mehr parallel an einem

Projekt arbeiten, desto öfter wird einer den von anderen Entwicklern geschriebenen Code lesen und verstehen müssen.

Ab einer gewissen Projektgröße macht sich Clean Code immer bezahlt. Das beginnt früher, als Sie vielleicht denken, denn ausschlaggebend ist nicht der geplante, sondern der tatsächliche Umfang.

Aber übertreiben Sie es nicht so sehr, dass der Projektfortschritt darunter unangemessen leidet.

Ändern bleibt schwierig

Clean Code erleichtert Ihnen zwar das Lesen und Verstehen des Codes, aber wenn es ans Verändern geht, werden Sie feststellen, dass sauberer Code bisweilen ebenfalls sehr schwer zu modifizieren ist.

Sie können sich das an einer Analogie veranschaulichen. Bestimmt haben Sie selbst schon einmal einen Text oder wenigstens einen Absatz geschrieben, der Ihnen rundum gelungen erschien. Und dann musste in diesem Text plötzlich noch etwas ergänzt werden. Es ist tragisch, doch in vielen Fällen gelingt es trotz immenser Bemühungen nicht, die ursprüngliche Geschlossenheit wiederherzustellen.

Besonders das Zerlegen des Codes in viele kleine Methoden erweist sich bisweilen als ernsthaftes Hindernis, wenn Änderungen nicht innerhalb dieser Struktur formuliert werden können und beispielsweise andere Parameter benötigt werden.

Einige Autoren schlagen deshalb vor, in solchen Fällen einen Umweg zu nehmen. Im ersten Schritt wird ein Teil der Methoden durch Inlining wieder entfernt. Dieser sozusagen »denormalisierte« Code wird dann verändert und in einem finalen Schritt wieder gesäubert, das heißt, es werden wieder die erwünschten kleinen Methoden extrahiert.

Inlining

Unter *Inlining* versteht man das Ersetzen von Methodenaufrufen durch eine Kopie des Methodenkörpers, unter Berücksichtigung der Parameter, in dem aufrufenden Programmtext. Ursprünglich war das vor allem eine Technik, die beim Kompilieren mit dem Ziel der Laufzeitverbesserung des übersetzten Programms angewandt wurde.

Inzwischen gibt es diese Funktion bereits auf Quellcode-Ebene in den meisten Entwicklungsumgebungen, um die Überarbeitung von Code zu erleichtern. Eine analoge Unterstützung existiert beim Gegenstück, dem Extrahieren von Methoden.

Dieses Vorgehen ist nicht ganz unproblematisch. Zum einen arbeiten die Funktionen zum Inlining und zur Extraktion – aus prinzipiellen Gründen – nicht immer völlig korrekt. Sie sollten die Ergebnisse stets überprüfen. Zum anderen unterscheiden sich dadurch aufeinanderfolgende Versionen im Detail schnell so stark voneinander, dass bei einem Vergleich, beispielsweise im Versionierungssystem, die eigentlich wichtigen Änderungen nur noch schwer zu erkennen sind.

Nicht zu vergessen ist auch, dass Sie die durch das Inlining verloren gegangenen Methodennamen beim abschließenden Extrahieren von Hand und möglichst übereinstimmend wieder vergeben müssen.

Nun lassen Sie sich bitte durch diese Diskussion nicht davon abhalten, sauberen Code zu schreiben. Möglicherweise werden Sie nie in die beschriebene Situation kommen. Aber ich finde, es wäre unfair von mir, Sie nicht auf solche Fallen aufmerksam zu machen.

Ganz nebenbei gibt es mir die Gelegenheit, Sie daran zu erinnern, dass sich Übertreibung – wie immer – auch beim Saubermachen nicht auszahlt. Wenn es um Code geht, ist nichts in Stein gemeißelt, alles muss unter Umständen einmal geändert werden. Die Weisheit liegt im richtigen Maß.

Manchmal passt es nicht

Selbst wenn Sie Ihre Mitstreiter überzeugt haben und entschlossen sind, nur noch sauberen Code zu schreiben, kann es vorkommen, dass Sie plötzlich vor einer Schranke stehen, die sich nur passieren lässt, wenn Sie Konzessionen machen.

Frameworks

Es lässt sich schwerlich eine größere Anwendung finden, die nicht auf einem Framework basiert oder zumindest Teile davon nutzt. Beispielsweise werden Datenbankanbindungen oft mittels einer Implementation des *Java Persistence Interface* (JPI) realisiert. Analoge Frameworks existieren für Webservices und weitere Aufgaben.

Das Problem in Bezug auf Clean Code besteht darin, dass Frameworks eine bestimmte Struktur vorgeben. Zwei Beispiele für Einschränkungen, die sauberen Code verhindern können, sind:

- ✔ Die Forderung, dass jede Klasse einen parameterlosen nicht `private` Konstruktor haben muss. Dadurch wird unter anderem die Deklaration von unveränderlichen (`final`) Feldern ausgeschlossen, und es kann nicht mehr gewährleistet werden, dass erzeugte Objekte zu jedem Zeitpunkt mit allen notwendigen Werten versehen sind.
- ✔ Die Forderung, dass jedes Feld (`x`) über eine Methode gesetzt (`setX`) und gelesen (`getX`) werden kann. Das verletzt die Datenkapselung, weil alle Objektinterna exponiert werden.

Glücklicherweise geht die Anzahl der Frameworks mit derartigen Anforderungen zurück, nicht zuletzt durch den verstärkten Einsatz von Annotationen. Letztere sind unter Clean-Code-Gesichtspunkten allerdings auch nicht nur positiv zu sehen. Mehr dazu in Kapitel 19 *Immer weiter*.

Projektvorgaben

Mit dem löblichen Ziel, die Codequalität zu heben, ist es vielerorts üblich, Code vor dem Einchecken in die Versionsverwaltung gegen einen Satz von Regeln zu überprüfen. Das ist eine begrüßenswerte Praxis, wenn den Entwicklern die Möglichkeit eingeräumt wird, bestimmte Verstöße als beabsichtigt zu markieren, sodass in diesen Fällen ihr Code passieren kann. Bekanntlich gibt es zu jeder Regel Ausnahmen, weil stoisches Befolgen manchmal unklug ist.

Ein besonders krasses Beispiel stellen die bisweilen geforderten Kommentare bei automatisch generierten Zugriffsmethoden (`set`/`get`) dar. Tatsächlich sind solche Kommentare eine Verschmutzung des Codes. Sie beeinträchtigen die Lesbarkeit ohne jeden positiven Effekt auf die Verständlichkeit. Das ist eine typische Folge des mechanischen oder erzwungenen Einhaltens von vorgegebenen Regeln.

Versuchen Sie in Ihren Projekten zu erreichen, dass sinnlose Regeln gelöscht werden. Vergessen Sie dann aber, wenn beispielsweise auf den Kommentarzwang verzichtet wird, nicht, an allen Stellen, an denen das notwendig ist, wie bei öffentlichen Schnittstellen, verständliche Kommentare zu schreiben.

Clean Code ist auch eine Sache des Vertrauens in die Entwickler. Rechtfertigen Sie dieses Vertrauen!

Einige andere Regeln, die zu hinterfragen sich lohnt, sind die folgenden:

- »Klassen, die nur statische Methoden enthalten, sogenannte *Utility-Klassen*, müssen einen `private` Konstruktor haben.«

  Mir ist in über zwanzig Jahren kein einziger Fall bekannt geworden, in dem diese Regel einen Fehler verhindert hätte oder jemand missbräuchlich ein entsprechendes Objekt erzeugt hat. Es wird aber mindestens eine überflüssige Zeile Code erzeugt.

- »Leere Standardkonstruktoren müssen einen Kommentar enthalten.«

  Diese Regel verursacht ebenfalls nur überflüssigen Code. Ein solcher leerer Konstruktor sollte ohnehin nur dann auftauchen, wenn das notwendig ist, weil weitere Konstruktoren existieren.

Schwierig wird es für Sie, wenn Sie in eine größere Organisationseinheit eingebunden sind, die ihren Entwicklern nur wenig Freiheit lässt. Dann bleibt Ihnen nur übrig, sich in das Unvermeidliche zu fügen und auf eine bessere Gelegenheit zu hoffen. Trotzdem, etwas Freiraum werden Sie immer haben. Nutzen Sie diesen!

Starre Abläufe

Normalerweise wird Clean Code im Zusammenspiel mit agilen Entwicklungsmethoden gesehen. Das passt gut zusammen, ist aber nicht zwingend.

Insbesondere in kleineren nicht agilen Projekten findet sich häufig ein liberales Klima, in welchem in gewissen Umfang Code-Überarbeitungen toleriert werden. Wahrscheinlich werden Sie dort nicht unbedingt das Niveau erreichen können, das Sie sich wünschen. Aber ein Schritt in die richtige Richtung ist besser als keiner.

Schließlich ist es leider nicht auszuschließen, dass das Projekt, für das Sie arbeiten, derart starren Vorgaben folgt, dass Sie nur sehr wenige der empfohlenen Praktiken umsetzen können. Entsprechend gering ist der zu erwartende Nutzen. Sie können eigentlich nur noch versuchen, wenigstens »im Kleinen« das eine oder andere Konzept anzuwenden.

Falsche Autoritäten

Ich hätte diesen Abschnitt auch »Vorurteile« oder »Volkstümliche Irrtümer« nennen können. Worum es mir geht, ist die Tatsache, dass es eine ganze Reihe von wenig vorteilhaften Mustern gibt, die unbesehen übernommen werden.

Nach meiner Erfahrung können Sie schnell in den Ruf des Außenseiters kommen, wenn Sie versuchen, solche Gewohnheiten infrage zu stellen. Aus dieser Position ist es dann schwierig bis unmöglich, andere Entwickler vom Clean-Code-Konzept zu überzeugen.

Bleiben Sie trotzdem bei der Ansicht, die Sie als richtig erkannt haben. Nur ist es manchmal besser, diese mit Fingerspitzengefühl und bei passender Gelegenheit ins Team zu tragen.

Es folgen zwei Beispiele für weit verbreitete Vorurteile.

Singleton

Singletons sind Klassen, von denen nur ein einziges Objekt instanziiert wird. Tatsächlich handelt es sich dabei um die alten, mit Recht kritisierten globalen Variablen im neuen objektorientierten Gewand. Es gibt eine Reihe von guten Gründen, sie nicht zu verwenden:

- ✔ Sie verstecken Abhängigkeiten im Code.
- ✔ Sie führen dazu, dass die Teile des Codes unnötig eng miteinander verbunden sind, und erschweren dadurch das Testen.
- ✔ Der Fakt, dass es nur eine Instanz gibt oder geben darf, sollte als Implementationsdetail verborgen bleiben.
- ✔ Die Beschränkung auf eine Instanz gilt – was leicht übersehen werden kann – nicht generell, sondern in Java beispielsweise bloß pro `ClassLoader`-Instanz.
- ✔ Im Allgemeinen bedarf es einer aufwendigen Änderung, wenn doch einmal mehrere Objekte benötigt werden.

Warum findet man Singletons dann trotzdem so häufig? Darauf gibt es eine einfache Antwort: Das Singleton ist ein prominentes Exemplar aus der bekanntesten Liste der *Entwurfsmuster* und dadurch praktisch über jede Kritik erhaben.

Entwurfsmuster

Entwurfsmuster oder *Design Patterns* sind Vorlagen für die Lösung von Entwurfsproblemen, die in der objektorientierten Programmierung häufig anzutreffen sind. Richtig bekannt wurde das Konzept durch das 1995 erschienene gleichnamige Buch von E. Gamma, R. Helm, R. Johnson und J. Vlissides. Die Autoren werden oft als *Gang of Four* (GoF) apostrophiert. Sie unterscheiden zwischen

- ✔ Erzeugungsmustern, dazu gehören unter anderem das *Singleton* und die *Abstrakte Fabrik*,
- ✔ Strukturierungsmuster, zum Beispiel *Fassade* und *Fliegengewicht*, sowie
- ✔ Verhaltensmuster, zum Beispiel *Beobachter* und *Besucher*.

Im Original-Buch werden die Muster anhand von Beispielen in C++ und Smalltalk diskutiert. Sie lassen sich jedoch problemlos auf andere objektorientierte Programmiersprachen übertragen. Eine gemeinhin übersehene Ausnahme stellt dabei allerdings das Singleton dar, weil dessen essenzielle Eigenschaft, nämlich nur in einem Exemplar zu existieren, in Programmiersprachen wie Java, die Klassen dynamisch und mehrfach laden können, nicht mehr gewährleistet ist.

Datensätze

Wenn Sie einen durchschnittlichen Entwickler fragen, ob es in Java eine Möglichkeit gibt, reine Datensätze, die keine Objekte sind, darzustellen, so wird die Antwort höchstwahrscheinlich nein sein. Rein formal ist das richtig.

Allerdings findet sich bereits in den ersten Versionen der *Java Language Specification* der Hinweis, dass es angebracht ist, in den Fällen, in denen eine Klasse lediglich als reine Datenstruktur verwendet wird, die Instanzfelder öffentlich zu machen und auf Zugriffsmethoden zu verzichten.

In der Praxis wird diese einfache Möglichkeit, zwischen Objekten und Datensätzen zu unterscheiden, leider fast vollständig ignoriert. Ich behandle dieses Thema in Kapitel 15 *Objekte und Datensätze* ausführlich.

Es liegt an Ihnen

Erinnern Sie sich, warum ich Ihnen das Clean-Code-Konzept nahebringen will? Es geht nicht darum, dass Sie einen Satz von strikt zu befolgenden Regeln lernen und beherzigen.

Sie sollen Code schreiben, der gut lesbar und verständlich ist und dadurch stabil weiterentwickelt werden kann. Wie sich das unter Ihren speziellen Bedingungen am besten bewerkstelligen lässt, müssen Sie entscheiden.

Übernehmen Sie Regeln und Vorlagen nicht ungeprüft. Das bedeutet natürlich nicht, dass Sie die Erfahrung derjenigen, die solche Vorgaben formuliert haben, ignorieren sollen.

Aber jede Regel ist an Voraussetzungen und Ziele gebunden. Versuchen Sie, diese zu verstehen. Nur wenn Sie eine genügende Übereinstimmung mit den Voraussetzungen und Zielen Ihres Projekts finden, passt die Regel.

Andernfalls treffen Sie Ihre eigene begründete Entscheidung und stehen Sie dazu!

Das Wichtigste in Kürze

- ✔ Clean Code ist ein Konzept der Professionalität. Der Umsetzung stehen allerdings bisweilen externe Schwierigkeiten entgegen.
- ✔ Kurzfristig entstehen vor allem erhöhte Kosten, deren Nutzen nur indirekt, etwa in Form höherer Arbeitszufriedenheit, zum Ausdruck kommt.
- ✔ Die wichtigsten Vorteile von Clean Code ergeben sich bei der langfristigen Weiterentwicklung der Software.
- ✔ Sauberer Code ist nicht automatisch einfach änderbar. Aber die erhöhte Verständlichkeit erleichtert gute Änderungen.

- ✔ Äußere Umstände wie Vorgaben oder Frameworks können das Schreiben sauberen Codes mehr oder weniger erschweren.
- ✔ Allgemein übliche Regeln und Muster sollten immer dahingehend überprüft werden, ob sie wirklich helfen, lesbaren und verständlichen Code zu erzeugen.

Teil II

An Herausforderungen wachsen

IN DIESEM TEIL …

- ✔ Lernen Sie das Wesen des Programmierens kennen
- ✔ Erfahren Sie, warum Softwareentwicklung so schwer ist
- ✔ Erhalten Sie einige Ratschläge, um die beschriebenen Probleme mit den richtigen Methoden anzugehen
- ✔ Werden Sie vor häufig anzutreffenden Hindernissen gewarnt

Kapitel 6

Mehr als Handwerkskunst

IN DIESEM KAPITEL

- Schwierigkeiten beim Schreiben sauberen Codes erkennen und akzeptieren
- Voraussetzungen professioneller Softwareentwicklung
- Handwerkliche Grundlagen sauberen Entwickelns und deren Grenzen

> Halte dir einen tüchtigen Feind! Er wird dir ein Sporn sein, dich zu tummeln. (K. F. Gutzkow)

In diesem Kapitel stehen die handwerklichen Fähigkeiten, die für das Schreiben sauberen Codes gebraucht werden, im Mittelpunkt. Sie erfahren, warum diese Fähigkeiten für eine professionelle Softwareentwicklung einerseits unverzichtbar sind, andererseits allein noch nicht ausreichen.

Programmieren ist schwer

Ich habe bereits in Kapitel 5 *In der Praxis: Stolpersteine* eingestanden, dass sauberer Code schwer zu erreichen ist. Das liegt weniger am Clean-Code-Konzept, sondern daran, dass Programmieren an sich schon schwierig ist, und guten Code zu schreiben, ist noch eine Stufe schwieriger. Aber lassen Sie sich dadurch bitte auf keinen Fall entmutigen.

Aus der Tatsache, dass Sie dieses Buch lesen, schließe ich, dass Sie sich intensiv mit Code auseinandersetzen und wahrscheinlich, beruflich oder als Hobby, Code schreiben. Die Schwierigkeiten,

um die es hier geht, müssen Sie daher größtenteils ohnehin bewältigen. Und Sie werden Sie besser bewältigen können, wenn Sie deren Ursachen richtig verstehen.

Aus diesem Grund ist das Lesen dieses Teils für Sie (hoffentlich) auch dann ein Gewinn, wenn Sie am Ende kein Clean-Code-Entwickler sein werden oder sein wollen. Um jedoch Clean Code praktisch erreichen zu können, sind diese Erläuterungen wichtig.

Weil Programmieren eben nun einmal schwer ist, gibt es seit Langem vielfältige Bemühungen, alles einfacher zu machen. Das hat mit Assemblern angefangen, die die lästige Eingabe von Zahlencodes überflüssig machten, und reicht über die bekannten Programmiersprachen bis zu hochspezialisierten und anwendungsgebietsspezifischen Fachsprachen. Parallel dazu wurden Entwicklungsmethoden und Programmierparadigmen ersonnen.

Alles mit dem Ziel und dem Versprechen, die Probleme der Softwareentwicklung zu lösen. Zu einem gewissen Grad ist das sogar gelungen. In der Folge können heute deutlich größere und kompliziertere Aufgaben bewältigt werden. Trotzdem gibt es immer noch eine viel zu große Zahl von Projekten, die ganz oder teilweise scheitern oder zumindest spürbar länger dauern und erheblich höhere Kosten verursachen als geplant.

Nach meiner festen Überzeugung werden gute und nützliche Methoden nicht oder nicht konsequent genug angewandt, weil ihre Propagierung zu oberflächlich erfolgt.

Aus nachvollziehbaren Gründen werden neue Methoden gern an allseits bekannten Beispielen demonstriert. Meistens ist das durchaus beeindruckend und überzeugend. Aber dann, und das ist auch mir schon so passiert, kommen Sie zu Ihrer täglichen Arbeit zurück, versuchen voller Elan die neuen Erkenntnisse umzusetzen – und scheitern.

Das ist frustrierend und führt schnell dazu, dass Sie die neuen Methoden als nicht praktikabel einstufen und vergessen. So sollte es jedoch nicht sein.

Häufig werden neue Verfahren bei der Vorstellung zu stark »verkauft«. Im Vordergrund stehen die Vorzüge, die es scheinbar erlauben, fast alle Probleme zu lösen. Nachteile und insbesondere die Randbedingungen für die sinnvolle Anwendung werden unterschlagen.

Das tiefere Wesen der Softwareentwicklung wird viel zu selten betrachtet. Sie können aber nur dann sauberen und verständlichen Code produzieren, wenn Sie nicht an der Oberfläche verharren.

Nicht an der Oberfläche verharren heißt zuerst einmal, dass Sie genauer erkennen müssen, was eigentlich Ihr wichtigstes Problem ist. Denn sehr oft sehen Sie sich einem ganzen Wust von miteinander verbundenen Schwierigkeiten gegenüber, die die zentralen Fragen ganz oder teilweise verbergen.

In diesem Teil finden Sie einige grundlegende Schwierigkeiten erklärt. Das kann Ihnen helfen, dasjenige (Kern-)Problem zu identifizieren, das Ihnen die meisten Schmerzen verursacht und auf dessen Lösung Sie sich daher zuerst konzentrieren sollten.

Den Grund dafür, dass ich so ausführlich auf Probleme (»Feinde«) eingehe, möchte ich Ihnen anhand eines Beispiels aus dem Alltag erläutern.

Stellen Sie sich vor, Sie machen eine Tour und stehen plötzlich vor der Notwendigkeit, eine Entscheidung über das weitere Vorgehen zu treffen. Natürlich können Sie ein Handbuch zurate ziehen, das nützliche Regeln enthält wie »Machen Sie eine Pause!«, »Achten Sie auf den Weg!« und so weiter. Alles sinnvoll und nützlich und anwendbar.

Aber um zu entscheiden, was Ihnen in der konkreten Situation wirklich weiterhilft, werden Sie sich zuerst überlegen müssen, was eigentlich die wichtigste zu entscheidende Frage ist. Denn wenn Sie sich verirrt haben, hilft eine Pause allein nicht weiter, wenn Sie erschöpft sind schon.

Es reicht nicht, zu prüfen, ob eine Regel in der konkreten Situation anwendbar ist. Viel wichtiger ist es, zu klären, ob ihre Anwendung helfen kann, eine wirksame Verbesserung zu erreichen.

Beim Softwareentwickeln werden Regeln oft nur deshalb angewendet, weil sie scheinbar passen, ohne genaue Prüfung, ob sie die grundlegende Schwierigkeit wirklich beheben. (Daneben gibt es eine, von Entwicklern verschiedentlich offen eingestandene Schwäche, neue Verfahren »wie ein neues Spielzeug« erst einmal für alles einzusetzen.)

Und ganz nebenbei: Auch außerhalb der Programmierung ist es meist eine gute Idee, sich in vermeintlich überfordernden Situationen die Frage zu stellen: Was ist eigentlich mein größtes Problem?

Software professionell entwickeln

Das Clean-Code-Konzept erklärt das Schreiben sauberen Codes zur Voraussetzung von Professionalität in der Softwareentwicklung. Die Beziehung zwischen diesen beiden Begriffen ist aber wesentlich vielfältiger. Denn mit gleicher Berechtigung kann man behaupten, dass Professionalität Voraussetzung für das Schreiben von Clean Code ist.

Es erscheint mir daher angebracht, einige Facetten professionellen Entwickelns in ihrem Bezug zu sauberem Code genauer anzuschauen. Professionalität umfasst ein ganzes Spektrum von Einstellungen, Werten und Kenntnissen:

- ✔ **Wissen**

  Ein umfassendes Wissen, das über die unmittelbar zur Aufgabenerfüllung notwendigen Kenntnisse hinausreicht und befähigt, das jeweilige Problem in einen größeren

Zusammenhang zu stellen, verschiedene Lösungsvarianten zu finden und bezüglich unterschiedlicher Kriterien zu bewerten.

- ✔ **Lernbereitschaft**

  Die Fähigkeit und der Wille, das eigene Wissen aktuell zu halten und fortlaufend zu erweitern.

- ✔ **Souveränität**

  Die Fähigkeiten, mit Fehlschlägen und plötzlich auftauchenden Schwierigkeiten angemessen umzugehen sowie berechtigte Kritik annehmen und unberechtigte sachlich zurückweisen zu können.

- ✔ **Verlässlichkeit**

  Die Einhaltung von Zusagen und das Befolgen vereinbarter Vorgehensweisen, keine unmotiviert oder sprunghaft wechselnden Meinungen und Entscheidungen.

- ✔ **Beitragsbereitschaft**

  Die Bereitschaft, nach Möglichkeit mehr als nur den unbedingt erforderlichen eigenen Anteil zum Gesamtergebnis beizutragen.

Professionelle Softwareentwicklung erfordert zusätzlich die Betonung einiger spezieller Fähigkeiten:

- ✔ Regeln und informelle Standards, die innerhalb eines Projekts oder eines Anwendungsgebiets üblich sind, müssen beachtet werden. Das schließt die Beteiligung an deren Weiterentwicklung ein.
- ✔ Das Ergebnis, also der Code und die begleitenden Dokumente, muss so gestaltet sein, dass es von einem anderen Entwickler mit angemessenem Aufwand verstanden und weiterbearbeitet werden kann.
- ✔ Nicht die beste Lösung in Bezug auf einen einzelnen Aspekt ist das Ziel, sondern die ausgewogenste bezüglich des Gesamtprojekts.

Auf der Basis dieser Eigenschaften sind Sie als echter Profi in der Lage, Ihre Aufgaben gezielt, fachgerecht und fundiert zu bearbeiten.

Insbesondere an den beiden Attributen *gezielt* und *fachgerecht* hapert es in der Praxis des Öfteren. Wahrscheinlich haben Sie es auch schon erlebt, dass Code »experimentell« bearbeitet wurde, um ein gewünschtes Ergebnis zu erreichen. So etwas ist ebenso wenig ein *gezieltes* Vorgehen wie ausufernde Debugging-Sessions. Allerdings lässt die Qualität des Codes – eine brauchbare Dokumentation gibt es dann gewöhnlich auch nicht – manchmal gar nichts anderes als das schrittweise Durchgehen zu.

Zusammenfassend kann man sagen: Clean Code ist zwar keine unabdingbar notwendige Voraussetzung für Professionalität, aber wenn Sie erfolgreich sauberen Code schreiben, ist das schon mal ein ganz wichtiger Baustein.

Softwareentwicklung braucht Handwerk

Der Vorschlag, Softwareentwicklung als Handwerkskunst zu verstehen, ist nicht neu. Im Kern geht es dabei ebenfalls darum, die Programmierung auf eine professionellere Basis zu stellen. Durch die Konzentration auf agile Entwicklungsmethoden waren die technischen Aspekte des Programmierens in der Praxis zeitweise etwas aus dem Blick geraten.

Der Begriff der Handwerkskunst scheint geeignet dieses Defizit zu beheben. Er bietet überdies den Vorzug, sowohl aus innerer als auch aus äußerer Sicht, ein positives Bild zu liefern.

Aus innerer Sicht hilft er den Entwicklern, sich als hochqualifizierte Fachkräfte zu sehen und entsprechende Eigenschaften verbunden mit einem gesunden Stolz zu entwickeln. Wer sich als Fachkraft fühlt, strebt ganz automatisch

danach, demgemäß akzeptable Ergebnisse zu liefern und seine Fähigkeiten auf dem erforderlichen hohen Niveau zu halten.

Aus äußerer Sicht wird klar, dass man es mit Meistern ihres Fachs zu tun hat, die man entsprechend wertschätzen muss, von denen man jedoch auch qualitativ hochwertige Erzeugnisse erwarten kann. Im Ergebnis ist das eine produktive Wechselwirkung, die einen ungemein positiven Effekt auf die Softwareentwicklung hat.

Software Craftsmanship

Die Idee entstand als Gegenentwurf zum unselbstständigen Entwicklungsingenieur oder zum besessenen Hobbyprogrammierer (auch *Nerd*). Sie stammt ursprünglich aus den angelsächsischen Ländern und ist nicht zuletzt dem Umstand geschuldet, dass für eine Tätigkeit als Programmierer oft keine formalen Voraussetzungen zu erfüllen waren oder noch sind. Berufsbezeichnungen wie »Software Craftsman« oder »Software Crafter« sollen ein höheres berufliches Fertigkeitsniveau und ein dementsprechendes Ethos sichtbar machen.

Außerdem sind die Produkte der Softwareentwicklung, wie bei handwerklicher Fertigung auch, immer mehr oder weniger spezialisierte Einzelstücke, die allerdings bei Bedarf beliebig oft kopiert werden können.

Bedauerlicherweise ist es ja so, dass in der Ausbildung von Informatikern und verwandten Berufen die praktischen Fertigkeiten, also das »Handwerk des Entwickelns«, eine völlig unzureichende Rolle spielen. Nach der Einführung in eine Programmiersprache gibt es zwar gewöhnlich noch das eine oder andere Praktikum oder kleine Projekte. Der Schwerpunkt liegt dabei aber auf der Erfüllung der funktionalen Anforderungen, manchmal auch noch auf der Entwicklungsmethodik.

Das Wichtigste, was einen guten Entwickler ausmacht, müssen Neulinge mühsam »on the job« lernen. Wie gut das gelingt, hängt stark davon ab, welche Entwicklungskultur in der Umgebung gelebt wird, in der sie starten.

Die vielen Kleinigkeiten, die gutes Handwerk ausmachen, finden generell viel zu wenig Aufmerksamkeit. Möglicherweise

empfinden Sie das ähnlich und interessieren sich gerade deshalb für das Thema Clean Code.

Die Analogie zur Handwerkskunst müssen Sie natürlich mit Vorsicht betrachten. Wenn ein Tischler massenweise gleichartige Stühle herstellt, kann man eigentlich nicht mehr von Einzelstücken reden. Den Software-Handwerker möchte ich eher mit einem Orgelbauer vergleichen. Dieser Beruf verlangt ebenfalls hohe und vielfältige Qualifikationen.

Jede Orgel muss den Wünschen des Auftraggebers entsprechen. Sie muss dem verfügbaren Platz und den akustischen Verhältnissen des jeweiligen Raums angepasst werden und soll auch noch gut aussehen.

Gerade weil so viele verschiedene Anforderungen und Wünsche gegeneinander abgewogen werden müssen, erscheint mir diese Analogie treffend. Denn das Produkt, also die Orgel, findet nur dann Anerkennung, wenn wirklich alles zueinanderpasst. Dieses Ziel, dass alles passt, sollten Sie als Softwareentwickler ebenfalls hochhalten.

Handwerk allein reicht nicht

Nachdem ich Ihnen erklärt habe, warum der Aspekt der Handwerkskunst so wichtig und fruchtbar für die Verbesserung der Softwareentwicklung ist, muss ich jetzt auf die Beschränkungen zu sprechen kommen. Um diese Grenzen zu illustrieren, brauche ich das Bild von der Handwerkskunst gar nicht zu verlassen.

Handwerker haben gewöhnlich ein recht genau umrissenes Sortiment von Produkten. Sie wissen, wie jedes einzelne davon herzustellen ist und was es voraussichtlich kosten wird. Sonderwünsche können in beschränktem Umfang berücksichtigt werden, oder auch nicht.

Der große Unterschied zur Softwareentwicklung besteht darin, dass bei vielen Software-Unternehmungen im Voraus nicht ausreichend klar ist, was zum Schluss herauskommen soll. Der Kunde hat nur vage Vorstellungen von dem, was er bestellt.

Falls die Anforderungen wirklich hinreichend exakt beschrieben sind und Sie bereits Erfahrungen mit ähnlichen Aufgaben sammeln konnten, werden Sie als geübter Software-Handwerker den Auftrag ohne größere Probleme erledigen können. Anders sieht es aus, wenn die Spezifikation schwammig formuliert wurde und dann während der Projektlaufzeit ständig Änderungs- und Ergänzungswünsche dazukommen.

Das kann die Folge einer nachlässigen Vorbereitung sein. Sehr oft liegt es jedoch daran, dass den Auftraggebern die Komplexität der betreffenden Fachdomäne nicht ausreichend bewusst ist. In solcher Lage reicht handwerkliche Perfektion nicht aus, weil notwendige Erkenntnisse erst noch gewonnen werden müssen.

Die bisweilen zu hörende flapsige Beschreibung solcher Situationen als »Jugend forscht« enthält mehr Wahrheit, als auf den ersten Blick zu vermuten ist. Aber das ist das Thema des Kapitels 7 *Entwickeln als kreative Wissenschaft*.

Festzuhalten bleibt, dass handwerkliche Fähigkeiten einen sehr wichtigen Beitrag zum Entwickeln hochwertiger Software leisten, bei großen und komplexen Aufgabenstellungen jedoch bei Weitem nicht ausreichen.

Ein beeindruckendes Beispiel dafür, dass handwerkliche Perfektion nicht hinreichend ist, um ein Desaster zuverlässig zu verhindern, kann man im Stockholmer Vasa-Museum besichtigen. Am Geschick und den exzellenten Fähigkeiten der Schiffszimmerleute lässt das dort ausgestellte, 1961 gehobene Schiffswrack noch heute keine Zweifel aufkommen. Trotzdem war das seinerzeit modernste Kriegsschiff der schwedischen Marine bei seiner Jungfernfahrt 1628 schon im Hafen gekentert und gesunken.

Die bereits damals ermittelten Ursachen der Katastrophe lesen sich wie die Beschreibung eines gescheiterten Softwareprojekts: Zeitdruck, viele Änderungen, kein dokumentierter Projektplan, Wechsel in der Projektleitung, übermäßige Innovation und so weiter.

Ganz besonders wichtig war der Umstand, dass die notwendigen *wissenschaftlichen* Methoden fehlten. Man kannte zu jener Zeit keine Wege, den Schwerpunkt, die Festigkeit und die Stabilität einer Schiffskonstruktion zu berechnen.

Das Wichtigste in Kürze

- ✔ Programmieren ist schwer und erfordert Meisterschaft.
- ✔ Um Probleme erfolgreich bewältigen zu können, müssen diese erst einmal klar erkannt und benannt werden.
- ✔ Professionelle Softwareentwicklung setzt einen soliden Grundstock an Kenntnissen und Fertigkeiten voraus, der durch ein entsprechendes Berufsethos ergänzt werden muss.
- ✔ Clean Code ist ein Weg, den Ansprüchen professionaler Softwareentwicklung zu genügen.
- ✔ Es fehlt in der Ausbildung weithin an der Vermittlung der grundlegenden Techniken für das Schreiben sauberen oder zumindest verstehbaren Codes.
- ✔ Obgleich die handwerklichen Fertigkeiten beim Entwickeln sauberen Codes nicht unterschätzt werden dürfen, sind sie, insbesondere bei komplexeren Projekten, nicht ausreichend.

Kapitel 7

Entwickeln ist (kreative) Wissenschaft

IN DIESEM KAPITEL

- Software ist formalisiertes Wissen
- Die Theorie hinter der Software
- Wege zur Theorie
- Konsequenzen für den Entwicklungsprozess

Sie wissen bereits, dass professionelle Softwareentwicklung solide handwerkliche Fähigkeiten erfordert und dass diese allein jedoch nicht ausreichen. In diesem Kapitel versuche ich, Ihnen eine umfassendere Sicht auf den Prozess des Entwickelns zu geben, bei dem das Schreiben von Code nur noch der letzte Schritt ist. Es folgen einige interessante Erklärungen für häufig beobachtbare Probleme und nützliche Schlussfolgerungen, die man aus der dargestellten Sichtweise ziehen kann.

Formalisiertes Wissen

Programme sind formalisierte Theorien. Alle Methoden und Regeln zur Lösung der unübersehbaren Probleme in der Softwareentwicklung, die diese Tatsache ignorieren, haben bisher nur zu wenig gebracht. Softwareentwicklung braucht zwar eine solide handwerkliche Basis. Das ist unbestreitbar. Allein, es reicht nicht.

Das liegt daran, dass reine Handlungsanweisungen zwar durchaus nützlich und hilfreich sind, dass sie aber grundlegend zu

kurz greifen. Sie dringen nicht zum Kern des Problems vor. (Das gilt übrigens auch für das Clean-Code-Konzept, wenn es nur oberflächlich betrachtet wird.)

Damit Sie wirklich guten Code produzieren und dessen Qualität langfristig erhalten können, müssen Sie das Wesen der Softwareentwicklung verstanden haben. Denn die Programmierung beginnt, lange bevor Sie die erste Zeile Code schreiben, mit dem Verstehen der Aufgabe.

Zuerst müssen Sie ausreichende Kenntnisse über die Anwendungsdomäne erworben haben, um die Intentionen des Auftraggebers nachvollziehen zu können. Das so erarbeitete Wissen muss die Basis für die weitere Arbeit bilden. Wenn Sie so vorgehen, ist Clean Code ein sehr effektives Werkzeug.

Was also ist nun das Wesen der Programmierung? Kurz gesagt ist es nicht mehr und nicht weniger als das Entwickeln einer formalen Theorie für bestimmte Sachverhalte der materiellen oder ideellen Realität.

Mit *Realität* meine ich dabei einfach die Tatsache, dass etwas unabhängig von der betrachteten Software existiert. Falls Ihnen diese Beschreibung zu abstrakt oder nebulös erscheint, kann ich Sie beruhigen: Im Verlauf dieses Kapitels werde ich diese Auslegung noch gründlich erläutern.

Um Ihnen den Gedankengang schon hier etwas verständlicher zu machen, lassen Sie mich die Softwareentwicklung mit dem Schreiben eines Fachbuchs oder einer Abschlussarbeit vergleichen. Ein Leitfaden, wie Sie den Text und eventuelle Formeln übersichtlich und verständlich strukturieren und Illustrationen anordnen können, entspräche dann gerade dem Clean-Code-Konzept.

Aber alle Regeln werden Ihnen wenig helfen, wenn Sie es nicht zuvor geschafft haben, Ihr Anliegen in klare Gedanken zu fassen. Und um dieses, ein Vorhaben oder ein Verfahren *in klare Gedanken fassen*, geht es.

Es sollte Sie nicht überraschen, dass es auf dem Weg zu sauberem Code immer wieder um die Herausarbeitung des jeweiligen Kerns geht. Sauberer Code kann nur auf der Grundlage klarer Konzepte und klarer Gedanken entstehen. Deshalb ist Klarheit so wichtig.

Was sind formale Theorien?

Formale Theorien sind Gebilde aus Elementen oder Symbolen, Voraussetzungen oder Axiomen und Regeln, die es erlauben, durch rein mechanisches Vorgehen, bestimmte Konstellationen von Elementen, oft Ausdrücke genannt, in andere Konstellationen umzuformen. Interessant wird das Ganze dann, wenn man die Elemente und Ausdrücke etwas Realem zuordnen kann, was verblüffend oft gelingt.

Die bekannteste formale Theorie ist die Mathematik. Es gibt viele andere. Formale Theorien werden auch symbolische Systeme genannt, weil bei ihnen anstelle realer Objekte nur Symbole manipuliert werden. Computer führen nichts anderes als solche symbolischen Operationen aus.

Jedes Programm ist somit ein symbolisches System oder eben eine formale Theorie. Wenn ein Programm ohne Fehler läuft, ist die implementierte Theorie in sich hinreichend konsistent oder zumindest widerspruchsfrei. Das ist für sich genommen schon mal nicht schlecht.

Die viel interessantere Frage lautet allerdings: Ist die formale Theorie wirklich ein Bild der betrachteten Realität? So gesehen sind semantisch fehlerhafte Programme einfach nicht passende formale Theorien. Diese Erkenntnis allein hilft Ihnen im Moment zwar nicht weiter, aber sie erlaubt es, einige grundlegende Schwierigkeiten, mit denen die Softwareentwicklung zu tun hat, genauer zu beschreiben.

Softwareentwicklung als Theorienbildung

Die Idee, Programme als formale Theorien und damit die Softwareentwicklung als Theorienbildung zu interpretieren, wurde bereits 1985 von dem

scharfsinnigen dänischen Informatiker Peter Naur publiziert. Obwohl sie viele offensichtliche Probleme der Softwareentwicklung überzeugend erklären kann, wurde sie leider nur wenig beachtet. Über die Gründe kann man bloß spekulieren. Möglicherweise liegt es daran, dass dieses Konzept in keine Schublade zu passen scheint und sich keine Werkzeuge und Methoden unmittelbar daraus ableiten lassen. Immerhin glimmt gelegentlich ein Funke davon wieder auf, wenn zum Beispiel Software richtigerweise als »Codified Knowledge« bezeichnet wird.

In Zeiten heftiger Diskussionen um das Potenzial künstlicher Intelligenz liefert dieser Ansatz überdies die tröstliche Gewissheit, dass für viele Entwicklungsaufgaben der menschliche Intellekt nicht so schnell überflüssig werden wird.

Wann braucht es eine (neue) Theorie?

Bevor ich diesen Punkt genauer betrachte, ist es angebracht, Software-Entwicklungsaufgaben in zwei grundverschiedene Gruppen zu unterteilen:

- ✔ Aufgaben, die einen, in der Regel nur mäßig komplexen, bereits gut verstandenen und weitgehend bekannten Sachverhalt betreffen.
- ✔ Aufgaben, die einen eher diffusen, nicht genau abgrenzbaren oder nicht völlig verstandenen, häufig sehr komplexen Sachverhalt betreffen.

Bei den Aufgaben der ersten Gruppe handelt es sich eher um Routineaufgaben, für die mitunter sogar schon Generatorprogramme oder Frameworks existieren. Das schließt nicht aus, dass Teilprobleme enthalten sein können, die der zweiten Gruppe zuzuordnen sind.

In der Praxis lassen sich die beiden Gruppen ohnehin nicht klar gegeneinander abgrenzen. Schließlich spielen bei der Bewertung dessen, was gut verstanden ist, persönliche Erfahrungen eine große Rolle. In der Kategorie der gut verstandenen Probleme

kommen Sie allein mit handwerklichen Fertigkeiten und Clean Code schon sehr weit.

Die interessanteren und herausfordernden Aufgaben gehören meistens zur zweiten Gruppe. Sie sind dadurch gekennzeichnet, dass ein umfassendes und kohärentes gedankliches Modell fehlt. Ein solches gedankliches Modell kann man auch Theorie nennen.

So gesehen unterscheiden sich die beiden Gruppen dadurch, dass bei der ersten eine Theorie bereits vorhanden ist, während bei der zweiten eine Theorie noch gefunden werden muss. Denn um eine Theorie formalisieren zu können, müssen Sie diese ja überhaupt erst einmal haben.

Wie Sie zu einer Theorie kommen?

Theoriebildung hört sich sehr akademisch an, ist aber in diesem Fall völlig praxisbezogen – was allerdings nicht bedeutet, dass es immer einfach wäre. Natürlich gebe ich Ihnen auch dazu Ratschläge.

Mentales Modell als Theorie

Eine wohlbekannte Theorie zu formalisieren, was in diesem Zusammenhang nur eine andere Formulierung für das Codeschreiben ist, kann bereits schwierig genug sein, weil sich häufig herausstellt, dass Ihr mentales Modell, also eben die Theorie, doch nicht so vollständig ist wie nötig. Trotzdem ist die Ausgangslage erheblich besser, als wenn Sie die Theorie erst noch finden müssen.

Denn das Entwickeln von Theorien ist wissenschaftliche Tätigkeit, und dafür gibt es keine zuverlässig zum Ziel führenden Methoden. Glücklicherweise sind Sie dem dennoch nicht völlig hilflos ausgeliefert. Im Lauf der Jahre wurden bewährte Vorgehensweisen und Heuristiken zusammengetragen.

An Universitäten werden derartige Sammlungen manchmal als »Leitfaden für wissenschaftliches Arbeiten« an Studenten verteilt. Und Sie selbst machen das im Alltag – ganz unbewusst – ständig. Jedes Mal, wenn Sie »verstehen wollen, wie das funktioniert«, sind Sie auf der Suche nach der passenden Theorie.

Diese Theorie muss nicht in einem abstrakten Sinn umfassend und vollständig sein. Sie sollte lediglich den konkreten Erfordernissen genügen. Der Hauptzweck ist, dass sie Ihnen hilft, die Folgen Ihrer Handlungen genügend zuverlässig vorauszusagen.

Im Alltag benutzen Sie und ich ständig unbewusst derartige Theorien. Sie haben beispielsweise eine Theorie, mit deren Hilfe Sie vorhersagen können, was passiert, wenn Sie in eine Pfütze springen: Es spritzt und Sie bekommen nasse Füße.

Ein Beispiel dafür, dass diese Vorstellungen auch sehr unterschiedlich ausgeprägt sein können, liefert das Auto. Wenn Sie etwas von Motoren und Elektronik verstehen, werden Sie eine viel detailliertere Theorie haben, als wenn das nicht der Fall ist. Um ein Auto benutzen zu können, reicht bereits eine relativ einfache Theorie aus – wenn Sie einen Fehler beheben wollen, brauchen Sie eine tiefergehende.

Um nun ein mentales Modell aufbauen zu können, müssen Sie zunächst Kenntnisse über den interessierenden Realitätsausschnitt erwerben. Das kann passiv durch Beobachtung und Befragungen erfolgen. Schneller, aber aufwendiger und nicht immer möglich, lernen Sie durch Experimente.

Wie weit Sie auf dem Weg zur Theorie gekommen sind, können Sie überprüfen, indem Sie versuchen, Eigenschaften oder Ergebnisse vorherzusagen. Wenn Ihnen das ohne wesentliche Fehler gelingt, sind Sie am Ziel. Der Weg dahin kann allerdings weit sein und Frustrationen bereithalten. Lassen Sie sich davon nicht unterkriegen und streben Sie nicht nach übertriebener Perfektion.

-  Vergessen Sie nicht, eine Theorie zu entwickeln, ist wissenschaftliche Forschung. In der Wissenschaft gibt es keine Schnellstraßen zum Ziel. Ausdauer und Kreativität sind gefragt.
- Diskussionen und Meinungsstreit sind unverzichtbare Quellen der Erkenntnis.
- Analysieren Sie Fehler gründlich. Sie liefern Ihnen nützlichere Informationen als Erfolge.

Wenn es so einfach wäre: Viele Hürden

Möglicherweise ahnen Sie es schon, ganz sicher werden Sie es noch erleben: In der Praxis ist das Ganze noch komplizierter und vor allem unübersichtlicher als bisher dargestellt. Die Ursachen der wichtigsten Schwierigkeiten lassen sich so beschreiben:

- **Ungenügende Abgrenzung**

  Sie bemühen sich, etwas zu verstehen, wissen aber nicht genau, was zu diesem Etwas dazugehört und was nicht. Manchmal sind die Aussagen, die Sie erhalten, auch einfach falsch, weil beispielsweise wesentliche Voraussetzungen vergessen wurden und Ihnen dadurch Grenzen unbekannt bleiben.
- **Fehlen wichtiger Informationen**

  Eigentlich versteht niemand das betroffene System korrekt: »Es funktioniert irgendwie.« So unwahrscheinlich das klingt, gerade die Einsicht, dass man die Steuerbarkeit verloren hat, ist oft der Auslöser für eine Software-Neuentwicklung. Überflüssig zu sagen, dass solche Projekte ein enormes Risiko bergen.
- **Widersprüchliche Sichten**

  Besonders bei großen »historisch gewachsenen« Organisationssystemen werden Sie auf viele unterschiedliche

Sichten treffen. Diese hängen davon ab, auf welchen Teil des Systems sie sich beziehen und wann der Betreffende begonnen hat, sich seine Vorstellung von dem Ganzen zu erarbeiten.

- **Informelle Kanäle**

  Unübersichtliche Organisationen funktionieren oft deshalb noch überraschend gut, weil es ein ganzes Geflecht von informellen Kanälen gibt. (Man weiß, wer einem in bestimmten Situationen weiterhelfen kann, weil man sich bei einem Lehrgang oder einer Feier kennengelernt hat.) Das kann für Sie sehr irritierend sein, wenn Sie sich beispielsweise wundern, wieso etwas, entgegen aller Erwartung, trotzdem läuft.

- **Nichtstoffliche Systeme**

  Ein großer Teil der Softwareentwicklung betrifft informationelle Systeme, das heißt Systeme, die vor allem in den Köpfen von Menschen existieren und auf Informationsaustausch beruhen. Sie als Entwickler oder Analyst können daher nichts wirklich selbst in die Hand nehmen und untersuchen. Alle Informationen müssen durch Menschen über fehleranfällige Kommunikation vermittelt werden.

Um das Maß vollzumachen, kommt dazu, dass Ihnen, um all das zu analysieren, in der Regel viel zu wenig Zeit zur Verfügung steht.

-  Akzeptieren Sie, dass Sie sich zunächst nur grob annähern können werden.
- Bereiten Sie sich mental auch darauf vor, dass Ihre Bemühungen scheitern können und Sie noch einmal von vorn beginnen müssen.
- Versuchen Sie – das ist das Schwierigste – Ihre Auftraggeber von der Notwendigkeit eines ausreichenden Verständnisses

der Aufgabe als Voraussetzung für den Projekterfolg zu überzeugen.

- ✔ Richtiges Abwägen aller Anforderungen ist der Schlüssel zum Erfolg.

Auch hier gilt, dass es mir nicht darum geht, ein Schreckensszenario aufzubauen. Vielmehr möchte ich Sie dafür empfänglich machen, die verschiedenen Problemursachen zu unterscheiden und in der Folge jeweils konzentriert anzugehen.

Und dann auch noch der kleine Rest

Idealerweise wird alles, was Sie für eine Theorie brauchen, in den Anforderungen beschrieben. Doch selbst im unwahrscheinlichen Fall, dass Sie dieses Wunder erleben sollten, fehlt noch eine schwer zu beschreibende Zutat, die nämlich aus den verfügbaren Fakten eine anwendbare Theorie macht.

Um aus einer Menge von Fakten, Voraussetzungen und Regeln eine anwendbare Theorie zu machen, muss dieses Theoriegerüst mit Ihrem bereits vorhandenen Wissen und Ihren Erfahrungen verknüpft werden. Das ist ein überaus komplizierter Prozess, der noch dazu kaum zu fassen ist.

Eine verbreitete Beschreibung spricht davon, »aus totem Wissen anwendungsbereites Wissen zu machen«. Man könnte auch sagen, die neue Theorie muss organisch in Ihr bereits vorhandenes Weltbild eingebaut werden.

Wahrscheinlich sollte ich das besser anhand eines Beispiels erklären. Sicher haben Sie es schon erlebt, dass sich Ihnen der Sinn einer Beschreibung oder einer Formel einfach nicht erschließen wollte, bis Sie ein Ihnen einleuchtendes Beispiel gefunden haben oder jemand Ihnen einen entscheidenden Hinweis gegeben hat. Plötzlich hatten Sie es »verstanden«. Um diesen Sprung vom Kennen zum Verstehen geht es.

> Übrigens ist es gerade die Fähigkeit, den Weg dahin zu verkürzen, die gute Lehrer auszeichnet. Je komplexer eine Theorie ist, desto schwieriger wird es, diese allein aus Beschreibungen zu begreifen, und umso wichtiger wird die Vermittlung des Verständnisses durch Menschen, die den Überblick haben und um typische Missverständnisse und sinnvolle Grundvorstellungen wissen.

Konsequenzen

Aus dem Gesagten ergibt sich ganz klar, dass Softwareentwicklung sehr viel mehr ist, als ordentlichen und sauberen Code zu schreiben. Und Software ist mehr als eine Menge Code und die dazugehörige Dokumentation.

Software ist vor allem auch eine Idee, eine Vorstellung davon, wie ein Teil der Welt funktioniert. Deshalb ist die Vermittlung dieser Idee an neu hinzukommende Entwickler eminent wichtig, wenn ein Programm langfristig am Leben erhalten werden soll.

Sauberer Code ist äußerst hilfreich, wenn es darum geht, die formalen Beschreibungen nachzuvollziehen, in die eine Idee umgesetzt wurde. Allein aus dem Code wird sich aber nur in sehr einfachen Fällen die ursprüngliche Idee rekonstruieren lassen.

Die Bedeutung des Entwicklers darf nicht unterschätzt werden

Wie aufwendig es ist, einem anderen Entwickler die Theorie hinter dem Code zu vermitteln, hängt von verschiedenen Faktoren ab und kann stark variieren. Den größten Einfluss haben bisherige Erfahrungen und eventuell bereits vorhandene Kenntnisse des Anwendungsgebiets.

Sie dürfen darüber hinaus nicht vergessen, dass zwei Menschen nie eine vollständig gleiche Vorstellung von der Welt haben, eine Tatsache, die in der Kommunikationstheorie ausführlich behandelt wird. Um miteinander kommunizieren und somit arbeiten zu können, muss für das betreffende Gebiet ein möglichst großer

Bereich übereinstimmenden Verständnisses erreicht werden. Falls die Teammitglieder schon länger gemeinsam arbeiten, wird sich das gemeinsame Verständnis schneller einstellen, als wenn sie ihre Erfahrungen in gänzlich unterschiedlichen Welten gewonnen haben.

Deshalb ist es wichtig, die Entwicklung sowohl des individuellen als auch des gemeinsamen Verständnisses als Investition zu verstehen. Dementsprechend verkörpern eingearbeitete Teams als Ganzes und jedes ihrer Mitglieder erhebliche Werte, die man zwar nutzen, aber nicht einfach transferieren kann, weil sie als Wissen in den Köpfen stecken.

Es ist wie im Sport: Eine gute Mannschaft aufzubauen, braucht Zeit. Dafür ist das Ergebnis mehr als die Summe seiner Teile. Und genauso schnell, wie Sie eine eingespielte Mannschaft durch unüberlegte Eingriffe zerstören können, geht das mit eingearbeiteten Teams.

Daraus ergeben sich diese Schlussfolgerungen:

- ✔ Ein Entwickler ist keine leicht austauschbare Ressource, sondern Träger von Wissen und daher für die betreffende Aufgabe sehr viel wertvoller als für andere Aufgaben oder als ein anderer Entwickler an seiner Stelle.
- ✔ Eingearbeitete Teams verkörpern durch das gemeinsame Verständnis einen Wert, den es für die Softwarewartung und die Weiterentwicklung zu bewahren gilt, indem zumindest ein Teil des Teams für den Wissenstransfer erhalten bleibt.
- ✔ Die Vermittlung des angehäuften Wissens ist letztlich nur durch direkten Kontakt zwischen den beteiligten Personen möglich. Alle indirekten Varianten, beispielsweise Dokumentationen, sind unzureichend oder unvertretbar aufwendig. Auch bei unmittelbarer Vermittlung darf der Zeitbedarf nicht unterschätzt werden.

Es werden verschiedene Qualifikationen gebraucht

Eine wichtige Folgerung, die sich aus der zu Beginn dieses Kapitels dargestellten Sicht auf die Softwareentwicklung ergibt, lautet: Entwickler brauchen Fähigkeiten zum Aufstellen und Formalisieren von Theorien. Dieser Gesichtspunkt spielt in der Ausbildung keine große Rolle und wird zu selten in der Weiterbildung angesprochen. Eher unterschwellig sind sich viele Auftraggeber aber schon bewusst, dass irgendetwas in dieser Richtung wichtig ist, wenn sie von Entwicklern Fach- oder Domänenwissen erwarten.

Wenn Sie es genauer überdenken, werden Sie unweigerlich zu dem Schluss kommen, dass zum Aufstellen von Theorien, also zum Entwickeln eines tiefergehenden Verständnisses, andere Fähigkeiten benötigt werden als zum Formalisieren. Letzteres ist die Tätigkeit, die jeder Programmierer ohnehin erlernen muss und die zumindest teilweise durch Modellierungsmethoden unterstützt wird.

Anders steht es um das Entwickeln von Theorien. Da es dafür keine zuverlässigen Methoden gibt, sind Begabungen wie Einfallsreichtum und Kreativität gefordert. Diese können jedoch in einem gewissen Spannungsverhältnis zu anderen für die Softwareentwicklung wichtigen Charaktereigenschaften stehen.

Kreative Köpfe sind häufig bei Routineaufgaben weniger konzentriert und zuverlässig. Außerdem neigen sie dazu, Entscheidungen durch neue Vorschläge ständig wieder infrage zu stellen. Das sollte man akzeptieren, ebenso wie den Fakt, dass nicht jeder Entwickler für jede anfallende Teilaufgabe gleich gut geeignet ist.

- Versuchen Sie – falls möglich – auf die Zusammenstellung Ihres Teams so Einfluss zu nehmen, dass kreative und gewissenhafte Potenziale in einem der Aufgabe angemessenen Verhältnis vertreten sind.
- Verteilen Sie die Arbeit so, dass die jeweiligen Fähigkeiten optimal genutzt werden.

Der zentrale Weg, um zu lernen, wie man eine Theorie entwickelt, ist Übung. Es ist leider auch der aufwendigste. Umso sorgfältiger sollten Sie deshalb mit diesem Erfahrungsschatz umgehen, wenn Sie ihn einmal angehäuft haben.

Versuchen Sie, durch Retrospektiven erfolgreicher Projekte Erkenntnisse nachhaltig zu machen, und fördern Sie den Erfahrungsaustausch, indem Sie die Theorienbildung bewusst zum Thema machen.

Daraus ergeben sich diese Schlussfolgerungen:

- Fähigkeiten zur Theorienbildung werden kaum vermittelt. Versuchen Sie, entsprechende Weiterbildungen zu finden.
- Theorienbildung erfordert in höherem Maße als andere Programmiertätigkeiten Kreativität. Die unterschiedlichen Persönlichkeitsprofile sollten bei der Teambildung und der Aufgabenverteilung beachtet werden.
- Der Erfahrungsaustausch über diesen Teil der Softwareentwicklung ist extrem wichtig.

Auch die Theorie muss weiterentwickelt werden

Ich habe es bereits mehrfach gesagt: Das Entwickeln eines inneren Modells zum Verständnis, wie einige Dinge in der uns umgebenden Welt zusammenhängen und funktionieren, ist kein einfacher und geradliniger Prozess.

Modellierung und Theorienbildung basieren auf *Abstraktion*. Die wichtigste Methode zur Erfassung komplizierter Zusammenhänge *abstrahiert* von unwesentlichen Merkmalen, das heißt, sie lässt etwas weg. Was übrig bleibt, ist das Modell.

Das hört sich zunächst einfach an, ist es unglücklicherweise in der Praxis aber nicht. Die Krux ist die Frage: Wie kann man wesentliche von unwesentlichen Merkmalen unterscheiden? Die Antwort hängt entscheidend von der zu lösenden Aufgabe ab und kann sich mit der Zeit verändern. Manchmal sind notwendige Abwandlungen Ausdruck eines vertieften Verständnisses, manchmal Folge geänderter externer Anforderungen.

Was ich damit sagen will, ist, dass Theorien genauso der Veränderung unterliegen wie Software allgemein. Sie müssen ebenso gewartet und überarbeitet werden. Allerdings wird das weniger bewusst wahrgenommen, weil es um Wissen geht, das zum großen Teil in den Köpfen steckt und nicht einfach versioniert werden kann.

Es gibt noch einen anderen Gesichtspunkt zu berücksichtigen. Irgendwann haben Sie endlich ein Modell oder eine Theorie. Aber wie wollen Sie die Richtigkeit überprüfen?

Üblicherweise wird eine Theorie durch Anwendung auf ihren Gegenstand getestet. In der Softwareentwicklung heißt das zumeist durch Implementierung und Anwendung. (Andere, nicht so aufwendige Verfahren, wie die manuelle Simulation, sind weniger beweiskräftig.)

Daraus ergibt sich quasi zwangsläufig die Schlussfolgerung, dass jede Erstimplementierung einer völlig neuen Anwendung ein *Prototyp* ist. Dieser Prototyp beweist im günstigsten Fall die prinzipielle Richtigkeit Ihrer Theorie.

Es wäre jedoch sehr verwegen, zu erwarten, dass Sie bereits in allen Details richtig gelegen haben. Viele Fragen stellen sich erst bei der konkreten Umsetzung, das heißt bei der Formalisierung.

Ganz sicher werden Sie noch einige Iterationen brauchen, bis Ihre Theorie hinreichend rund und in sich schlüssig ist. Den Aufwand

für diesen Reifeprozess sollten Sie nicht unterschätzen. Gerade bei Individualsoftware ist das wegen der damit verbundenen Kosten ein extrem kritischer Punkt.

- Seien Sie skeptisch gegenüber dem »großen Wurf«, das heißt gegenüber der völlig neuen Theorie, die alle bekannten und vorstellbaren Probleme löst. Sie fangen damit wieder bei null an und müssen sich auf viele Iterationen und damit verbundene hohe Kosten einstellen.
- Versuchen Sie, in jede neue Theorie so viel Erfahrung und bewährte Theorien einfließen zu lassen wie möglich. Streben Sie bei Erweiterungen ein evolutionäres Vorgehen an.

Das Wichtigste in Kürze

- Programme sind formalisierte Theorien über abgegrenzte Bereiche der Wirklichkeit.
- Softwareentwicklung setzt das Verständnis des abzubildenden Gegenstandsfelds voraus. Ein Modell des Gegenstandsfelds zu erlangen, ist oft wissenschaftliche Forschungstätigkeit.
- Der Teil des Wissens, der nur in den Köpfen steckt, ist für das Verständnis unverzichtbar. Entwickler und Teams sind deshalb keine einfach austauschbaren Ressourcen.
- Theorien können nicht aus dem Stand entwickelt werden. Ihre iterative Verbesserung erfordert Implementierungen und ist daher kostspielig. Das ist vor allem auch bei kompletten Neuentwicklungen »auf der grünen Wiese« zu bedenken.

Kapitel 8

Modellierungsdilemma und Entscheidungsmüdigkeit

IN DIESEM KAPITEL

Das Modellierungsdilemma erkennen

Lernen, dass Entscheiden ermüdet

Die Welt der Softwareentwicklung kennt nicht ohne Grund eine Unmenge teilweise und ganz gescheiterter Projekte. Es kann vieles schiefgehen. Manche Klippen lassen sich durch gewissenhafte Planung und sorgfältige Steuerung umschiffen.

Es gibt allerdings auch Probleme, denen Sie nicht ausweichen können. Ihre einzige Chance besteht dann darin, diese Probleme zu erkennen, zu akzeptieren und sich darauf einzustellen. Alles andere wäre sinnlose Energieverschwendung. Um zwei dieser Probleme geht es in diesem Kapitel.

Das Modellierungsdilemma

In Kapitel 7 *Entwickeln ist kreative Wissenschaft* habe ich ausführlich erläutert, dass jedes Programm eine formalisierte Theorie darüber ist, wie die Software bestimmte Dinge der Realität beeinflusst oder simuliert.

Dabei habe ich stillschweigend vorausgesetzt, dass eine zutreffende Theorie beziehungsweise ein zutreffendes Modell zumindest prinzipiell existiert. Die Aufgabe besteht dann darin, dass Sie dieses Modell finden oder »entdecken«. Genau

genommen ist die Annahme, dass zu jedem Problem ein geeignetes Modell existiert, jedoch eine starke Vereinfachung.

Was macht ein Modell aus?

Die genaue Unterscheidung zwischen *Modell* und *Theorie* ist nicht einfach, weil es einen großen Überdeckungsbereich gibt. In diesem Kapitel bevorzuge ich die Begriffe Modell und die Modellierung, weil es in erster Linie um das Verständnis einer Situation oder eines Sachverhalts und weniger um die Formalisierung geht.

Die wesentliche Funktion eines Modells ist es, die Komplexität von Erscheinungen auf ein Maß zu reduzieren, das geistig verarbeitbar ist, sodass Sie sich eine Vorstellung davon machen können. Gedankliche Modelle spielen eine zentrale Rolle beim Vorhersehen von Ereignissen, weil sie es ermöglichen, virtuelle Experimente vorzunehmen, die in der Realität so nicht durchführbar sind.

In diesem Sinne ist ein Modell durch folgende drei Eigenschaften gekennzeichnet:

- ✔ Es ist ein Modell »von etwas«, das heißt, es existiert eine Projektion für die Elemente eines Urbildbereichs (Originale) auf die Modelle (Bilder).
- ✔ Ein Modell vereinfacht das Original durch Weglassen von unwesentlichen Eigenschaften.

  Modellbildung ist daher eine Form der Abstraktion. Abstraktion ist die wichtigste Methode, die Menschen haben, um mit Komplexität umzugehen. Man beschränkt sich auf eine möglichst kleine Menge von Merkmalen, die für das betrachtete Problem interessant sind. Diese Auswahl ist oft nicht ganz einfach und manchmal nur mithilfe von Versuchen möglich.
- ✔ Ein Modell braucht immer einen Modellierungszweck. Erst dieser Zweck macht die Unterscheidung zwischen

wesentlichen und unwesentlichen Eigenschaften überhaupt möglich.

Da jede objektive Erscheinung eine immens große Anzahl von Merkmalen hat, können Sie im Prinzip eine unübersehbare Menge unterschiedlicher Modelle gewinnen. Welche davon sinnvoll sind, hängt allein am Zweck der Modellierung. Diesen Zweck möglichst genau zu erfassen, ist Ihre erste wichtige Aufgabe.

Machen Sie sich als Erstes, noch bevor Sie mit der eigentlichen Modellierung beginnen, klar, was Ihre Ziele sind.

Welche Fragen wollen Sie mithilfe des Modells beantworten beziehungsweise welches Verhalten simulieren?

Darüber hinaus sollte ein Modell in sich weitgehend widerspruchsfrei oder konsistent sein. Das muss zumindest für den im konkreten Fall betrachteten Ausschnitt gelten.

Ein Modell für alle Anforderungen gesucht

Allen Menschen recht getan ist eine Kunst, die niemand kann. (Sprichwort)

Menschen benutzen ständig eine Vielzahl unterschiedlicher gedanklicher Modelle, um auf deren Basis mit ihrer Umwelt zu interagieren. Dabei machen sie sich zunutze, dass es einfacher ist, ein Modell zu erzeugen und zu verstehen, welches nur einem einzigen Zweck dienen soll.

Wenn Sie beispielsweise eine Radtour planen, könnten Sie sowohl ein Höhenprofil als auch eine Wegekarte benutzen. Beides sind leicht fassbare zweidimensionale Modelle des Geländes. Vielleicht haben Sie aber auch noch ein

zeitabhängiges Modell der zu erwartenden Windverhältnisse. Zusammenführen können Sie die Erkenntnisse aus diesen Modellen dann aber nur noch punktuell, weil eine komplette Gesamtschau die Gehirnkapazität überfordert.

Mit Blick auf die Softwareentwicklung ist die Lage leider nicht so einfach. Sie müssen in der Regel verschiedene Anforderungen mit nur einem Modell erfüllen. Das ist ein grundlegendes Problem der Softwareentwicklung: Denn Sie haben es höchst selten nur mit einem einzigen Zweck zu tun.

Eine signifikante Ausnahme sind die zahlreichen kleinen Werkzeugprogramme der Unix-Welt, die jeweils genau eine Aufgabe ausführen und nicht mehr. Das dahinterstehende einfache Modell ist möglicherweise ein Grund der bemerkenswerten Stabilität dieser Software.

Die Konsequenzen, die sich daraus ergeben, dass unterschiedliche Anforderungen vorliegen, sind verschieden und abhängig von den jeweiligen Sichten auf den Gegenstandsbereich.

Prinzipiell gibt es dabei zwei Fälle:

- Die Sichten sind *orthogonal*, das heißt, sie betreffen unterschiedliche Merkmale, die zudem voneinander unabhängig sind, beispielsweise technische und kaufmännische Parameter.
- Die Sichten weisen große Überschneidungen auf, sind also *nicht orthogonal*, und möglicherweise unterscheidet sich sogar die Interpretation von einzelnen Parametern, beispielsweise das Verständnis eines Auftrags aus Sicht der Produktion und Abrechnung.

Im ersten Fall wird das Modell lediglich etwas umfangreicher und komplizierter, bleibt aber verständlich, weil Sie sich ohne Gefahr auf jeweils eine Sicht beschränken können. Leider ist das Leben selten so einfach.

Der zweite Fall beschreibt die sehr oft anzutreffende Situation, dass die verschiedenen zu erfüllenden Zwecke das Modell komplexer machen. Diese Komplexität wird dadurch verursacht, dass jedem Zweck zwar weiterhin eine Sicht entspricht, aber das Verhalten des Modells nun in jeder Sicht durch verborgene Abhängigkeiten zwischen den Merkmalen beeinflusst wird.

Das sind Einflüsse von Merkmalen, von denen in einer Modellsicht abstrahiert wird, die aber in einer anderen Sicht wesentlich sind. Der Hauptvorteil der Modellierung, nämlich das einfachere Verständnis, geht dadurch dummerweise mehr oder weniger stark verloren. Aus diesem *Modellierungsdilemma* gibt es für den Entwickler keinen Ausweg.

Da das Modellierungsdilemma trotz seiner Wichtigkeit selten betrachtet wird, will ich es an zwei Beispielen aus der Welt der Compiler erläutern. Dieses Gebiet hat den großen Vorteil, dass die Anforderungen genau definiert und gut verstanden sind. Die auftretenden Diskrepanzen können also mit Sicherheit nicht auf ein möglicherweise noch nicht ausreichendes Verständnis zurückgeführt werden, sondern sie sind wirklich dem Modellierungsgegenstand innewohnend.

Das erste Beispiel stammt aus der hohen Zeit des Compilerbaus in den 1980er-Jahren. Damals gab es Versuche, durch eine neutrale Zwischensprache die Entwicklung neuer Compiler zu erleichtern.

Die Idee bestand darin, dass für jeden Prozessortyp die Zwischensprache nur einmal implementiert werden muss und für jede neue Programmiersprache nur noch ein rechnertypunabhängiger Übersetzer in die Zwischensprache gebraucht wird. Durch insgesamt `Z + P` Übersetzer für `Z` Prozessoren (Zwischensprache in Maschinencode) und `P` Programmiersprachen (jeweils in Zwischensprache) könnte man auf einen Schlag `Z * P` komplette Compiler erhalten.

Es stellte sich jedoch heraus, dass praktisch jede Programmiersprache einige Besonderheiten hat, die spezielle Konstrukte und damit Erweiterungen der Zwischensprache notwendig machte, sodass im Endeffekt die Zwischensprache zu einem großen Teil aus Elementen bestand, die jeweils nur für ein bis zwei Quellsprachen benötigt wurden. Die erhofften Vorteile ließen sich so nicht realisieren, weil das Modell einfach zu kompliziert geworden wäre.

Ganz ähnlich verhält es sich mit dem Bytecode der Java Virtual Machine (JVM), der ein etwas elementareres Konzept einer Zwischensprache darstellt. Allerdings ist die Entwicklung hier umgekehrt verlaufen.

Weitere Sprachen, die in Bytecode übersetzt werden, kamen erst im Laufe der Zeit dazu. Das Problem, bestimmte Spracheigenschaften nicht optimal unterstützen zu können, ergab sich jedoch ebenfalls.

Mit Java 7 wurde dafür eine höchst bemerkenswerte Lösung gefunden: *Invokedynamic*. Das ist ein spezieller Bytecodebefehl ohne vollständig spezifizierte Semantik.

Praktisch wird nur eine Schnittstelle definiert, an die für verschiedene Quellsprachen entsprechende Implementierungen angebunden werden können. Auf diese Weise kann die sonst unumgängliche Aufblähung der Liste der Bytecodebefehle vermieden werden.

Das grundlegende Problem wird auch dadurch nicht beseitigt, aber seine Lösung wurde verlagert. Der Preis dafür muss beim Implementieren einer neuen Sprache gezahlt werden. Denn die Übersetzung in Bytecode reicht nicht mehr aus. Unter Umständen muss Code für die notwendige Ergänzung der JVM-Funktionalität beigesteuert werden, und zwar für jede JVM-Plattform.

Sie können dieses Beispiel als interessante Anregung für Aufgaben sehen, bei denen Sonderfälle das Modell zu

überfordern drohen.

Und wenn es kein umfassendes Modell gibt?

Da Sie dem Modellierungsdilemma nicht ausweichen können, stellt sich sofort die Frage, wie Sie damit umgehen sollten. Einige Ratschläge:

- Erkennen und akzeptieren Sie das Dilemma! Wenn Ihnen das gelingt, haben Sie bereits einen großen Schritt getan, um mit den Konsequenzen umgehen zu können. Er ermöglicht es Ihnen, die folgenden Regeln anzuwenden.
- Investieren Sie keinen, letztendlich verlorenen, Aufwand dahinein, ein Problem zu lösen, das keine ideale Lösung hat. Wenn kein allumfassendes Modell sinnvoll gebildet werden kann, ist vielleicht eine Aufspaltung oder Zerlegung möglich.
- Hüten Sie sich vor übermäßig komplexen und alles auf einen Schlag lösenden Projekten, insbesondere wenn nur wenige verwertbare Erfahrungen vorliegen. Anforderungen von zu vielen verschiedenen Seiten führen beinah zwangsläufig zu komplexen und schwer zu beherrschenden Modellen.

Denken Sie daran, die »eierlegende Wollmilchsau« gibt es nicht und kann es nicht geben, auch nicht als Modell. (Nebenbei: Eier legen und Milch geben sind jeweils an sich gegenseitig ausschließende Vermehrungsmethoden gebunden.)

Mitunter gelingt es, durch gründliche Überlegungen ein allgemeineres Verständnis zu erreichen, auf dessen Basis dann ein Modell entwickelt werden kann, das verschiedenartige Ansichten erlaubt. Das ist vergleichbar mit der Einführung eines neuen Konzepts in der Wissenschaft und eine vergleichbare Sternstunde.

Normalerweise müssen Sie versuchen, durch Zerlegung in verschiedene Modelle dem Dilemma zu entgehen. Der Nachteil, dass Sie dann zusätzliche Schnittstellen und mehrere Komponenten haben, wird durch die bessere Beherrschbarkeit aufgewogen – jedenfalls dann, wenn Ihr Dilemma echt und nicht in fehlender Analyse begründet ist.

Ein Beispiel dafür, dass es auch in der Natur Dinge gibt, die sich der einheitlichen Modellierung entziehen, ist das Licht. Es hat sowohl Eigenschaften von Wellen als auch solche von Teilchen. Bis heute gibt es kein einheitliches Modell, das beide Ausprägungen verständlich zu erklären vermag. Je nach konkreter Frage muss man das eine oder das andere Modell zur Beantwortung heranziehen.

Als Softwareentwickler müssen Sie mit der Tatsache leben, dass jedes Programm nur genau ein Modell formalisieren kann. Und dieses eine Modell kann nur eine beschränkte Menge von Fragen beantworten.

Nehmen Sie sich bei komplexen Aufgaben ein Vorbild an der Natur. Versuchen Sie, den »großen Sprung« zu vermeiden und stattdessen in kleinen Schritten zu entwickeln. Dann sind auch die unvermeidlichen und nützlichen Irrtümer keine Katastrophen.

Zum Abschluss dieses Themas will ich Ihnen das grundlegende Problem noch einmal anhand des Entwurfs eines Pkw demonstrieren. Dabei kommen viele völlig unterschiedliche, aber in der Wirkung nicht unabhängige Modelle zum Einsatz. Das beginnt mit dem Design, der Fahrdynamik, der Stabilität, dem Crash-Verhalten und endet noch lange nicht bei Produktionsverfahren und Kostenanalysen.

Jedes dieser Modelle wird zwar einzeln betrachtet, Änderungen an einem haben aber fast immer direkte Auswirkungen auf andere. Eine Abwandlung im Design

beispielsweise kann unter anderem über die Stabilität das Gewicht und damit den Verbrauch oder die Herstellung beeinflussen.

Eine Gesamtoptimierung kann daher nur in sehr vielen Iterationen erfolgen. Und bei aller Sorgfalt bleibt immer ein Restrisiko, dass irgendein Zusammenhang übersehen oder eine Schnittstelle nicht korrekt bedient wurde.

Die materielle Realisierung von Modellen – hier der Bau eines Prototyps – hat allerdings gegenüber der virtuellen – in einem Programmsystem – einen großen Vorteil: Man sieht leichter, wenn etwas überhaupt nicht zusammenpasst. (Vielleicht erinnern Sie sich noch an die Probleme, die es beim Start der A380-Montage wegen zu kurzer Kabel gab.)

Bei Software lässt sich manches verbinden, was nicht dafür gedacht war, mit der Folge, dass selbst schwere Fehler, die im Nachhinein völlig offensichtlich sind, oft erst sehr spät entdeckt werden. Daran, dass solche Fehler passieren, lässt sich aus den geschilderten Gründen nur relativ wenig ändern.

Glücklicherweise sind viele Softwareprojekte nicht so komplex wie moderne Autos oder Flugzeuge. Trotzdem stößt man schon bei der Analyse scheinbar einfacher Konzepte, wie zum Beispiel das des Inhabers eines Bankkontos, auf eine verblüffende Vielzahl von Sichten, je nachdem, ob es um Verfügungsrechte, Boykottlisten, wirtschaftliche und steuerliche Verantwortlichkeiten oder Marketingfragen geht.

Entscheiden ermüdet

Nachdem ich im ersten Teil dieses Kapitels ein Problem betrachtet habe, das der Entwicklung unlösbar eigen ist, folgt nun eine ebenso reale Tatsache, der Sie als Entwickler unterliegen. Da man die Realität zwar verdrängen, ihr aber letztlich nicht entrinnen kann, sollten Sie diese zumindest zur Kenntnis nehmen.

Besser wäre es natürlich, wenn Sie Ihr Verhalten danach ausrichten. Es mag sein, dass in der Folge Ihre unmittelbare

Produktivität, gemessen in Zeilen Code pro Tag, nicht unbedingt steigen wird, auf jeden Fall wird aber die Qualität Ihres Codes zunehmen. Langfristig ist das produktiver.

Entwickeln heißt entscheiden

Viele Entwickler lieben ihren Beruf auch deshalb, weil er ihnen ein Gefühl von Freiheit gibt. Sie können entscheiden, wie eine bestimmte Funktion umgesetzt wird, wie Variablen benannt werden und vieles mehr. Am Ende steht dann hoffentlich der Erfolg, wenn alles wie gedacht zusammenspielt. Das ist die positive Seite.

Dem gegenüber steht, dass Menschen offensichtlich beim Entscheiden ermüden. Dieser *Entscheidungsmüdigkeit* oder *Decision Fatigue* sind auch Entwickler ausgeliefert. In der Folge nimmt die Güte der getroffenen Entscheidungen ab. Bauchgefühl, Stimmungen und Gewohnheiten verdrängen dabei zunehmend rationales Abwägen.

Eine große Gefahr ergibt sich daraus, dass Sie diese schleichende Verschlechterung nicht wahrnehmen, besonders wenn Sie gar nicht wissen, dass es so etwas gibt. Außerdem wird ja in den meisten Fällen Ihr Code auch nicht sofort erkennbar schlechter. Bösartigerweise schleichen sich aber gerade in solchen Situationen gern subtile Fehler ein, deren Entstehen man sich später nicht mehr erklären kann.

Ein Beispiel für diese Ermüdung hat fast jeder schon erlebt. Wenn Sie einmal längere Zeit mit Aufräumen beschäftigt waren, werden Sie bemerkt haben, dass es Ihnen nach einer Weile guten Fortschritts immer schwerer fällt, Dinge richtig einzusortieren. Manche fangen an, dann alles oder gar nichts mehr wegzuschmeißen. Besser wäre es, eine Pause zu machen. Am nächsten Tag läuft es dann wieder viel besser.

Wer um die Entscheidungsmüdigkeit weiß, kann – falls es sich einrichten lässt – statt der einen großen Aufräumaktion

gleich mehrere kleine planen. Das Ergebnis wird, ohne insgesamt mehr Zeit zu beanspruchen, besser sein.

Entscheidungsmüdigkeit

Wenn man die Anzahl der gegen Ende des Arbeitstags oder noch später getroffenen wichtigen Entscheidungen als Maßstab nimmt, muss diese Art der Ermüdung fast unbekannt sein.

Ins Licht der Öffentlichkeit geriet die Entscheidungsmüdigkeit 2011 durch einen Zeitungsbericht über die unterschiedliche Rate von Strafvollzugsaussetzungen auf Bewährung. Wissenschaftler hatten in Israel untersucht, ob die Erfolgsaussichten von der Herkunft oder anderen sozialen Faktoren abhängen.

Das vollkommen überraschende Ergebnis war: Am Morgen verhandelte Fälle wurden weitaus großzügiger entschieden als die gegen Ende des Tags. Diese Abhängigkeit war weitaus signifikanter als alle anderen, die betrachtet wurden.

Als Ursache kam nur eine Ermüdung beim Treffen von Entscheidungen infrage. Je weiter die Zeit fortschritt, desto mehr wich die notwendige Risikobereitschaft einem Auf-Nummer-sicher-Gehen.

Nach der Mittagpause zeigte sich übrigens ein kleiner Erholungseffekt, der aber schnell wieder verebbte. In mehreren Untersuchungen konnten seither derartige Effekte bestätigt werden.

Mittlerweile gibt es bereits Empfehlungen, Kunden in Onlineshops durch voreingestellte Auswahlen, die kein explizites Entscheiden mehr erfordern, in eine gewünschte Richtung, das heißt zum Kauf zu lenken.

Entscheidungskraft optimal nutzen

Da ganz offensichtlich Ihre Entscheidungskraft oder mentale Energie – wie bei allen Menschen – nun einmal begrenzt ist, sollten Sie sich Gedanken über deren möglichst sinnvollen Einsatz machen. Im Folgenden finden Sie einige Vorschläge, die Ihnen dabei helfen können. Aber seien Sie bei der Anwendung bitte nicht dogmatisch: Das richtige Maß entscheidet über den Erfolg.

Entscheidungen vermeiden

Die wichtigste Art, mentale Energie zu sparen, ist das Vermeiden von Entscheidungen. Die Menschheit hat ein ganzes Arsenal von

Techniken entwickelt, die diesen Zweck haben. Gewohnheiten, Rituale, Regeln und Konventionen gehören dazu. Obschon manchmal kritisch belächelt, helfen sie im Alltag, den Kopf für wichtige Dinge frei zu halten.

Auch beim Programmieren gibt es ausreichend Möglichkeiten, Entscheidungen zu vermeiden, indem Sie beispielsweise ähnliche Aufgaben immer auf die gleiche Weise lösen oder auf Entwurfsmuster zurückgreifen.

Natürlich bedeutet das nicht, dass Sie Ihre Tätigkeit gar nicht mehr kritisch reflektieren sollen. Ich plädiere lediglich dafür, in Zeiten hoher Arbeitsbelastung nur unbedingt erforderliche Entscheidungen zu treffen und das Hinterfragen des eigenen Handelns auf ruhigere Tage zu verschieben.

Wichtiges am Morgen entscheiden

Entscheiden Sie die wirklich wichtigen Dinge am Morgen. Fürs Entwickeln heißt das, bearbeiten Sie die schwierigen Aufgaben möglichst zeitig am Vormittag. Die verbreitete Sitte, den Tag mit einer Besprechung eigentlich weniger wichtiger Angelegenheiten zu beginnen, an die sich dann häufig noch längere Diskussionen zweitrangiger Fragen anschließen, ist absolut kontraproduktiv.

Wenn Sie es nicht vermeiden können, wichtige Entscheidungen oder schwierige Probleme spät am Tage zu bearbeiten, so sollten Sie unbedingt am nächsten Morgen noch mal einen kritischen Blick auf Ihr Ergebnis werfen. Der Volksmund empfiehlt zu Recht, über schwierige Fragen »erst mal eine Nacht zu schlafen«.

Entscheidungen verschieben

Viele Entscheidungen sind nicht so dringend, dass sie sofort getroffen werden müssen. Sparen Sie Ihre Energie für das auf, was unmittelbar entschieden werden muss. Sie werden sehen, manches erledigt sich von selbst, bevor es unaufschiebbar geworden ist.

Mach mal Pause

In Pausen erholt sich auch Ihre Entscheidungskraft, zumindest ein wenig. Vergessen Sie nicht, dass das Gehirn ein Energiegroßverbraucher ist. Also essen Sie ausreichend und treffen Sie nie Entscheidungen, wenn Sie hungrig sind. Der negative Einfluss von Hungergefühlen ist in Studien nachgewiesen worden.

Zweifeln Sie nicht

Vermeiden Sie es, einmal getroffene Entscheidungen immer wieder zu überdenken, es sei denn, es gibt einen wichtigen Grund dafür. In den meisten Fällen reicht es, eine Entscheidung zu treffen, die »gut genug« ist.

Sie haben nichts gewonnen, wenn Sie Ihre Energie investieren, um für eine Sache das Optimum zu finden, und währenddessen anderes schlecht, also nicht gut genug, entscheiden.

Haushalten

Mit Mitteln, die nur beschränkt verfügbar sind, empfiehlt sich gemeinhin ein sparsamer Umgang. Entscheidungskraft sollte ebenso wenig verschwendet werden wie Arbeitszeit. Denn so wie sich verlorene Zeit nicht zurückholen lässt, ist auch nutzlos aufgewandte Entscheidungskraft dahin.

Die Energie, die Sie für eine Entscheidung aufwenden, muss durch die Folgen gerechtfertigt sein. Verzetteln Sie sich nicht in unbedeutenden Detailfragen.

Bedenken Sie außerdem, dass Sie – ähnlich wie bei körperlich anstrengender Arbeit – kein je eigenes Budget für Arbeit und Freizeit haben. Sie haben nur einen Vorrat, und der muss für alles reichen.

-  Denken Sie schon bei der Planung Ihrer Tätigkeit daran, dass Ihre Entscheidungskraft erschöpfbar ist. Berücksichtigen Sie auch Anforderungen im Privatleben.
- Aufgebrauchte Entscheidungskraft lässt sich nicht durch längere Arbeitszeit ersetzen. Im Gegenteil steigt dadurch die

Gefahr, dass Sie Fehler machen, stark an.

Das Wichtigste in Kürze

- ✔ Es gibt Schwierigkeiten, denen Sie nicht ausweichen können, weil sie in der Sache selbst begründet liegen. Um damit erfolgreich umgehen zu können, müssen Sie diese Klippen kennen und erkennen.
- ✔ Mentale Modelle sind die Grundlage jeder Software. Die Art der Modellierung hängt stark davon ab, welche Fragen das Modell beantworten soll.
- ✔ Zu unterschiedliche Anforderungen können dazu führen, dass kein widerspruchsfreies oder überschaubares Modell mehr entwickelt werden kann. Sie stehen dann vor einem Modellierungsdilemma.
- ✔ Die Fähigkeit, sinnvolle Entscheidungen zu treffen, ist ähnlich der physischen Leistungsfähigkeit begrenzt. Man nennt das Entscheidungsermüdung.
- ✔ Da Softwareentwicklung eine Folge kleiner und großer Entscheidungen ist, spielt die Entscheidungsermüdung eine wichtige Rolle.
- ✔ Finden Sie Wege, um mit Ihrer Entscheidungskraft sorgsam und sparsam umzugehen.

Kapitel 9

Fallen vermeiden

IN DIESEM KAPITEL

- Erst mal loslegen geht oft schief
- Unnötige Flexibilität ist teuer
- Modularisierung hat ihren Preis

> Du musst deinen Feind kennen, um ihn besiegen zu können. (Sunzi, chinesischer Philosoph und Stratege, ca. 500 vor Christus)

Es gibt auf dem Weg zu guter Software jede Menge böser Fallen. Einige sind leicht zu erkennen, zum Beispiel enge Budgets und unrealistische Zeitpläne. Die gefährlicheren Fallen verstecken sich jedoch hinter plausibel erscheinenden Verfahren, deren Anwendungsbereich maßlos überdehnt wird. In diesem Kapitel erfahren Sie, wie ein manchmal sinnvolles Vorgehen in anderen Fällen in die Irre führen kann.

Erst mal loslegen

Mittlerweile sollte es sich eigentlich überall herumgesprochen haben, dass jedes Projekt ein klares Ziel und einen abgegrenzten Umfang, gern *Scope* genannt, braucht. Trotzdem werden Sie immer wieder auf Vorhaben stoßen, bei denen diese Grundregel verletzt wird. Die Folgen sind gewöhnlich dramatisch, auch wenn es dem Management bisweilen gelingt, dies zu vertuschen.

Agil heißt nicht »kein Konzept«

Ein Grund dafür, dass die konzeptionelle Vorbereitung vernachlässigt wird, ist ein falsches Verständnis von Agilität. *Agile Softwareentwicklung* ist ursprünglich als leichtgewichtiges Gegenstück zu traditionellen Entwicklungsprozessen entstanden, weil diese zu umständlich erschienen. Als besonderer Vorteil gilt, dass der große Initialaufwand durch Analyse und Design entfällt und durch ein iteratives und inkrementelles Vorgehen ersetzt wird.

Diese einseitige Betonung der Schnelligkeit birgt jedoch eine Gefahr: Es wird allzu leicht übersehen, dass ein grundlegendes Verständnis des Problembereichs trotz allem vorhanden sein muss. Genau diese Voraussetzung ist oftmals nicht erfüllt.

Sie können das mit der Orientierung im Gelände vergleichen. Beim traditionellen Vorgehen erstellen Sie zunächst eine möglichst präzise Karte und legen dann den Weg detailliert fest. Das schließt natürlich nicht aus, dass Fehler passieren, die Sie dann im Gelände ausbügeln müssen.

Beim agilen Vorgehen haben Sie zwar auch eine Vorstellung davon, wo Sie ungefähr rauskommen wollen, aber keinen exakten Plan. Sie gehen sozusagen auf Sicht.

Damit das nicht in eine unvertretbar aufwendige Folge von »Versuch und Irrtum« ausartet, brauchen Sie jedoch zumindest eine ungefähre Gesamtübersicht, die weit über die unmittelbar zu lösenden Aufgaben hinausreicht. Diese Karte muss nicht übermäßig exakt sein, aber entscheidende Hindernisse, also zum Beispiel eine unpassierbare Schlucht kurz vor dem Ziel, sollten erkennbar sein. Ganz ohne grobe Übersicht geht es einfach nicht.

Wenn Sie mit dem Programmieren beginnen, ohne eine halbwegs schlüssige Theorie zu haben, ist es fast sicher, dass Sie viel Lehrgeld werden bezahlen müssen. Schließlich formalisieren Sie dann bereits Arbeitsstände und Fragmente Ihres Modells, lange bevor die Modellierung abgeschlossen ist.

Jede Modifikation Ihres Modells hat unvermeidlich eine Überarbeitung des bereits erstellten Codes zur Folge. Die daraus entstehenden Kosten werden Sie nicht dauerhaft ignorieren können, sodass Sie unweigerlich irgendwann anfangen, nicht mehr nach dem besten Modell zu suchen, sondern nach demjenigen, das sich mit dem geringsten Aufwand an Ihre bisherigen Vorstellungen anpassen lässt. So gehen Sie den ersten Schritt in Richtung unsauberer Software. Eine Theorie, die nicht so klar und einfach wie möglich ist, lässt sich auch schwerer begreifen und formalisieren.

Bis eine Theorie die notwendige Reife erreicht hat, sind viele Zyklen der Abstraktion, des Aufstellens und Prüfens von Hypothesen sowie der Neuformulierung notwendig. Der Großteil dessen muss vor Beginn der eigentlichen Entwicklung erledigt sein. Alles andere führt zu erhöhten Kosten, faulen Kompromissen und Frust bei den Entwicklern.

Versuchen Sie stets zu erreichen, dass der Gegenstandsbereich einer Entwicklungsaufgabe so gut verstanden ist, dass größere Überraschungen unwahrscheinlich sind.

Abgrenzung ist alles

Ohne klares Konzept sind Sie nicht in der Lage, den Umfang Ihres Projekts abzugrenzen. Die möglichst genaue Abgrenzung ist aus den folgenden Gründen jedoch unverzichtbar:

- ✔ Ohne vereinbarte Grenzen sind ärgerliche Diskussionen darüber, ob die vereinbarten Ziele erreicht wurden oder nicht, kaum zu vermeiden.
- ✔ Ohne genaue Beschreibung der Projektgrenzen, das heißt dessen, was die zu schaffende Software leisten soll, können Sie den Modellierungszweck nicht effektiv bestimmen.
- ✔ Sowohl beim Modellieren als auch später beim Implementieren müssen Sie immer wieder zwischen

konkurrierenden Anforderungen abwägen. Für die notwendige Bewertung der Varianten müssen Sie wissen, was erwartet wird und was nicht.

- ✔ Wenn Sie sehen, wie weit sich durch Änderungswünsche die ursprünglichen Grenzen verschieben, können Sie den notwendigen Umfang und damit die resultierenden Kosten leichter abschätzen.

Natürlich müssen Sie dokumentieren, welche Abgrenzungen aus welchen Gründen wesentlich sind. Projekte können nicht von heute auf morgen realisiert werden. Das Umfeld bleibt nicht unverändert. Deshalb sind neue und modifizierte Wünsche nicht vermeidbar. Sie werden nur selten im Voraus wissen, was da kommen wird.

Trotzdem sollten Sie es vermeiden, die Grenzen vorsichtshalber sehr weit zu ziehen. Zum einen verursacht das zusätzlichen Aufwand, dessen Nutzeffekt zweifelhaft ist. Zum anderen kommen neue Anforderungen häufig aus völlig überraschenden Richtungen. Mit einer sauberen und kompakten Struktur können Sie nachhaltiger darauf reagieren, als wenn Sie versuchen, irgendetwas, das zwar für Ergänzungen gedacht war, aber nur halbwegs passt, umzubiegen.

Und wenn eine Modifikation einen umfangreichen Umbau erfordert, dann ist das eben so. Eine weniger eng gezogene Grenze hätte in den allermeisten Fällen den Umfang des Umbaus auch nicht verringert.

Saubere Abgrenzung äußert sich nicht zuletzt in klaren Begriffen. Definitionen sind nichts anderes als die Zuordnung möglichst genau begrenzter Bedeutungen zu einem sprachlichen Ausdruck.

Natürliche Sprachen leben freilich von und durch eine gewisse Unschärfe ihrer Begriffe. Je näher eine Theorie jedoch der Formalisierung kommt, desto wichtiger wird es, diese Unschärfe zu reduzieren. Dementsprechend basieren alle formalisierten Theorien auf einer Grundmenge exakter Begriffsbestimmungen, oder anders ausgedrückt: Bedeutungsabgrenzungen.

Und weil der Code einen Formalismus repräsentiert, realisiert er zwangsläufig abgegrenzte Bedeutungen. Dummerweise sind diese aus dem Code meist schwer zu erschließen und in ungünstigen Fällen auch noch schwer sprachlich zu beschreiben.

Wenn es nicht anders geht

Das Leben an sich und das eines Entwicklers im Besonderen ist kein Wunschkonzert. Manchmal wird es sich nicht umgehen lassen, dass Sie eine Aufgabe ohne ausreichende Vorbereitung angehen müssen. Als erfahrener Clean-Coder wissen Sie natürlich sofort, dass die Versäumnisse bei der Vorbereitung ihren Preis haben. Das Konzept muss dann eben inkrementell im Laufe der Entwicklung erstellt werden.

Vergessen Sie das nie, und lassen Sie sich nicht leichtfertig zu optimistischen Terminzusagen überreden. Das Ganze ist und bleibt ein Erkundungsunternehmen, bei dem jederzeit mit überraschenden Wendungen und Schwierigkeiten zu rechnen ist.

Stellen Sie sich vor, Sie müssten ein Ziel durch wegloses und unübersichtliches Gelände erreichen. Wie würden Sie vorgehen? Auf jeden Fall empfiehlt es sich, genügend Proviant mitzunehmen, denn Sie wissen nicht, wie lange Sie brauchen werden.

Sehr wichtig ist es auch, den jeweils zurückgelegten Weg so zu dokumentieren, dass Sie im Falle einer Sackgasse gefahrlos zurückkehren können, um eine andere Variante zu probieren. Wenn Sie eine Weile darüber nachdenken, fallen Ihnen gewiss weitere Ratschläge ein. Analog sollten Sie in Projekten mit unausgereiften Konzepten vorgehen.

Ein paar Ratschläge für den Anfang, die Sie gern ergänzen dürfen:

- ✔ Reservieren Sie Zeit für die Erarbeitung des Konzepts, das heißt vor allem für die möglichst exakte Beschreibung des Modells und die Festlegung des begrifflichen Rahmens. Ohne

gemeinsames Verständnis gibt es keinen verständlichen Code.

- ✔ Unternehmen Sie »Expeditionen ins Unbekannte«, indem Sie *technologische Durchstiche* zu realisieren versuchen, um auf diese Weise die prinzipielle Umsetzbarkeit von Ideen zu prüfen.
- ✔ Sichern Sie Zwischenstände des Projekts, die als Wiederaufsetzpunkte geeignet sind, falls sich im weiteren Verlauf ein bestimmtes Vorgehen als ungeeignet erweist.
- ✔ Versuchen Sie, Teilaufgaben zu isolieren, die gut verstanden sind und deren Umsetzung relativ unkritisch ist. Da es im Allgemeinen nicht sinnvoll sein wird, zu viele Erkundungen parallel vorzunehmen, kann ein Teil des Teams mit diesen Aufgaben schon produktiv arbeiten.
- ✔ Sie werden in solch einem Projekt sehr viel lernen. Vernachlässigen Sie deshalb die Dokumentation der gewonnenen Erfahrungen nicht, auch wenn Sie agil arbeiten. Fast jedes Projekt hat ein Folge- oder Nachbarprojekt. Wenn nichts dokumentiert wird, ist das von Ihnen gezahlte Lehrgeld zum Fenster hinausgeworfen.

Bei allen Bemühungen werden Sie es nicht vermeiden können, dass ein ohne ausreichende Konzeption gestartetes Projekt aufwendiger und fordernder ist. Aber Sie können helfen, die Kosten in Grenzen zu halten, wenn Sie sich der Herausforderungen bewusst sind und die zusätzlichen Aufgaben nicht verdrängen, sondern engagiert anpacken.

Schön flexibel bleiben

> Erstens kommt es anders und zweitens als man denkt. (Redensart)

Sobald es um die Zukunft geht, hört es sich immer gut an, wenn jemand versichert, er sei in der Lage, »flexibel« auf die zu

erwartenden Unwägbarkeiten zu reagieren. Die Softwareentwicklung macht da keine Ausnahme.

Trotzdem sollten Sie bei solchen Sätzen stets genau hinhören, was wirklich gemeint ist. Manchmal verbirgt sich dahinter nämlich einfach das Ausweichen vor Entscheidungen nach dem Motto: Ich mache erst mal gar nichts und überlege es mir später. Im Leben mag das zuweilen eine angemessene Maxime sein, beim Programmieren ist es das meist nicht.

Flexible Programme

Flexibel sind Programme streng genommen immer, da sie mit unterschiedlichen Daten umgehen können. Das ist der große Vorteil der Steuerung durch Software. Die flexibelsten Programme überhaupt sind Interpreter. Bei ihnen steuert ein Teil der Daten, der zu interpretierende Code, die Verarbeitung der restlichen Daten.

Aber diese Betrachtung führt nicht wirklich weiter. Hier geht es mir vorrangig um die Flexibilität, die in Software eingebaut wird, um für noch nicht bekannte, aber möglicherweise aufkommende Anforderungen gerüstet zu sein.

Die Gründe für dieses vorauseilende Bereitsein sind verschiedene:

- ✔ Die Erfahrung, dass Software gewissermaßen »lebt«. Die äußeren Umstände ändern sich ständig, und der Code muss dementsprechend angepasst werden.
- ✔ Unsicherheiten, die sich aus dem Umfeld ergeben, beispielsweise durch noch laufende Vertragsverhandlungen oder anstehende Organisationsveränderungen.
- ✔ Unsicherheiten, die sich aus der ungenügenden konzeptionellen Vorbereitung ergeben.
- ✔ Der Unwille mancher Verantwortlicher, Entscheidungen zu treffen.

- ✔ Unklarheiten über die relevanten Projektziele. (Obwohl das kaum glaubhaft erscheint, habe ich das mehrfach erlebt, meist als Folge langwieriger Entscheidungsrunden.)

Sie als Softwareentwickler haben dann die undankbare Aufgabe, zu entscheiden, wo und mit welchen Mitteln die erwünschte Anpassbarkeit ermöglicht werden soll. Bei einer oft genutzten Variante wird beispielsweise die Menge der zulässigen Werte erst zur Laufzeit aus einer Datenbank oder Datei eingelesen.

Programmiersprachen wie Java erlauben es außerdem, Code, der möglicherweise erst später erstellt werden wird, während der Laufzeit dynamisch nachzuladen. In der Tat gibt es auf dem Weg zur flexiblen Software kaum ernsthafte technische Grenzen, wenn Sie die Beherrschbarkeit außer Acht lassen.

Flexibilität bläht auf

Der Wunsch, flexibel zu bleiben, hört sich oft gar nicht so unvernünftig an. Es gibt nicht wenige Entwickler, die diesen Wunsch als berufliche Herausforderung sehen und gern darauf eingehen.

Was dabei allzu leicht übersehen wird, ist die damit – unter der Hand – einhergehende Ausweitung des Scopes. Durch die außerplanmäßig eingebaute Variabilität deckt die Software einen größeren Bereich ab als ursprünglich vorgesehen.

Die unvermeidliche Konsequenz jeder Umfangserweiterung sind steigende Kosten. Diese entstehen in allen Phasen: Entwicklung, Dokumentation, Test und Betrieb. Letztere sind allerdings ein eher kleiner Posten, der aus dem durch die dynamische Verarbeitung verursachten und oft nicht sehr relevanten höheren Ressourcenverbrauch entsteht.

Sehr viel wichtiger ist der negative Einfluss auf das vorhandene Modell. Flexibilität wird gewöhnlich lokal realisiert, das heißt mit eingeschränktem Blick auf das Gesamtmodell. Wenn die Auswirkungen ebenso lokal begrenzt bleiben, ist das nicht so schlimm. Aber häufig werden Sie diese Lokalität nicht vollständig

gewährleisten können. Auch das ist im Einzelfall noch kein Beinbruch.

Gefährlich wird es erst, wenn mehrere »nicht ganz lokale« Modifikationsmöglichkeiten zusammenkommen. Denn dann formalisieren Sie nicht mehr Ihr initiales Modell, sondern ein in subtiler Weise durch die flexiblen Elemente parametrisiertes Modell.

Es kann passieren, dass dieses erweiterte Modell spürbar von dem Verständnis abweicht, das Sie sich erarbeitet haben. Bemerken werden Sie das schlimmstenfalls zur Laufzeit, wenn unerwartete Wechselwirkungen zu überraschenden Fehlern führen.

Wie schwierig die einzelnen Steuerungsmöglichkeiten und ihre jeweiligen Wirkungen zu beschreiben sind, entdecken Sie spätestens beim Dokumentieren. Und sorgfältige Dokumentation ist zwingend, wenn jemand später die aufwendig eingebaute Flexibilität nutzen soll.

Die Sinnfrage

Abschließend nun die Sinnfrage, die Sie sich aber besser schon am Anfang stellen sollten: Wo liegt der Nutzen erhöhter Flexibilität? Wenn man unter Flexibilität die Anpassbarkeit an bei Projektbeginn noch nicht bekannte Anforderungen versteht, fällt es sehr schwer, eine überzeugende Antwort zu geben.

Es wird ein erheblicher Aufwand getrieben, um für Ereignisse in Zukunft gewappnet zu sein. Wohlgemerkt, für Ereignisse, die möglicherweise nie eintreten. So gesehen ist Flexibilität ein reines Spekulationsgeschäft.

Wenn es nur um die Kosten ginge und der Projektsponsor bereit wäre, das Risiko einzugehen – warum nicht? Entscheidend ist jedoch, dass Sie mit der Flexibilität die Software komplexer und damit letztlich schwerer wartbar machen.

Bei jeder anstehenden Änderung stehen Sie dann vor der Frage: Kann das im Rahmen der vorhandenen Flexibilität gelöst werden,

oder muss eine ganz normale Modifikation implementiert werden? Und natürlich werden Sie dabei einem gewissen Druck ausgesetzt sein, denn »das sollte doch alles flexibel sein«. Um die Frage trotzdem wohlbegründet beantworten zu können, benötigen Sie jemanden, der die Theorie der Software, einschließlich aller Parameter, kennt. Nur so ist garantiert, dass die Änderungen in der vorhergeplanten Weise erfolgen.

Leider wird diese Voraussetzung oft nicht beachtet. Weil es so einfach erscheint, Parameterwerte zu ändern, wird das viel zu leichtfertig vorgenommen – mit möglicherweise desaströsen Folgen für die Überlebensfähigkeit der Software.

Ein anderer Punkt, der bei flexibler Software ebenfalls gern übersehen wird, ist die erreichbare Testabdeckung. Da durch die zusätzlichen Parameter der eigentlich notwendige Testumfang förmlich explodiert, wird in der Praxis meist nur mit den initial geplanten Werten und bestenfalls noch ein paar ausgewählten Szenarien getestet. Das heißt, dass Sie bei einer späteren Anpassung oft gar nicht wissen, ob die dadurch entstehende Konfiguration jemals geprüft wurde.

Wenn Sie diese Risiken gegen den möglichen Nutzen abwägen, sollte klar sein, dass Flexibilität, die keiner genau beschriebenen Anforderung folgt, als Entwicklungsziel sehr kritisch zu bewerten ist. In den allermeisten Fällen ist das kein geeigneter Weg, wenn Sie hochwertige Software, die leicht weiterzuentwickeln ist, liefern wollen.

- Vermeiden Sie, wo immer möglich, nicht angeforderte Flexibilität. Die zusätzliche Komplexität rechtfertigt fast nie die vermeintlichen, unsicheren Vorteile.
- Reduzieren Sie bei Überarbeitungen nicht mehr benötigte Flexibilität. Das erhöht auf die Dauer die Stabilität der Software.
- Versuchen Sie, nicht vermeidbare Flexibilität auf exakt definierte Bereiche zu beschränken, die Sie in Tests

einbeziehen können.

Modularisierung übertreiben

Modularisierung gilt gemeinhin als Königsweg, als nicht in Zweifel zu ziehendes Ziel für die Gestaltung guter Software. Nicht ganz zu Unrecht, verspricht sie doch eine Reihe von Vorteilen, sowohl bei der Entwicklung als auch später bei der Pflege. Und mit einem Verweis auf die Analogie zu einem bekannten Steckbaukastenprinzip lassen sich die letzten Zweifler überzeugen.

Selbstverständlich sind die aufgeführten Argumente alle richtig, aber sie zeigen nicht das ganze Bild. Wie so häufig steckt der Teufel im Detail.

Davon verschwindet die Komplexität nicht

Modularisierung kann helfen, Komplexität beherrschbarer zu machen. Insofern ist sie eine Ausprägung des Abstraktionsprinzips. Allerdings übersehen viele, dass es Probleme gibt, die inhärent komplex oder kompliziert sind. Solche inhärente Komplexität lässt sich nicht »wegmodularisieren«. Im schlimmsten Fall kann eine ungeschickte Modularisierung sogar die Komplexität erhöhen.

Wenn Sie ein Beispiel für eine Anforderung suchen, die Ihnen jedes vernünftige Modul zur Umsatzsteuer-Berechnung sprengt, sehen Sie sich die Steuersätze für Weihnachtsbäume an.

Die anzusetzende Mehrwertsteuer (zwischen null und 19 Prozent) hängt unter anderem davon ab, wie und wo (artgerecht, Plantage, Wald) der Baum gewachsen ist und wer (Gewerbetreibender, Landwirt, Kleinunternehmer) ihn

verkauft. Das sind Daten, die Sie für andere Produkte wohl nicht brauchen, die aber die Modulschnittstelle aufblähen.

Auch wenn dieses Beispiel spektakulär willkürlich erscheint, werden Sie überall vergleichbar aus dem Raster fallende Gegebenheiten finden, die sich – im Gegensatz zum genannten Beispiel – oft sogar einleuchtend begründen lassen.

Die Erfahrung zeigt, dass Prozesse und Produkte dazu tendieren, im Laufe ihres Lebens komplexer zu werden. Das ist nicht überraschend, denn sobald etwas einen gewissen Reifegrad erreicht hat, sind weitere Verbesserungen vorrangig durch das Ausnutzen spezieller Wechselbeziehungen, auch *Synergien* genannt, möglich.

Beispiele für die Tendenz zur Integration zunächst eher unabhängiger »Module« finden Sie überall, um nur ein paar zu nennen:

- ✔ Durch die Luftdrucküberwachung der Reifen sind Räder heute viel stärker in das Gesamtsystem Auto integriert als noch vor ein paar Jahren. Das gilt für viele andere Komponenten ebenso.
- ✔ Kochautomaten mit Internetanbindung integrieren Küchenmaschinen und Computer.
- ✔ Webshops benutzen Informationen aus Produktsuche und Statistik im Verkaufsmodul für Angebotsvorschläge.

Diese Tendenz wirkt sich auf die Modularisierung auf zweierlei Art aus:

- ✔ Wenn ein bereits länger bestehendes System ersetzt werden soll, zeigen sich bei der genauen Analyse viele zunächst übersehene Abhängigkeiten. Das Gleiche gilt, wenn etablierte gewachsene Prozesse erstmalig automatisiert werden sollen.

- ✔ Falls die vorliegende Anwendung schon gut modularisiert ist, werden Sie zunehmend mit Wünschen konfrontiert sein, die sich im Rahmen dieser Struktur nicht realisieren lassen, weil sie beispielsweise Schnittstellenerweiterungen nötig machen.

Kurz gesagt: Modularisierung kann Ihnen helfen, Ordnung und damit die Übersicht zu behalten – aber echte Komplexität lässt sich dadurch nicht beseitigen.

Zerlegung will geübt sein

Bevor Sie anfangen, eine Modulstruktur zu entwerfen, müssen Sie sich Gedanken über das Ziel der Modularisierung machen. Wollen Sie in erster Linie

- ✔ parallele unabhängige Entwicklungen ermöglichen oder
- ✔ die Austauschbarkeit von Komponenten gewährleisten oder
- ✔ die Wiederverwendbarkeit von Komponenten unterstützen oder
- ✔ ein an Kundenwünsche anpassbares Produkt bereitstellen oder
- ✔ die selektive Weiterentwicklung erleichtern?

Das angestrebte Ziel bestimmt, wie die Zerlegung sinnvollerweise erfolgen sollte. Wenn Ihre Software auf unterschiedlichen technischen Plattformen laufen soll, werden Sie zumindest einen Teil Ihrer Module nach technischen Gesichtspunkten schneiden. Geht es eher um verschiedene fachliche Funktionen, so sind dementsprechend fachliche Aspekte entscheidend.

Sie können sich auch hier sehr leicht einem Dilemma ausgesetzt sehen, wenn Sie sowohl fachlichen als auch technischen Betrachtungsweisen genügen müssen. Ein Modul einer Sichtweise kann nämlich durchaus Funktionen umfassen, die sich aus einer anderen Sichtweise auf mehrere Module verteilen.

Dann müssen Sie entweder eine der beiden Sichtweisen zur führenden erklären (lassen) oder aus den Schnittmengen viele

kleine Module erzeugen, was meist keine gute Idee ist.

Unterschätzen Sie nicht die Schwierigkeiten beim Entwurf der Modulstruktur und nehmen Sie den Entwurf nicht auf die leichte Schulter. Die Struktur entscheidet letztlich über den Erfolg.

Schon wieder: Kosten

Modularisierung gibt es nicht zum Nulltarif. Konzeption und Umsetzung verursachen Aufwände. Ein Teil davon rentiert sich bereits in der Entwicklung durch die genau abgegrenzten Aufgaben und die klaren Schnittstellen.

Aber gerade die Schnittstellen verursachen auch Mehraufwand in der Entwicklung und verschlechtern die Leistung im Betrieb.

Zum einen müssen im Sinne einer defensiven Programmierung alle von außen, und das heißt hier von außerhalb des Moduls, kommenden Daten überprüft werden. Jede Schnittstelle ist auch ein potenzieller Angriffspunkt.

Zum anderen verlangen Schnittstellen in der Regel die Datenübergabe in festgelegten kanonischen Formaten, die für die Verarbeitung nicht immer gut geeignet sind. So ist es beispielsweise oft üblich, Datumswerte als Zeichenketten im ISO-Format zu übergeben. Das kann dazu führen, dass Werte eigentlich unnötig aus einer internen Form in die externe und wieder zurück konvertiert werden müssen. Zuweilen verursacht das eine messbare Mehrbelastung.

Unabhängig von solchen direkt nachweisbaren Kosten sollten Sie sich aber immer fragen, welchen konkreten Nutzen die Modularisierung in Ihrem Fall bringt. Klar, ein gewisser Grad von Strukturiertheit ist stets nützlich. Aber es gibt in jedem Projekt eine Grenze, nach deren Überschreiten der zusätzliche Aufwand keinen angemessenen Effekt mehr erzeugt.

Wachsen im Korsett

Kaum jemand denkt daran, dass überdies die Modularisierung eine entwicklungsfeindliche Seite hat. Sind die Modulstruktur und die Schnittstellen erst einmal festgezurrt, bilden sie gewissermaßen ein Korsett für die weitere Entwicklung.

Solange sich neue Anforderungen vollständig innerhalb der Module implementieren lassen, stört das nicht. Wie bereits angedeutet, ist es jedoch äußerst wahrscheinlich, dass irgendwann Anforderungen auftauchen, die Sie in der bestehenden Struktur nicht effektiv erfüllen können.

Mit etwas Glück reicht es dann, eine Schnittstelle geringfügig »aufzubohren«. Schon das ist keine einfache Arbeit und erfordert umfangreiche sogenannte Regressionstests, die nach der Fehlerkorrektur oder Änderung sicherstellen, dass keine neuen Fehler oder unerwünschte Seiteneffekte eingebaut wurden. Richtig aufwendig wird es aber, wenn Sie die Modulstruktur umbauen müssen, das heißt, wenn das Korsett zu eng geworden ist.

Wenn sich die bestehende Modulstruktur als ernstes Hindernis für die erforderliche Weiterentwicklung herausgestellt hat, sollten Sie vor einer einfachen Überarbeitung prüfen, ob nicht eine grundsätzliche Neuentwicklung – vielleicht unter Weiternutzung noch brauchbarer Komponenten – die wirtschaftlichere Lösung ist.

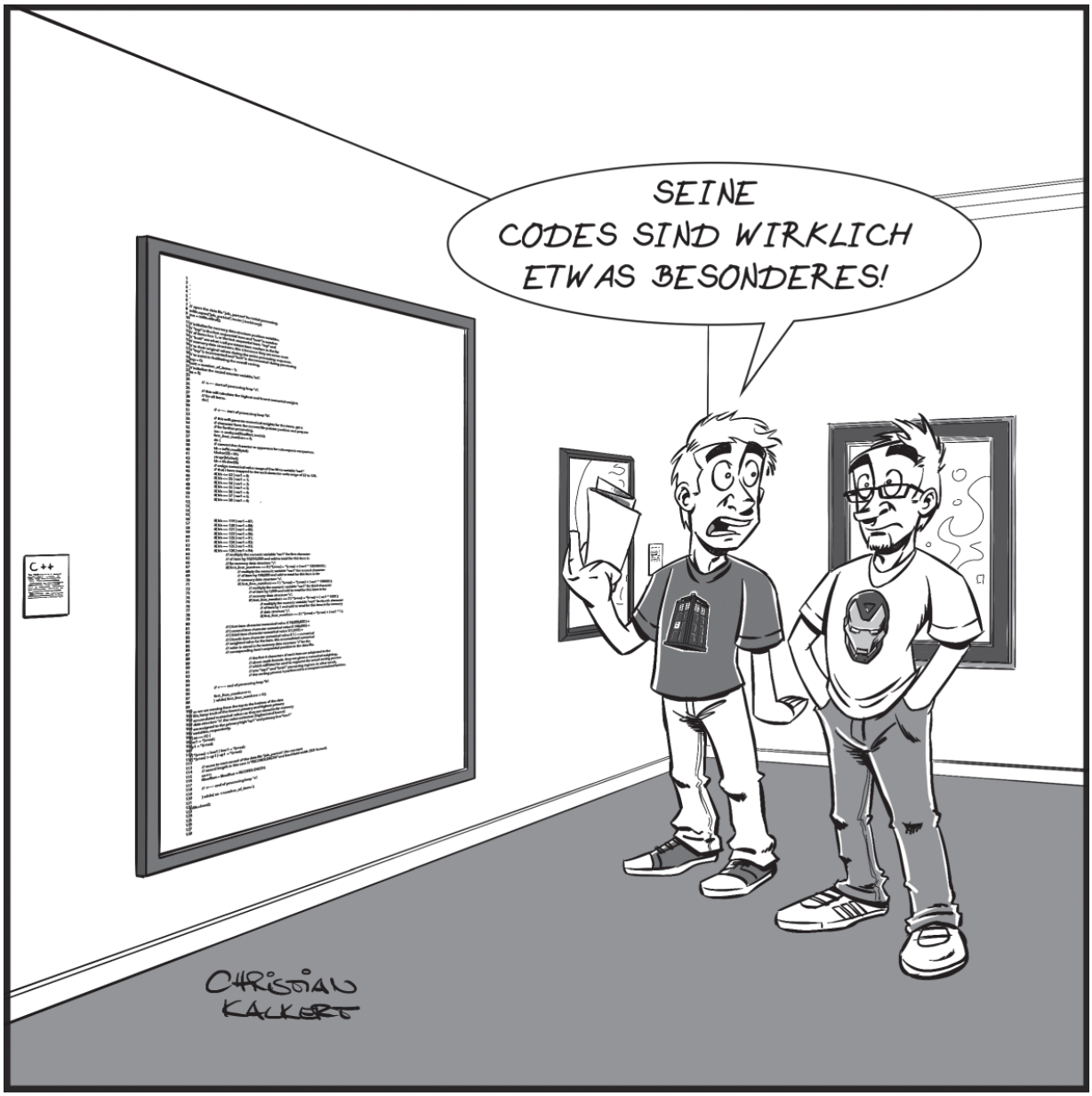
Das Wichtigste in Kürze

- Auf dem Weg zu Clean Code lauern viele Fallen. Halten Sie deshalb die Augen offen, um das Hineintappen zu vermeiden.
- Ohne klares Konzept gleicht jede Unternehmung, Software zu entwickeln, einer Expedition in unbekanntes Gelände. Man erweitert zwar sein Wissen, erhält aber nur mit viel Glück ein brauchbares Produkt.

- ✔ Die Kosten, die für die Konzeption anfallen, lassen sich auch durch agiles Vorgehen nicht vermeiden. Vielmehr verursachen die unvermeidlichen Fehlschläge durch die parallel verlaufende Entwicklung dort zusätzlichen Aufwand.
- ✔ Flexibilität vermittelt die trügerische Sicherheit, auf die unbekannten Herausforderungen der Zukunft gut vorbereitet zu sein. Tatsächlich bindet sie nur Kapazitäten, die besser verwendet würden, um die Software so zu gestalten, dass sie bei Bedarf möglichst einfach angepasst werden kann.
- ✔ Modularisierung ist in verschiedener Hinsicht nützlich, hat aber auch ihren Preis. Ob der gerechtfertigt ist, hängt entscheidend vom Zweck der Zerlegung ab.
- ✔ Die Definition von Schnittstellen ist schwierig und kritisch. Einerseits wird dadurch eine saubere Abgrenzung erreicht, andererseits können feste Schnittstellen bei der Weiterentwicklung hinderlich werden.

Teil III

Sauberen Code schreiben



IN DIESEM TEIL …

- ✔ Lernen Sie Regeln und Grundsätze kennen, um Clean Code zu schreiben
- ✔ Gestalten Sie Code durch gute Namen und passende Formatierung besser lesbar
- ✔ Begreifen Sie Schnittstellen als Schlüssel für eine verständliche Struktur
- ✔ Erfahren Sie einiges über den Umgang mit Fehlern und Fehlersituationen
- ✔ Lernen Sie neue Sprachelemente zu bewerten und zu nutzen

Kapitel 10

Namen sind nicht Schall und Rauch

IN DIESEM KAPITEL

- Namen sind wichtig für das Vermitteln des Verständnisses
- Gute Namen zu finden, ist nicht einfach
- Woran Sie gute Namen erkennen können
- Die Qual der Sprachwahl

In diesem Kapitel dreht sich alles um Namen. Was zeichnet gute Namen aus? Welche Fehler sollten Sie vermeiden und wie lassen sich treffende Bezeichnungen finden? Abschließend werden einige Gesichtspunkte, die bei der Wahl der Sprache bedeutsam sind, betrachtet.

Benennungen

Wenn Goethe seinen Doktor Faust »Name ist Schall und Rauch« sagen lässt, drückt der sich nicht nur um die Antwort auf die berühmte Gretchenfrage, sondern verweist darauf, dass *Benennungen* letztlich immer willkürlich sind. Trotzdem spielen sie eine unverzichtbare Rolle in der menschlichen Kommunikation.

Ohne die Übereinkunft, einen Sachverhalt mit einem bestimmten Namen zu bezeichnen, kann es keinen Austausch von Informationen geben. Je besser dabei die jeweiligen Interpretationen von Sprecher und Hörer beziehungsweise von

Sender und Empfänger übereinstimmen, desto effektiver und leichter ist die Verständigung.

Auch wenn das gern übersehen wird, ist Code in ganz erheblichem Umfang ein Verständigungsmittel zwischen Menschen, und zwar zwischen denjenigen, die ihn schreiben, und denen, die ihn lesen und verstehen wollen. (Das müssen bei größerem zeitlichem Abstand nicht unbedingt verschiedene Personen sein.)

Deshalb war es nur logisch, dass einer der ersten Schritte nach der Erfindung des Computers darin bestand, die kryptischen Zahlenfolgen der frühen Programme durch sogenannte *mnemotechnische Codes* etwas lesbarer zu machen. Statt der Zahlenfolge `0500020` konnte man dann beispielsweise `ADD 20` schreiben, um den Inhalt der Speicherzelle 20 zum Inhalt des Akkumulatorregisters zu addieren.

In der Programmierung haben Namen allerdings nie die ihnen gebührende Aufmerksamkeit gefunden. Bis heute werden Sie in der Literatur fast nichts darüber finden, wie Sie effektiv zu ausdrucksstarken Namen kommen können. Clean Code betont zwar immerhin die Wichtigkeit guter Bezeichnungen, hält aber, wenn es darum geht, solche zu finden, auch keine wirkliche Hilfe bereit. Das ist schade, denn es handelt sich um ein Kernprinzip, das alles andere als einfach umzusetzen ist.

Namen versus Bezeichner

In der Informatik spricht man statt von *Namen* häufig auch von *Bezeichnern* oder *Identifikatoren*. Gemeint ist damit immer eine bestimmten Regeln folgende Zeichenfolge, die ein an sich abstraktes Element eines Programms möglichst eindeutig identifiziert. Ich habe dabei ganz bewusst den Begriff »Zeichenfolge« und nicht »Wort« gewählt, weil das lange die vorherrschende Sicht auf Namen war und in einigen Bereichen notwendigerweise immer noch ist.

Bezeichner in Programmiersprachen

Da Speicherplatz teuer war, gab es ursprünglich in fast allen Programmiersprachen Längenbeschränkungen für Bezeichner. Üblicherweise waren nur sechs oder acht Zeichen erlaubt, bei kleinen Computern auch weniger. Manchmal wurde zwischen Groß- und Kleinbuchstaben unterschieden, manchmal nicht.

Immerhin erlaubten es einige Sprachen, dass im Code längere Bezeichnungen verwendet werden konnten. Allerdings mussten sich diese dann in den »signifikanten« ersten Stellen unterscheiden, weil nur diese vom Compiler oder Interpreter berücksichtigt wurden.

Erst mit Java wurde der Verwendung beliebig langer Bezeichner wirklich zum breiten Durchbruch verholfen. Relikte aus der »guten alten Zeit« können Sie aber noch an vielen Stellen finden, beispielsweise in der Gewohnheit, Abkürzungen durch Weglassen von Vokalen zu gewinnen, etwa `List` zu `Lst` oder `println` für `printline`.

Namen versus Begriffe

> Bei den Bezeichnungen ist darauf zu achten, dass sie für das Erfinden bequem sind. Dies ist am meisten der Fall, so oft sie die innerste Natur der Sache mit Wenigem ausdrücken und gleichsam abbilden. So wird nämlich auf wunderbare Weise die Denkarbeit vermindert. (G. W. Leibniz)

Wahrscheinlich haben Sie selbst schon öfter bemerkt, dass es vielfach gar kein Problem ist, passende Namen zu finden. Bei anderer Gelegenheit erscheint es dagegen fast unmöglich. Die Ursache für diesen Unterschied lässt sich leicht erklären. Einmal haben Sie es mit Denkstrukturen oder Konzepten zu tun, die nicht nur Ihnen bereits bekannt sind. Im anderen Fall beackern Sie echtes Neuland und haben erste Konzepte oder Begriffe geprägt.

Was Ihnen in dieser Situation noch fehlt, ist ein Name, der dem neuen Begriff Ausdruck verleiht. Wissenschaftler haben es da im Allgemeinen etwas einfacher. Sie nehmen ein ihnen passend erscheinendes Wort (oder eine Phrase) und definieren es als Namen des neuen Begriffs. Von Ihnen als Softwareentwickler wird hingegen erwartet, dass Sie nicht nur den neuen Begriff inhaltlich

kreieren, sondern auch noch einen Namen finden, der allen Lesern sofort klarmacht, was Sie damit meinen.

Der Name soll implizit die Definition mitliefern. Das wird Ihnen nur selten auf Anhieb gut gelingen und zuweilen überhaupt nicht. Angesichts der Schwere dieser Aufgabe sollten Sie jedoch nicht resignieren. Versuchen Sie, so weit zu kommen, wie es geht. Üben hilft, und Sie werden dabei überdies das ursprüngliche Problem besser verstehen lernen.

- ✔ Diskutieren Sie im Team die zu benennenden Konzepte.
- ✔ Benutzen Sie Synonymverzeichnisse und sammeln Sie Vorschläge. Nutzen Sie dabei das Wissen derjenigen, die mit dem Anwendungsgebiet vertraut sind.
- ✔ Versuchen Sie, den neuen Begriff zu definieren – selbst wenn Sie noch keinen Namen dafür haben.
- ✔ Versuchen Sie, das Konzept einem völlig unbeteiligten Menschen zu erklären. Benutzen Sie dabei Analogien. Das kann der Suche nützliche Impulse geben.
- ✔ Wenn Sie sich für einen Vorschlag entschieden haben, verwenden Sie diesen konsequent in allen Dokumenten.
- ✔ Konsequent sein heißt allerdings nicht, dass Sie einen Namen nicht später durch einen besseren ersetzen sollten.

Gerade zu Beginn eines Projekts werden Sie häufig feststellen, dass eine gewählte Bezeichnung doch nicht so gut passt. Das ist natürlich und der Weg der Erkenntnis. Vermeiden Sie aber um jeden Preis, dass der alte, schlechtere Name noch irgendwo erhalten bleibt. Nichts ist verwirrender als Code, bei dem man anhand der verwendeten Namen erkennen kann, in welcher Projektphase er entstanden ist. Ein unbefangener Leser kennt die Entstehungsgeschichte ja nicht und wird durch die unterschiedlichen Namen irritiert.

Selbstverständlich ist es nützlich, wenn Sie eine Sammlung der Definitionen oder gar eine komplette Ontologie anlegen. Der Aufwand dafür lohnt jedoch nur dann, wenn Sie sicherstellen können, dass diese Verzeichnisse über die gesamte Softwarelebenszeit gepflegt und genutzt werden.

Begriffe und Bezeichnungen

In der Umgangssprache wird unglücklicherweise kaum noch zwischen *Begriff* und *Bezeichnung* unterschieden. Lediglich in Redewendungen wie »etwas begriffen haben« oder »sich keinen Begriff machen« scheint die echte Bedeutung noch durch.

Wenn man von etwas »einen Begriff hat«, heißt das nicht unbedingt, dass man auch ein Wort dafür hat, um ihn zu benennen. Ein Begriff ist eine Einheit des Denkens. Begriffe umfassen Bedeutungen, Konzepte oder Ideen, die unabhängig von sprachlichen Ausdrücken existieren und abgegrenzt werden können.

Wörter, also die Namen oder Bezeichner, sind dabei eine Art Henkel oder Beschriftung, deren sich die Menschen bei der Kommunikation bedienen. In diesem Sinn wird das Wort »Begriff« hier verwendet.

Woher nehmen?

Grundsätzlich gibt es zwei Bereiche, aus denen Sie die Bestandteile neuer Namen nehmen können. Ihre Entscheidung, welchen Sie wählen, sollte dabei vor allem davon abhängen, ob es vorwiegend um fachliche oder eher implementationstechnische Fragen geht.

Lösungsdomäne

Das ist die Domäne, in der die Entwickler zu Hause sind. Da finden Sie Bezeichnungen wie *Collection*, *Map* und *Tree*. Ebenso gehören die bekannten Algorithmen der Informatik und die gängigen Entwurfsmuster dazu. Vergessen Sie nicht, Ihr Programm soll vorrangig von anderen Entwicklern verstanden werden. Nutzen Sie deshalb so weit wie möglich die Fachsprache der Softwareentwicklung.

Vermeiden Sie es aber auch, dabei unnötige Implementierungsdetails einfließen zu lassen. Eine *LinkedList* sollte wirklich nur dann so heißen, wenn es vom Konzept her wesentlich ist, gerade diesen Listentyp und keinen anderen zu verwenden.

Anwendungsdomäne

Software wird entwickelt, um irgendein Problem aus einem Anwendungsbereich zu lösen. Dementsprechend können Sie es kaum vermeiden, dass Ihr Code Funktionen enthält, deren Sinn sich nur aus der Kenntnis des Anwendungsfalls ergibt. Das sollte sich dann auch in den Namen widerspiegeln.

Wenn Sie beispielsweise Zuschläge berechnen müssen, ist es verständlicher, Sie nennen die `ZuschlagNachParagraf17` und `ZuschlagNachVerordnung22` als `Zuschlag1` und `Zuschlag2`.

Der Nachteil solcher Namen ist, dass sie eventuell die algorithmische Struktur verdecken können. Es ist ein Teil der Kunst, guten Code zu schreiben, die richtige Balance zwischen Namen aus der Lösungs- und solchen aus der Anwendungsdomäne zu finden.

Eigenschaften guter Namen

Sie wissen jetzt, dass es schwer sein kann, gute Namen zu finden. Immerhin gibt es ein paar Kriterien, anhand derer Sie überprüfen können, ob Sie einen guten Namen gefunden haben.

Den Sinn vermitteln

Durch Namen können Sie dem Leser des Codes den Bezug zur ursprünglichen Aufgabe vermitteln, und zwar nur dadurch – wenn Sie Kommentare und Dokumentationen außer Betracht lassen. Ein Stück Code

```
a.add(b);
```

sagt Ihnen bestenfalls, dass zu einem Containerobjekt `a` ein Objekt `b` hinzugefügt wird. Daran ändert sich auch nichts, wenn Sie längere Namen verwenden, solange diese keinen Bezug zur Aufgabe haben. Ganz anders sieht es aus, wenn Sie lesen

```
auftrag.add(teilauftrag);
```

Die Tatsache, dass das bewusste Containerobjekt einen aus Teilaufträgen bestehenden Auftrag realisiert, könnten Sie ansonsten nur durch Kommentare erklären. Im Gegensatz zu einem Kommentar bei der Deklaration

```
Container a; // Auftragsobjekt
```

wird durch einen guten Namen bei jeder Verwendung dieser Sinn gleich mitgeliefert.

Das heißt natürlich auch, Namen konsistent zu benutzen. Wenn Sie das nicht tun, stellen sich dem Leser sofort eigentlich überflüssige Fragen, zum Beispiel: Was unterscheidet einen `Teilauftrag` von einem `Unterauftrag` oder einem `Auftragselement`? Unglücklicherweise enthalten die Java-Bibliotheken selbst solche Inkonsistenzen, beispielsweise für das Löschen (`delete`, `remove`) oder den Umfang von Verbundobjekten (`size`, `length`, `lengthOf`).

Nicht in die Irre führen

Wenn Namen wie dargestellt, den außerhalb des Codes liegenden Sinn ausdrücken sollen, dann ist es sehr wichtig, dass sie das korrekt tun. Ein in die Irre führender Name richtet mehr Schaden an als eine nichtssagende Bezeichnung. Bei Letzterer ist sofort klar, dass sich der Leser die Bedeutung aus dem Kontext erschließen muss. Das muss er bei einem irreführenden Namen letzten Endes auch, aber vorher verschwendet er seine Zeit damit, die Irreführung aufzudecken.

Ein eigener Punkt sind in vorrangig fachlich geprägtem Code Endungen, die – ohne Not – auf die Art der Implementierung verweisen, zum Beispiel `AuftragsList` (besser

`Auftragsverzeichnis`) oder `RechnungsMap` (besser `Rechnungszuordnung`).

Generell sollten Sie solche Namen bei fachlichem Bezug nur mit großer Zurückhaltung verwenden, weil dadurch häufig Implementierungsdetails viel zu früh, das heißt auf einer zu hohen Ebene, festgelegt werden. Wenn es aber fachlich notwendig ist, dann muss eine `AuftragsListe` auch wirklich eine Liste sein. Falls sich der Typ dann doch einmal ändern sollte, müssen Sie unbedingt auch die Namen entsprechend anpassen. Überlegen Sie deshalb genau, ob Sie wirklich auf diese Art der Namensbildung angewiesen sind.

Irreführende Namen sind meistens die Folge von Überarbeitungen und Umbauten.

Prüfen Sie nach jeder Änderung, ob die Namen noch passen. Wenn das nicht der Fall ist, ersetzen Sie die betroffenen Namen durch bessere.

Sinnvolle Unterschiede

Namen sollen die Dinge unterscheidbar machen. Das ist leicht gesagt. Hin und wieder treffen Sie jedoch auf eine Situation, in der Sie unterschiedliche Objekte haben, für die auf den ersten Blick der gleiche Name angemessen erscheint.

Eine verbreitete Unsitte besteht darin, dann einen rein lexikalischen Unterschied zu erzeugen, zum Beispiel durch `object` und `objekt` – keine Erfindung von mir. Sie können sicher sein, im resultierenden Code wird dann an mindestens einer Stelle der falsche Name stehen.

Diese Stelle zu finden, ist eine Höllenarbeit, weil das mentale System des Menschen einfach nicht auf derartige kleine Unterschiede ausgerichtet ist. Für einen Compiler ist das kein Problem, für Sie als Entwickler, der den Code verstehen will, aber ein gewaltiges, weil Sie dabei versuchen müssen, die unbewusste Fehlerkorrektur Ihres kognitiven Systems zu unterdrücken.

Redundanz in Wörtern

Zumindest die indoeuropäischen Sprachen sind so aufgebaut, dass es nur sehr wenige Wörter gibt, die sich in genau einem Buchstaben unterscheiden, wenn man von Flexionen (beispielsweise des Baums, dem Baum), die sich meistens auch aus dem Kontext erschließen lassen, absieht. Die geschriebene Sprache ist also hochgradig redundant.

Eine Studie aus dem Jahr 1976 weist nach, dass Texte durch Verfremdung nicht sofort unlesbar werden, wenn die Wortlänge nicht verändert wird und der erste und der letzte Buchstabe erhalten bleiben. Versuchen Sie es selbst an folgendem Beispiel: »Luat enier sidtue an eienr elgnhcsien uvrsnäiett, ist es eagl in wcheler rhnfgeeloie die bstuchbaen in eniem wrot snid.«

Das kognitive System ist offensichtlich in der Lage, solche Fehler stillschweigend zu korrigieren. So nützlich dies im Alltag sein mag, bei der Fehlersuche ist es ein nicht zu unterschätzendes Handicap.

Eine andere schlechte Praxis ist es, Namen durch mehr oder weniger nichtssagende Bestandteile unterscheidbar zu machen, indem man beispielsweise `Data` oder `Info` anhängt. Sehr beliebt ist es auch, Klassen `IrgendeinManager` zu nennen.

Sicher mag es manchmal einen akzeptablen Grund für derartige Benennungen geben. Sie sollten aber ein gewisses Misstrauen dagegen entwickeln und sich fragen, ob es nicht doch bloß bequem ist. Besser ist es immer, einen Namen zu wählen, der möglichst genau beschreibt, worum es sich handelt.

Die genannten Zusätze `Data`, `Info` oder `Manager` sind dabei nicht grundsätzlich abzulehnen. Derartige Anfügungen können für die Kennzeichnung technischer Unterschiede sogar notwendig sein. Wenn Sie beispielsweise eine Klasse `Connection` haben, ist es vielleicht sinnvoll, eine Klasse `ConnectionData` für die wesentlichen Daten der `Connection`-Objekte zu haben. Verwenden Sie dann aber den gewählten Zusatz konsistent nur in diesem einen Sinn.

Eine ernsthafte Schwierigkeit will ich Ihnen nicht verschweigen. Die ergibt sich, wenn Sie zwischen einem und mehreren Objekten

unterscheiden müssen:

```
Account account = getAccount();
Accounts accounts = getAccounts();
```

Das ist einfach leicht zu verwechseln und einer der Gründe, weshalb Sie sehr oft etwas in der Art wie

```
AccountList accountList = getAccountList();
```

finden. Wenn es sich wirklich um eine Liste handelt, mag das als Kompromiss hinnehmbar sein. Eine wirklich gute Lösung kann ich Ihnen leider auch nicht präsentieren. Vielleicht finden Sie eine geeignetere Endung. Doch ganz gleich wofür Sie sich entscheiden, verwenden Sie dann immer diese Variante.

Verschlüsselungen vermeiden

Als es noch keine integrierten Entwicklungsumgebungen (IDE) gab, wurden Namen vielfach benutzt, um in ihnen zusätzliche Informationen zu verschlüsseln. Beispielsweise indem Parameternamen ein »p« und Feldern (members) ein »m« vorangesetzt wurde. Dafür gibt es heute keinen Grund mehr. Solche unnötigen Kennzeichen sind beim Lesen eher hinderlich. Sie sollten sie deshalb vermeiden.

Ebenso sollten Variablennamen keine Informationen über den Typ enthalten. Wenn Sie diese Angabe wirklich brauchen, liefert sie Ihnen die IDE. Und es enthebt Sie der Notwendigkeit, nach einer Typänderung oder Typumbenennung auch noch alle betroffenen Variablen umbenennen zu müssen.

Eine Ausnahme von dieser Regel liegt allerdings vor, wenn Sie einen Wert ganz bewusst in zwei Formen benötigen, weil zum Beispiel, wie beim Datum, die Konvertierung zu aufwendig ist, um sie wiederholt durchzuführen:

```
Date datum;
String datumAsString;
```

Abschließend muss ich noch auf Interfaces zu sprechen kommen. Im Allgemeinen sollten Sie diese nicht explizit kennzeichnen. Vor

allem darum, weil es sich letztlich um ein Implementierungsdetail handelt. Für große Teile des Codes ist es schlichtweg unerheblich, ob es um ein Interface oder eine Klasse geht.

Möglicherweise werden Sie sogar einmal aus einer Klasse ein Interface extrahieren. Dann sparen Sie viel Aufwand, wenn Sie keine spezielle Namensreglung befolgen und keine Umbenennung vornehmen müssen.

Allerdings gilt auch hier: Konsistenz ist wichtig. In einer Umgebung, wo beispielsweise wie bei .NET die Konvention herrscht, Interface-Namen ein großes I voranzustellen, sollten Sie sich daran halten. Die resultierenden Nachteile sind meist geringer als die Verwirrung durch uneinheitliche Benennungen.

Verwendbarkeit

Nicht vernachlässigen sollten Sie bei der Auswahl von Namen deren praktische Verwendbarkeit, allerdings als nachrangigen Aspekt. Wichtig ist zuerst immer die Ausdruckskraft, aber wenn Sie unter verschiedenen Vorschlägen auswählen können, sollten Sie die folgenden Gesichtspunkte berücksichtigen. Die Reihenfolge sagt dabei nichts über die Priorität aus, denn die kann je nach Situation variieren.

Aussprechbare Namen

Namen sollten grundsätzlich aussprechbar sein. Erstens kann das Gehirn mit aussprechbaren Wörtern besser umgehen, und zweitens erleichtert es die Diskussion über die Benennung von Begriffen und über den Code im *Code Review*. (Code Reviews werden in Kapitel 20 *Miteinander Lernen – Code Reviews* besprochen.)

Länge

Längere Wörter können mehr Informationen vermitteln, aber sie verlängern den Code und verursachen dadurch häufigere Zeilenumbrüche, wodurch unter Umständen das Erkennen von Strukturen erschwert wird. Außerdem müssen Sie einfach mehr Text lesen. Beides *kann* die Lesbarkeit verschlechtern.

Optische Unterscheidbarkeit

Die Namen sollten bereits auf den ersten Blick möglichst gut unterscheidbar sein, um Verwechslungen zu vermeiden. Gerade dadurch, dass moderne IDE nach dem Eintippen der ersten Zeichen eine Liste möglicher Fortsetzungen anbieten, besteht eine große Gefahr, dass Sie bei zu ähnlich aussehenden Wörtern irrtümlich das falsche auswählen. Klar unterscheidbare Namen können sich alle Beteiligten überdies deutlich einfacher merken.

Suchbarkeit

Ganz gleich, ob es darum geht, einen Fehler zu finden oder eine Ergänzung zu implementieren, als Erstes müssen Sie einen geeigneten Einstiegspunkt für Ihre Codeanalyse finden. Dabei ist es von Vorteil, wenn Sie einen Namen oder einen Namensteil haben, nach dem Sie suchen können.

Natürlich wünschen Sie sich dabei möglichst wenige »falsche« Treffer. Sie können sich vorstellen, dass eine Suche nach `get` kaum ein brauchbares Ergebnis liefern wird. Deshalb sollten Sie Namen bevorzugen, die eine hohe Signifikanz für die jeweilige Funktion aufweisen.

Außerdem ist klar, dass Namen für lokale Variablen nur innerhalb ihres Kontexts aussagekräftig sein müssen. Wobei auch dabei Unterschiede zu beachten sind. Während es beispielsweise völlig angebracht sein kann, wenn Sie eine Indexvariable `i` oder den Parameter der `equals`-Methode `o` nennen, sollten Sie im Allgemeinen doch ausdrucksstärkere Namen verwenden.

Bei den privaten Feldern einer Klasse sollten Sie zudem bei der Benennung bedenken, dass diese möglicherweise über Getter- und Setter-Methoden in einem deutlich weiteren Bereich sichtbar sind und dort verstanden werden müssen.

Klassen und Methoden

Klassennamen sollten Substantive sein. Versuchen Sie, Ausdrücke mit `Manager`, `Processor`, `Data` oder `Info` zu vermeiden.

Sie sind zu unspezifisch, um wirklich zum Verständnis beizutragen.

Versuchen Sie, Klassennamen so spezifisch zu wählen, dass es sehr unwahrscheinlich ist, dass der gleiche Name an anderer Stelle nochmals verwendet wird. Gleiche Klassennamen in unterschiedlichen Paketen wirken irritierend, weil sie im Code gewöhnlich ohne Paketpräfix verwendet werden.

Es erschwert überdies das Verständnis, wenn Sie als Leser unterschiedliche Konzepte mit gleichem Namen auseinanderhalten müssen.

Die Namen von Methoden sollten die jeweilige Wirkung durch ein Verb oder eine Verbalphrase beschreiben. Dabei wird logischerweise der Name umso länger sein, je spezifischer die Funktion einer Methode ist. Haben Sie keine Angst vor langen Methodennamen, denn diese machen den Code verständlicher. Im Idealfall liest sich eine Folge von Methodenaufrufen dann wie eine Handlungsanweisung.

Verwenden Sie immer nur ein Wort für ein Konzept und umgekehrt. Nichts ist irritierender, als wenn ein Wort für zwei verschiedene Begriffe verwendet wird. Fast genauso lästig ist es, wenn Sie für ein und dasselbe Konzept willkürlich verschiedene Bezeichnungen verwenden.

Konsistent zu benennen, kann durchaus einiges Nachdenken erfordern, zumal die Java-Bibliotheken auch in dieser Hinsicht nicht vorbildlich sind. Es ist beispielsweise gar nicht so leicht, eingängig zu beschreiben, unter welchen Bedingungen eine Methode, die etwas hinzufügt, `add`, `append`, `put` oder `insert` heißen sollte.

Die Qual der Sprachwahl

Entwickler außerhalb des englischen Sprachraums stehen zusätzlich vor der Frage, welche Sprache sie für ihre Namen wählen sollen – Englisch oder die jeweilige Landessprache? Die Antwort darauf ist nicht einfach und hängt von verschiedenen Faktoren ab.

Englisch

Englisch als Grundsprache der Namen hat einige Vorteile:

- ✔ Es passt sich nahtlos in den durch die Programmiersprache gegebenen sprachlichen Kontext ein.
- ✔ Es gibt keine Probleme mit Umlauten, Akzenten, Tilden oder anderen Buchstabenbesonderheiten.
- ✔ Es gibt kaum Probleme mit Beugung und Zusammensetzungen.
- ✔ Die Wörter sind relativ kurz und gut kombinierbar.
- ✔ Es ist gut geeignet, um in multinationalen Teams zu arbeiten.

Deutsch

Die Verwendung der jeweiligen Landessprache, also des Deutschen, bietet ebenfalls Vorteile:

- ✔ Es sind keine guten Kenntnisse des Englischen bei den Entwicklern notwendig. (Es entfallen auch Probleme, die sich – eher selten – aus dem Unterschied zwischen englischem und amerikanischem Sprachgebrauch ergeben können.)
- ✔ Es ist keine Übersetzung von Fachbegriffen erforderlich.
- ✔ Benennungen sind durch das ausgeprägtere Sprachgefühl oft präziser möglich.
- ✔ Es gibt keine Probleme mit unterschiedlicher Begrifflichkeit. Vor allem in stark reglementierten Anwendungsbereichen (Recht, Finanzen) ist es oft schwierig bis unmöglich, hinreichend genaue Übersetzungen zu finden, wenn es

beispielsweise kein vergleichbares Konzept im englischen Sprachraum gibt.

Insbesondere der letzte Punkt kann entscheidend sein, sich – zumindest bei wichtigen Begriffen – für die Landessprache und damit für einen Sprachmix zu entscheiden.

Keine Empfehlung

Die Entscheidung liegt bei Ihnen. Falls Sie in einem Bereich arbeiten, in dem keine oder nur wenige nationale Besonderheiten zu beachten sind, wird Englisch im Allgemeinen eine gute Wahl sein.

Andernfalls müssen Sie festlegen, wie weit Sie mit den deutschen Bezeichnungen gehen wollen. Denn hybrid wird diese Lösung stets sein, weil Sie ja durch die Programmiersprache und die verwendeten Bibliotheken englische Bestandteile nicht vermeiden können.

Schließlich kann es noch passieren, dass Sie Englisch verwenden müssen, obwohl die Fachlichkeit sehr spezifisch ist. Legen Sie dann eine möglichst für das Gesamtprojekt verbindliche Übersetzungsliste an, am besten einschließlich kurzer Definitionen, pflegen Sie diese sorgfältig und sorgen Sie dafür, dass wirklich alle Entwickler damit arbeiten.

Was zu tun ist

Der Ratschlag, gute Namen zu wählen, ist leicht gegeben. Wenn Sie ihn in der Praxis erfolgreich umsetzen wollen, sollten Sie zuerst einmal die damit verbundenen Schwierigkeiten akzeptieren.

Um Ihren Code durch passende Namen besser lesbar zu machen, muss diese Aufgabe als integraler Teil des Entwicklungsprozesses verstanden und eine entsprechende Kultur etabliert werden.

- ✔ Jeder Entwickler muss spüren, dass Namen wichtig sind.

- ✔ Die Projektverantwortlichen müssen den Wert guter Namen kennen und würdigen.
- ✔ Es muss akzeptiert werden, dass das Finden guter Namen Überlegung und Zeit braucht und oft nicht auf Anhieb gelingt.
- ✔ Namen müssen immer wieder im Team diskutiert werden. Ihr Nutzen beruht auf einem gemeinsamen Grundverständnis, das nur so entstehen kann.
- ✔ Konventionen müssen vereinbart und eingehalten werden.
- ✔ Die Verwendung guter und einheitlicher Namen muss in den Anforderungsdokumenten beginnen.

Selbst wenn Sie sich entscheiden sollten, das Clean-Code-Konzept nicht vollständig umzusetzen, kann Ihnen das Befolgen dieser Ratschläge helfen, die Codequalität zu verbessern. Schließlich geht es im Kern darum, gedankliche Klarheit über die zu lösende Aufgabe zu erreichen. Das ist immer nützlich.

Das Wichtigste in Kürze

- ✔ Verständliche und einleuchtende Bezeichnungen sind ein wichtiges Kernprinzip von Clean Code.
- ✔ Konzeptionelle Klarheit ist eine notwendige Voraussetzung für das Finden verständlicher Namen.
- ✔ Gute Namen sind unter anderem selbsterklärend, aussprechbar, nicht leicht zu verwechseln und möglichst eindeutig.
- ✔ Die Wahl der Sprache, in der die Namen gebildet werden sollen, muss in jedem Einzelfall gut überlegt werden. Es gibt dafür keine perfekte Lösung.
- ✔ Namen und die dahinterstehenden Konzepte bilden das begriffliche Gerüst der im Code beschriebenen Funktionalität. Das muss sich in der Entwicklungskultur widerspiegeln.

Kapitel 11

Reine Formfrage – Formatierung

IN DIESEM KAPITEL

- Richtige Größe von Klassen
- Vertikale und horizontale Codeformatierung
- Vor- und Nachteile automatischer Formatierungen

Während Namen für die Funktion des Programms noch eine gewisse Bedeutung haben – die sich allerdings in der Identifizierung der benannten Objekte erschöpft –, geht es in diesem Kapitel um eine Codeeigenschaft, die dem Computer völlig gleichgültig ist, die Formatierung. Ebenso wie es Kommentare sind, ist die zweidimensionale Struktur nur für den menschlichen Leser von Code relevant.

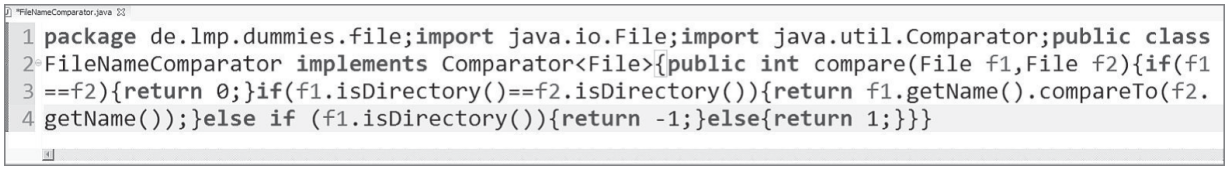
Das Auge liest mit

Mit der nicht ganz ernst gemeinten Überschrift will ich Sie daran erinnern, dass Code deutlich mehr ist als eine lineare Zeichenfolge, wie Sie sie etwa bei einem Prosatext haben. In einem natürlichsprachlichen Artikel beeinflussen Zeilenlänge, Umbrüche und Wortabstände zwar ebenfalls die Lesbarkeit. Bei Code und anderen Formalismen – denken Sie zum Beispiel an die Mathematik – ist dieser Einfluss jedoch noch bedeutend ausgeprägter.

Vergleichen Sie die beiden Darstellungen identischen Codes. Der erste Eindruck vom nicht formatierten Code in Abbildung 11.1 ist der eines unstrukturierten Klumpens. Erst bei genauer

Betrachtung, das heißt unter Aufwendung mentaler Energien, können Sie bestimmte Strukturen erkennen. Dabei bleibt allerdings die Identifizierung zusammengehörender Klammerpaare eine nahezu unlösbare Aufgabe.

Ganz anders ist das in der Abbildung 11.2. Die Struktur springt sofort ins Auge, ohne dass Sie wirklich alle Klammerpaare mühsam zusammensuchen müssen – vorausgesetzt, die Einrückungen sind korrekt.



```
package de.lmp.dummies.file;import java.io.File;import java.util.Comparator;public class
FileNameComparator implements Comparator<File>{public int compare(File f1,File f2){if(f1
==f2){return 0;}if(f1.isDirectory()==f2.isDirectory()){return f1.getName().compareTo(f2.
getName());}else if (f1.isDirectory()){return -1;}else{return 1;}}}
```

Abbildung 11.1: Unformatierter Code

```
1  package de.lmp.dummies;
2
3  import java.io.File;
5
6  public class FileNameComparator implements Comparator<File> {
7      public int compare(File f1, File f2) {
8          if (f1 == f2) {
9              return 0;
10         }
11         if (f1.isDirectory() == f2.isDirectory()) {
12             return f1.getName().compareTo(f2.getName());
13         } else if (f1.isDirectory()) {
14             return -1;
15         } else {
16             return 1;
17         }
18     }
19 }
```

Abbildung 11.2: Formatierter Code

Tatsächlich gibt es formale Sprachen, bei denen Strukturen – wahlweise oder obligatorisch – nicht durch Klammerpaare wie `begin-end`, `if-fi` oder `{-}`, sondern durch Einrückungen definiert werden. Dazu gehören Python, F# und Haskell für Programme oder YAML für Schnittstellen.

Klammern und Formatierung zusammen erhöhen die Wahrscheinlichkeit, dass Fehler schnell erkannt werden.

Überdies erleichtern Sie die Gestaltung von Tests, weil Kontrollflüsse schnell erkannt werden.

Sie sehen, Formatierung ist wichtig. Sie ist wichtig, um Code lesbar und verständlich zu gestalten. Nur solcher Code kann seine Funktion als Kommunikationsmittel erfüllen. Ihnen als professioneller Entwickler ist bewusst, dass die Lesbarkeit ebenso wichtig ist wie die funktionelle Richtigkeit.

Das mag provokant klingen. Aber denken Sie daran, dass sich die Aufgaben Ihres Codes wahrscheinlich ändern werden. Was sich im Laufe der Anpassungen nicht verschlechtern darf, ist die Lesbarkeit. Diese muss von Anfang an gegeben sein.

Wenn Sie wirklich sauberen Code produziert haben, wird diese Qualität in der Software selbst dann noch zu finden sein, wenn fast keine Zeile des ursprünglichen Codes mehr erhalten ist. Denn lesbarer Code stellt bei jeder Überarbeitung – unausgesprochen – die Forderung, diese Qualität zu erhalten. Niemand wird sich gern nachsagen lassen, den Code weniger verständlich gemacht zu haben.

Weil eine gute Formatierung viel leichter umzusetzen ist als das Finden guter Namen, sollten Sie sich dabei keine Nachlässigkeit erlauben. Es droht sonst der *Broken-Windows-Effekt*, und der Code wird erschreckend schnell verwahrlosen.

Broken-Windows-Effekt

Soziologische Untersuchungen haben gezeigt, dass ein kaputtes Fenster, das nicht schnell repariert wird, dazu führt, dass schon nach kurzer Zeit alle weiteren Fenster eines ungenutzten Gebäudes zerstört werden. Es deutet sehr viel darauf hin, dass das nicht nur für Häuser gilt, sondern dass Menschen dazu neigen, in einer Umgebung, die bereits Anzeichen von Unordnung oder Verwahrlosung zeigt, automatisch weniger korrekt zu handeln, als sie das tun würden, wenn die Umgebung aufgeräumt und sauber wäre.

Wenn Sie darauf achten und ganz ehrlich sind, werden Sie dieses Verhalten höchstwahrscheinlich auch bei sich selbst – zumindest in der Tendenz – beobachten können.

Vertikales Formatieren

So richtig ist es einem gar nicht bewusst, dass beim Formatieren zwei Richtungen mit durchaus unterschiedlichen Eigenheiten betroffen sind. Ich beginne mit derjenigen, an die man eher weniger denkt, nämlich der vertikalen Struktur.

Codelänge

Formatieren bedeutet, den eigentlich linearen Programmcode auf zweidimensionale Weise zu organisieren: vertikal und horizontal. Die erste Frage, vor der Sie dabei stehen, lautet: Wie viel Platz habe ich zur Verfügung? Für die vertikale Dimension bedeutet das, wie lang sollte eine Quellcode-Datei, das heißt für Java in der Regel eine Klasse, sein? Darauf gibt es keine einfache Antwort, weil der Umfang einer Klasse ganz wesentlich von der funktionalen Zerlegung abhängt und daher schwer durch formale Vorgaben zu steuern ist.

Erfahrungen lassen den Schluss zu, dass 200 bis 500 Zeilen eine empfehlenswerte Größe sind. Das schließt nicht aus, dass Sie bei besonders komplexen Anforderungen, die nicht vernünftig aufgeteilt werden können, als Ausnahme auch einmal mehrere Tausend Zeilen erreichen können.

Prüfen Sie auffällig lange Dateien kritisch, ob der Umfang wirklich berechtigt ist. Es gibt solche Fälle, aber oft ist auch eine Aufteilung möglich. Kleinere Dateien sind deutlich einfacher zu verstehen.

Selbstverständlich sollten Sie sich auch vor dem anderen Extrem hüten. Unmengen von sehr kurzen, nur einige Zeilen lange Dateien erleichtern das Verständnis meistens auch nicht, weil jede einzelne dann zu wenig Information enthält und Sie beim Lesen die ganze Zeit mit dem Navigieren zwischen diesen Teilstücken beschäftigt sind.

Vorbild Zeitung

Guter Code sollte wie eine ansprechende Zeitungsseite organisiert sein. Oben stehen die Schlagzeilen mit den wichtigsten Informationen in Kurzform. Es folgen Zusammenfassungen der wesentlichen Inhalte und schließlich die ausführlichen Texte.

Gegliedert wird der Text durch Teilüberschriften. An erster Stelle im Code sollten die wesentlichen Konzepte und Verfahren stehen. Je weiter Sie nach unten gehen, desto spezieller und detaillierter wird es.

Wie bei einer Zeitung, die vor allem aus kürzeren Artikeln besteht, sollte auch Ihr Code vor allem aus kürzeren Teilen bestehen und nur wenige, auf unterster Ebene angesiedelte lange Methoden enthalten. Nur wenn der Leser bereits aus der Struktur die wichtigen Punkte erkennen kann, ist der Code gut lesbar. Arrangieren Sie also Ihren Code wie ein Redakteur!

Vertikale Abstände

Wo der Zeitungsredakteur verschieden große Lettern zur sichtbaren Strukturierung verwenden kann, haben Sie als Entwickler nur Leerzeilen, also vertikale Abstände, um mehr oder weniger Zusammengehöriges zu markieren.

Trennung zwischen Konzepten

Gewöhnlich werden Sie Code von links nach rechts und von oben nach unten lesen. Dabei sind einige Zeilen stärker aufeinander bezogen als andere. Solche Gruppen aufeinander bezogener Zeilen sollten optisch durch Leerzeilen getrennt sein.

Vergleichen Sie die beiden Darstellungen in Abbildung 11.3 und Abbildung 11.4. Versuchen Sie nicht, den Code zu verstehen, sondern konzentrieren Sie sich auf den visuellen Gesamteindruck.

In Abbildung 11.3 sind einige Muster noch erkennbar, aber die Grundstruktur, also das, was Sie zuerst erkennen möchten, verbirgt sich im diffusen Ganzen. Dagegen sehen Sie in

Abbildung 11.4 sofort die Basisstruktur. Erreicht wird dies einfach dadurch, dass die einzelnen Konzepte durch Leerzeilen getrennt werden.

In den meisten Fällen ist eine Leerzeile genau das richtige Maß. Es kann aber vorkommen, dass Sie in komplexem Code der Übersichtlichkeit wegen bereits einige einzelne Leerzeilen eingefügt haben. So etwas passiert vor allem, wenn Code wenig strukturierende Anweisungen und relativ lange Namen, die zusätzliche Zeilenumbrüche verursachen, enthält.

```
1 package de.lmp.dummies;
2 import java.util.HashMap;
3 import java.util.List;
4 import java.util.Map;
5 public class Bestellannahme {
6   private Map<String, Bestellung> bestellungen= new HashMap<>();
7   public String[] getAnbieternamen() {
8     return Anbieterverwalter.getAnbieterListe().toArray(new String[0]);}
9   public Angebotsposition[] getAngebote(String anbietername) {
10    Angebotsliste angebotsliste= Anbieterverwalter.getAngebote(anbietername);
11    List<Angebotsposition> angebote= angebotsliste.getAngebote();
12    return angebote.toArray(new Angebotsposition[0]);}
13  public String startBestellung(String anbietername) {
14    String bestellId= anbietername.hashCode()+"_"+System.currentTimeMillis();
15    Bestellung bestellung= new Bestellung(bestellId, anbietername);
16    bestellungen.put(bestellId, bestellung);
17    return bestellId;}
18 }
```

Abbildung 11.3: Code ohne Leerzeilen

Bestellannahme.java

```java
package de.lmp.dummies;

import java.util.*;

public class Bestellannahme {
  private Map<String, Bestellung> bestellungen= new HashMap<>();

  public String[] getAnbieternamen() {
    return Anbieterverwalter.getAnbieterListe().toArray(new String[0]);
  }

  public Angebotsposition[] getAngebote(String anbietername) {
    Angebotsliste angebotsliste= Anbieterverwalter.getAngebote(anbietername);
    List<Angebotsposition> angebote= angebotsliste.getAngebote();
    return angebote.toArray(new Angebotsposition[0]);
  }

  public String startBestellung(String anbietername) {
    String bestellId= anbietername.hashCode()+"_"+System.currentTimeMillis();
    Bestellung bestellung= new Bestellung(bestellId, anbietername);
    bestellungen.put(bestellId, bestellung);
    return bestellId;
  }
}
```

Abbildung 11.4: Code mit Leerzeilen

Abgesehen davon, dass Sie versuchen sollten, derartigen Code durch Überarbeitungen besser zu strukturieren, ist es dann angebracht, die umfassenden Einheiten durch zwei Leerzeilen zu trennen. Denken Sie aber immer daran: Code, der das erfordert, hat sehr wahrscheinlich ein inhaltliches Problem.

Zerreißen vermeiden

Optisches Zusammenfassen von unterschiedlichen Konzepten ist der eine Fehler – die optische Trennung von inhaltlich zusammenhängendem Code ein anderer. Diese Trennung wird oft durch unnötig platzgreifende, meist völlig überflüssige, Kommentare verursacht, wie in Abbildung 11.5 bei den beiden Deklarationen.

```
public class Angebotsliste {

    /**
     * Die Liste der Angebote
     */
    private List<Angebotsposition> angebote;

    /**
     * Der Name des Anbieters
     */
    private String anbietername;

    public List<Angebotsposition> getAngebote() {
        return angebote;
    }
```

Abbildung 11.5: Störende vertikale Trennung

Denken Sie daran, der Bereich, den Sie »auf einen Blick« erfassen können, ist beschränkt. Es geht darum, die visuelle Aufnahmekapazität des Lesers nicht zu über-, aber auch nicht zu unterfordern. Zu viel Leerraum vergeudet Aufmerksamkeit und Zeit beim Lesen.

In diesem Beispiel zerstört er überdies die Abstandshierarchie, weil die beiden Deklarationen weiter auseinanderliegen als Felddeklarationen und Methodendefinitionen. In Abbildung 11.6 wird gezeigt, wie es besser aussieht.

```
public class Angebotsliste {

    private List<Angebotsposition> angebote;
    private String anbietername;

    public List<Angebotsposition> getAngebote() {
        return angebote;
    }
```

Abbildung 11.6: Angemessene vertikale Kompaktheit

Vertikale Ordnung

Jede Klasse besteht aus einer Reihe verschiedener Elemente: Importen, Definitionen, Vereinbarungen und Methoden. Außer für Importdeklarationen gibt es in Java keine syntaktischen Beschränkungen, in welcher Reihenfolge diese Elemente in einer Datei stehen. Lediglich bei der Initialisierung von statischen

Variablen muss unter Umständen auf die Reihenfolge geachtet werden.

So viel Freiheit birgt Gefahren. Es vereinfacht das Lesen enorm, wenn Sie ein Element nicht in der ganzen Datei suchen müssen. Deshalb sollten Sie die folgenden beiden Grundregeln einhalten:

- ✔ Legen Sie für die Elemente eine Reihenfolge fest und halten Sie diese strikt ein.
- ✔ Elemente eines Typs müssen immer in einer zusammenhängenden Gruppe erscheinen.

Nichts kann den Leser effektvoller irritieren, als wenn Sie die eine oder andere Feldvereinbarung zwischen Ihren Methoden verstecken. Nicht ganz so verwirrend, aber trotzdem nicht empfehlenswert ist es, Deklarationen von Konstanten, statischen und Instanzfeldern zu vermischen.

Grundsätzlich ist es egal, für welche Reihenfolge Sie sich entscheiden, aber es hat Vorteile, wenn Sie dabei nicht allzu stark von den inzwischen etablierten Gepflogenheiten abweichen. Die folgende Reihenfolge können Sie als meinen Vorschlag verstehen.

Konstanten

Üblicherweise beginnt der Klassenkörper mit der Definition der verwendeten Konstanten (`static final`). Diese sind nach Sichtbarkeit zu gruppieren, normalerweise stehen `public`-Werte zuerst und `private`-Werte am Ende.

Falls diese Reihenfolge nicht möglich ist, weil Sie interne Abhängigkeiten berücksichtigen müssen, überdenken Sie Ihren Code grundsätzlich, verwenden Sie einen statischen Initialisierungsblock oder verändern Sie die Reihenfolge. Letzteres aber nur, wenn es sich wirklich nur um sehr wenige Definitionen handelt und jede andere Lösung unvertretbar aufwendig wäre.

Statische Felddeklarationen und Initialisierungsblöcke

Die Deklaration der statischen Felder muss klar von derjenigen der Instanzfelder getrennt werden. Nicht ganz so eindeutig ist die Frage zu beantworten, ob statische Initialisierungsblöcke am besten im Anschluss an die statischen Deklarationen oder erst nach den Instanzfeldern platziert werden sollten.

Logisch spricht einiges dafür, diese Blöcke zwischen die beiden Deklarationsarten zu schieben. Aus visueller Perspektive lässt sich einwenden, dass dadurch die Instanzfelddeklarationen versteckt werden. Das gilt vornehmlich bei umfangreichen Blöcken und wenigen Instanzvariablen. Wenn Ihr Code kurz und übersichtlich ist, spielen diese Erwägungen aber keine große Rolle.

Instanzfelder

Ordnen Sie Instanzfelder – so gut das geht – nach inhaltlichen Gesichtspunkten. Was eng zusammenhängt, sollte auch zusammen stehen. Achten Sie dabei darauf, dass Ausdrücke zur Initialisierung so angeordnet werden, dass sie beim schnellen Auffinden von Deklarationen visuell nicht übermäßig stören.

Konstruktoren

Alle Konstruktoren sollten ebenfalls in einem Block erscheinen und so angeordnet sein, dass zuerst die mit der größten Sichtbarkeit und den wenigsten Parametern erscheinen.

Methoden

Auch bei den Methoden gilt jeweils der Grundsatz vom Allgemeinen zum Speziellen. Am Anfang steht stets eine Methode, die von außen aufgerufen werden kann. Es folgen die Methoden, die durch die erste aufgerufen werden, und so fort. Der Leser kann sich dann praktisch wie ein Kunde der Klasse durch den Code bewegen. Außerdem sollten Methoden, die konzeptionell eng verbunden sind, weil sie beispielsweise die gleiche Funktion realisieren, nur mit unterschiedlichen

Parameterlisten, unmittelbar hintereinanderstehen – so wie die oben erwähnten Konstruktoren.

Sicher werden Sie diesen Kurs nicht exakt befolgen können, allein schon deshalb nicht, weil Methoden sich möglicherweise wechselseitig oder zyklisch aufrufen. Versuchen Sie es trotzdem! Denken Sie an die Analogie zur Zeitungsseite, erst das Wichtigste im Überblick, dann in der detaillierten Darstellung. Und wenn Sie keinen relevanten Unterschied finden, dann ist die Reihenfolge eben nicht wichtig – kommt vor.

Horizontales Formatieren

In diesem Absatz geht es nun um das, was einem beim Thema Formatierung gewöhnlich zuerst in den Sinn kommt: Leerzeichen, Einrückungen und Zeilenumbrüche.

Zeilenlänge

In den alten Zeiten »als Programmieren noch Handarbeit war«, durfte der Code nach Zeilen zwar fast beliebig lang werden, doch für die Breite gab es harte Beschränkungen. Obwohl auch elektrische Schreibmaschinen, Drucker und Bildschirme technische Grenzen setzten, ging der wichtigste Einfluss von der in den 1930er-Jahren eingeführte Lochkarte mit ihren 80 Spalten aus.

Diese Breite liegt offenbar in einem Bereich, der gut zum Lesen geeignet ist. Für Code erscheint die Grenze inzwischen als etwas zu einschränkend:

- ✔ Durch Einrückungen gehen auf der linken Seite Spalten verloren, sodass die Augen ihren Fokus ganz automatisch ein Stück nach rechts verschieben und dort mehr Raum erfasst werden kann.
- ✔ Verständliche Namen sind in der Regel länger. Das führt bisweilen dazu, dass selbst einfache Ausdrücke zum Beispiel der Struktur `a = b + c * d;`, etwas eingerückt, nicht mehr in

die 80 Spalten passen. Umbrüche lassen sich zwar nie ganz vermeiden, verringern aber die vertikale Dichte und machen den Code dadurch schwerer lesbar.

Eine Zeilenlänge von 120 bis 130 Zeichen ist deshalb akzeptabel. Längere Zeilen sind kaum noch auf einen Blick zu erfassen und erschweren den Augen das Erkennen der Struktur.

Außerdem sollten Sie bedenken, dass diese Grenze die Zeilen als Ganzes, einschließlich aller Einrückungen betrifft. Der Text selbst sollte unabhängig davon eher maximal 80 bis 100 Zeichen umfassen. Auf jeden Fall muss horizontales Scrollen beim Lesen vermieden werden.

Horizontale Abstände

Innerhalb einer Zeile stehen Ihnen zur Strukturierung nur Leerzeichen zur Verfügung. An einigen Stellen sind diese als Trennzeichen ohnehin obligatorisch, an anderen können Sie beliebig eingefügt werden.

Im Allgemeinen ist es keine gute Idee, Zwischenräume aus mehr als einem Zeichen zu verwenden. Im Gegensatz zu Leerzeilen ist das Auge nicht besonders gut darauf trainiert, unterschiedlich breite Leerräume wahrzunehmen, ein Fakt, der beim Blocksatz bewusst ausgenutzt wird. Deshalb ist es schwierig, für die horizontale Gliederung einfache Regeln zu formulieren. Verstehen Sie meine Hinweise daher bitte nur als Anstoß, nicht als verbindliche Vorgaben.

Zusammenfassen und trennen

Benutzen Sie Leerzeichen, um enger von weniger eng Zusammengehörendem zu unterscheiden. Die Grenze ist nicht immer eindeutig definiert. Nehmen Sie als Beispiel eine Wertzuweisung:

```
stringLength=line.lenght();
```

So geschrieben wird verwischt, dass linke und rechte Seite völlig unterschiedliche Dinge verkörpern, nämlich links eine Variable

und rechts ein durch einen Ausdruck berechneter Wert und dass das Gleichheitszeichen eine wichtige Operation beschreibt. Dementsprechend ist es sinnvoll, Leerzeichen zur Gliederung zu setzen:

```
stringLength = line.lenght();
```

Weil die Wertzuweisung eine asymmetrische Operation ist, bei der die Seiten nicht vertauscht werden können, wird oft auch

```
stringLength= line.lenght();
```

geschrieben. (Die Sprachen der Pascal-Familie verwenden für die Zuweisung die aus formaler Sicht richtige, aber mit zwei Zeichen aufwendigere Schreibweise `:=`.)

Nach öffnenden und vor schließenden Klammern sollten Sie kein Leerzeichen einfügen. Diese Symbole trennen optisch angemessen. Dagegen ist nach Kommas, speziell in Listen und wie in normalen Texten, ein Leerzeichen angebracht:

```
Position(int id, String beschreibung, BigDecimal preis)
```

Kein Leerzeichen sollte auch vor und nach Punkten in zusammengesetzten Bezeichnungen stehen, weil diese eng zusammenhängen. Schließlich wird genau ein Element benannt:

```
Iterator<String> kursIterator= alleKurse.kurse().iterator();
```

Ausdrücke

Mathematische Ausdrücke stellen für die Formatierung, insbesondere in Verbindung mit ausdrucksstarken Namen eine spezielle Herausforderung dar. Nicht ohne Grund verwenden Mathematiker in ihren Formeln gewöhnlich sehr kurze Symbole, die sie vorher definieren.

Nutzen Sie gegebenenfalls eine ähnliche Vorgehensweise und verwenden Sie dabei lokale Variablen. Also schreiben Sie statt

```
gleitenderMittelwert = (gleitenderMittelwert * anzahl
      + aktuellerWert ) / (anzahl + 1);
```

vielleicht

```
gmw = (gmw*n + x) / (n+1);
```

Dem Vorteil der besseren Erfassbarkeit des mathematischen Ausdrucks stehen die weniger verständlichen Namen gegenüber. Stattdessen können Sie die Berechnung natürlich auch in eine eigene Methode mit entsprechend benannten Parametern auslagern, was mit Sicherheit der bessere Weg wäre. Dieser Vorschlag geht allerdings schon über die reine Formatierung hinaus.

Ausrichtung

Ein Code wie in Abbildung 11.7 sieht zwar beeindruckend aufgeräumt aus, bietet aber außer diesem guten ersten Eindruck keine praktischen Vorteile. Die Lesbarkeit wird nicht verbessert, weil die Aufmerksamkeit von der Gesamtstruktur abgezogen wird. Der zusätzliche Aufwand beim Formatieren zahlt sich nicht aus. Stattdessen besteht die Gefahr, dass durch Umbenennungen oder andere Änderungen das schöne Bild durch Unachtsamkeit oder Trägheit über kurz oder lang verdorben wird.

```
public class FileZipper  {

    private int             pufferLaenge = 8192;
    private byte[]          puffer       = new byte[pufferLaenge];
    private ZipOutputStream zipStream;
    private int             rootLaenge   = 0;
    public  int             anzahl       = 0;

    public FileZipper(String zipfileName) throws Exception {
        zipStream= new ZipOutputStream(new FileOutputStream(zipfileName));
    }
```

Abbildung 11.7: Unnütze Ausrichtung

Einrückungen

Einrückungen nehmen eine Sonderstellung zwischen horizontaler und vertikaler Formatierung ein. Obgleich sie zweifellos horizontale Elemente sind, dienen sie eigentlich der vertikalen Strukturierung, indem Sie zusammenhängende Zeilenblöcke visualisieren.

Noch wichtiger ist aber ihre Funktion bei der Sichtbarmachung der hierarchischen Gestalt des Codes. Einrückungen – richtig angewandt – können die syntaktische Struktur formaler Texte spiegeln, ganz gleich, ob es sich um Programme oder Daten handelt. Dieses Potenzial, welches das Lesen enorm erleichtert, muss unbedingt genutzt werden.

Regeln

- ✔ Wenden Sie Einrückungen konsistent an. Ganz gleich, ob Sie pro Ebene um zwei, drei oder vier Stellen einrücken, stellen Sie sicher, dass Elemente der gleichen Ebene immer um die gleiche Anzahl Stellen eingerückt sind.
- ✔ Rücken sie nicht in zu großen Schritten ein. Mehr als vier Zeichen sind selten angebracht, weil die sonst notwendige seitliche Fokusverschiebung der Augen den Lesefluss beeinträchtigt.
- ✔ Korrigieren Sie die Einrückungen nach Änderungen am Code. Das erhält nicht nur die Lesbarkeit, sondern unterstützt auch eine optische Kontrolle, ob nicht doch eine schließende Klammer an der falschen Stelle gelandet ist.
- ✔ Wenn Sie doch einmal den Eindruck haben, an einer Stelle auf die Einrückungen verzichten zu können und vielleicht alles in eine Zeile zu schreiben, überlegen Sie bitte noch einmal, welchen Vorteil das wirklich bringen würde. Oder sparen Sie sich diese Zeit und formatieren Sie es gleich mit korrekten Einrückungen.

Zeilenumbrüche

Hin und wieder werden Sie nicht darum herumkommen, Codezeilen umzubrechen, weil sie zu lang sind. Als Folge der Verwendung verständlicher Bezeichnungen, vor allem wenn diese deutsch sind, wird das relativ häufig passieren. Da solche Umbrüche die Lesbarkeit stark beeinflussen, sollten Sie diese bewusst gestalten.

Die Fortsetzungszeilen müssen zusätzlich eingerückt werden, und zwar so, dass sie sich von der vorhandenen Einrückungsstruktur sichtbar abheben. Dabei sollten Sie die innerhalb der Zeile vorhandene Struktur berücksichtigen. Das kann weitere Umbrüche und Einrückungen sinnvoll machen. Denken Sie immer daran, dass es dem Leser leicht gemacht werden soll, die Struktur zu erkennen.

Besondere Beachtung verdienen umgebrochene Parameterlisten. Wenn Sie beispielsweise folgendermaßen umbrechen und an der Position der Klammer ausrichten

```
mmmmmmmmm(ppppppppppppppp,
          ppppppppppppppp);
```

wird das nach einer Umbenennung möglicherweise so aussehen

```
mmm(ppppppppppppppp,
          ppppppppppppppp);
```

und damit nicht mehr Ihrer Absicht entsprechen.

Deshalb empfehle ich dringend, Regeln für den Zeilenumbruch immer so zu wählen, dass das Resultat stabil gegen Längenänderungen der Bezeichner ist. Das kann zum Beispiel durch eine jeweils feste Anzahl von Leerzeichen erfolgen.

Schlecht strukturierter Code wird auch durch aufwendiges Formatieren nicht leicht verständlich. Beginnen Sie immer damit, Ihre Codestruktur zu verbessern. Eine ordentliche Darstellung hilft Ihnen dabei.

Bereiche, die sich nur schlecht so darstellen lassen, dass sie eingängig sind, haben eine Überarbeitung verdient.

Automatische Formatierung

Es gibt verschiedene Werkzeuge, die Ihren Code automatisch formatieren oder die Formatierung prüfen können. In manchen

Projekten ist es üblich, den Code beim Speichern oder beim Einchecken in die Versionsverwaltung damit zu bearbeiten.

Vorteile

Für die automatische Formatierung sprechen gute Gründe:

- ✔ Der Code ist stets einheitlich und den vereinbarten Regeln entsprechend formatiert.
- ✔ Die Entwickler brauchen keine Zeit und keine Gedanken auf die Formatierung zu verwenden.

Diese Vorteile kommen vor allem bei großen Teams und häufigem Mitarbeiterwechsel zum Tragen. Weitere Gründe für den Einsatz solcher Werkzeuge können sich aus den Persönlichkeiten der beteiligten Entwickler ergeben.

Junge, noch unerfahrene Programmierer unterschätzen oft die Wichtigkeit der äußeren Form und halten dann die Regeln nur ungenügend ein. Profis mit jahrelanger Erfahrung sind manchmal unwillig, von ihrem gewohnten Verhalten abzuweichen. In beiden Fällen ist die Automatisierung hilfreich, weil sie lästige Diskussionen erspart.

Nachteile

Der wichtigste Nachteil automatischer Formatierung liegt allerdings darin, dass die Entwickler nicht mehr gezwungen sind, sich explizit mit dieser Frage auseinanderzusetzen. Sauberer Code soll aber ganz bewusst geschrieben werden. Es ist ein Zeichen der angestrebten Professionalität, Code optisch so zu gestalten, dass er gut lesbar ist. Dazu müssen Sie Disziplin entwickeln. Das geht nur durch Übung.

Weitere Probleme können sich ergeben, wenn unterschiedliche Entwicklungsumgebungen mit ebenso unterschiedlichen Formatierungswerkzeugen verwendet werden, die jeweils eigene Regelformate haben. Auch der Aufwand für die Erstellung und die Pflege der Regel-Sets sollte nicht unterschätzt werden.

Nicht vergessen dürfen Sie darüber hinaus, dass die Regeln alles über einen Kamm scheren. Ausnahmen sind nicht vorgesehen. Besonders bei Zeilenumbrüchen sind die Ergebnisse häufig unschön. Das wiederum kann dazu führen, dass Entwickler, die die Formatierung ernst nehmen, zu Hilfsmitteln wie leere Kommentare am Zeilenende zum Schutz von Zeilenumbrüchen greifen müssen, um bestimmte Regeln an einigen Stellen unwirksam zu machen.

Das Wichtigste in Kürze

- ✔ Ohne angemessene Formatierung ist Code für Menschen praktisch nicht lesbar. Formatierung ist deshalb unverzichtbar.
- ✔ Ziel einer guten Darstellung ist es, die grundlegenden Strukturen leicht erkennbar zu visualisieren. Zusammenhängendes muss zusammenhängend dargestellt werden. Unterschiedliche Konzepte erfordern eine klare Trennung.
- ✔ Die Anordnung der Elemente erfolgt dabei vom Allgemeinen zum Speziellen, sodass ein Leser nur so weit in die Tiefe vordringen muss, wie das jeweils erforderlich ist.
- ✔ Formatierungsregeln müssen im Team erarbeitet und beschlossen werden. Einmal vereinbart, sind sie konsequent anzuwenden.
- ✔ Gute Formatierung, so wichtig sie ist, hilft nicht gegen schlechten Code.

Kapitel 12

Code zuerst – sind Kommentare nötig?

IN DIESEM KAPITEL

- Die Funktion von Kommentaren im Quelltext
- Nützliche Kommentare erkennen und schreiben
- Störende Kommentare vermeiden

In diesem Kapitel erfahren Sie, warum Kommentare manchmal eher verwirren als zum Verständnis beitragen und demzufolge weniger mehr sein kann.

Außerdem wird mit der bisweilen zu hörenden Ansicht aufgeräumt, dass sauberer Code Kommentare und Dokumentationen überflüssig macht.

Obgleich Kommentare Teil des Quellcodes sind, bleibt in diesem Kapitel – in Ermangelung eines geeigneteren Namens – der Begriff *Code* dem ausführbaren Teil des Quellcodes vorbehalten.

Code allein reicht nicht

Code beschreibt zwar weitestgehend exakt, was der Computer machen soll, aber diese Beschreibung ist für Menschen meist eher schwierig zu verstehen.

Erklärung gesucht

Solange es Code gibt, standen Entwickler vor der Aufgabe, ihre Produkte anderen Entwicklern verständlich zu machen. Was liegt da näher, als dies durch Erklärungen zu versuchen. Und am

besten schreibt man die Erklärungen, wenigstens zum Teil, gleich in den Code – als Kommentare.

Über Jahrzehnte haben Programmierer gelernt, dass guter Quellcode ausführlich dokumentierter Code ist. Diese Haltung hat die gesamte Softwareindustrie so stark durchdrungen, dass Sie auch heute noch jede Menge Quellcode finden können, bei dem die Anzahl der Kommentarzeilen die der Codezeilen um ein Mehrfaches übertrifft.

In den Zeiten, als Entwickler mit engen Grenzen bei Namenslänge und verfügbarem Speicher leben mussten, mag es dafür gute Gründe gegeben haben. Aber das ist inzwischen lange her. Java ist über 20 Jahre alt. Höchste Zeit, die alten Zöpfe abzuschneiden!

Das große Missverständnis: Code spricht nur den Computer an

Das grundlegende Missverständnis besteht darin, Code nur oder vorrangig als Ausführungsvorschrift für Computer zu sehen. Code ist immer auch Kommunikationsmittel zwischen Menschen. Spätestens bei der Fehlersuche begreift selbst der blauäugigste Programmierer, dass Code für Menschen ebenfalls lesbar sein muss.

Verständlicher Code beginnt damit, dass er bereits ohne Kommentare gut strukturiert und weitgehend einleuchtend ist. Allein die Notwendigkeit einer zusätzlichen Erläuterung muss bei Ihnen die Frage aufwerfen: Hätte ich das nicht verständlicher schreiben können? Mit Sicherheit werden die Antworten darauf unterschiedlich ausfallen. Nicht alles lässt sich im Code ausdrücken. Trotzdem lautet die Schlussfolgerung, dass Kommentare nur die zweitbeste Möglichkeit sind, Code verständlich zu machen.

Der ideale Code ist derjenige, der ohne zusätzliche Erläuterungen verstanden werden kann. Allerdings lebt niemand in einer idealen

Welt, und daher kommen Sie in der Regel nicht völlig ohne Kommentare aus.


Kommentare sind nicht geeignet, kryptischen Code besser verständlich zu machen. Nutzen Sie die Zeit und schreiben Sie stattdessen lesbaren, sauberen Code!

Kommentare – hilfreich oder störend?

Wie so vieles in der Welt sind Kommentare eine zweischneidige Angelegenheit. Der Grat zwischen hilfreich und störend ist oft sehr schmal. Auf der einen Seite können Kommentare beim Verständnis wirklich helfen, wenn sie beispielsweise Gedanken und Absichten erklären, die anders nicht zu vermitteln sind.

Auf der anderen Seite sind sie Ballast, der den Quelltext verlängert, beim Schreiben und Pflegen zusätzliche Kosten verursacht und schließlich den eigentlichen Code vernebelt. Sie als Leser müssen dadurch mehr Zeichen betrachten und öfter Scrollen, insgesamt also einen höheren Aufwand betreiben, um den Sinn des Codes zu erfassen.

Achten Sie deshalb stets darauf, Kommentare so einzusetzen, dass die zu transportierende Information nicht im Rauschen überflüssigen Texts untergeht.


Fragen Sie sich bei jedem Kommentar, ob die vermittelte Information den benötigten Platz und damit den beim Lesen verursachten kognitiven Aufwand wert ist.

Kommentare lügen – oft

Sind Sie sicher, nach jeder Korrektur im Quellcode gewissenhaft geprüft zu haben, ob da nicht auch noch ein Kommentar

angepasst werden müsste? Ich kann diese Frage für mich leider nicht mit ja beantworten. Im besten Fall habe ich solche Unterlassungen später bemerkt und korrigiert, aber manchmal sind sie mir sehr wahrscheinlich auch völlig entgangen. Der Blick auf fremde Codes tröstet mich etwas, weil er zeigt, ich bin nicht allein mit diesem Vergehen.

Je öfter ein Code bearbeitet wird, desto größer wird die Wahrscheinlichkeit, dass zumindest ein Teil der Kommentare veraltet oder sogar völlig falsch ist. Manchmal irritiert das nur, weil der Widerspruch zwischen Code und Beschreibung offenbar ist. Schlimmer sind die Fälle, die den Leser auf eine gänzlich falsche Fährte führen und damit unnötig Zeit kosten.

Ein ganz spezielles Problem ergibt sich, wenn die für Kommentare verwendete Sprache von Entwicklern nur rudimentär beherrscht wird. Ich habe es selbst erlebt, dass Kommentare erst verstanden werden konnten, nachdem sie in die Muttersprache der Schreiber zurückübersetzt worden waren. Aber nicht immer werden Sie jemanden haben, der Ihnen dabei helfen kann. Deshalb ist gerade in mehrsprachigen Teams wichtig, im Code Review die Kommentare ebenso kritisch zu begutachten wie den Code.

- Lesen Sie Kommentare kritisch. Ihr Inhalt könnte veraltet sein.
- Schreiben Sie Kommentare in kurzen und klaren Formulierungen, um möglichen Missverständnissen vorzubeugen.

Sinnvolle Kommentare

Auch wenn Sie sich die allergrößte Mühe geben, gibt es Situationen, in denen Kommentare nicht vermeidbar sind. Einige Beispiele werden im Folgenden aufgezählt.

Rechtshinweise

In vielen Firmen ist es Vorschrift, jede Datei mit einem Hinweis auf Copyright und Autorschaft zu versehen. Gute IDE klappen solche Texte auf Wunsch zusammen, sodass die Lesbarkeit dadurch nicht nennenswert beeinträchtigt wird. Halten Sie solche Hinweise aber trotzdem möglichst kurz und verweisen Sie gegebenenfalls auf getrennt abgelegte vollständige Lizenz- oder Nutzungsbedingungen.

```
/* Copyright 2019 s. www.xlmlx.de/license
 *  @author J. Lampe
 */
```

Unerledigtes

Typischerweise wird beim agilen Entwickeln nicht alles sofort und vollständig erledigt. Um offene Punkte nicht aus den Augen zu verlieren, können Sie die betreffenden Stellen im Code kennzeichnen:

```
//TODO not implemented yet
```

Die IDE hilft Ihnen dabei, diese Markierungen zu verwalten, sodass Sie nicht so schnell die Übersicht verlieren.

Trotzdem ist es angeraten, die Anzahl derartiger Markierungen nicht ausufern zu lassen. Versuchen Sie, den fehlenden Code beim Refactoring zu ergänzen. Manchmal kann eine solche Markierung auch ganz gelöscht werden, weil die Aufgabe zwischenzeitlich entfallen ist oder anders gelöst wurde.

Klarstellungen und Warnungen

Nicht alles funktioniert so, wie es sollte. Es wird Ihnen immer wieder passieren, dass Sie eine Aufgabe wegen eines Fehlers im fremden Code nicht so lösen können, wie das naheliegend wäre. Nicht immer ist genügend Zeit, um auf ein Update zu warten, das die Probleme behebt. Dann müssen Sie einen Weg finden, den Fehler zu umgehen, also einen *Workaround* realisieren.

Gar nicht selten führt das zu Codekonstruktionen, die auf den ersten Blick unverständlich und unnötig kompliziert aussehen. Diese sollten Sie dann unbedingt mit einem Kommentar versehen, der andere Entwickler vor unbedachten Änderungen warnt beziehungsweise Hinweise gibt, unter welchen Umständen dieser Workaround hinfällig werden könnte, beispielsweise durch Angabe einer registrierten Fehlernummer und der Version der fehlerhaften Software.

```
// Version V2.7.0 Workaround for Bug#2557
```

Algorithmen

Manche Algorithmen sind so populär, dass es reicht, ihren Namen aufzuführen. So werden Sie zum Beispiel die Bezeichnung »Quicksort« recht leicht mit Tony Hoares berühmtem Sortierverfahren assoziieren. Nicht ganz so sicher ist das hingegen für »Timsort«, einem anderen Sortieralgorithmus, den Oracle in neueren Java-Versionen implementiert hat. Allerdings hilft eine Suche im Internet hier schnell weiter.

Bei weniger bekannten Verfahren kann es deshalb nützlich sein, neben dem Namen auch erreichbare Quellen für die Beschreibung anzugeben. Solche Hinweise sind insbesondere dann unverzichtbar, wenn es Versionen gibt, die sich in wichtigen Merkmalen unterscheiden, oder wenn ein Algorithmus noch neu oder sehr spezifisch ist.

Dabei muss der Kommentar selbst keine ausführlichen Erläuterungen enthalten. Es reicht ein Verweis auf andere Dokumente, die aber für die potenziellen Leser problemlos zugreifbar sein sollten.

```
// Hash-Berechnung optimiert auf verwendete Schluessel
// siehe Perfekte Hashfunktionen in XXX
```

Spezifikationen

Entwickler haben im Allgemeinen viele Freiheiten bei der Umsetzung ihrer Aufgaben, aber nicht immer.

Externe Vorgaben

Es gibt Bereiche, in denen aufgrund gesetzlicher oder vertraglicher Regelungen genaue Vorgaben einzuhalten sind.

In Berechnungsvorschriften des Bundesfinanzministeriums fand sich beispielsweise von 1975 bis 2003 die Aussage, dass Rechenschritte in der Reihenfolge auszuführen sind, die sich nach dem Horner-Schema ergibt. Ergänzt wurde diese Vorgabe durch detaillierte Anleitungen für die vorzunehmenden Rundungen.

Ebenso ausführlich sind die Vorschriften für die Zinsberechnung im Bankbereich. Aber auch aus Datenschutzbestimmungen können sich ganz spezielle Handlungsanweisungen ergeben, die ein Entwickler berücksichtigen muss.

Dann ist es wichtig, dass jederzeit sofort deutlich wird: Dieser Code darf, beispielsweise bei Performance-Optimierungen, nur in engen Grenzen verändert werden. Kommentare, die diese Informationen vermitteln, sind daher unerlässlich.

```
// laut EStG §32a Absatz 1 (Fassung 1.1.2018)
```

Projektanforderungen

Weil es in kleineren Projekten nicht so wichtig ist und weil auch große Projekte klein anfangen, wird viel zu häufig vergessen, dass die Zuordnung von einzelnen Anforderungen zu den umsetzenden Codeteilen und umgekehrt nicht trivial ist. Das Gleiche gilt für die zugeordneten Testfälle.

Vielleicht kommt Ihnen das Folgende bekannt vor. Es handelte sich um eine große Anwendung, an deren Entwicklung ich eine Zeit lang beteiligt war und die über hundert einzelne Eclipse-Projekte umfasste. Die Daten waren nach genau spezifizierten Regeln zu verarbeiten.

Sowohl bei der Fehlersuche als auch bei der Umsetzung von Änderungen habe ich damals schrecklich viel Zeit damit

verbracht, herauszubekommen, in welchem Dokument eine bestimmte Funktion spezifiziert wurde beziehungsweise wo überall im Code eine bestimmte Vorgabe Auswirkungen hatte.

Der Aufwand wäre noch viel größer gewesen, wenn ich nicht auf die Erinnerungen erfahrener Kollegen hätte zurückgreifen können. Um so etwas zu vermeiden, sollten Sie alle Anforderungen von Anfang an mit eindeutigen Kennzeichen versehen und diese als Kommentare in den jeweiligen Code einfügen.

Das gilt ebenso für Verweise auf Change Requests, Einträge im Konfigurationsmanagementsystem und ähnliche Unterlagen, die beim Programmieren berücksichtigt werden müssen.

Wenn Sie die Kennzeichen so wählen, dass bei einer textuellen Suche »falsche« Treffer weitgehend ausgeschlossen sind, haben Sie sich ein effektives Mittel geschaffen, das Ihnen bei der Weiterentwicklung der Software sehr viel Zeit sparen wird:

```
// #SPEC-22.51#
```

Pragmatisches

Beweggründe

Für das Verständnis von Code kann es nützlich sein, wenn Sie wissen, warum etwas auf die vorliegende Art und Weise gemacht wurde:

```
// Parameter gewährleistet die Abwärtskompatibilität
```

Schließlich kann es viele Gründe für bestimmte Entscheidungen geben. Eine kurze Erläuterung erspart es dem Leser, diese jeweils selbst herausfinden zu müssen oder gar einen unzutreffenden Grund anzunehmen.

Ganz wichtig ist es, dass Sie derartige Kommentare konsequent entfernen, wenn der erwähnte Grund einmal weggefallen ist.

Begrifflicher Kontext

Wie bereits erwähnt, ist es schon recht schwierig, gute Namen zu finden. Es kommt hinzu, dass viele Bezeichnungen mehrere

Bedeutungen haben, je nachdem, in welchem Kontext sie verwendet werden.

Oft erklärt sich dieser begriffliche Kontext zwar aus dem Code, aber wenn Mehrdeutigkeiten nicht sicher ausgeschlossen werden können, sollten Sie einen aufklärenden Kommentar verfassen.

Wie alle Kommentare muss auch dieser möglichst präzise und kurz sein. Denken Sie daran, dass Sie Quellcode für den Computer **und** den Leser schreiben.

Abgrenzung

Wenn Sie den Code strukturieren, dann unter anderem auch deshalb, um durch passende Abstraktionen die Lesbarkeit zu verbessern. Allerdings gibt es kaum Mittel, um Eigenschaften und Restriktionen zu formulieren. So gilt für viele `int`-Ausdrücke, dass sie für Werte in der Nähe von `Integer.MAX_VALUE` oder `Integer.MIN_VALUE` nicht das erwartete Ergebnis liefern.

Sie werden trotzdem nur höchst selten einen Hinweis auf solche Einschränkungen finden. Solange das nur die Extremwerte betrifft, mag das tolerierbar sein. Sobald aber schärfere Begrenzungen gelten, müssen Sie diese explizit aufführen – als Kommentar.

Das Gleiche gilt für inhaltliche Restriktionen. Das bekannteste Beispiel ist die Pflicht, die beiden Methoden `hashCode` und `equals` stets gemeinsam zu überschreiben, sodass die Implikation

```
a.equals(b)    ⇒    a.hashCode() == b.hashCode()
```

immer erfüllt ist. Derartige Anforderungen lassen sich eben leider nur in Kommentaren ausdrücken.

Schlechte Kommentare

Eigentlich ist es ganz einfach, schlechte Kommentare zu charakterisieren: alle Kommentare, die den Lesefluss stören und nicht zum Verständnis beitragen. Leider sind das stark subjektiv geprägte Kriterien. Deshalb werde ich ein paar Punkte

beleuchten, auf deren Basis Sie die Sinnhaftigkeit von Kommentaren besser einschätzen können.

Wie für alle bewertenden Tätigkeiten brauchen Sie auch für die Einschätzung von Kommentaren Übung. Nehmen Sie sich darum die Zeit, fremden Code auch im Hinblick auf die Kommentare zu analysieren: Was hat Ihnen geholfen? Was ist überflüssig und was stört?

Als Nächstes interessiert natürlich, warum Sie einen Kommentar in eine der Kategorien eingeordnet haben. Schließlich könnten Sie noch überlegen, wie es besser zu machen wäre.

Schon nach kurzer Zeit werden Sie bemerken, dass Sie immer schneller die Qualität erkennen können und sich im gleichen Maße Ihre Kommentare verbessern.

Nichtssagendes

Sehr viele Kommentare fallen in die Gruppe der belanglosen Bemerkungen. Sie wirken als Rauschen, das die Kommunikation behindert. Im Wesentlichen gibt es zwei Ursachen für gehaltlose Kommentare.

Aus Konvention

Aus Gewohnheit, weil sie es so von anderen übernommen haben oder weil sie es nicht besser wissen, kommentieren einzelne Entwickler immer noch jede Feldvereinbarung und jede Methodendeklaration.

Leider wird das an manchen Stellen auch noch erwartet und bisweilen sogar durch automatische Überprüfungen vor dem Einchecken erzwungen. Sehr oft sind die Kommentare dann nur geringfügig veränderte Versionen dessen, was moderne IDE auf Tastendruck generieren.

Das ist ein absoluter Anachronismus angesichts der Tatsache, dass an anderen Stellen versucht wird, trivialen Code –

sogenannten *Boilerplate Code* – aus dem Quelltext zu verbannen und automatisch durch den Compiler generieren zu lassen. Kommentare der Art

```
/**
 * @param anzahl the anzahl to set
 */
public void setAnzahl(int anzahl) {
```

nehmen nur unnötig Platz in Anspruch und tragen rein gar nichts zum Verständnis bei.

Aus Unsicherheit

Es fällt wahrscheinlich schwer, das zuzugeben, aber Kommentare entspringen zum Teil dem Gefühl, dass der Code nicht so aussieht, wie er eigentlich sollte. Das kann aus Unwissenheit, Zeitmangel oder Unlust passieren.

Versuchen Sie nicht, Ihr schlechtes Gewissen dadurch zu besänftigen, dass Sie schlechten Code mit einem Kommentar zu rechtfertigen suchen. Verbessern Sie den Code oder – aber nur als absolute Ausnahme – markieren Sie ihn mit einem

```
//TODO muss überarbeitet werden!
```

wenn die Zeit dafür wirklich nicht vorhanden ist. Alles andere macht die Sache nur noch schlimmer.

Auskommentierter Code

Eine weitere schlechte Angewohnheit aus längst vergangenen Zeiten ist es, nicht mehr benötigten Code als Kommentar zu behalten. Wenn Sie kein Quellcode-Verwaltungssystem haben, kann das sinnvoll sein. Aber wer entwickelt heute noch so?

Deshalb löschen Sie überflüssigen Code sofort. Wenn sich später herausstellen sollte, dass das etwas voreilig war, hilft die Versionsverwaltung, diesen Fehler ohne großen Aufwand rückgängig zu machen.

Sollte es wirklich einmal vorkommen, dass Code nur vorübergehend deaktiviert werden muss, dann vermerken Sie

eine klare Begründung. Andernfalls wird dieser auskommentierte Codeblock wahrscheinlich für alle Zeiten erhalten bleiben, weil jeder zukünftige Bearbeiter annimmt, dass es für das Nichtlöschen wichtige Gründe gibt, die er zwar nicht kennt, die ihn aber gerade deshalb davon abhalten, endlich aufzuräumen.

Unterschiedliche Sprachen

Vermeiden Sie HTML oder andere Auszeichnungssprachen in Kommentaren. Quellcode wird normalerweise im Editorfenster einer IDE betrachtet. Auch wenn es im Browser noch so gut aussieht, im Editor stören die Formatierungsanweisungen nur.

Was Sie auf gar keinen Fall zulassen dürfen, sind Kommentare, die teilweise in Deutsch und teilweise in Englisch geschrieben sind. Entscheiden Sie sich für eine Sprache und bleiben Sie dabei.

Kontextwechsel sind mental aufwendig. Der Leser muss bereits zwischen Programmier- und natürlicher Sprache umschalten. Sie sollten ihm das Leben nicht noch dadurch erschweren, dass er auch noch zwischen verschiedenen natürlichen Sprachen wechseln muss.

Fehlender Bezug

Kommentare ohne konkreten Bezug zum kommentierten Code stören den Lesefluss und das Verständnis.

Ausführlichkeit

Beim Versuch, den Code zu verstehen, braucht ein Leser keine Informationen darüber, in welchen Schritten bestimmte Entscheidungen gefallen sind und worüber dabei diskutiert wurde. Das Gleiche gilt für alle Details, die im betrachteten Code ohne Belang sind.

Wenn Sie beispielsweise eine Hash-Funktion implementieren, schreiben Sie keine Abhandlung über deren Verwendungsmöglichkeiten als Kommentar. Falls jemand doch zusätzliche Informationen wünscht, sollte der Name als Stichwort

für die Suche genügen oder allenfalls im Kommentar ein Verweis gegeben werden.

Nichtlokale Informationen

Ein typisches Beispiel für nichtlokale Informationen ist das folgende:

```
/** Liefert die Waehrung, Standard ist "EUR"
 * @return Waehrungscode (3-Buchstaben-Code)
 */
public String getWaehrung() {
    return waehrung;
}
```

Der kommentierte Code definiert keinen Standardwert. Das gehört nicht hierher. Die Festlegung geschieht an einer ganz anderen Stelle, und dementsprechend groß ist die Gefahr, dass bei einer eventuellen Änderung des Standardwerts dieser Kommentar vergessen wird und dann falsch ist.

Unklare Begriffe

Ein Kommentar muss für sich verständlich sein. Insbesondere muss klar erkennbar sein, worauf er sich bezieht, und keine Informationen voraussetzen, die nicht in unmittelbarer Nähe zu finden sind.

Verstöße gegen diesen Grundsatz passieren häufig, wenn Sie wirklich »tief in einer Sache drin sind« und dabei vergessen, dass der Leser letztlich nur den Code verstehen will. Dagegen hilft, im Geiste kurz einen Schritt zurück zu treten und den Text noch einmal aus der Distanz zu lesen.

JavaDoc

Die weite Verbreitung von Java ist nicht zuletzt der guten Dokumentation der öffentlichen Schnittstellen – kurz *API* für *Application Programming Interface* genannt – aller mitgelieferten Bibliotheken zu verdanken. Wesentlichen Anteil daran hat die JavaDoc-Technologie, die es erlaubt, die Dokumentationen direkt aus den im Quellcode enthaltenen Kommentaren zu generieren.

Für die Lesbarkeit des Quellcodes ist das aber weniger förderlich, wie Sie sich leicht überzeugen können. Speziell längere Passagen können Sie durch HTML-Tags zwar so gestalten, dass die generierten Dokumente recht ansehnlich sind. Auf der Codeebene ist das jedoch oft kaum noch verständlich. Oder erkennen Sie sofort, was Folgendes bedeutet?

```
( + ) &#92;u0131 I -&gt; i
```

Es ist nicht völlig unverständlich, aber hindert den Lesefluss. Trotzdem bleibt JavaDoc ein ausgezeichnetes Werkzeug für die Dokumentation öffentlicher API. Wie bei jeder wirksamen Arznei gibt es jedoch Gegenindikationen. Verwenden Sie deshalb JavaDoc-Kommentare nicht leichtfertig, sondern nur, wenn Sie sicher sind, dass

- ✔ die Dokumentation nach Änderungen neu generiert wird und somit immer dem Codestand entspricht und
- ✔ die Anwender wirklich vorrangig die Dokumentation und nicht den Quellcode als Informationsquelle benutzen.

Nur wenn Sie durch JavaDoc-Kommentare einen echten Mehrwert erhalten, sind die damit verbundenen Nachteile gerechtfertigt. Letztlich können diese Kommentare genauso gut oder schlecht sein wie jeder andere Kommentar.

Nehmen Sie JavaDoc-Kommentare so wichtig wie den Code selbst und formulieren Sie sie so sorgfältig, dass alle benötigten Informationen enthalten sind. Niemand sollte genötigt sein, im Code nachzusehen, weil die Dokumentation unvollständig ist.

Dokumentationen

Allen anderslautenden Gerüchten zum Trotz macht Clean Code weder Kommentare noch Dokumentationen überflüssig. Ganz im Gegenteil geht es darum, die Kommunikation über Software zu verbessern, indem für jede Information das jeweils am besten geeignete Medium gewählt wird.

Natürlich soll das mit möglichst wenig Zusatzarbeit passieren, aber es gibt Dinge, die sich im Quellcode nicht sinnvoll formulieren lassen:

- ✔ Bilder und grafische Übersichten sind unverzichtbar für die Veranschaulichung von Strukturen und Zusammenhängen.
- ✔ Tabellen können Daten verständlicher darstellen.
- ✔ Beweggründe, Erklärungen und übergreifende Aspekte erfordern zu ihrer Darstellung häufig längere Textpassagen, die im Code nichts zu suchen haben.

Halten Sie sich an die Regel: Was im Quellcode besser, das heißt insbesondere präziser und verständlicher dargestellt werden kann, gehört nicht in die Dokumentation. Das gilt selbstverständlich auch umgekehrt.

Schönheit

Wenn Sie das Thema hier überrascht, lassen Sie sich sagen, dass dieser Aspekt eigentlich in jedes Kapitel gehören würde. Hier ist er deshalb gut aufgehoben, weil es sich analog Formatierung und Kommentare um eine Codeeigenschaft handelt, die ausschließlich für Menschen bedeutsam ist.

Charakteristisch für Schönheit sind neben anderem Einfachheit, Symmetrie, Harmonie und Ausgeglichenheit, also Gesichtspunkte, die für sauberen Code ebenfalls eine wichtige Rolle spielen. Sie sollten den Einfluss auf die Lesbarkeit und Weiterentwickelbarkeit nicht unterschätzen.

-  Vergessen Sie nicht, Ihr Quellcode darf schön aussehen, aber das ist nicht das primäre Ziel. Freuen Sie sich, wenn er Ihnen gut gelungen ist, aber verschwenden Sie keine Energien, etwas um jeden Preis schön zu machen. Manche Aspekte der Realität sind nicht schön, sondern kompliziert und unansehnlich.
- Lassen Sie sich nicht von einer »ästhetischen Welle« zu weit hinaustreiben. Sie sind zuerst Softwareentwickler und nicht Künstler. Überdies sind Schönheitsideale nicht universell und unterliegen dem Wandel.

Rezeption und Schönheit

In den formalen Wissenschaften, besonders auch in der Mathematik, wird Schönheit bisweilen als Indiz für die Wahrheit einer Aussage oder Theorie gesehen.

> Das entscheidende Kriterium ist Schönheit; für hässliche Mathematik ist auf dieser Welt kein beständiger Platz. (G. H. Hardy 1877–1947)

Ein Zusammenhang zwischen Schönheit und beurteilter Wahrheit lässt sich experimentell nachweisen. Wahrscheinlich ist, dass dem Schönheitsempfinden ähnlich den Wahrheitsurteilen eine besonders flüssige mentale Rezeption zugrunde liegt. Unabhängig von solchen psychologischen Detailfragen ist offensichtlich, dass es sich beim Schönheitsempfinden um eine entwicklungsgeschichtlich alte Fähigkeit handelt.

Trotz aller durch die gesellschaftlichen Konventionen gegebenen Variabilität muss die Wahrnehmung und Nutzung von Schönheit einen evolutionären Vorteil geboten haben. Insofern ist Schönheit eine durchaus relevante Kategorie, wenn es um Wahrnehmung und Verstehen geht, die Sie bewusst nutzen sollten.

Das Wichtigste in Kürze

- Auch Clean Code braucht Kommentare, denn Quellcode wird nicht nur für den Computer geschrieben.

- ✔ Kommentare müssen genauso sorgfältig gepflegt werden wie der eigentliche Code. Passiert das nicht, sollte man sie besser ganz weglassen.
- ✔ Kommentare müssen genau die Informationen vermitteln, die in den Quellcode gehören, die der Code aber selbst nicht ausdrücken kann.
- ✔ Damit Kommentare das Verständnis unterstützen können, müssen sie kurz und präzise formuliert sein.
- ✔ Überflüssige und nichtssagende Kommentare vermüllen den Quellcode.
- ✔ Kommentare können eine Dokumentation nicht ersetzen.
- ✔ Schönheit ist zwar kein primäres Ziel von Clean Code, kann dessen Ziele aber sehr wohl unterstützen.

Kapitel 13

Kleine Schritte – saubere Methoden

IN DIESEM KAPITEL

- Methoden als Bauteile für die Strukturierung von Abläufen
- Methoden optimal zuschneiden
- Parameter richtig einsetzen

Algorithmen bestehen aus wohldefinierten Einzelschritten. Jeder dieser Schritte leistet seinen Beitrag, um das Gesamtziel zu erreichen. Die Festlegung, was genau einen Schritt umfasst, ist jedoch willkürlich.

Komplexe Algorithmen lassen sich besser verstehen und implementieren, wenn man sie in überschaubare Schritte zerlegt. In diesem Kapitel erfahren Sie, wie diese Teile zweckmäßigerweise aussehen sollten und welche Fehler es bei der Zerlegung zu vermeiden gilt.

Methoden

> Zerlegen Sie jede zu untersuchende Aufgabe in so viele Teile, wie Sie können und wie Sie benötigen, um sie leicht lösen zu können. (Descartes)
>
> Diese Regel ist wenig effektiv, da die Kunst der Zerlegung unergründlich bleibt … Indem der unerfahren Lösende die Aufgabe in ungeeignete Teile zerlegt, kann er seine Schwierigkeiten auch vergrößern. (Leibniz)

Die beiden Zitate illustrieren sehr schön das Spannungsfeld, um das es hier geht. Zerlegung ist einerseits das Mittel der Wahl für die Lösung anspruchsvoller Aufgaben – im Allgemeinen gibt es kein anderes. Andererseits ist die Zerlegung selbst wiederum nicht trivial und erfordert Übung und Erfahrung.

Die Ratschläge dieses Kapitels werden Ihnen hoffentlich helfen. Trotzdem wird das Erreichen eines guten Ergebnisses jedes Mal wieder Ihre gesamten Fähigkeiten herausfordern.

Begriffliche Klärung

Um die Diskussion von Methoden auf eine solide Grundlage zu stellen, ist es nützlich, vorab die verwendeten Begriffe zu klären.

Abgrenzung

Ganz gleich ob man sie nun *Methoden*, *Funktionen*, *Prozeduren* oder *Unterprogramme* nennt, sie sind die Basis der funktionalen Abstraktion und damit die Bausteine jeder Software. Es gibt mehr oder weniger feine Unterschiede, abhängig von den Programmiersprachen oder Programmierparadigmen, in deren Kontext man sich bewegt.

Für Clean Code sind diese Differenzen nicht wesentlich. Ich verwende hier den Begriff *Unterprogramm*, wenn es um grundlegende allgemeine Eigenschaften geht, und *Methode* sobald ein konkreterer Bezug vorliegt.

Intentionen

In der Frühphase der Programmierung dienten Unterprogramme vor allem der strukturellen Dekomposition, indem sich wiederholende Codesequenzen herausgelöst und von verschiedenen Aufrufstellen aus abgearbeitet wurden. Das half nicht zuletzt bei der besseren Ausnutzung des seinerzeit teuren und daher knappen Hauptspeichers.

Der zweite wichtige Aspekt, die funktionelle Dekomposition, trat zwar erst später hinzu, wurde dafür dann aber schnell der bedeutendere Gesichtspunkt.

Eigenschaften

Damit Unterprogramme sinnvoll wiederverwendet werden können, müssen sie mit unterschiedlichen Daten versorgt werden können. Dazu gibt es prinzipiell zwei Möglichkeiten:

- Durch sogenannte *globale Variablen*, das sind Variablen, die außerhalb des Unterprogramms deklariert sind und von diesem benutzt werden können. Die Wirkung tritt nach außen hin nur als Seiteneffekt in Erscheinung.
- Mittels formaler Parameter, die beim Aufruf durch Werte oder Variablen ersetzt werden. Eventuell kann ein Wert, der *Rückgabewert*, dem Aufruf als Ergebnis zugewiesen werden. Entsprechende Unterprogramme können ähnlich wie mathematische Funktionen in Ausdrücken benutzt werden.

Objektorientierte Sprachen nehmen äußerlich eine Zwitterstellung ein, weil Methoden auf die Datenfelder ihres Objekts zugreifen können, als ob diese global deklariert wären. Tatsächlich jedoch ergänzt der Compiler jeden Methodenaufruf um einen Parameter `this`, der das jeweilige Objekt implizit als Parameter übergibt.

Der Inhalt

Nach der Klärung der äußeren Form einer Methode geht es nun um ihre Funktion bei der Formulierung eines Lösungsalgorithmus.

Abstraktion

Methoden sind die Abstraktionseinheiten, durch die komplexe Abläufe in überschaubare Teile zerlegt werden. Dazu wird die Aufgabe in eine Folge von Abstraktionsniveaus gegliedert. Die Abstraktionen einer Ebene müssen so gewählt sein, dass sie erfass- und verstehbar sind.

Gleichzeitig muss es möglich sein, sinnvolle Analysen und Manipulationen durchzuführen, ohne dabei das jeweilige Niveau

zu verlassen. Abstraktionen, die diese Bedingungen erfüllen, sind die angemessene Basis für Methoden.

In der Regel wird das – der beschränkten mentalen Kapazitäten wegen – zu kurzen und kleinen Methoden führen, aber nicht notwendigerweise.

Das entscheidende Kriterium ist, dass alle Aktionen innerhalb einer Methode auf dem gleichen Abstraktionsniveau liegen. Wenn Sie beispielsweise erst einen Datensatz beschaffen und dann einzelne Elemente bearbeiten, ist dieses Kriterium verletzt. Beschaffung und Bearbeitung liegen auf verschiedenen Abstraktionsebenen. Teilen Sie deshalb die Arbeit auf zwei Methoden auf.

Halten Sie sich an die altbewährte Regel: Eine Methode soll genau eine Sache erledigen. Das soll sie gut machen, und sie soll nur das machen.

Trennung von Bearbeitung und Abfrage

Jede Bearbeitung von Daten lässt sich auf zwei grundsätzliche Aktionen zurückführen:

1. *Bearbeitung*, dabei werden die vorliegenden Daten in irgendeiner nach außen sichtbaren Form verändert.
2. *Abfrage*, dabei werden aus den vorliegenden Daten Informationen gewonnen, ohne diese Daten in einer nach außen sichtbaren Weise zu verändern.

Beachten Sie bitte, dass dabei der auf dem entsprechenden Abstraktionsniveau *nach außen* sichtbare Zustand gemeint ist. Das schließt nicht aus, dass beispielsweise ein interner Cache bei einer Abfrage aktualisiert wird.

Es hat sich vielfach bewährt, Methoden nach den genannten Kriterien auszurichten. Bearbeitungsmethoden haben dabei im

Unterschied zu Abfragemethoden in der Regel keinen Rückgabewert. So können Sie bereits aus der Signatur, das heißt der durch Namen und Parameter bestimmten Schnittstelle einer Methode, erkennen, welcher Typ vorliegt.

Wenn Sie diesen Grundsatz einhalten, erleichtert das den Umgang enorm, weil Sie Abfragemethoden in einem weiten Bereich ohne Gefahr streichen, ergänzen oder verschieben können. Bei Bearbeitungsmethoden ist dagegen mehr Sorgfalt erforderlich.

Trennung von Bearbeitung und Abfrage

Dieses auch unter der Bezeichnung *Command Query Separation* bekannte Prinzip wurde bereits 1992 von Bertrand Meyer geprägt. Es besagt informell gesprochen, dass eine Abfrage die Antwort nicht verändern sollte, so wie man das etwa auch bei einer SQL-SELECT-Abfrage erwartet.

Verstöße gegen diesen Grundsatz gibt es jedoch viele. Größtenteils sind sie der überkommenen Codesparsamkeit geschuldet. Über Jahrzehnte haben Entwickler versucht, möglichst kurzen Code zu schreiben, ohne dabei große Rücksicht auf Abstraktionsebenen zu nehmen.

Manchmal folgt die Verletzung aber auch einem breit akzeptierten Verständnis, wie das bei der aus Abfrage und Entfernen des obersten Elements bestehenden Pop-Operation für Stacks der Fall ist.

Testen

Vergessen Sie nicht, dass sauberer Code getestet werden muss. Das Schreiben von Testfällen ist integraler Bestandteil Ihrer Entwicklung. Auch wenn ich das nicht als erstrangiges Erfordernis sehe, sollten Sie beim Zuschnitt der Methoden deren Testbarkeit nicht völlig außer Acht lassen.

Die Größe

Die vorteilhafte Länge einer Methode wird in Kapitel 12 *Reine Formfrage — Formatierung* unter vorwiegend optischem Aspekt

betrachtet. Hier wird die Antwort unter inhaltlichen Gesichtspunkten präzisiert.

Eine Aufgabe

Die Frage nach der vorteilhaftesten Größe einer Methode ist alt. Frühe Antworten lauteten etwa, dass eine Seite oder ein Bildschirm angemessene Grenzen wären. Das war insofern nicht falsch, als es sich auf das auf einen Blick Erfassbare bezog. Inzwischen sind Bildschirme so groß, dass weit mehr dargestellt werden kann, als auf einen Blick erfassbar ist.

Der beschriebene Ansatz hat allerdings noch einen wesentlicheren Fehler: Er nähert sich dem Problem von der falschen Seite, nämlich allein von den kognitiven Kapazitäten her. Die darf man zwar nicht übersehen, aber das Ziel der Zerlegung ist die intellektuell beherrschbare Darstellung.

Eine Methode soll genau eine Aufgabe lösen und die dazu notwendigen Aktionen auf einer Abstraktionsebene formulieren. Lange Methoden sind ein Indiz dafür, dass der Abstand zwischen der Abstraktionsebene, auf dem die Methode definiert ist, und demjenigen der Realisierungsebene wahrscheinlich zu groß ist.

Zeilenzahl

Entscheidend ist nicht die Anzahl der Zeilen, sondern die Anzahl der Konzepte. Beispielsweise wäre eine Methode, die einen umfangreichen Datensatz kopiert, trotz ihrer Länge akzeptabel, während andererseits eine relativ kurze Methode, die aber die Abstraktionsebenen so vermischt wie das Beispiel in Listing 13.1, abzulehnen ist.

```
public static void printExcelFileAsCsv(String excelFileName,
          PrintStream printStream)
        throws FileNotFoundException, IOException {
  InputStream inputStream= new
FileInputStream(excelFileName);
  HSSFWorkbook workbook = new HSSFWorkbook(inputStream);
  for (int k = 0; k < workbook.getNumberOfSheets(); k++) {
```

```
    HSSFSheet sheet = workbook.getSheetAt(k);
    int noOfRows = sheet.getPhysicalNumberOfRows();
    for (int r = 0; r < noOfRows; r++) {
      HSSFRow row = sheet.getRow(r);
      int noOfCells = row.getPhysicalNumberOfCells();
      for (int c = 0; c < noOfCells; c++) {
        if (c> 0) {
          printStream.print(',');
        }
        HSSFCell cell = row.getCell(c);
        if (cell!=null) {
          switch (cell.getCellType()) {
          case HSSFCell.CELL_TYPE_FORMULA:
            printStream.print(cell.getCellFormula());
            break;
          case HSSFCell.CELL_TYPE_NUMERIC:
            printStream.print(cell.getNumericCellValue());
            break;
          case HSSFCell.CELL_TYPE_STRING:
            printStream.print(cell.getStringCellValue());
            break;
          default:
          }
        }
      }
      printStream.println();
    }
  }
}
```

Listing 13.1: Beispiel einer zu umfangreichen Methode

Ich habe daher große Bedenken, eine Empfehlung bezüglich der maximalen Zeilenzahl auszusprechen. Gemeinhin werden 20 Zeilen als Obergrenze angesehen.

Anhand der Methode aus Listing 13.1, die unter Nutzung einer externen Bibliothek (HSSF) eine Excel-Datei als kommagetrennte

Werte (CSV) ausgibt, werde ich Ihnen im Verlauf dieses Kapitels zeigen, wie Sie es besser machen können.

Sehen Sie Vorgaben wie die einer empfohlenen Methodenlänge nicht als blind zu befolgende Regel. Nehmen Sie Verstöße dagegen vor allem zum Anlass für eine Prüfung, ob die gefundene Struktur wirklich schon so einfach wie möglich ist.

Schachtelungsstruktur

Die Schachtelungsstruktur, das heißt die Tiefe der Einrückungen, ist ein guter Indikator für die Komplexität einer Methode. Mehr als zwei Einrückungsstufen sind ein sicheres Indiz dafür, dass die Methode auf mehr als einem Abstraktionsniveau angesiedelt ist.

Die erwähnte Beispielmethode in Listing 13.1 zeigt das deutlich. Ihre Aufgabe besteht darin, eine Excel-Datei zu lesen und die enthaltenen Daten im CSV-Format in einen `PrintStream` zu schreiben. Dazu muss die Datei zunächst geöffnet und dann über die enthaltenen Tabellenblätter, Zeilen und Zellen iteriert werden.

Sie sehen bereits an der Beschreibung, da wird mehr als eine Sache erledigt und es werden verschiedene Abstraktionen verwendet: Datei, Blatt, Zeile und Zelle. Jedes neue Abstraktionsniveau verursacht eine weitere Einrückungsstufe.

Es ist also angebracht, das Beispiel zu überarbeiten. Mithilfe der in der IDE verfügbaren Unterstützung ist das relativ schnell gemacht. Aber lassen Sie sich nicht dazu verleiten, das als einfache Aufgabe quasi nebenher zu erledigen.

Erinnern Sie sich daran, wie wichtig gute Namen für die Lesbarkeit sind! Verwenden Sie die Zeit, die Ihnen die IDE erspart, um wirklich aussagekräftige Bezeichnungen zu finden. Je spezieller die ausgeführte Funktion ist, desto aussagekräftiger – und damit häufig auch länger – sollte der gewählte Name sein.

Listing 13.2 zeigt das Ergebnis der ersten Überarbeitung. Sie sehen, wie viel übersichtlicher alles geworden ist. Wenn jetzt

jemand von Ihnen verlangt, vor den Daten jedes Blatts eine Leerzeile einzufügen, kostet Sie das nur noch einen Augenblick.

Das erste Ziel, nämlich kurze Methoden auf jeweils einem einzigen Abstraktionsniveau, ist damit schon erreicht. Nur ein Methodenkörper hat mehr als sieben Zeilen, aber selbst dieser eine ist nur 15 Zeilen lang.

Trotzdem gibt es noch ein paar Dinge, die Sie besser machen könnten. Doch dazu später.

```
public static void printExcelFileAsCsv(String excelFileName,
          PrintStream printStream)
    throws FileNotFoundException, IOException {
  printExcelStreamAsCsv(new FileInputStream(excelFileName),
    printStream);
}
static void printExcelStreamAsCsv(InputStream inputStream,
          PrintStream printStream)
    throws IOException {
  printWorkbook(new HSSFWorkbook(inputStream), printStream);
}
static void printWorkbook(HSSFWorkbook workbook,
          PrintStream printStream) {
  for (int k = 0; k < workbook.getNumberOfSheets(); k++) {
    printSheet(workbook.getSheetAt(k), printStream);
  }
}
static void printSheet(HSSFSheet sheet, PrintStream
printStream) {
  for (int r = 0; r < sheet.getPhysicalNumberOfRows(); r++) {
    printRow(sheet.getRow(r), printStream);
  }
}
static void printRow(HSSFRow row, PrintStream printStream) {
  for (int c = 0; c < row.getPhysicalNumberOfCells(); c++) {
    if (c> 0) {
      printStream.print(',');
    }
```

```
    printCell(row.getCell(c), printStream);
  }
  printStream.println();
}
static void printCell(HSSFCell cell, PrintStream printStream)
{
  if (cell!=null) {
    switch (cell.getCellType()) {
    case HSSFCell.CELL_TYPE_FORMULA:
      printStream.print(cell.getCellFormula());
      break;
    case HSSFCell.CELL_TYPE_NUMERIC:
      printStream.print(cell.getNumericCellValue());
      break;
    case HSSFCell.CELL_TYPE_STRING:
      printStream.print(cell.getStringCellValue());
      break;
    default:
    }
  }
}
```

Listing 13.2: Beispielmethode überarbeitet

Parameter

Parameter geben Methoden die Flexibilität, ihre Anweisungen auf unterschiedliche Daten anzuwenden. Sie können die Lesbarkeit des Codes allerdings beeinträchtigen.

Parameter sind gewissermaßen Türen zwischen den Abstraktionsebenen. Durch diese Türen werden Informationen ausgetauscht, die auf beiden Seiten der Wand gleich interpretiert werden müssen. Das macht das Ganze etwas komplizierter zu verstehen.

Anzahl

Wenn Sie fragen, wie viele Parameter eine Methode haben sollte, lautet die Antwort: so wenig wie möglich. Durch Parameter wird zwar die Vielseitigkeit erhöht, aber Sie sollten die etwas abgewandelte Form eines bekannten Spruchs beherzigen: so vielseitig wie nötig, aber nicht vielseitiger.

Kein Parameter

Verschiedentlich werden parameterlose Methoden als anzustrebendes Ideal gepriesen. Den konzeptionellen Vorteilen steht aber die damit verbundene geringere Ausdruckskraft entgegen. Ganz abgesehen davon haben auch diese Methoden fast immer Parameter, nur eben implizit.

Bei Objektmethoden ist das das zugehörige Objekt, welches auf Bytecode-Ebene als eigener Parameter übergeben wird, bei Klassenmethoden übernimmt die komplette Ausführungsumgebung diese Rolle, wie zum Beispiel bei `System.currentTimeMillis()` für die aktuelle Zeit oder `Toolkit.getDefaultToolkit()` für den Zugriff auf Funktionen der Hardware.

Wirklich vollständig parameterlose Methoden liefern immer das gleiche Ergebnis, und es gibt sogar vernünftige Anwendungsfälle dafür, wie das Beispiel `Collections.emptyList()` – liefert immer das gleiche leere Listenobjekt – beweist.

Was Sie mit parameterlosen Methoden nicht können, ist eine Beziehung zwischen zwei oder mehr Objekten herstellen. Abgesehen davon sollten Sie jedoch versuchen, wo immer es geht, auf Parameter zu verzichten.

Ein bis drei Parameter

In gut verständlichem Code sollten Methoden nicht mehr als drei Parameter haben. Natürlich sind Ausnahmen von dieser Regel möglich, aber die sollten schon sehr gut begründet sein.

Variable Argumentlisten, auch *Varargs* genannt, sind kein Verstoß gegen diese Regel, weil sie nur die Schreibweise des Aufrufs vereinfachen. Das ist bei der Deklaration deutlich erkennbar. Die formale Parameterliste in der Form

```
methode(int… args)
```

ist eben nur eine bequemere Schreibweise für

```
methode(int[] args)
```

mit einem Argument.

Wenn Sie variable Argumentlisten zusammen mit weiteren Parametern verwenden, müssen Sie darauf achten, dass das nicht zu Verwirrungen führt. So ist die folgende Deklaration im Allgemeinen keine sehr gute Idee

```
methode(int i, int k, int… args)
```

weil beim Aufruf der Beginn der Liste nicht erkennbar ist. Keine Probleme bereitet hingegen eine Methode

```
format(Locale l, String format, Date… dates)
```

denn durch die unterschiedlichen Datentypen ist keine Verwechslungsgefahr gegeben.

Mit diesem leicht vereinfachten Beispiel aus der Praxis möchte ich Ihnen zeigen, was Sie *auf keinen Fall* machen sollten, um scheinbar die Anzahl der Parameter zu verringern:

```
public static String createReasonText(String… info) {
  return "R:" + info[0] + ";" + info[1]+ ";" + info[2] +
";"
      + info[3] + ";"  + info[4] + ";" + info[5] + ";";
}
```

Auf den ersten Blick könnte man denken: Sechs Parameter durch einen ersetzt ist doch in Ordnung. Auf den zweiten Blick erkennen Sie hoffentlich sofort, dass der Preis dafür zu hoch ist: Der Code verletzt nämlich den Schnittstellenvertrag.

Durch die Deklaration einer variablen Parameterliste sichern Sie dem Aufrufer zu, dass die Methode mit einer beliebigen Anzahl von Argumenten aufgerufen werden kann. Der Aufruf

`createReasonText("P0", "P1");` wird zwar vom Compiler klaglos akzeptiert, verursacht jedoch eine `ArrayIndexOutOfBoundsException`.

Das ist umso hässlicher, als die Methode für das Loggen von Exceptions vorgesehen war. Ein möglicherweise fehlerhafter Aufruf wäre also mit hoher Wahrscheinlichkeit bis zum Auftreten eines anderen Fehlers unentdeckt geblieben.

Die Variante

```
public static String createReasonText(String… info) {
  if (info.length<6) throw new
IllegalArgumentException("2short");
  return "R:" + info[0] + ";" + info[1]+ ";" + info[2] +
";"
      + info[3] + ";"  + info[4] + ";" + info[5] + ";";
}
```

führt in der entsprechenden Situation ebenfalls zu einem Laufzeitfehler und ist daher nicht besser.

Eine akzeptable Lösung darf keine von der Anzahl der Argumente abhängige Exception verursachen und könnte so aussehen:

```
public static String createReasonText(String… info) {
  StringBuilder sb = new StringBuilder("R:");
  for (String infopart : info) {
    sb.append(infopart).append(';');
  }
  return sb.toString();
}
```

Es ist eine nützliche Übung, wenn Sie selbst versuchen, weitere und vielleicht sogar schönere Lösungen zu finden.

Stellung

Die Zuordnung von aktuellen zu formalen Parametern erfolgt auf der Basis der Stellung. Das macht es schwer, beispielsweise bei

einem Aufruf `list.add(1, 2);` zu erkennen, ob der Wert 2 an erster Stelle eingefügt werden soll oder der Wert 1 an zweiter Stelle. Um dem vorzubeugen, könnten Sie die entsprechende Methode besser `addAtElement` oder `addElementAt` nennen.

Ganz wichtig ist es, bei der Reihenfolge der Parameter konsistent zu sein. Wenn Sie beispielsweise mit Bildschirm-Koordinaten hantieren, wäre es sehr verwirrend, Methoden `moveTo(int x, int y)` und `drawCircle(int y, int x, int radius)` nebeneinander zu haben.

Leider werden Sie dieses Problem nicht immer vermeiden können, weil es in den meisten Fällen keine allgemein verbindliche Reihenfolge gibt. Code aus unterschiedlichen Bibliotheken kann möglicherweise sich widersprechenden Konventionen folgen.

Um die Auswirkungen auf Ihren Code zu minimieren, ist es dann empfehlenswert, die externen Methoden nicht direkt zu nutzen, sondern in eigene – sogenannte *Wrapper-Methoden* – zu verpacken, die die Parameter wie gewünscht umordnen, zum Beispiel:

```
public void drawCircle (int x, int y, int r) {
    imported.drawCircle(y, x, r);
}
```

Der geringe Mehraufwand ist durch den Nutzen beim Lesen und durch vermiedene Fehler gerechtfertigt. In Kapitel 14 *Passend schneiden – Schnittstellen* wird die Einbindung von externen Programmkomponenten noch genauer betrachtet.

Parameter vermeiden

An dem Beispiel aus Listing 13.2 lässt sich ein einfaches Verfahren zur Vermeidung von Parametern zeigen. Alle Methoden haben einen Parameter vom Typ `PrintStream`. Warum also nicht diesen Parameter zum Feld und die statischen Methoden zu Objektmethoden einer neuen Klasse machen? Wie das Ergebnis aussehen könnte, ist in Listing 13.3 skizziert.

```
… class ExcelToCsvPrinter {

private final PrintStream printStream;

public ExcelToCsvPrinter(PrintStream printStream) {
  this.printStream= printStream;
}

public static void printExcelFileAsCsv(String excelFileName,
          PrintStream printStream)
    throws FileNotFoundException, IOException {
  ExcelToCsvPrinter printer= new
ExcelToCsvPrinter(printStream);
  printer.printExcelStreamAsCsv(
    new FileInputStream(excelFileName));
}

void printExcelStreamAsCsv(InputStream inputStream) throws
       IOException {…
void printWorkbook(HSSFWorkbook workbook) {…
void printSheet(HSSFSheet sheet) {…
void printRow(HSSFRow row) {…
void printCell(HSSFCell cell) {…
```

Listing 13.3: Im Beispiel Parameter durch neue Klasse ersetzen

Testen

Parameter erhöhen den Testaufwand. Das ist nicht überraschend. Allerdings wird leicht übersehen, dass es nicht reicht, das Verhalten für jeden Parameter einzeln zu testen. Sie müssen auch einen Teil der Kombinationen prüfen, und deren Anzahl steigt sehr schnell mit der Anzahl der Parameter.

Stellen Sie sich vor, sie hätten eine Methode mit drei Parametern geschrieben, von denen – obgleich das schlechter Stil ist – jeder auch `null` sein darf. Allein um alle Kombinationen von Null- und Nicht-Null-Parametern zu

testen, benötigen Sie bereits acht Tests. Bei vier Parametern wären es sechzehn.

Falls Sie für jeden Parameter beispielsweise drei verschiedene Werte testen müssen, kommen Sie bereits auf 27 beziehungsweise 81 Kombinationen. Das ist ein Aufwand, den Sie wahrscheinlich nicht treiben wollen oder können.

Lange Parameterlisten sind also nicht nur schlecht zu lesen. Sie sind darüber hinaus schlecht für die Qualität, weil es sehr aufwendig sein kann, eine ausreichend gute Testabdeckung zu erreichen.

Flag-Parameter

Boolesche Parameter, die nur die zwei Wahrheitswerte annehmen können, gemeinhin als *Flags* bezeichnet, sind fast immer ein Zeichen dafür, dass eine Methode mehr als eine Funktion hat. Die eine wird ausgeführt, wenn das Flag den Wert `true` hat, die andere beim Wert `false`. Dann ist es besser, zwei Methoden zu haben, deren Namen die jeweilige Funktion exakt beschreiben. Das Problem ergibt sich vor allem daraus, dass es ausgesprochen schwierig ist, solche Flag-Parameter wirklich verständlich zu benennen.

Sie werden aber auf Konstellationen stoßen, in denen sich derartige Parameter nicht völlig vermeiden lassen:

- ✔ Wenn es sich wirklich um einen booleschen Wert handelt, also kein »Flag« im eigentlichen Sinn vorliegt.
- ✔ Wenn Sie ein vorgegebenes Interface implementieren müssen.
- ✔ Wenn Sie in einem Konstruktor eine Variante brauchen, ist die Definition einer weiteren Klasse oft nicht vertretbar. Beispielsweise bietet die Java-Klasse `FileOutputStream` Konstruktoren mit einem zweiten Flag-Parameter, der die Unterscheidung zwischen Überschreiben und Anhängen steuert. Die Abtrennung einer speziellen

`FileOutputAppendStream`-Klasse hätte entsprechende Erweiterungen bei allen abgeleiteten Klassen zur Folge.

- Wenn der Wert des Flags erst in einer tiefen Schicht gebraucht wird, müssten Sie unter Umständen eine ganze Kaskade von jeweils zwei Methoden implementieren. Das ist nicht immer vertretbar. Prüfen Sie aber vorher, ob es sich nicht um einen Entwurfsfehler handelt.

Falls Sie glauben, boolesche Parameter verwenden zu müssen, überprüfen Sie gründlich, ob wirklich einer der akzeptablen Fälle vorliegt. Sehr oft geht es nämlich nur darum, scheinbar »unnütze« Schreibarbeit zu vermeiden.

Resultate

Methoden werden aufgerufen, um bestimmte Funktionen auszuführen. Es gibt mehrere Möglichkeiten, bei der Rückkehr ein Ergebnis zu übermitteln, wenn der Aufrufer das erwartet.

Rückgabewerte

Der normale Weg, auf dem eine Methode ein Ergebnis an den Aufrufer zurückgibt, ist die `return`-Anweisung. Nun können Sie auf diese Weise nur einen Wert oder ein Objekt übertragen. Das ist kein Nachteil, weil es Sie zwingt, verständlich zu programmieren. Sollten Sie nämlich feststellen, dass das nicht reicht, so liegt fast immer einer der folgenden Fehler vor:

- Die Methode erledigt mehr als eine Aufgabe und muss zerlegt werden.
- Die Methode hängt am falschen Objekt. Verschieben Sie sie an die passende Stelle.
- Es fehlt eine Klasse für den benötigten Ergebnistyp.

In objektorientiertem Code lassen sich solche Situationen fast immer problemlos bereinigen.

Rückgabewert null

Eine gesonderte Betrachtung erfordert der Rückgabewert `null`. Genau genommen ist das kein Wert, und das heißt, dass der Methodenaufruf nicht das erwartete Ergebnis liefert. Für den aufrufenden Code ist das ein problematisches Verhalten, das Sie daher möglichst vermeiden sollten.

Kein Ergebnis bedeutet gleichzeitig, dass Sie keinerlei Information über die Ursache erhalten. Ist ein interner Fehler aufgetreten oder wurde nur ein Wert fehlerhaft oder willentlich nicht gesetzt?

Die Folge dieser Unsicherheit sind Massen von `if (x!=null)`-Anweisungen im Code, die die Lesbarkeit nicht verbessern. Wird zudem der `else`-Zweig weggelassen, so können an weit entfernten Stellen unerwartete Fehler auftreten

Oft ist die Vermeidung gar nicht schwierig. Geben Sie statt `null` beispielsweise leere `Collections`-Objekte zurück:

```
public List<Cell> getCells(int row) {
  if (… /*es gibt keine*/ …)
    return Collections.emptyList();
}
```

Es gibt allerdings Fälle, in denen Ihnen das nicht so einfach gelingen wird. Ein gutes Beispiel dafür ist die `Map`-Methode `get(key)`. Um die Rückgabe eines Nullwerts zu vermeiden, müsste stattdessen eine Exception geworfen werden. Das wäre aber mitnichten eine Verbesserung – ganz im Gegenteil. Um diese Exception zu vermeiden, müssten Sie jedes Mal

```
if (map.contains(key) {
  value = map.get(key); …
```

schreiben. Das ist nicht nur umständlich, sondern auch ineffizient, weil gegebenenfalls zweimal nach dem Schlüssel gesucht werden muss. Einen einfachen Ausweg gibt es nicht. Wie Sie in Listing

13.4 sehen können, existieren nun allerdings Methoden, die Sie in typischen Konstellationen von der Behandlung des Nullwerts befreien.

```
//  Fall 1:
value = map.get(key);
if (value == null)
  value = default;
// ersetzt durch
value = map.getOrDefault(key, default);
// Fall 2:
value = map.get(key);
if (value == null)
  map.put(key, newvalue);
// ersetzt durch
map.putIfAbsent(key, newvalue);
```

Listing 13.4: Vermeiden von Tests auf `null` durch spezielle Methoden

Wenn Sie sich an diesem Vorbild orientieren, werden Sie zwar nicht alle Null-Rückgabewerte vermeiden können, aber den Code trotzdem übersichtlicher gestalten.

Die mit Java 8 eingeführte Klasse `Optional` hilft Ihnen zwar dabei, `if`-Anweisungen mit expliziten Tests auf `null`-Werte zu vermeiden. Das ist besonders bei funktionaler Schreibweise hilfreich. Das grundsätzliche Problem von »Nicht-Ergebnissen« wird dadurch jedoch nicht gelöst.

Ergebnisparameter

Methoden werden aufgerufen, um – durch die aktuellen Parameter gesteuert – die gewünschten Resultate zu produzieren. Sie sehen bereits an der Beschreibung, dass Parameter gewöhnlich als Steuergrößen für die Ausführung verstanden werden, also als etwas, das in die Methode hineinwirkt.

Technisch ist es zwar möglich, Ergebnisse in Parametern zu übergeben, für einen Leser ist das jedoch nur umständlich zu erkennen. Betrachten Sie die drei Methoden in Listing 13.5.

```
public void exec1(List list) {
  list.clear();
}
public List exec2(List list) {
  list.clear();
  return list;
}
public List exec3(List list) {
  return new CustomList();
}
```

Listing 13.5: Resultatparameter vermeiden

Beim Aufruf `exec1(meineListe);` ist nicht erkennbar, ob die Liste verändert wird. Ganz im Unterschied dazu wird bei `meineListe = exec2(meineListe);` deutlich, dass diese Methode eine Liste zum Ergebnis hat.

Ob es sich dabei um die veränderte ursprüngliche oder, wie in der dritten Methode, um eine gänzlich andere Liste handelt, bleibt als Implementierungsdetail verborgen. Das ist auch richtig so, denn auf der Anwendungsebene sollte das unerheblich sein.

Resultatparameter sind ein Relikt aus der Vergangenheit. Ältere Programmiersprachen erlauben mit Rücksicht auf effiziente Implementierbarkeit nur einfache Werte bei der Rückgabe. Aus diesem Grund musste ein Weg gefunden werden, um mit strukturierten Werten arbeiten zu können. Meist kam ein *Call by Reference* genanntes Verfahren zum Einsatz, bei dem als Parameter eine Referenz auf eine außerhalb des Unterprogramms vereinbarte Variable übergeben wird. Die Nachteile für die Codequalität wurden schnell erkannt, eine effektivere Lösung war unter den gegebenen Bedingungen jedoch nicht möglich.

Vermeiden Sie Ergebnisparameter. Wenn Methoden den Zustand von etwas ändern müssen, sollte das immer durch Methoden des Objekts geschehen, denen dieses Etwas gehört.

Rückkehrcodes

In den alten Tagen, als Computer noch richtig große Kästen waren, war es üblich, den Abarbeitungserfolg von Unterprogrammen an den Aufrufer durch sogenannte *Rückkehrcodes* zu vermitteln. Auf Ebene des Betriebssystems ist das auch heute noch so. In Ihren Programmen sollten Sie das jedoch vermeiden. Denn Rückkehrcodes haben mindestens zwei große Nachteile:

- Wenn der Wert eines Methodenaufrufs immer ein Code sein muss, können Sie auf diesem Weg keine anderen Ergebnisse mehr liefern. Sie müssten also Ergebnisparameter verwenden, was aber unbedingt vermieden werden sollte.
- Sie müssen den Rückkehrcode an der Aufrufstelle auswerten. Das führt leicht zu unübersichtlich verschachtelten `if-else`-Konstruktionen, in der Art

```
if (methode(a)==ok) {
  if (andereMethode(a)==ok) {…
  } else {
    logger.log("Fehler in andereMethode");
  }
} else {
  logger.log("Fehler in methode");
}
```

Der eigentlich wichtige Code für den normalen Ablauf, nämlich die beiden Methodenaufrufe, geht dabei fast unter.

Mit Rückkehrcodes kommen Sie außerdem schnell in die Versuchung, alle zulässigen Werte, beispielsweise in Form eines `Enum`s, zu zentralisieren. Sie schaffen dadurch zusätzliche Abhängigkeiten. Diese können leicht dazu führen, dass später vorsichtshalber keine neuen Codes mehr ergänzt werden, weil die dann möglicherweise notwendige Neuübersetzung aller abhängigen Klassen zu aufwendig erscheint. Wie Sie mit den Folgen von Fehlern besser umgehen sollten, erfahren Sie in Kapitel 18 *Ausnahmen regeln – Exceptions*.


Verwendwen Sie keine Rückkehrcodes. Für die Fehlerbehandlung sind Exceptions sehr viel besser geeignet.

Seiteneffekte

Eine Methode sollte stets genau das machen, was ihr Name beschreibt – und nicht mehr. Denn jedes Mehr bedeutet eine unerwartete *Nebenwirkung*. In Anlehnung an den englischen Begriff »side effect« wird eine solche Nebenwirkung häufig auch als *Seiteneffekt* bezeichnet.

Listing 13.6 zeigt Ihnen ein gar nicht so abwegiges Beispiel. Die Methode `createExample` erzeugt eben nicht nur – wie der Name sagt – ein neues Objekt, sondern erhöht außerdem noch einen Zähler. Deshalb sollte Sie richtigerweise `createExampleAndIncrementCount` heißen.

```
public static int count;
public Example createExample() {
  Example expl = new Example();
  count++;
  return expl;
}
```

Listing 13.6: Beispiel für einen Seiteneffekt

Beachten Sie bitte, dass es nicht darum geht, Implementationsdetails von Objekten über den Namen nach

außen zu geben. Der Zähler im Beispiel ist aber kein internes Feld eines Objekts, sondern gehört zum globalen Zustand, weshalb seine Veränderung ein typischer Seiteneffekt ist.

Vermeiden Sie Seiteneffekte! Methoden sollen nichts Verborgenes tun. Ein Seiteneffekt ist oft etwas Zweites, was in eine eigene Methode gehört.

Wenn eine Aufteilung nicht geht, zeigen Sie es wenigstens im Namen, dass da noch etwas mehr gemacht wird.

Auswahlanweisungen

Um es ganz klar zu sagen, *Auswahlanweisungen* oder *Switch-Anweisungen* widersprechen in vielerlei Hinsicht den Anforderungen sauberen Codes. Es ist sehr schwer, eine solche Anweisung zu schreiben, die klein ist und nur genau eine Sache erledigt.

Trotzdem können Sie nicht völlig darauf verzichten. Was sie jedoch können, ist, Auswahlanweisungen in untere Abstraktionsniveaus verbannen. Auf den darüber liegenden Ebenen brauchen Sie dann, dank Polymorphismus, keine expliziten Fallunterscheidungen mehr.

Ein natürlicher Platz für Auswahlanweisungen sind *Factory*-Methoden, in denen die jeweils passenden Objekte erzeugt werden.

Lassen Sie mich das anhand des eingangs dieses Kapitels benutzten Beispiels aus den Listings 13.1 bis 13.3 demonstrieren. Bisher ist bei allen Überarbeitungen die `Switch`-Anweisung unverändert geblieben.

Die Methode `CellForPrint.printCell` fällt daher von Umfang und Funktion immer noch aus dem Rahmen. Die Unterscheidung zwischen den unterschiedlichen Zelltypen gehört nicht hierher und wird, wie in Listing 13.7 zu sehen ist, in eine eigene

Konvertierungsfunktion ausgelagert. Auf diese Weise werden die zuvor vermischten Funktionen klar getrennt:

- ✔ Wenn Sie das Druckbild insgesamt ändern wollen, tun Sie das in den Print-Methoden.
- ✔ Wenn Sie einen neuen Zelltyp ergänzen müssen, ist die Factory-Methode dafür zuständig.
- ✔ Wenn Sie schließlich die Darstellung für einen bestimmten Zelltyp bearbeiten wollen, so erfolgt das in der zuständigen Klasse.

```
interface CellForPrint {
  String asPrintString();
}
class CellForPrintFactory {
  static CellForPrint createCellForPrint(final HSSFCell cell)
{
    switch (cell.getCellType()) {
    case HSSFCell.CELL_TYPE_FORMULA:
      return new CellForPrint(){
        public String asPrintString() {
          return cell.getCellFormula();
        }
      };
    case HSSFCell.CELL_TYPE_NUMERIC:
      return new CellForPrint(){
         public String asPrintString() {
          return Double.toString(cell.getNumericCellValue());
        }
      };
    case HSSFCell.CELL_TYPE_STRING:
      return new CellForPrint(){
        public String asPrintString() {
          return cell.getStringCellValue();
        }
      };
```

```
    default:
      return new CellForPrint(){
        public String asPrintString() {
          return "";
        }
      };
    }
  }
}
-------------------------------------------------------------
  void printCell(HSSFCell cell) {
    if (cell!=null) {
      CellForPrint cellPrintString=
              CellForPrintFactory.createCellForPrint(cell);
      printStream.print(cellPrintString.asPrintString());
    }
  }
```

Listing 13.7: Extraktion einer Factory-Methode

Zweckmäßigerweise sollten Sie in der Regel keine anonymen Klassen, sondern erklärende Klassennamen verwenden.

Sobald Sie mehr als eine Methode implementieren müssen, sind vollwertige Klassen ohnehin Pflicht.

Alles fließt

Sehr oft werden sich im Laufe der Zeit die funktionalen Anforderungen für Ihr Projekt ändern. Das kann bitter sein, wenn Sie gerade eine besonders gelungene Struktur für Ihren Code gefunden haben, in der die gewünschte Änderung aber nicht umgesetzt werden kann.

Deshalb möchte ich Ihnen abschließend noch folgende Hinweise mit auf den Weg geben:

- ✔ In jeder Hinsicht optimal geschriebene Methoden gibt es nicht. Je näher Sie einem vermeintlichen Optimum kommen, desto heftiger ist Ihre Struktur durch zukünftige Änderungserfordernisse bedroht. Schreiben Sie einfach guten Code.
- ✔ Die Struktur der Methoden ist nichts Statisches. Mit jeder Änderung oder Weiterentwicklung können sich neue Gesichtspunkte ergeben, die umfangreiche Überarbeitungen rechtfertigen. Führen Sie diese durch, um den Code auf einem gleichbleibenden Qualitätsniveau zu halten.
- ✔ Egal wie viel Erfahrung Sie bereits haben, Sie werden immer noch etwas dazulernen. Setzen Sie das neue Wissen ein.

Das Wichtigste in Kürze

- ✔ Hierarchische Dekomposition ist die Schlüsseltechnik, um komplexe Algorithmen in handhabbare Methoden zu zerlegen.
- ✔ Methoden sind so zu schneiden, dass sie genau eine Sache gut erledigen. Die Unterscheidung zwischen Bearbeitung und Abfrage hilft bei der Umsetzung dieser Regel.
- ✔ Alle Anweisungen einer Methode müssen auf einem Abstraktionsniveau angesiedelt sein.
- ✔ Methoden sollten möglichst wenig Parameter haben. Viele Parameter sind ein Indiz für die Überfrachtung mit mehr als einer Aufgabe.
- ✔ Methoden sollten keine Seiteneffekte haben. Wenn das nicht realisierbar ist, muss der Seiteneffekt im Namen sichtbar gemacht werden.
- ✔ Flag-Parameter sind ein Zeichen dafür, dass eine Methode zwei unterschiedliche Aufgaben erledigt und daher aufgespaltet werden sollte.
- ✔ Ergebnisse von Methoden sind immer als Rückkehrwert zu übermitteln. Ergebnisparameter sind ein Überbleibsel aus

Zeiten, in denen zusammengesetzte Rückkehrwerte nicht erlaubt waren. Sie sollten auf keinen Fall verwendet werden.

- ✔ Rückkehrcodes verschmutzen den aufrufenden Code. Zur Fehlerbehandlung sind Exceptions besser geeignet.
- ✔ Auswahlanweisungen sind nur in Factory- und analogen Methoden zur Erzeugung dedizierter Objekte zulässig. Ansonsten ist Polymorphie anzuwenden.

Kapitel 14

Passend schneiden – Schnittstellen

IN DIESEM KAPITEL

- Die Funktion der verschiedenen Schnittstellenarten
- Die Vorteile kleiner und zusammenhängender Schnittstellen
- Saubere Anbindung externer Software

In diesem Kapitel geht es um Schnittstellen. Schnittstellen sind die Scharniere zwischen den Komponenten einer Anwendung. Qualität und Angemessenheit entscheiden über die Stabilität des Gesamtsystems und über seine Elastizität gegenüber geänderten Anforderungen.

Sauberer Code ist ohne klare und sauber definierte Schnittstellen nicht denkbar.

Die Rolle von Schnittstellen

Schnittstellen beeinflussen die Architektur einer Anwendung auf verschiedenen Wegen. Welche das sind, werde ich Ihnen nun auseinandersetzen.

Mehr als ein Interface

Als Entwickler denken Sie beim Stichwort »Schnittstelle« möglicherweise sofort an die syntaktische Einheit *Interface*. Das ist nicht falsch, und Interfaces sind ein wichtiges Beispiel, aber diese Sicht ist viel zu beschränkt.

Ich verstehe hier den Begriff Schnittstelle viel umfassender. Schnittstellen in diesem Sinn sind neben den Interface-Klassen auch Methoden-Signaturen, Web- und REST-Servicedefinitionen sowie die öffentlichen Felder und Methoden von Klassen und Bibliotheken.

Öffentliche Schnittstelle

Als *öffentliche Schnittstelle* oder *API* werden diejenigen Schnittstellen einer wie auch immer gearteten Softwarekomponente bezeichnet, die dazu bestimmt sind, von anderen Programmen oder Programmteilen benutzt zu werden, ohne dass deren Entwickler den Quellcode der Komponente kennen.

Aus diesem Grund müssen die Klassen und Methoden der öffentlichen Schnittstelle ausreichend dokumentiert sein. Häufig werden die notwendigen Dokumentationen aus den JavaDoc-Kommentaren generiert.

Isoliert betrachtet

Erinnern Sie sich bitte, wie wichtig Abstraktionen für die Beherrschung komplexer Sachverhalte sind. Beim Abstrahieren entfernen Sie, unter Umständen in mehreren Stufen, alles, was in der konkreten Situation unwichtig ist, bis Sie schließlich bei einer überschaubaren Zahl von Eigenschaften enden.

Das, was dabei übrig bleibt, also die restlichen Eigenschaften zusammen mit der durch Sie gewählten Bezeichnung, bildet die *Oberfläche* oder *Schnittstelle* der von Ihnen gebildeten Abstraktion. Damit sind zwei entscheidende Funktionen von Schnittstellen gefunden:

- ✔ Sie definieren eine Abstraktion nach außen.
- ✔ Sie verbergen alle unwesentlichen Details.

Ebenso klar erkennbar sind die kritischen Punkte in diesem Prozess. Auf die Auswahl eines angemessenen Namens gehe ich hier nicht nochmals ein. Diese Frage wird in Kapitel 10 *Nicht Schall und Rauch – Namen* ausführlich beleuchtet. Es bleibt aber noch zu klären, unter welchen Gesichtspunkten und auf welchem

Niveau abstrahiert wird und wie Sie dabei wesentliche von unwesentlichen Eigenschaften unterscheiden können.

In der Programmierpraxis werden Schnittstellen nicht immer konsequent abgegrenzt. Dann sind Dinge sichtbar, die es eigentlich nicht sein sollten. Manchmal bedarf es dazu eines Tricks, manchmal liegt es offen zutage.

Vielleicht haben Sie im Quellcode der Java-Bibliotheken schon einmal den Kommentar »This method is public as an implementation side effect. Do not call or override.« gefunden. Leider lässt es sich nämlich nicht immer vermeiden, dass eine Schnittstelle faktisch mehr umfasst, als sie von der Theorie her sollte.

Halten Sie sich im Zweifel an die erkennbar intendierte Schnittstelle, auch wenn sie technisch realisierte verlockende Möglichkeiten bietet.

Im Verbund

Natürlich hilft Ihnen eine einzelne Abstraktion nur sehr bedingt weiter. Sie wollen ja ein komplexes System modellieren. Dabei zeigt sich dann, wie gut Sie Ihre Schnittstellen ausgewählt haben. In aller Regel werden Ihre Schnittstellen verschiedenen Abstraktionsebenen angehören.

Beim Zusammenbau jedes Teilmodells sollten ausschließlich Abstraktionen einer Ebene verwendet werden. Nur dadurch bleiben Sie unabhängig von speziellen Realisierungen, was ja der tiefere Sinn des ganzen Vorgehens ist. Schließlich soll die gegenseitige Abhängigkeit zwischen den Komponenten möglichst gering oder lose sein.

Lose Kopplung

Mit dem Begriff *lose Kopplung* wird in der Softwarearchitektur ein Zustand beschrieben, bei dem die verbundenen Komponenten nur über eine möglichst schmale Schnittstelle verbunden sind. Die Verbindung wird dabei einzig und

allein durch die Schnittstelle definiert, sodass sich die beteiligten Komponenten gegenseitig nicht zu kennen brauchen – und auch nicht sollen.

Lose Kopplung wird generell als erstrebenswertes Ziel angesehen, weil es die eigenständige Wartung und Weiterentwicklung der Teile erleichtert. Es gibt allerdings auch nicht zu unterschätzende Nachteile. Dazu gehört vor allem der höhere Entwicklungsaufwand, weil jede über die Schnittstelle verfügbar gemachte Funktion vollständig implementiert und getestet werden muss, auch wenn das weit über die momentan erforderliche und tatsächliche Nutzung hinausgeht.

Außerdem können eventuell notwendige Datenkonvertierungen und -prüfungen den Aufwand zur Laufzeit erhöhen. Letzteres gilt besonders für Schnittstellen, die ein externes Textformat wie XML oder JSON verwenden.

Das größte Problem bereitet jedoch eine hinreichend genaue, Missverständnisse ausschließende Definition der Semantik, die der Schnittstelle zugrunde liegt.

Das Gerüst der Schnittstellen definiert die Architektur der Anwendung, von der Ebene der Methoden bis zur Ebene der Nutzerinteraktionen. Behalten Sie das stets im Auge und denken Sie daran, dass es später viel leichter ist, eine schlecht implementierte Funktion zu verbessern, als eine schlechte Schnittstelle durch eine bessere zu ersetzen.

In den frühen Entwicklungsphasen, wenn ohnehin noch »alles im Fluss« ist, sollten Sie bei Überarbeitungen Ihre Energie deshalb vorrangig dafür einsetzen, die Schnittstellen zu verbessern.

Komponenten

Schnittstellen sind die natürlichen Grenzen von Teilen oder Komponenten. Sie grenzen logische Einheiten ab. Orientieren Sie sich bei der Organisation Ihrer Testfälle daran.

Vergessen Sie dabei bitte nicht, dass Ihre Komponente eigenständig ist und deshalb alle Funktionen der Schnittstelle berücksichtigt werden müssen, nicht nur diejenigen, die im aktuellen Projekt gerade gebraucht werden. *Contract-based Testing* ist eine Methode, die genau für diese Aufgabe entwickelt wurde.

Wenn Sie diese Strategie gewissenhaft umsetzen, gewinnen Sie die beruhigende Sicherheit, dass bei Überarbeitungen und Weiterentwicklungen sehr wahrscheinlich keine außerhalb der betroffenen Komponente liegenden Funktionalitäten beeinträchtigt werden.

Denken Sie beim Testen von Schnittstellen daran, dass ein Anwender alle definierten Funktionen nutzen kann und möglicherweise auch nutzen wird.

Die Schnittstelle ist ein Angebot für einen Vertrag, den Sie bei jeder Realisierung erfüllen müssen. Versuchen Sie deshalb, eine möglichst vollständige Abdeckung aller Möglichkeiten zu erreichen.

Interface Segregation

Das *Interface-Segregation-Prinzip* (ISP) ist ein Grundbaustein der objektorientierten Programmierung. Es dient dem Ziel, modulare Anwendungen zu entwickeln, deren Bestandteile möglichst wenige Abhängigkeiten untereinander haben. Kompakte Interfaces bieten viele Vorteile. Aber wie bei jedem guten Vorsatz steckt auch beim Umsetzen des ISP der Teufel in den Details.

Das *Interface-Segregation-Prinzip* besagt, dass Klienten einer Schnittstelle nicht gezwungen werden sollten, von Methoden abhängig zu sein, die sie gar nicht benötigen.

Im Folgenden werde ich mich der Übersichtlichkeit wegen auf Interfaces im Sinne der Java-Syntax beschränken. Die angeführten Überlegungen gelten jedoch ganz allgemein für alle Arten von Schnittstellen.

Schlanke Schnittstellen

Schlanke Schnittstellen erleichtern es Ihnen, Ihre Software klar zu strukturieren und dabei gleichzeitig die Abhängigkeiten zwischen

den Teilen klein zu halten.

Keine Überforderung

Schnittstellen sollten nicht mit Funktionalität überfrachtet werden. Ein ausufernder Umfang ist oft das sichtbare Zeichen konzeptioneller Defizite. Sie erinnern sich, dass Schnittstellen sozusagen das »Gesicht« von Abstraktionen sind, die eingeführt wurden, um komplexe Dinge besser behandeln zu können.

Das funktioniert aber bloß, wenn die Abstraktion dem mentalen Fassungsvermögen des Menschen angepasst ist. Letzteres ist dummerweise recht beschränkt. Große Schnittstellen erhöhen deshalb auch die Gefahr, dass infolge mentaler Überforderung etwas übersehen wird.

Weniger Abhängigkeiten

Breite Schnittstellen können leicht dazu führen, dass zwischen den beteiligten Elementen starke Abhängigkeiten entstehen. Das kann Ihnen vor allem bei der späteren Weiterentwicklung Probleme bereiten, verschlechtert also die Wartbarkeit des Codes.

Wenn Sie schlanke Schnittstellen entwickeln wollen, zwingt Sie das ganz automatisch zu einer genaueren Analyse. An deren Ende sollten Sie erkannt haben, welche Beziehungen essenziell sind und welche abgeleitet und daher verzichtbar. Schnittstellen, die Sie auf diese Weise definieren, werden zwischen ihren Elementen weniger, aber klar ausgeprägte Abhängigkeiten haben.

Testbarkeit

Je schlanker eine Schnittstelle ist, desto einfacher ist für Sie der Test. Da es wenige Funktionen gibt, müssen Sie auch nur wenige Wechselwirkungen beachten. Das ermöglicht Ihnen, einen hohen Testabdeckungsgrad zu erreichen.

Bei sehr breiten Schnittstellen kann allein die hohe Zahl von potenziellen Wechselwirkungen die eigentlich notwendige Abdeckung unmöglich machen.

Betrachten Sie beispielsweise ein Interface aus dem Java-Collections-Framework und versuchen Sie, abzuschätzen, wie viele Tests Sie schreiben müssten, um eine eigene Implementation abzusichern.

Da in der Regel nur ein kleiner Teil der Methoden wirklich benutzt wird, gibt es in vielen Projekten Implementierungen, die nur für diese ausgewählten Methoden getestet sind oder sogar nur diese implementieren. Das widerspricht natürlich ganz klar dem Zweck einer Schnittstellendefinition, ist aber eine beinahe unvermeidliche Folge des großen Umfangs.

Kohäsion

Falls ich Sie nun so weit überzeugen konnte, dass Sie eine bewusstere Gestaltung von Schnittstellen versuchen wollen, stellt sich schnell die Frage: Woran erkenne ich, dass eine Schnittstelle das richtige Maß hat?

Das Zauberwort für die Beantwortung dieser Frage lautet *Kohäsion*. Die Elemente einer Schnittstelle sollen »möglichst eng« zusammenhängen. Das ist ein qualitatives Maß, das ich im Folgenden näher erläutern werde.

Abstraktion

Ich habe bereits gesagt, dass Schnittstellen die programmtechnische Realisierung von abstrakten Konzepten sind. Daraus ergibt sich als erste Charakterisierung für einen engen Zusammenhang:

- ✔ Eine Schnittstelle verkörpert genau ein abstraktes Konzept.
- ✔ Alle Elemente der Schnittstelle sind Teile dieses Konzepts.

Dieser Grundsatz wird leider häufig verletzt. Möglicherweise haben Sie bereits wie die meisten Entwickler, die Funktion einer IDE genutzt, um aus einer Klasse ein Interface zu extrahieren. Üblicherweise denkt man dabei zuerst daran, welche Methoden

benötigt werden, und eher selten daran, welche Abstraktion dem Interface zugrunde liegt.

Interfaces, die auf diese Weise definiert werden, sind oft weniger kohärent und stringent als solche, die bewusst vor der Implementierung angelegt werden. Ein sehr wichtiger Gesichtspunkt geht dabei nämlich leicht unter: die begriffliche Abgrenzung der Abstraktion. Die Definition einer solchen ist weitgehend willkürlich, braucht aber ein gemeinsames Verständnis.

Selbst scheinbar allgemein verstandene Begriffe wie »Liste« erlauben im Detail unterschiedliche Interpretationen. Für sauberen Code brauchen Sie jedoch eine exakte Festlegung.

Dass die erwähnte Abgrenzung alles andere als einfach ist, demonstriert Javas Collection-Framework. Dazu gehört beispielsweise auch die Klasse `Stack`. Ganz abgesehen davon, dass hinter diesem Namen besser ein Interface und nicht eine von `Vector` abgeleitete Klasse stecken sollte, ergibt sich die Frage: Ist ein Stack wirklich eine Collection?

Theoretische Betrachtungen definieren einen Stack gewöhnlich nur durch die Operationen `empty`, `pop`, `push` und `top`. Von den Collections kommen jedoch mindestens Iterierbarkeit und Elementanzahl dazu – doch ist das richtig? Sie sehen damit am konkreten Beispiel, in welche Art von Problemen Sie beim nachträglichen »Einbau« von Interfaces, wie das für die Collections erfolgt ist, laufen können.

Versuchen Sie immer, Schnittstellen ganz bewusst von der zugrunde liegenden Abstraktion her zu definieren. Insbesondere beim nachträglichen Extrahieren eines Interfaces aus einer bereits vorliegenden Klasse kann das einige Abänderungen verursachen. Lassen Sie sich dadurch nicht davon abhalten, sauber zu arbeiten.

Ich will nicht verhehlen, dass es manchmal schwierig bis unmöglich sein kann, eine allseits zufriedenstellende Lösung zu finden. Das gilt besonders dann, wenn Sie allgemein übliche Begriffe zu erfassen und in ihrer Beziehung darzustellen versuchen.

Das Collection Framework von Java ist auch für diese prinzipiellen Probleme ein hervorragendes Beispiel. Bereits das Vorhandensein einer `UnsupportedOperationException` verweist auf das Dilemma. Alle sechs Operationen, mit denen eine Collection modifiziert werden kann, sind – einzeln – als optional gekennzeichnet.

Deshalb ist es auch nicht möglich, beispielsweise aus einem erfolgreichen `remove`-Aufruf zu schließen, dass ein `clear()` ebenfalls ausgeführt werden kann. Die Konsequenz, dass es selbst bei korrekter Verwendung des Interfaces zu Laufzeitfehlern kommen kann, ist höchst unbefriedigend.

Allerdings ist es tatsächlich kaum möglich, das Verhältnis von änderbaren zu nicht änderbaren Collections vernünftig zu definieren. Erben die nicht änderbaren von den änderbaren, wie das der Stand ist, entsteht das beschriebene Problem mit unzulässigen Operationen.

Die umgekehrte Vererbungshierarchie würde zu schwer vermittelbaren Folgen führen, weil sich die Nichtänderbarkeit auf das Fehlen von Änderungsoperationen beschränken müsste. Eine Methode beispielsweise, die eine nicht änderbare Liste als Parameter hat, könnte diese dann zwar selbst nicht modifizieren, müsste sie aber so behandeln, als ob sie änderbar wäre. Schließlich könnte das tatsächlich übergebene Objekt ja eine änderbare Liste sein, die im ungünstigsten Fall in einem parallel laufenden Thread modifiziert wird.

Das Notwendige

Eine gute Schnittstelle beschreibt genau eine abstrakte Einheit. Das kann wie im Falle der funktionalen Interfaces eine zustandslose Berechnungsfunktion sein oder ein zustandsbehaftetes Objekt, beispielsweise eine Zeichenkette. Nachdem Sie diesen Gegenstand ausgewählt haben, müssen Sie sich fragen: Welche Operationen sollten dafür definiert werden?

Bei der Beantwortung gehen Sie zunächst rein pragmatisch vor. Schließlich definieren Sie die Schnittstelle aus einem ganz bestimmten Grund: Sie wollen dadurch eine konkrete Anforderung erfüllen. Aus dieser Anforderung ergeben sich notwendige Operationen. Wenn Sie alles beisammenhaben, ist es an der Zeit für eine kritische Überarbeitung.

Stellen Sie sich dazu folgende Fragen:

- ✔ Können alle notwendigen Operationen ausgedrückt werden?
- ✔ Sind alle Methoden auf der gleichen Abstraktionsebene angesiedelt?
- ✔ Gibt es Methoden, die sich gegenseitig ersetzen können? Oder, anders gefragt, gibt es überflüssige Methoden?
- ✔ Gibt es Methoden, die problemlos weiter gefasst oder verallgemeinert werden können?
- ✔ Sind bei den Methoden jeweils wirklich alle Parameter erforderlich?
- ✔ Können Parameter- und Resultatklassen durch allgemeinere Klassen oder Interfaces ersetzt werden?

Das ungeliebte Kind

Ein schönes Beispiel für ein gut zugeschnittenes Interface stellt `CharacterSequence` dar. Es enthält alle Methoden, die zur Bearbeitung einer Zeichenfolge gewöhnlich benötigt werden. Unglücklicherweise wurde es erst mit Java 4 eingeführt. Es wird selbst in den Java-Bibliotheken nur selten verwendet und ist daher auch unter Entwicklern viel zu wenig bekannt.

Mark Reinhold, Java-Chefarchitekt, musste einräumen, dass dieses nützliche Interface das Schicksal eines »unbeloved child« fristet. An vielen Stellen

könnte der Typ `String` problemlos durch `CharacterSequence` ersetzt werden. Dadurch würden nicht nur zahllose `toString()`-Aufrufe überflüssig, sondern durch das Einsparen von Konvertierungen wäre gleichzeitig ein Leistungsgewinn möglich. Bei textintensiven Prozessen wie beispielsweise der Serialisierung/Deserialisierung großer Datenmengen mit XML oder JSON sind derartige Vorteile schnell spürbar.

Auf alle diese Fragen werden Sie oft genug nicht *die* eine richtige Antwort finden. Manches hängt von den Umständen ab. Eine funktional vollständige Schnittstelle kann trotzdem Probleme verursachen, wenn sie nicht performant genug ist. In solchen – seltenen – Fällen ist es möglicherweise angebracht, ergänzende Methoden, meist auf einem niedrigeren Abstraktionsniveau, zu definieren, die bei Bedarf verwendet werden können.

Am Beispiel `CharSequence` lässt sich gut zeigen, wie Sie ein Interface sinnvoll ergänzen könnten, um in bestimmten Fällen eine bessere Leistung zu erreichen:

```
public interface CharSequence {
    int length();
    char charAt(int index);
    CharSequence subSequence(int start, int end);
    String toString();
// diese Methode gibt es im Original nicht:
    char[] toCharArray();}
```

Sie können diese Methode bei Bedarf problemlos selbst implementieren:

```
char[] toCharArray(CharSequence s){
    char[i] ca= new char[s.length()];
    for (int i= 1; i<s.length(); i++) {
       ca[i]= s.charAt(i);
    }
    return ca;
 }
```

Da aber praktisch alle Implementierungen von `CharSequence` intern bereits ein Char-Array benutzen, könnte die vorgeschlagene Methode unter Verwendung von `System.arraycopy` deutlich einfacher und besser optimierbar umgesetzt werden. Wenn Sie sehr viele lange neue Zeichenketten erzeugen müssen, kann das nützlich sein.

Vergessen Sie aber bitte nicht, dass dieses Beispiel nur zeigt, welcher Art eine entsprechende Ergänzung sein kann. In den allermeisten Fällen werden Sie Derartiges nicht brauchen.

Das Überflüssige

> Wann immer du einen Satz kürzen kannst, tu es. (Anatole France)

Sich kurzzufassen und nur das Wesentliche zu erkennen, ist eine schwierige Kunst. Daher ist es nur allzu verständlich, dass neu formulierte Schnittstellen Überflüssiges enthalten und ihnen manchmal sogar Notwendiges noch fehlt. Das können Sie genauer überprüfen, indem Sie eine erste, provisorische Realisierung für beide Seiten in Angriff nehmen.

Mit den beiden Seiten meine ich zum einen den Anbieter, also denjenigen, der die Schnittstellenfunktionen realisiert, und zum anderen einen Klienten, der die angebotenen Funktionen nutzt.

Überprüfen Sie anschließend Ihre Schnittstelle im Licht der gewonnenen Erfahrung. Viele Schnittstellen enthalten Bestandteile, die derzeit gar nicht gebraucht werden, »aber später vielleicht einmal nützlich sein könnten«.

Vermeiden Sie Spekulationen. Das Meiste, was in der Zukunft vielleicht gebraucht werden könnte, wird nie verwendet werden, sei es aus Unkenntnis – weil keiner sich erinnert, dass Sie das schon vorbereitet haben – oder wegen völlig anders gearteter Anforderungen.

Vermüllen Sie Ihre Schnittstellen nicht mit Dingen, für die absehbar kein wirklicher Bedarf besteht.

Auch wenn es Ihnen schwerfällt, weil Sie vielleicht viel Energie auf die Definition von ein paar besonders »schönen« Methoden verwendet haben, entfernen Sie alles, was nicht wirklich benötigt wird, und trösten Sie sich damit, dass die wirksamsten Verbesserungen am Code oft durch Streichungen zustande kommen.

Streichen heißt allerdings nicht, dass Sie alles sofort und endgültig verwerfen. Vielleicht ist ja Ihre gut ausgedachte Methode nur an dieser Stelle falsch und in einer anderen Schnittstelle genau richtig.

Kombination

Schlanke Schnittstellen haben allerdings zur Folge, dass an manchen Stellen eine Definition nicht ausreicht, weil Sie gerade die Kombination aus mehreren Schnittstellen benötigen. Sie können dieses Problem lösen, indem Sie kombinierte Schnittstellen definieren, die von allen benötigten erben.

Mehrfachvererbung von Interfaces ist auch in Java kein Problem. Der Nachteil dieses Vorgehens liegt darin, dass Sie zusätzliche Schnittstellen einführen, deren Existenz, zumindest teilweise, nur technisch begründet ist. So etwas sollte in sauberem Code besser vermieden werden.

Ein weiteres Manko ergibt sich, wenn die Klassen bereits vorhanden sind, die die benötigten Interfaces implementieren, und Sie keine Möglichkeit haben, diese Klassen zu ändern. Java bietet Ihnen mit den *Generics* einen Weg, der Ihnen verschiedentlich hilft, das beschriebene Dilemma zu vermeiden.

Mit den Klassen `StringBuffer` und `StringBuilder` verfügt Java über zwei, abgesehen von der Synchronisierung, gleichwertige Verfahren, Zeichenketten zu konstruieren. Beide erben von der gleichen Superklasse, die aber leider

package-private ist und deshalb von Ihnen nicht genutzt werden kann.

Falls Sie nun eine Methode haben, die auf den String-Erzeuger lesend und schreibend zugreift und mit beiden Klassen funktionieren soll, müssen Sie diese eigentlich zweimal schreiben: einmal für jeden Typ.

Wenn Sie sich allerdings etwas einschränken und auf einige Bequemlichkeitsmethoden verzichten, gibt es eine bessere Lösung. Beide Klassen implementieren nämlich sowohl `CharSequence` als auch `Appendable`.

Der Verzicht besteht also darin, sich auf die Methoden dieser Interfaces zu beschränken. Dann können Sie zum Beispiel folgenden Code schreiben, wobei der generische Typparameter der Konvention entsprechend mit `AC` bezeichnet wird:

```
public <AC extends Appendable & CharSequence> AC
    appendXX(AC sb, String xx) {
  try {
    if (',' != sb.charAt(sb.length()-1)){
      sb.append(',');
    }
    sb.append(xx);
  } catch(IOException e){throw new RuntimeException(e);}
  return sb;
}
```

Diese Methode akzeptiert `StringBuffer`- und `StringBuilder`-Objekte als Parameter.

Einen kleinen Schönheitsfehler bringt das `Appendable`-Interface leider mit sich. Damit es auch von Streams implementiert werden kann, muss die Signatur der `append`-Methode eine `IOException` aufweisen. Das ist gewissermaßen eine »Altlast«, die durch das nachträgliche Einfügen der Interfaces entstanden ist.

Lassen Sie sich dieses negative Beispiel als Ermahnung dienen, Schnittstellen immer möglichst frühzeitig festzulegen.

Keine Missverständnisse

Schnittstellen beschreiben Kommunikationskanäle zwischen Anbieter und Benutzer einer Funktion, aber auch zwischen verschiedenen Entwicklern. Um diesen zweiten Aspekt geht es mir in diesem Abschnitt.

Kommunikation ist stets durch Missverständnisse bedroht, das heißt, es besteht die Gefahr, dass der Empfänger einer Nachricht diese anders versteht als vom Sender beabsichtigt. Dieses Risiko kann nicht völlig beseitigt werden, aber Sie sollten alles tun, um es möglichst klein zu halten.

Exakte Beschreibung

In aller Regel sind Schnittstellendefinitionen syntaktischer Natur. Sie legen fest, welchen formalen Kriterien die beteiligten Elemente genügen müssen. Die Semantik, das heißt die Bedeutung dahinter, kann nur durch möglichst gut gewählte Namen vermittelt werden.

Nun ist es allerdings so, dass es im Alltag praktisch keine hinreichend genau definierten Bezeichnungen gibt. Selbst mathematische Artikel beginnen gewöhnlich mit der Abgrenzung der verwendeten Definitionen.

Da eine Schnittstelle aber keine wissenschaftliche Abhandlung ist, müssen Sie auf andere Weise versuchen, dem Leser des Codes Ihre Überlegungen möglichst genau zu vermitteln.

Voraussetzungen aufführen

Mit jeder Schnittstelle definieren Sie eine begriffliche Einheit *Ihres* Denkens auf der Basis *Ihrer* Vorstellungen von der Welt. Diese Vorstellungen werden niemals völlig mit denen eines anderen Menschen übereinstimmen. Deshalb ist es außerordentlich wichtig, dass Sie wesentliche implizite Annahmen sichtbar

machen. Das kann durch Kommentare, Testfälle oder Beispiele erfolgen.

Denken Sie dabei daran, dass es nicht reicht, positive Fälle darzustellen. Versuchen Sie, die Grenzen zu beschreiben: Was gilt noch, und was gilt nicht mehr?

Zeichenketten sind ein Beispiel dafür, wie wichtig eine genaue Beschreibung der angenommenen Voraussetzungen ist. Bei Java wurde relativ erfolgreich versucht, eine möglichst weitgehende Abdeckung aller möglichen Anwendungsfälle zu erreichen, um den Preis einer recht komplexen und nicht leicht zu verstehenden Umsetzung.

Denn Zeichenketten werden nicht nur als Folgen von Zeichen verstanden, sondern in gewissem Sinne als *Texte*. Das heißt, sie haben zusätzliche Eigenschaften. Beispielsweise erfolgt die Umwandlung zwischen Klein- und Großbuchstaben sprachabhängig, sodass `"ß".toUpperCase()` – liefert SS – und `Character.toUpperCase('ß')` – liefert ß – verschiedene Ergebnisse haben. (Auch wenn dieses Beispiel in Zukunft durch den Großbuchstaben ß obsolet werden sollte, gibt es in anderen Sprachen Vergleichbares.)

Insbesondere Entwickler, die hauptsächlich mit ASCII-Zeichenketten arbeiten, sind sich dieser Komplexität oft nicht ausreichend bewusst oder berücksichtigen sie nicht, ohne die entsprechende Einschränkung hinreichend deutlich zu formulieren.

Eine Eigenschaft, die daher nicht jedem sofort klar sein dürfte, ist, dass im Allgemeinen

```
einString.length() == einString.toUpperCase().length()
```

nicht gilt. Nämlich immer dann, wenn der String Zeichen enthält, die bei der Umwandlung in mehrere Zeichen konvertiert werden.

Beschreiben Sie möglichst genau, welche Eigenschaften für eine Schnittstelle gelten und welche nicht. Insbesondere solche Eigenschaften, deren Fehlen nicht unmittelbar einsichtig ist, müssen deutlich benannt werden.

Vollständige Definition

Definieren Sie Ihre Schnittstellen vollständig. Sie tun niemandem einen Gefallen, wenn Sie etwas offenlassen, »um effiziente Implementierungen zu ermöglichen«. Derartige Formulierungen sind Ihnen bestimmt schon begegnet.

Was sich auf den ersten Blick wie Freiheit und Großzügigkeit anhört, verbirgt tatsächlich einen gewaltigen Pferdefuß. Denn Code verhält sich bei der Ausführung – zumindest in der Regel – nicht »undefiniert«. Irgendetwas passiert immer, und damit müssen Sie umgehen.

Konsequenter Umgang mit undefiniertem Verhalten oder einem undefinierten Wert würde erfordern, alle denkbaren Möglichkeiten in Betracht zu ziehen – und zu testen. Abgesehen davon, dass Sie wahrscheinlich doch eine Möglichkeit vergessen würden, ist das schon vom Umfang her selten machbar.

Lücken in der Definition sind Einfallstore für Fehlinterpretationen und daraus folgende Missverständnisse. Es ist schon schwer genug, unbeabsichtigte Lücken zu vermeiden. Bauen Sie deshalb nicht auch noch bewusst weitere ein.

Tests und Mocks

Auch wenn es nicht immer möglich ist, sind Testfälle und *Mocks* eine hervorragende Möglichkeit, Schnittstellen inhaltlich zu beschreiben.

Ein *Mock*, vom englischen Wort für Attrappe, ist ein Objekt, das nach außen hin zwar eine Schnittstelle realisiert, aber ohne echte Funktionalität.

So könnte ein Mock für einen Zufallszahlengenerator immer die gleiche Zahl liefern. Mocks werden vor allem beim Testen genutzt, um definierte Umgebungen für das zu testende Element bereitzustellen.

Der größte Vorteil besteht darin, dass es sich nicht um Kommentare oder Ähnliches, sondern ebenfalls um Code handelt. Der kann auf Compile-Fehler geprüft, ausgeführt und versioniert werden. Sie können so bereits Tests schreiben, ohne dass eine konkrete Realisierung der Schnittstelle verfügbar ist.

Nutzen Sie deshalb diese Chance. Leider gibt es auch Situationen, in denen dieser Vorschlag relativ wenig bringt, weil die betroffene Schnittstelle eine Informationssenke ist, das heißt, sie nimmt nur Daten entgegen. Beispiel für eine solche Senke ist das `Appendable`-Interface.

Als Testrahmen benutze ich hier das weit verbreitete JUnit-Framework. Das folgende Beispiel zeigt die Grundstruktur für Unittests, die eine Interface-Definition ergänzen können, anhand des `CharSequence`-Interfaces. Mit dem Interface könnte der folgende abstrakte Testfall ausgeliefert werden:

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
public abstract class AbstractCharSequenceTest {

  abstract CharSequence createNonEmpty();
    @Test
  public void testSubSequence() {
    CharSequence sequence= createNonEmpty();
    int length= sequence.length();
    for (int i= 0; i < length; i++) {
```

```
            assertEquals(sequence.toString().substring(i),
                sequence.subSequence(i, length).toString());
        }
    }
}
```

Eine Implementation kann dann einfach getestet werden, beispielsweise so:

```
public class StringBufferTest extends
AbstractCharSequenceTest {

  @Override
  CharSequence createNonEmpty() {
    return new StringBuffer("Ein String zum Testen");
  }
}
```

Tests dienen nicht nur der Codequalität. Sorgfältig entworfene abstrakte Testfälle können das Verständnis von Schnittstellen entscheidend verbessern.

Kein Code ohne Fremdcode

Abgrenzung zwischen Komponenten ist wichtig, um bei vorzunehmenden Änderungen unbeabsichtigte Beeinflussungen zu verhindern. Wenig offensichtliche Grenzen begünstigen das Übersehen von Abhängigkeiten. So gut wie keine Anwendung kommt ohne die Einbindung von Fremdcode aus. Streng genommen sind ja bereits die mit einer Programmiersprache ausgelieferten Bibliotheken »fremder Code«, der nicht von Ihnen geschrieben wurde und dessen Wartung nicht in Ihrer Hand liegt. Der Einfachheit halber wird dieser Code oft völlig bedenkenlos verwendet.

Im Allgemeinen entstehen daraus auch keine ernsthaften Probleme, selbst wenn für viele Instanzen von `Map` oder `List` nur ein sehr kleiner Teil der Funktionen benötigt wird.

Etwas anders sieht es aus, wenn Sie diese Interfaces mit eigenen Implementierungen benutzen. Dann wird Ihr Code plötzlich extrem von diesem Fremdcode abhängig. Das ist kein abstraktes Beispiel. Ich kenne Code, der `java.sql.Statement` implementiert und bereits mehrmals ergänzt werden musste, weil im Zuge der Java-Weiterentwicklung neue Methoden hinzugekommen sind.

Eine so direkte Abhängigkeit von Fremdcode sollte deshalb vermieden werden.

Eine unsichtbare Grenze

Durch die Nutzung des Fremdcodes entsteht eine weitgehend unsichtbare oder nicht deutlich wahrnehmbare Grenze zwischen dem Code, der Ihnen gehört, und diesem fremden Code.

Die Schnittstellen fremden Codes sind unterschiedlich stabil. Bei zur Programmiersprache gehörigen Bibliotheken ändern sich Namen und Signaturen nur selten, allerdings sind Erweiterungen häufig.

Für andere Software lässt sich das so nicht sagen. Da müssen Sie sogar damit rechnen, dass irgendwann ein Austausch erfolgen muss, weil die ursprünglich ausgewählte Bibliothek nicht mehr gepflegt wird oder sich deren Nutzungsbedingungen nachteilig ändern. Spätestens dann wird die schlechte Sichtbarkeit der Grenze zum Problem.

- Lassen Sie keine »unsichtbaren Grenzen« zu. Machen Sie die Schnittstellen zu Fremdcode sichtbar.
- Vermeiden Sie Ableitungen von im Fremdcode definierten Klassen. Nutzen Sie stattdessen Delegation.

Delegation

Delegation ist eine Form der Objektkomposition. Sie ermöglicht es unter anderem, Klassenvererbung durch Objektkomposition zu ersetzen. Bei der Delegation werden Methodenaufrufe an ein *Delegationsobjekt* weitergereicht.

Da der Aufbau der Objektstruktur erst zur Laufzeit erfolgt, sind hochgradig flexible Lösungen möglich.

Diese Flexibilität kann sich allerdings auch nachteilig auswirken, da der Code dadurch schwieriger zu verstehen ist. Delegation sollte deshalb vorrangig im Rahmen bekannter Entwurfsmuster benutzt werden.

Abhängigkeiten isolieren

Um Probleme der oben aufgeführten Art zu vermeiden und gleichzeitig die Grenzen im Code klar erkennbar zu machen, ist es wichtig, Abhängigkeiten zu isolieren. Das geht, indem Sie den Fremdcode hinter einer eigenen Schnittstelle verbergen.

Praktisch handelt es sich dabei um eine Anwendung des Entwurfsmusters *Adapter Pattern*. Die Schnittstellen des fremden Codes werden dabei auf die eigenen abgebildet.

Definieren Sie die eigene Schnittstelle so schmal wie möglich, sodass diese nur die benötigte Funktionalität umfasst. Das vereinfacht nicht nur den Adapter, sondern macht Sie auch flexibler, wenn Sie den Fremdcode einmal austauschen müssen.

Wenn Sie zusätzlich Ihre Schnittstellendefinition mit Mocks und Testfällen ergänzen, können Sie die Entwicklung Ihres Codes noch stärker vom Fremdcode entkoppeln.

Die Mocks erlauben es, Ihren Code unabhängig von der Verfügbarkeit der externen Software zu testen. Und mithilfe der Testfälle können Sie leicht überprüfen, ob neu gelieferte Versionen des Fremdcodes noch Ihren Anforderungen genügen.

Entwurfsmuster *Adapter Pattern*

Dieses Muster beschreibt die Verbindung zweier nicht miteinander kompatibler Schnittstellen durch einen Adapter. Dabei werden die Methoden des durch den Adapter implementierten Interfaces in Aufrufe von Methoden des adaptierten Interfaces umgesetzt.

Je nach Standpunkt sind für derartige Konvertierer auch die Bezeichnungen *Wrapper* oder *Proxy* üblich. Es handelt sich dabei stets um eine Form der Delegation.

Wie es gehen könnte

Zum Abschluss dieses Kapitels komme ich noch einmal auf das Beispiel aus Kapitel 13 *Kleine Schritte – saubere Methoden* zurück. Bei allen Schritten zur Verbesserung der Struktur wurde die Integration der Bibliothek für den Zugriff auf die Excel-Daten unberührt gelassen. In einem letzten Umbau wird nun auch dieses Defizit behoben werden.

Das Beispiel importiert vier Klassen des Fremdcodes:

```
import org.apache.poi.hssf.usermodel.HSSFCell;
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
```

Für diese Klassen werden jetzt eigene Wrapper definiert. Zweckmäßigerweise wird dafür ein neues Package angelegt.

```
package de.lmp.dummies.exceladapter;
import java.io.IOException;
import java.io.InputStream;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

public class Workbook {

  private HSSFWorkbook hssfWorkbook;

    public Workbook(HSSFWorkbook hssfWorkbook) {
    this.hssfWorkbook= hssfWorkbook;
  }
    public int getNumberOfSheets() {
    return hssfWorkbook.getNumberOfSheets();
  }
  public Sheet getSheetAt(int idx) {
```

```
    return new Sheet(hssfWorkbook.getSheetAt(idx));
  }
  public static Workbook readWorkbook(InputStream
inputStream)
      throws IOException {
    return new Workbook(new HSSFWorkbook(inputStream));
  }
}
```

Listing 14.1: Klasse `Workbook`

```
package de.lmp.dummies.exceladapter;
import org.apache.poi.hssf.usermodel.HSSFSheet;

public class Sheet {
  private final HSSFSheet hssfSheet;

  public Sheet(HSSFSheet hssfSheet) {
    this.hssfSheet= hssfSheet;
  }
 public int getNumberOfRows() {
   return hssfSheet.getPhysicalNumberOfRows();
 }
 public Row getRowAt(int idx) {
   return new Row(hssfSheet.getRow(idx));
 }
}
```

Listing 14.2: Klasse `Sheet`

```
package de.lmp.dummies.exceladapter;
import org.apache.poi.hssf.usermodel.HSSFRow;

public class Row {
  private final HSSFRow hssfRow;

  public Row(HSSFRow hssfRow) {
    this.hssfRow= hssfRow;
  }
```

```
  public int getNumberOfCells() {
    return hssfRow.getPhysicalNumberOfCells();
  }
  public CellForPrint getCellAt(int idx) {
    return CellForPrintFactory.
        createCellForPrint(hssfRow.getCell(idx));
  }
}
```

Listing 14.3: Klasse `Row`

Da es von den Zellen verschiedene Ausprägungen gibt, deren Unterscheidung für die Beispielanwendung jedoch nicht wichtig ist, wird das bereits eingeführte Interface `CellForPrint` in das Adapter-Package verschoben.

```
package de.lmp.dummies.exceladapter;
interface CellForPrint {
  String asPrintString();}
```

Listing 14.4: Interface `CellForPrint`

Nun fehlt nur noch die Factory (Listing 14.5) für die Erzeugung der benötigten Zellenobjekte.

```
package de.lmp.dummies.exceladapter;
import org.apache.poi.hssf.usermodel.HSSFCell;

class CellForPrintFactory {
  static CellForPrint createCellForPrint(final HSSFCell cell)
{
    if (cell!=null) {
      switch (cell.getCellType()) {
      case HSSFCell.CELL_TYPE_FORMULA:
        return createFormulaCell(cell);
      case HSSFCell.CELL_TYPE_NUMERIC:
        return createNumCell(cell);
      case HSSFCell.CELL_TYPE_STRING:
        return createStringCell(cell);
      default:
```

```
        return createEmptyCell();
      }
    } else {
      return createEmptyCell();
    }
  }
  static CellForPrint createFormulaCell(final HSSFCell cell)
{
    return new CellForPrint() {
      public String asPrintString() {
        return cell.getCellFormula();
      }
    };
  }
  static CellForPrint createNumCell(final HSSFCell cell) {
    return new CellForPrint() {
      public String asPrintString() {
        return Double.toString(cell.getNumericCellValue());
      }
    };
  }
  static CellForPrint createStringCell(final HSSFCell cell) {
    return new CellForPrint() {
      public String asPrintString() {
        return cell.getStringCellValue();
      }
    };
  }
  static CellForPrint createEmptyCell() {
    return new CellForPrint() {
      public String asPrintString() {
        return "";
      }
    };
  }
}
```

Listing 14.5: Klasse `CellForPrintFactory`

Durch die Auslagerung der Factory in das Adapter-Package hat sich die Kernanwendung, wie in Listing 14.6 dargestellt, weiter vereinfacht. Alle Importe aus fremden Bibliotheken sind verschwunden. Damit können Sie in Zukunft sicher sein, dass dieser Code bei etwaigen Veränderungen im Fremdcode **nicht** angefasst werden muss. Ein Austausch des Fremdcodes kann völlig unabhängig vorbereitet und getestet werden. Selbst ein Umbau der Art, dass Sie `Row`, `Sheet` und `Workbook` zu Interfaces machen, würde Ihren Anwendungscode nicht beeinflussen.

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStream;
import java.io.PrintStream

import de.lmp.dummies.exceladapter.CellForPrint;
import de.lmp.dummies.exceladapter.Row;
import de.lmp.dummies.exceladapter.Sheet;
import de.lmp.dummies.exceladapter.Workbook;

public class ExcelToCsvPrinter {

  private PrintStream printStream;

  public ExcelToCsvPrinter(PrintStream printStream) {
    this.printStream= printStream;
  }
  public static void printExcelFileAsCsv(String
excelFileName,
       PrintStream printStream) throws FileNotFoundException,
        IOException {
    ExcelToCsvPrinter excelToCsv = new ExcelToCsvPrinter(
        printStream);
    excelToCsv.printExcelStreamAsCsv(new FileInputStream(
        excelFileName));
    }
```

```
  void printExcelStreamAsCsv(InputStream inputStream)
       throws IOException {
    printWorkbook(Workbook.readWorkbook(inputStream));
  }
  void printWorkbook(Workbook workbook) {
    for (int k = 0; k < workbook.getNumberOfSheets(); k++) {
      printSheet(workbook.getSheetAt(k));
    }
  }
  void printSheet(Sheet sheet) {
    for (int r = 0; r < sheet.getNumberOfRows(); r++) {
      printRow(sheet.getRowAt(r));
    }
  }
  void printRow(Row row) {
    for (int c = 0; c < row.getNumberOfCells(); c++) {
      if (c> 0) {
        printStream.print(',');
      }
      printCell(row.getCellAt(c));
    }
    printStream.println();
  }
  void printCell(CellForPrint cell) {
    printStream.print(cell.asPrintString());
  }
}
```

Listing 14.6: Die überarbeitete Anwendung

Das Wichtigste in Kürze

- ✔ Schnittstellen definieren als Verbindungselemente zwischen den Komponenten das Gerüst jeder Anwendung.
- ✔ Die Güte der Schnittstellendefinitionen ist entscheidend für die Wartbarkeit der Software und ihre Elastizität gegenüber neuen Anforderungen.

- ✔ Schnittstellen sind die technische Formulierung ideeller Abstraktionen und müssen deshalb den mentalen Fähigkeiten des Menschen angemessen sein.
- ✔ Gute Schnittstellen sind schmal und kohärent, das heißt, sie umfassen nur ein einziges Konzept und nur die dazu notwendigen Methoden.
- ✔ Schnittstellen dienen nicht zuletzt der Kommunikation zwischen Entwickler und Leser des Codes. Dabei müssen Missverständnisse durch möglichst genaue Beschreibungen vermieden werden.
- ✔ Die Abgrenzung zu Fremdcode ist eine oft unterschätzte wichtige Schnittstelle. Durch Isolierung in speziellen Adapter-Packages wird die Wartbarkeit des Codes verbessert.

Kapitel 15

Objekte und Datensätze unterscheiden

IN DIESEM KAPITEL

Objekte und Datenstrukturen unterscheiden

Prozeduralen Code sinnvoll einsetzen

Gesetz von Demeter

Java kennt – abgesehen von den sogenannten einfachen Typen (`int`, `long` …) – nur die Typart *Klasse* (`java.lang.Class`), deren Instanzen *Objekt* (`java.lang.Object`) heißen. Das erleichtert zweifellos das Erlernen der Sprache, hat auf die Dauer gesehen aber auch einen ungünstigen Einfluss auf die Lesbarkeit des Codes, weil wichtige Unterschiede verwischt werden. In diesem Kapitel geht es darum, wie Sie durch klare Trennung zweier elementarer Konzepte die Verständlichkeit verbessern können.

Für die folgenden Überlegungen ist es hilfreich, sich erst einmal genauer an die Grundlagen der beim Programmieren benutzten Abstraktionen zu erinnern.

Was ist ein Objekt?

Objekt ist ein reichlich unbestimmter Ausdruck, laut Duden ein »Gegenstand, auf den das Interesse, das Denken, das Handeln gerichtet ist«, was verschiedene Interpretationen zulässt und selbst in der Informatik abweichende Schattierungen kennt. Ich beschränke mich hier auf die Unterscheidung zwischen

- einem Objekt im Sinne der objektorientierten Programmierung, im Folgenden als *Objekt* bezeichnet, und
- einem Objekt im Sinne der Java-Begrifflichkeit, im Folgenden *Java-Objekt* genannt.

Für das erstgenannte Objekt können Sie in der Literatur verschiedene Definitionen finden. Allen gemeinsam ist aber,

- dass ein Objekt einen inneren Zustand kapselt (*Datenkapselung*, *Information Hiding*),
- dass dieser Zustand über den Aufruf von Methoden beeinflusst werden kann und umgekehrt das Resultat von Methodenaufrufen von diesem Zustand abhängen kann und
- dass der Zustand über einen Methodenaufruf hinweg erhalten bleibt.

Kurz gesagt: Objekte haben ein Verhalten und verbergen ihren inneren Zustand hinter einer Schnittstelle. Natürlich gibt es weitere wichtige Eigenschaften wie Vererbung und Polymorphie, die aber für die Betrachtungen in diesem Kapitel weniger interessant sind.

Objekte kapseln damit nicht nur Daten, sondern auch den Code, der ihr Verhalten implementiert. Jedes Objekt ist über seine Schnittstelle Anbieter von Services, die es, gegebenenfalls unter Aufruf von Services anderer Objekte, ausführen kann.

Und ein Datensatz?

Datensätze sind Zusammenfassungen einzelner Daten in einer festgelegten Struktur. Die einzelnen Elemente eines Datensatzes werden *Felder* genannt. Im Allgemeinen kann ein Feld nur Daten eines bestimmten Typs annehmen. In vielen Fällen darf ein Feld selbst wieder einen Datensatz enthalten, was zu hierarchisch strukturierten Datensätzen führt.

Datensätze

Die Zusammenfassung von Daten zu Datensätzen ist schon alt. Nachdem sie jahrhundertelang vor allem als Zeile in Konto- oder Kirchenbüchern erschien, wurde mit der Erfindung der Lochkarte dem Datensatz eine eigene physische Erscheinungsform gegeben. Diese lochkartenbasierte *Datenverarbeitung* ist einer der Ursprünge der heutigen IT. (Die andere Ahnenlinie betrifft mathematische Berechnungen und reicht über A. Turing und K. Zuse bis zu Ch. Babbage zurück.)

Nicht überraschend war die für Aufgaben der geschäftlichen Datenverarbeitung entworfene Sprache COBOL (COmmon Business Oriented Language) 1959 die erste wichtige Programmiersprache, die über ausgebaute Möglichkeiten zur Verarbeitung von Datensätzen verfügte.

In den eher wissenschaftlich orientierten Sprachen dauerte es noch fast zehn Jahre, bis entsprechende Konstrukte standardmäßig aufgenommen wurden. PL/I kopierte diesbezüglich COBOL. Wirklich populär wurden Datensätze jedoch erst mit dem Record-Konzept der Sprache Pascal.

Datensätze haben über die Definition ihres Aufbaus hinaus keine weiteren Eigenschaften. Sie verbergen nichts und hindern im Allgemeinen niemanden daran, ihren Inhalt zu bearbeiten. In dieser Hinsicht unterscheiden sie sich nicht von einfachen Variablen und sind deshalb auch ein typischer Bestandteil prozeduralen Programmierens. Dadurch sind sie in den Augen dogmatischer Entwickler für die objektorientierte Programmierung diskreditiert.

Die Praxis

Java hat zwar viel dazu beigetragen, objektorientiertes Programmieren populär zu machen, aber leider sind dabei ein paar wichtige Grundsätze unter die Räder gekommen. Daraus, dass alle Klassen von `Object` abgeleitet sind, entstand die falsche Überzeugung, dass alle Java-Objekte automatisch auch Objekte sind. Der Unterschied zwischen Datensätzen und Objekten wird dabei einfach übersehen.

Ohne Zweifel benötigen Sie auch in objektorientierten Programmen Datensätze als Behälter für Nachrichten, etwa um einem Objekt eine strukturierte Postadresse zu senden. Das übliche Vorgehen besteht dann darin, eine entsprechende Java-Klasse zu definieren. Da es sich bei den Instanzen dann scheinbar um Objekte handelt, wird die innere Struktur verborgen, das heißt, alle Datenfelder sind privat. Allerdings werden gleichzeitig für jedes der »verborgenen« Felder *Getter* und *Setter* geschrieben, und dadurch wird die innere Struktur vollständig explizit gemacht. Für die genannte Postadresse sieht das dann zum Beispiel so aus:

```
public class MailAddress {
  private String name, street, zip;
  public String getName(){
    return name;
  }
  public setName(String name){
    this.name = name;
  }
  public String getStreet(){
    return street;
  }
  public setStreet(String street){
    this.street = street;
  }
  public String getZip(){
    return zip;
  }
  public setZip(String zip){
    this.zip = zip;
  }
}
```

Das Ergebnis ist ein Zwitter, ein Datensatz in Gestalt eines Objekts, das aber keines ist. Denn dieses Java-Objekt verbirgt weder seine innere Struktur noch hat es ein Verhalten. In der

Java-Terminologie werden solche verhaltenslosen Pseudoobjekte auch als *Bean* bezeichnet.

Um wie viel übersichtlicher ist dagegen:

```
public class MailAddress {
  public String name;
  public String street
  public String zip;
}
```

Sie sparen dadurch Unmengen von Code für die Zugriffsmethoden, der so einfach ist, dass Sie ihn durch Ihre IDE generieren lassen könnten. Es gibt auch Werkzeuge, die Ihnen die Zugriffsmethoden für private Felder verdeckt beim Laden erzeugen, sodass diese im Quellcode überhaupt nicht mehr auftauchen. Solche »Magie« macht den Code allerdings nicht unbedingt besser lesbar.

Der größte Gewinn durch die Verwendung öffentlicher Felder ergibt sich aber daraus, dass die Konzepte explizit erkennbar sind. Jedem Betrachter ist sofort klar, dass es sich bei einem Java-Objekt mit öffentlichen Feldern um einen Datensatz handelt, der nichts verbergen soll.

Übrigens wird die Verwendung von öffentlichen Datenfeldern für die Realisierung von Datensätzen bereits in der Java-Sprachdefinition ausdrücklich vorgeschlagen.

Die klare Unterscheidung verlangt allerdings auch von Ihnen als Entwickler konsequentes Vorgehen. Sie müssen sich bei jeder neuen Klasse fragen: Objekt oder Datensatz? Vermengungen müssen Sie unbedingt vermeiden. Entweder ein Java-Objekt hat öffentliche Datenfelder und dann auch keine Objektmethoden, oder es ist ein echtes Objekt mit Methoden und ohne öffentliche Felder.

Versuchen Sie, Datensätze zu erkennen und konsequent als solche zu realisieren. Die Vorteile werden Sie schnell überzeugen. Lassen Sie sich nicht entmutigen, wenn Ihnen dabei (organisatorische) Hindernisse begegnen.

Die Objekt-Datensatz-Antisymmetrie

Objekte und Datensätze sind nicht nur unterschiedliche Abstraktionen. Sie haben unterschiedliche Eigenschaften und damit verschiedene Anwendungsfelder.

Java und Objektorientierung

Java ist von Anfang an als hybride Sprache gestaltet worden, mit der Sie sowohl *objektorientiert* als auch *prozedural* – und seit einiger Zeit und mit Einschränkungen auch *funktional* – programmieren können. Einige Puristen haben das als inkonsequent kritisiert, aber der Erfolg hat den Sprachschöpfern recht gegeben.

Jedes dieser Programmierparadigmen hat seine Vor- und Nachteile und ist für bestimmte Aufgabenstellungen besser geeignet als für andere. Eine universell einsetzbare Sprache muss dem Rechnung tragen und entsprechende Freiheiten gewähren.

Im Gegensatz zu Werken der Kunst wird Software eher an ihrer Effektivität und Effizienz als an der Reinheit des Stils gemessen. Die Verantwortung für die Klarheit des Codes liegt damit in erster Linie bei Ihnen als Entwickler.

Sie müssen aus dem gut ausgestatteten Werkzeugkasten die richtigen Teile auswählen und fachgerecht kombinieren.

Trotz aller Vielseitigkeit ist es weder Java noch einer der anderen neueren Sprachen gelungen, im kaufmännischen Bereich zu einem adäquaten Ersatz für COBOL zu werden, das bereits seit 1959 im Einsatz ist. Jüngeren Schätzungen zufolge werden immer noch 70 Prozent aller Transaktionsverarbeitungssysteme mit COBOL gebaut. Es gibt zwar Migrationen weg von COBOL, der Grund dafür ist aber häufiger der Mangel an erfahrenen Entwicklern als Defizite der Programmiersprache.

Vergleichbares gilt übrigens auch für das 1957 eingeführte und seitdem mehrfach erweiterte Fortran, das dank effizienter Ausdrucksmittel für mathematische Aufgaben seine Position bei anspruchsvollen Algorithmen, wie beispielsweise zur Simulation und Optimierung, erfolgreich verteidigt.

Prozeduraler Code

Prozedurale Programmierung ist in Verruf gekommen oder gilt zumindest als überholt – ich glaube zu Unrecht. In diesem Abschnitt werde ich versuchen, Ihnen meine Einschätzung nahezubringen.

Eigenschaften

Nach wie vor wird in großem Maße Datenverarbeitung im wörtlichen Sinn getrieben, das heißt Daten rein – Daten raus. Dafür ist prozeduraler Code genau das Richtige, denn bei dieser Verarbeitung ändern sich die Datenstrukturen nur selten. Was hingegen viel volatiler ist, sind die Verarbeitungsprozesse.

Denken Sie beispielsweise an Buchungssätze in einer Bank. Deren Struktur hat sich oft über Jahrzehnte nicht verändert, zumindest bis zur verpflichtenden Einführung der IBAN vor einigen Jahren. So eine (seltene) Umstellung verursacht dann auch erhebliche Aufwände.

Die Verarbeitung wird hingegen durch regulatorische Vorgaben wie Embargolisten und Geldwäscheprüfungen ständig komplexer. Fast jedes Jahr kommen neue Gesetze hinzu, die kurzfristig umgesetzt werden müssen.

Außerdem entwerfen die Geschäftsbereiche laufend neue Produkte, die spezielle Anforderungen stellen. Betroffen davon sind aber auch in der Regel nur die Verarbeitungsprozeduren, nicht die Datenstrukturen.

Der Vorteil prozeduralen Codes besteht darin, dass neue Prozeduren relativ leicht implementiert werden können, weil sie unabhängig von bereits vorhandenen sind. Problematisch und aufwendig ist die Veränderung von Datenstrukturen, weil alle betroffenen Prozeduren angepasst werden müssen.

Prozeduraler Code ist überdies immer dann angebracht, wenn Sie es mit offenen Daten ohne Verhalten zu tun haben. In Java gilt das ohne Zweifel für die einfachen Typen.

Der Typ `String` hingegen hat kein Verhalten, obwohl er über zahlreiche Methoden verfügt. Diese wären alle als Prozeduren realisierbar, weil das zugehörige Objekt nie verändert wird. Für die Zeichenkette selbst ist ein Objekt als Datenkapsel jedoch angebracht, weil dadurch Implementierungsdetails verborgen werden können, was bereits für verschiedene Optimierungsansätze genutzt wurde.

Vorteile

Die Vorteile prozeduralen Codes stellen sich nur dann ein, wenn dieses Paradigma der Aufgabenstellung angemessen ist. In den allermeisten Fällen werden Sie weder ausschließlich in dem einen noch in dem anderen Stil entwickeln wollen und können. Aber wenn die reine Verarbeitung von Daten im Vordergrund steht, können sich durch die Verwendung von Datenstrukturen einige Pluspunkte für Sie ergeben:

- ✔ Deutlich weniger Code bei Verwendung von Datensätzen statt Bean-Pseudoobjekten.

- ✔ Gute Isolierung einzelner Verarbeitungsschritte in unabhängigen Prozeduren.
- ✔ Einfaches Einfügen oder Weglassen von Verarbeitungsschritten.
- ✔ Datensätze sind darüber hinaus geeignet, als Transportmittel von Informationen zwischen den Objekten – in der objektorientierten Terminologie *Botschaften* oder *Nachrichten* genannt – zu dienen.

Probleme

Beim Versuch, mit Java prozedural zu programmieren, werden Sie jedoch auf ein paar Probleme stoßen, die sich aus dem hybriden Charakter der Sprache ergeben:

- ✔ Wie bereits erwähnt, dürfen Datensätze keine Methoden haben. Da alle Klassen von `Object` erben, können Sie das jedoch nicht vollständig realisieren. In den meisten Fällen lässt sich dieser Fakt ignorieren, indem Sie die ererbten Methoden einfach nicht verwenden.
- ✔ Sie können den Inhalt kompletter Datensätze nicht durch eine einfache Zuweisung kopieren, sondern müssen spezielle Kopierprozeduren schreiben.

  Das erfordert zusätzlichen Code und wirkt sich negativ auf die Laufzeiteffizienz aus (verglichen mit prozeduralen Sprachen, bei denen das häufig durch den Transfer eines einzigen Datenblocks realisiert wird).
- ✔ In sich strukturierte (hierarchische) Datensätze, dazu reicht bereits ein enthaltenes Array, benötigen zur Erzeugung mehrere Konstruktor-Aufrufe.

  Aus Sicht der Speicherorganisation sind dadurch selbst scheinbar einfach aufgebaute Datensätze bereits verknüpfte Java-Objekt-Strukturen, die einen entsprechenden internen Verwaltungsaufwand erzeugen.
- ✔ Bei der Navigation in hierarchischen Datensätzen müssen Sie in aller Regel mit Nullpointern rechnen.

Explizite Tests oder die Verwendung von *Optionals* machen den Code dann unübersichtlicher. (C# bietet mit dem sogenannten »Elvis-Operator« `?.` dafür eine elegante Lösung.)

Bedenken Sie aber bitte, dass alle diese Nachteile nicht verschwinden, wenn Sie stattdessen in der üblichen Art und Weise Bean-Pseudoobjekte benutzen. Dadurch werden die Defizite nur verschleiert, weil der Vergleich mit prozeduralen Sprachen nicht so offen auf der Hand liegt.

Objektorientierter Code

Vieles was in der äußeren Gestalt objektorientierten Codes daherkommt, ist bei genauer Betrachtung nur verkleideter prozeduraler Code. Deshalb halte ich es für angebracht, die Grundgedanken der Objektorientierung noch einmal zu hervorzuheben.

Eigenschaften

Die Entwicklung der objektorientierten Programmierung war eng verbunden mit der Verbreitung grafischer Benutzeroberflächen. Das ist kein Zufall. Denn genau für derartige Aufgabenstellungen eignet sich die Objektorientierung besonders gut.

Die in den Oberflächen darzustellenden grafischen Elemente sind höchst unterschiedlich und benötigen daher sowohl verschiedene Daten als auch verschiedene Methoden für die Darstellung. Für diese Kombination sind Objekte genau die richtige Lösung: Sie kapseln Daten und Verhalten gemeinsam. Prozeduraler Code könnte diese Aufgabe nur schwer in vergleichbarer Allgemeinheit lösen.

Eine Bedingung muss durch die Objekte dabei allerdings erfüllt sein. Alle Objekte müssen eine vereinbarte Schnittstelle für die Darstellung bereitstellen. Das kann auf zweierlei Weise geschehen:

- ✔ Durch Vererbung von einer gemeinsamen Basisklasse, das ist die eher klassische Methode.
- ✔ Durch Implementation eines entsprechenden Interfaces, das ist flexibler und wird oft der Vererbung vorgezogen.

Eine Änderung der Daten für ein solches Objekt, etwa eine weitere Farbe, ist ohne großen Aufwand möglich. Das Gleiche gilt, wenn Methoden einzelner Objekte modifiziert werden sollen.

Schwierig wird es hingegen, wenn Sie eine völlig neue Methode einfügen möchten. Denn das bedeutet eine Änderung der Schnittstelle und erfordert zumeist entsprechende Ergänzungen in allen Klassen.

Objektorientierter Code erleichtert das Einfügen neuer Objekte und macht es schwer, neue Methoden einzuführen.

Vorteile

Objektorientierung hat sich in der Softwareentwicklung weitgehend durchgesetzt. Dieser Erfolg ist den guten Eigenschaften zu verdanken, von denen ich nur einige aufzählen möchte. Allerdings gilt auch hier, dass die Vorteile nur bei passenden Problemstellungen vollständig zum Tragen kommen:

- ✔ Objekte unterstützen die kleinteilige Modularisierung des Codes. Das erleichtert die arbeitsteilige Entwicklung und vereinfacht das Testen.
- ✔ Objekte fassen Daten und Code zusammen. Das entspricht eher dem menschlichen Strukturierungs- und Klassifizierungsvorgehen und ist intellektuell gut zu bewältigen.
- ✔ Es gibt eine klare Unterscheidung, was öffentlich und was verborgen ist. Durch das Verbergen von Implementationsdetails sind Anpassungen und Verbesserungen, beispielsweise der Effizienz, weitgehend lokal möglich.

- ✔ Die Wiederverwendung von Code wird erleichtert, wenn auch nicht in dem Umfang, den man sich ursprünglich einmal erhofft hat.

Probleme

Bei allen Vorzügen hat Objektorientierung auch Schattenseiten, die Sie kennen sollten:

- ✔ Komplexe Verarbeitungen können durch die Verteilung auf viele Objekte schwierig nachvollziehbar sein. Das ist häufig ein Problem in tiefen Vererbungsstrukturen, weshalb heute statt Vererbung oft Delegation verwendet wird.
- ✔ Das Erzeugen von Objekten ist aufwendig, vor allem wenn diese viele interne Daten benötigen.
- ✔ Die heute übliche automatische Speicherverwaltung trägt zwar viel zur sicheren Anwendung bei, hat aber ihren Preis und bringt bezüglich Antwortzeiten und Speicherbedarf ein gewisses Maß an Zufälligkeit in die Programmausführung.
- ✔ Um einen akzeptablen Grad an Effizienz zu erreichen, müssen prozedurale Elemente wie zum Beispiel einfache Typen eingesetzt werden. Das kann leicht dazu führen, dass der Code unsauber wird und letztlich die Nachteile beider Paradigmen vereint.

Nachdem ich am Anfang dieses Kapitels Ihnen mit der Adresse bereits ein typisches Beispiel für eine Datenstruktur gegeben habe, folgt jetzt ein Adressprüfer als Beispiel dafür, was möglicherweise als Objekt realisiert werden kann.

Dazu wird eine Schnittstelle festgelegt. Hier genügt eine Methode:

```
interface AddressChecker {
  public boolean isValidAddress(MailAddress);
}
```

Sie können dieses Interface jetzt durch verschiedene Klassen implementieren, wobei die Umsetzung nach außen vollkommen verborgen bleibt. Der Anwender der Methode `isValidAddress` soll und muss nichts darüber wissen.

Das ermöglicht es Ihnen, innerhalb einer Anwendung verschiedene Prüfer zu verwenden und sogar dynamisch auszutauschen.

Wenn allerdings die Adressprüfung aufgrund fester externer Vorgaben erfolgen muss, ist eine prozedurale (statische) Methode die bessere Wahl.

Schlussfolgerungen

Auch wenn Sie eine Anwendung objektorientiert entwickeln, werden Sie immer einen Anteil von Aufgaben haben, für die prozeduraler Code angemessen ist. Sie werden schwerlich ein Java-Programm finden, das nicht wenigstens eine Handvoll Beans, also letztlich Datensätze, enthält.

Sehr oft ist der Anteil, und damit die durch Umstellung auf öffentliche Felder erreichbare Codeeinsparung beachtlich. Diese Schlankheitskur macht den Code besser lesbar und vereinfacht die Wartung.

In vielen Projekten werden heute Werkzeuge eingesetzt, die eine vorgegebene Testabdeckung des Codes einfordern. Unter Umständen zwingt Sie das, für die trivialen Beans-Methoden auch noch Testfälle zu schreiben. Auch dafür gibt es Hilfsmittel, aber es ist zweifellos besser, diesen unnötigen Aufwand gleich ganz einzusparen.

Machen Sie es sich nicht zu einfach, indem Sie bedenkenlos dem »Java-Objekt = Objekt«-Schema folgen, sondern beherzigen Sie die folgenden Regeln:

- ✔ Prozedurale Programmierung ist nicht per se schlecht. Es gibt an vielen Stellen Algorithmen, die aus übergebenen Daten

neue Daten ermitteln, ohne dass dabei irgendein innerer Zustand eine Rolle spielt. Künstlich eingeführte Objekte verschlechtern dann nur die Verständlichkeit.

- ✔ Packen Sie prozeduralen Code in statische Methoden. Das ermöglicht Ihnen zusätzlich eine einfache Kontrolle, ob Ihre Entscheidung richtig war: Wenn der Test erfordert, dass Sie statische Methoden durch einen Mock ersetzen müssen, dann sollten Sie besser Objekte benutzen.
- ✔ Machen Sie sich stets bewusst, dass Datensätze und Objekte verschiedenen Anforderungsprofilen gerecht werden. Finden Sie heraus, welches Profil vorliegt, und wählen Sie danach die passende Programmierweise aus.

Das Gesetz von Demeter

In der täglichen Arbeit hat das unbewusste Vermischen von Objekten und Datensätzen immer wieder zu Problemen geführt, wenn die für Bean-Pseudoobjekte angebrachte Offenlegung der internen Daten leichtfertig auf echte Objekte übertragen wurde. Die daraus resultierenden Nachteile waren Anlass für die Formulierung einer Regel.

Internes intern halten

Es ist allgemein anerkannt, dass das Geheimnisprinzip, das heißt das Verbergen von Implementierungsdetails, einer der Grundpfeiler der objektorientierten Programmierung ist. Trotzdem wird gegen dieses Prinzip so oft verstoßen, dass es angebracht erschien, die unter dem Namen *Gesetz von Demeter* bekannte Regel zu formulieren.

Die Einhaltung dieser Regel fördert die lose Kopplung der Komponenten untereinander. Im Kern geht es darum, dass ein Kunde oder Benutzer eines Objekts keine inneren Einzelheiten dieses Objekts kennen soll.

Gesetz von Demeter: Eine Methode *m* einer Klasse *K* soll ausschließlich auf folgende Programmelemente zugreifen:

- ✔ Methoden der eigenen Klasse,
- ✔ Methoden der Parameter von *m*,
- ✔ Methoden von Objekten, die als Instanzvariablen in *K* gehalten werden, und
- ✔ Methoden von in *m* erzeugten Objekten.

Praktisch heißt das, es dürfen von außen keine Methoden von Objekten aufgerufen werden, die zum inneren Zustand gehören.

Bisweilen können Sie auch lesen, dass keine Methoden solcher Objekte aufgerufen werden sollen, die Sie als Resultat eines regelkonformen Aufrufs erhalten. Das ist natürlich eine unsinnige Vereinfachung, weil es unter anderem jede Factory-Methode als unzulässig erklären würde.

Trotzdem kommunikativ sein

Da der Unterschied zwischen den nach dieser Regel zulässigen und unzulässigen Methoden etwas diffizil ist, will ich ihn an einem Beispiel erläutern. Listing 15.1 zeigt einen Ausschnitt aus der Definition einer Klasse `Person`.

```
public class Person {
  private List<Merkmal> merkmalListe;
  …
}
```

Listing 15.1: Klasse mit internem Objekt

Nun sollen Sie die Daten der internen Liste nach außen geben. Die naheliegendste und sehr häufig zu findende, aber schlechte Lösung zeigt Listing 15.2.

```
public class Person {
  private List<Merkmal> merkmalListe;

  public List<Merkmal> getMerkmalListe() {
    return merkmalListe;
  }
  …
}
```

Listing 15.2: Wie es nicht sein sollte

Warum ist das verkehrt? Weil Sie dadurch die Kontrolle über ein objektinternes Objekt verlieren. Ohne Ihr Wissen können Elemente in die Liste eingefügt oder aus dieser entfernt werden, und das – was die Sache noch schlimmer macht – zu völlig unkontrollierbaren Zeitpunkten.

Das Gesetz von Demeter

Tatsächlich handelt es sich dabei nicht um ein Gesetz im eigentlichen Sinn. Richtig wäre die Bezeichnung Regel oder Prinzip. Das Gesetz von Demeter ist eine präzise formulierte Heuristik, die von Codeanalyse-Werkzeugen überprüft werden kann.

Ein Zusammenhang zwischen Fehlerquoten und Verstößen konnte empirisch gezeigt werden. Als nachteilig erweist es sich bisweilen, dass die Regelbefolgung zu einer größeren Zahl von Methoden führen kann und der innere Zusammenhang des Codes durch einfache »Weiterleitungsmethoden« geschwächt wird.

Der Name steht übrigens nur mittelbar in Beziehung zur Göttin des Getreides, der Saat und der Jahreszeiten. Die entsprechenden Untersuchungen zur objektorientierten Methodik waren Teil der Implementierung einer Hardware-Beschreibungssprache »Zeus«, sodass der Name einer Schwester des Zeus für das Implementierungswerkzeug einfach nahe lag.

Verfallen Sie nun aber nicht in das andere Extrem und enthalten Sie den Nutzern die von diesen benötigten Informationen vor, nur um nicht gegen das Gesetz von Demeter zu verstoßen. Niemand verbietet Ihnen, Informationen nach außen zu geben, nur die internen Objekte selbst sind davon ausgeschlossen.

```
public class Person {
  private List<Merkmal> merkmalListe = new ArrayList<>();

  @SuppressWarnings("unchecked")
  public List<Merkmal> getMerkmalListe1() {
    return Collections.unmodifiableList(
        (List<Merkmal>)((ArrayList<Merkmal>) merkmalListe)
         .clone());
  }

  public List<Merkmal> getMerkmalListe11() {
    return Collections.unmodifiableList(
        new LinkedList<Merkmal>(merkmalListe));
  }

  @SuppressWarnings("unchecked")
  public List<Merkmal> getMerkmalListe2() {
    return (List<Merkmal>)((ArrayList<Merkmal>) merkmalListe)
         .clone();
  }
}
```

Listing 15.3: Zwei gute und eine nicht so gute Lösung

Listing 15.3 zeigt drei Möglichkeiten, wie Sie regelkonform die gewünschten Informationen liefern können. Gemeinsam ist allen Varianten, dass das interne Listenobjekt **nicht** nach außen gegeben wird. Ein Aufrufer erhält jeweils eine Kopie. Unterschiede ergeben sich durch die Art der Kopie:

- ✔ `getMerkmalListe1`: Die interne Liste wird geklont und als nicht modifizierbare Liste zurückgegeben. Das ist die sauberste Lösung, weil dadurch niemand dem Irrtum erliegen kann, auf diese Weise die innere Liste manipulieren zu können. Ungünstig ist nur, dass derartige Fehler erst zur Laufzeit bemerkt werden können.
- ✔ `getMerkmalListe11`: Diese Variante illustriert zusätzlich die Möglichkeit, ein vom Typ des inneren Objekts abweichendes

Objekt zurückgeben zu können.

- `getMerkmalListe2`: Die interne Liste wird ebenfalls geklont und dieser Klone als veränderbare Liste zurückgegeben. Der Nachteil oder die Gefahr besteht darin, dass beim Anwender möglicherweise die Illusion entsteht, dass es sich um die interne Liste handeln würde.

Bei allen drei Varianten wird überdies vorausgesetzt, dass die Java-Objekte vom Typ `Merkmal` unveränderlich sind.

Das gilt auch umgekehrt

Auch wenn Sie das vielleicht nicht gleich erkannt haben sollten, das Gesetz von Demeter gilt auch für Methoden, die Objekte als Parameter entgegennehmen. Wenn Sie im obigen Beispiel eine Methode ergänzen wollen, die dem `Person`-Objekt eine Liste von Merkmalen hinzufügt, dürfen Sie diese nicht einfach in Ihr Objekt übernehmen.

Sonst ist der Effekt der gleiche wie beim Publizieren durch eine `get`-Methode. Sie hätten ein internes `List`-Objekt, das auch von außerhalb zugreifbar ist, ohne dass Sie das kontrollieren können.

Die Regel lautet in diesem Fall: Als Parameter an eine Methode übergebene Objekte dürfen nicht in den Zustand des Empfängerobjekts übernommen werden. Listing 15.4 zeigt ein Beispiel dafür, wie Sie es richtig machen.

```
public void setMerkmalListe(List<Merkmal> liste) {
  merkmalListe = new ArrayList<>(liste);
}
```

Listing 15.4: Zulässige Methode zum Setzen eines Objekts

Nicht ganz unerwähnt soll an dieser Stelle bleiben, dass die inneren Daten von Objekten natürlich strukturiert sein können, also ihrerseits weitere Objekte enthalten. Unter Umständen kann es deshalb notwendig sein, dass die in den Beispielen verwendeten »flachen« Kopien nicht ausreichen und durch

»tiefe«, das heißt den ganzen Objektbaum erfassende Kopien ersetzt werden müssen.

Aufrufketten

Als typisches Indiz für die Verletzungen des Gesetzes von Demeter gelten Aufrufketten der Art

```
getKonto().getPerson().getMerkmalListe().add(merkmal);
```

Verwechseln Sie diese jedoch nicht mit den bei der Umsetzung von *Fluent Interfaces* vorkommenden Aufrufketten wie beispielsweise in

```
EasyMock.andThrow(new NullPointerException()).atLeastOnce();
```

Letztere Ketten sind Ausdruck eines Stils, der explizite Variablen überflüssig macht und dadurch den Schreibaufwand reduziert. Sie geben dabei keine Interna nach außen.

Fluent Interface

Unter diesem Namen wird ein Konzept verstanden, bei dem bestimmte Folgen von Methodenaufrufen quasi erzwungen werden können. Der Code soll dadurch einfacher lesbar und Fehler in der Aufrufreihenfolge sollen vermieden werden. Beispiel:

```
Book b =
Book.author("Lampe").title("Dummies").year(2020).build();
```

Wenn die Methoden `author`, `title` und `year` dabei jedes Mal ein spezielles *Builder*-Objekt zurückliefern, das nur die jeweils nächste Methode anbietet, kann man erreichen, dass nur vollständig initialisierte Objekte erzeugt werden.

Das Ganze ist zweifellos verständlicher als eine lange Liste von Parametern im Konstruktor. Nachteilig ist der relativ große Aufwand, sowohl im Code als auch zur Laufzeit, den die notwendigen Builder-Objekte verursachen.

Tatsächlich ein Datensatz

Hier geht es um Ketten der ersteren Art. Bei genauer Betrachtung werden Sie feststellen, dass es sich in vielen Fällen dabei gar nicht um echte Objekte, sondern um strukturierte Datensätze

handelt und die Kette einfach dem Zugriffspfad auf ein untergeordnetes Element entspricht.

Da Datensätze offen zugänglich sind, ist ein derartiger Aufruf keine Verletzung des Gesetzes von Demeter. Falls Sie dem Vorschlag aus dem vorigen Abschnitt folgend keine Bean-Pseudoobjekte verwenden, wird das auch sofort erkennbar:

```
einKonto.person.merkmalListe.add(merkmal);
```

Statt der aufgeführten Aufrufketten sieht man im Code oft auch Konstruktionen der Art:

```
Person p = konto.getPerson();
List<Merkmal> merkmalListe = p.getMerkmalListe();
merkmalListe.add(merkmal);
```

So etwas erleichtert zwar das Debuggen und täuscht eventuell ein Prüfwerkzeug, in der Sache ändert sich dadurch aber nichts. Wenn `Konto` und `Person` echte Objekte sind, bleibt es ein Verstoß gegen das Gesetz.

Delegieren!

Trotzdem werden Ihnen vielleicht einige Aufrufketten bleiben, die sich nicht auf Datensätze reduzieren lassen. Die zwei häufigsten Ursachen sind diese:

- ✔ Hybride Strukturen, das heißt, Objekte und Datensätze wurden vermengt. Durch Isolierung der beiden Aspekte in jeweils eigene Java-Klassen lässt sich diese Situation bereinigen.
- ✔ Unsauber gewählte Abstraktionsebenen. Dieses Problem ist etwas schwieriger zu beseitigen und wird im Folgenden genauer betrachtet.

Aufrufketten entstehen häufig dadurch, dass ein Kunde des Objekts eine Aktion ausführen will, zu der Informationen aus einem tieferen Abstraktionsniveau benötigt werden.

Ein Beispiel dafür ist die Frage nach dem Verzeichnisnamen der Persistenzschicht eines Objekts, um dort eine temporäre Datei anlegen zu können. Um derartige Aktionen sollte sich jedoch besser das angesprochene (oder ein anderes) Objekt kümmern. Auf der Ausgangsebene ist doch nur wesentlich, dass man Zugriff auf die gewünschte Datei erhält.

Es ist extrem wichtig, immer wieder nach der wirklich zu lösenden Aufgabe zu fragen, ohne sich zu zeitig darum zu kümmern, wie diese gelöst werden kann.

Das ist eine frappierende Parallele zu einem häufig zu beobachtendem Verhalten in der Geschäftswelt. Nicht wenige Vorgesetzte scheitern daran, dass sie zu wenig Vertrauen in ihre Mitarbeiter haben und sich unnötigerweise in Details der Ausführung verlieren.

Da sie das schon rein mengenmäßig überfordert, leidet die Arbeitsqualität. Also, vermeiden Sie diesen Fehler, haben Sie Vertrauen in Ihre Objekte und delegieren Sie so viel wie möglich!

Fazit

Es gibt Datensätze und es gibt Objekte. Beide werden in Java als Klassen definiert, was diese Unterscheidung zulasten der Codestruktur verwischt.

Objekte zeigen ein Verhalten und verbergen ihre Daten. Dadurch ist es leicht, neue Objektarten einzuführen, ohne dass bereits vorhandenes Verhalten geändert wird. Allerdings ist es schwierig und aufwendig, existierende Objekte um neues Verhalten zu erweitern.

Bei Datensätzen ist es genau anders herum, sie haben kein relevantes Verhalten und legen ihre Daten offen. In der Folge ist es einfach, neue Verarbeitungen zu implementieren, und aufwendig, die Datenstruktur zu ändern.

In jedem nicht trivialen Programm werden Sie auf Situationen stoßen, in denen eher die eine oder die andere Flexibilität benötigt wird. Es ist Ihre Verantwortung als Entwickler, stets die angemessene Strategie auszuwählen und konsequent umzusetzen.

Das Wichtigste in Kürze

- ✔ Objekte und Datenstrukturen sind unterschiedliche Abstraktionen mit jeweils eigenen Anwendungsfeldern.
- ✔ Durch das Klassenkonzept von Java wird die Trennung verwischt. Verwenden Sie für Datenstrukturen öffentliche Felder statt Bean-Pseudoobjekte, um den unterschiedlichen Charakter deutlich zu machen.
- ✔ Objekte zeigen Verhalten und verbergen ihre Daten, das erleichtert das Hinzufügen neuer Objektarten mit neuem Verhalten. Es ist jedoch schwierig, existierendes Verhalten zu ändern.
- ✔ Datensätze haben praktisch kein Verhalten und zeigen ihre Daten. Das macht es einfach, neue Verarbeitungen zu realisieren, aber aufwendig, die Datenstruktur zu verändern.
- ✔ Eine Methode darf keine Methoden aufrufen, die zu inneren Objekten eines anderen Objekts gehören (Gesetz von Demeter).

Kapitel 16

Wege im Dschungel – Regeln

IN DIESEM KAPITEL

Wiederholungen im Code vermeiden

Nur liefern, was verlangt wird

Jede Aufgabe für sich erledigen

Die SOLID-Regeln

Verschiedene Probleme tauchen immer wieder auf. Dann ist es hilfreich, wenn man auf verallgemeinerte Erfahrungen zurückgreifen kann. In diesem Kapitel lernen Sie einige bewährte Regeln für das Schreiben sauberen Codes kennen.

Denken Sie stets daran: Regeln sind Angebote – keine Vorschriften. Es bleibt Ihrem Geschick überlassen, unter den jeweils anwendbaren die am besten passenden Regeln auszuwählen. Das erfordert Übung und Erfahrung und wird Ihnen am Anfang nicht gleich perfekt gelingen.

Falls Sie es bisher noch nicht gemacht haben, sollten Sie sich vielleicht in diesem Zusammenhang das Kapitel 3 *Alles muss unter einen Hut* genauer ansehen.

Wiederholungen vermeiden

Beim Lernen ist Wiederholung der Schlüssel zum Erfolg – beim Programmieren sollten Sie Wiederholungen aber besser vermeiden. Der Code wird dadurch nur ohne echte Notwendigkeit umfangreicher. Sie müssen mehr schreiben, und der Leser mehr

lesen. Das nützt keinem, zumal damit weitere Nachteile verbunden sind.

Die Regel

Die diesbezügliche Regel ist vor allem unter dem englischen Namen *Don't Repeat Yourself* beziehungsweise der daraus gebildeten Abkürzung *DRY* bekannt.

Wiederholungen vermeiden: Jedes Stück Kenntnis oder Information muss an genau einer Stelle im Code klar verständlich und verbindlich formuliert sein.

Das Vermeiden von Dopplungen ist ganz allgemein eine weit über Code hinaus nützlich Methode zur Komplexitätsverminderung.

Die Technik, Wiederholungen zu vermeiden, ist ein uraltes Grundprinzip der Wissenschaften, insbesondere der Mathematik. Jede Definition hat diesen Zweck. In der Programmierung fand DRY seinen ersten Ausdruck in Unterprogrammen. Später war das Vermeiden von doppelter Datenhaltung ein wesentlicher Aspekt bei der Entwicklung der relationalen Datenbanken.

Eng verwandt oder weitgehend bedeutungsgleich sind die Bezeichnungen *Single Point of Truth* oder *SPOT*, die vorrangig mit Bezug auf Daten verwendet werden, und *Once and Only Once*.

Motivation

Grundsätzlich gibt es zwei Arten von Codedopplungen:

- Formal identischer Code: Das sind leicht erkennbare Dopplungen, auf die Sie auch durch entsprechende Analysewerkzeuge aufmerksam gemacht werden.
- Inhaltlich weitgehend identischer, aber äußerlich unterschiedlicher Code: Derartige Dopplungen entstehen im

einfachsten Fall durch Umbenennungen. Wichtiger sind jedoch die Fälle, bei denen gleiche Aufgaben auf unterschiedliche Weise erledigt werden, wie zum Beispiel das Sortieren einer Liste durch verschiedene Verfahren.

Dopplungen dieser Art sind meist schwierig zu erkennen.

Problematisch sind die logischen Dopplungen, die in Bezug auf die Anwendung gleiche Funktionen realisieren. Sie können auf beide Arten im Code in Erscheinung treten, weil formal identischer Code stets auch logisch identisch sein *kann*.

Wahrscheinlich ist es Ihnen auch schon passiert, dass eine Änderung schiefgelaufen ist, weil nicht alle Stellen, an denen das notwendig gewesen wäre, modifiziert wurden, oder irrtümlich eine Stelle verändert wurde, an der das nicht erfolgen durfte. So etwas hat häufig – nicht immer – mit Codedopplungen zu tun.

Die wichtigsten nachteiligen Folgen überflüssiger Wiederholungen sind:

- ✔ Erhöhter Aufwand beim Schreiben und vor allem beim Lesen durch den größeren Umfang und die unterlassene Abstraktion.
- ✔ Bei Änderungen müssen Sie als Leser erkennen, dass es sich um »echte« Doppelungen handelt. Nicht alles, was gleich aussieht, ist auch logisch gleich – dafür aber manches, was äußerlich gar nicht gleich erscheint.
- ✔ Es entsteht ein hohes Fehlerpotenzial. Logische Dopplungen können wegen der unvermeidlichen Differenzen dazu führen, dass sich Teile der Anwendung in Randbereichen unterschiedlich verhalten, beispielsweise wenn Sie an einer Stelle stabile und an anderer nicht stabile Sortierverfahren einsetzen. Außerdem müssen Sie bei Änderungen alle betroffenen Dopplungen finden und bearbeiten, das heißt, die gedoppelten Codestellen müssen synchron gehalten werden.

Ein erfreulicher Nebeneffekt des Vermeidens von Wiederholungen ist, dass Sie auf diesem Wege Tests einsparen, denn Code, der nur an einer Stelle steht, muss auch nur einmal getestet werden.

Umsetzung

So einleuchtend diese Regel in der Theorie ist, so schwierig ist ihre Umsetzung in der Praxis. Unglücklicherweise ist es durch die Funktionen zum Kopieren und Einsetzen ja außerordentlich leicht, Code zu duplizieren und danach eventuell anzupassen. Unnötige Wiederholungen zu vermeiden, erfordert hingegen ein systematisches Vorgehen, dessen wichtigste Punkte ich im Folgenden erläutere.

Entwurf

Das Vermeiden von Dopplungen muss bereits in der Konzeptionsphase beginnen. Dopplungen sind nicht vorgenommene Abstraktionen, also letztlich Entwurfsmängel.

Aber nehmen Sie diese Feststellung nicht gar so schwer. Niemand ist in der Lage, sofort einen völlig korrekten Entwurf zu liefern. Es ist ganz natürlich, dass Sie im Laufe der Entwicklung zu tieferen Einsichten kommen. Nur sollten Sie diese immer in den Kontext des Konzepts oder der Theorie stellen, um so ein besseres Gesamtverständnis zu gewinnen.

Wenn Sie die Aufgabe wirklich umfassend verstanden haben, können Sie den Code auch so gestalten, dass ein zukünftiger Leser ihn gut erfassen kann.

Unterscheidung

Code kann auf unterschiedlichen logischen Ebenen interpretiert werden. Sie haben in jedem Fall zwei Ebenen:

- ✔ die logisch technische Ebene, beispielsweise ein spezielles Sortierverfahren
- ✔ die anwendungslogische Ebene, beispielsweise das Sortieren von Aufträgen

Dabei nutzt die anwendungslogische Ebene Funktionen der technischen Schicht. Beide Ebenen können in sich weiter untergliedert sein. Wenn Sie versuchen, Dopplungen zu

vermeiden, ist es außerordentlich wichtig, dass Sie die entsprechenden Abstraktionsniveaus beachten.

Echte logische Dopplungen können nur auf dem gleichen Niveau auftreten. Bei unsauberer Programmierung kann es allerdings vorkommen, dass Sie auf Vermischungen der Ebenen treffen. Dann sollten Sie zuerst dieses Problem beheben. Andernfalls könnte ein Umbau alles nur verschlimmern.

Mit diesem Beispiel möchte ich die Bedeutung der unterschiedlichen Abstraktionsebenen noch einmal veranschaulichen.

Betrachten Sie die folgenden beiden Funktionen zur Berechnung eines durch einen Prozentwert bestimmten Teilbetrags:

```
public double steuer(double betrag, double steuersatz) {
 return betrag*steuersatz/100d;
}
public double honorar(double betrag, double provision) {
  return betrag*provision/100d;
}
```

Auf den ersten Blick liegt es nahe, stattdessen folgende Funktion zu verwenden:

```
public double anteil(double betrag, double prozentsatz)
{
  return betrag*prozentsatz/100d;}
```

Das sollten Sie allerdings besser nicht tun, weil Sie damit für die anwendungslogischen Funktionen `steuer` und `honorar` eine Funktion der technischen Ebene benutzen würden.

Die Folgen spüren Sie in dem Moment, in dem sich beispielsweise die Honorarberechnung durch Einführung eines Mindestbetrags ändert. Sie müssten dann alle Aufrufe von `anteil` dahingehend prüfen, ob sie der Honorarberechnung dienen.

Korrekt wäre es deshalb, wenn Sie in den Anwendungsfunktionen die technische Funktion benutzen:

```
public double steuer(double betrag, double steuersatz) {
  return anteil(betrag, steuersatz);
}
public double honorar(double betrag, double provision) {
  return Math.max(MINHONORAR, anteil(betrag,
provision));
}
```

Wie Sie sehen, ist die angenommene Änderung dann wirklich nur noch an einer einzigen Stelle vorzunehmen.

Realisierung

Java stellt mit den Generics ein effektives Mittel bereit, das Ihnen dabei hilft, Wiederholungen zu vermeiden. Daneben bieten sich altbewährte Verfahren wie die Nutzung von vorhandenen und selbst entwickelten Funktionsbibliotheken an.

Schwierigkeiten

Im Zuge der Beschreibung sind einige der sich bei Anwendung der Regel ergebenden Schwierigkeiten bereits angeklungen. Die folgende Zusammenstellung soll Sie nicht davon abhalten, Wiederholungen zu minimieren, ganz im Gegenteil.

Ich möchte Ihnen nur zeigen, dass es sinnvoll ist, auch Teilerfolge zu akzeptieren und nicht zu verzagen, wenn Sie die angestrebte Ideallösung nicht finden können – wahrscheinlich gibt es die nämlich gar nicht.

- ✔ Es ist oft nicht einfach zu erkennen, ob eine zu beseitigende Dopplung vorliegt oder nicht. Die Abgrenzung logischer Funktionen ist immer auch subjektiv interpretierbar.
- ✔ Herauszulösende Artefakte müssen ein bestimmtes Mindestmaß an Funktion beziehungsweise Umfang haben. Dafür gibt es keine festen Vorgaben. Was vernünftig ist, stellt sich oft erst im Laufe des Projekts heraus.

- Die entstehende Menge von verfügbaren Teilen muss durch die Entwickler intellektuell beherrscht werden.

  Bereits vorhandene Funktionen müssen bekannt oder mit geringem Aufwand zu finden sein. Das ist eine mit zunehmendem Projektumfang schwerer zu lösende Herausforderung.

Fragen der Effizienz bei der Codeausführung spielen heute kaum noch eine Rolle. Durch den Compiler oder die virtuelle Maschine können eventuell entstehende Nachteile fast immer beseitigt werden.

»Vorbeugend« sollte deshalb nie von der DRY-Regel abgewichen werden. Wie immer sind Bemühungen um Optimierung erst dann angebracht, wenn Sie einen Schwachpunkt nachgewiesen haben und die Verbesserung durch Messungen bestätigen können.

- Versuchen Sie, faul zu sein! Prüfen Sie vorher, ob es das, was gebraucht wird, nicht schon irgendwo gibt.

  Im Zweifel ist selbst ein begrenzter Umbau, der vorhandenen Code für eine breitere Anwendung zugänglich macht, für das Gesamtprojekt nützlicher als eine Doppelung.
- Kopieren Sie nicht leichtfertig! Prüfen Sie immer, ob Herausziehen und Wiederverwenden nicht die auf Dauer bessere Lösung ist.
- Nutzen Sie vorhandene Bibliotheken. Prüfen Sie, ob eine Funktionalität nicht besser in einer eigenen Bibliothek aufgehoben wäre, sodass sie leichter wiederverwendbar ist.

  Dokumentieren Sie Code so, dass ein anderer Entwickler leicht entscheiden kann, ob er für eine anstehende Aufgabe geeignet ist.

Liefern, was verlangt wird

Wer sich dem Notwendigsten widmet, geht überall am sichersten zum Ziel. (Goethe)

Die Regel

Wie so oft in der Softwareentwicklung wird auch die Regel *Du wirst es nicht brauchen* gewöhnlich in ihrer englischen Formulierung *You Ain't Gonna Need It* oder kurz *YAGNI* zitiert.

Du wirst es nicht brauchen: Nur das soll und darf umgesetzt werden, was tatsächlich gefordert ist – und nicht mehr.

Motivation

Viele Entwickler, ich eingeschlossen, neigen dazu, bei ihrer Arbeit nicht nur das zu tun, was unmittelbar gefordert ist, sondern gleich noch einiges zu erledigen, von dem sie annehmen, dass es früher oder später ohnehin benötigt werden wird.

Manchmal ist es auch nur der Wunsch, eine Funktion vollständig zu realisieren, »die Sache rund zu machen«. Dahinter verbirgt sich der Gedanke, dass es leicht ist, dies jetzt gleich mit zu erledigen, und dass dadurch keine großen Kosten entstehen. Leider ist der zweite Teil dieses Gedankens meistens falsch.

Es gibt gewichtige Gründe, die gegen die Implementierung noch nicht erforderlicher Funktionen sprechen:

- Auch scheinbar geringe Mehraufwände summieren sich im Laufe eines Projekts zu relevanten Größen: »Kleinvieh macht auch Mist.«
- Die Komplexität des Codes wird erhöht. Der Umfang vergrößert sich, und beim Lesen müssen auch die nicht benutzten Codeteile betrachtet und verstanden werden.
- Der zusätzliche Code muss getestet werden, denn ein Projekt darf keinen ungeprüften Code enthalten. Dadurch entsteht

nochmals zusätzlicher Aufwand.

- ✔ Prognosen sind risikobehaftet. Wenn die zusätzlichen Funktionen entgegen der Voraussicht nie benutzt werden, kann es vorkommen, dass der eigentlich überflüssige Code notwendige Änderungen komplizierter macht.
- ✔ Es kann leicht passieren, dass die vorsorglich eingebauten Funktionen später ein zweites Mal gebaut werden, weil niemand mehr weiß, dass sie bereits vorhanden sind.

Wie Sie an diesen Punkten sehen, sind zusätzliche Funktionen, vor allem in der Summe über das Gesamtprojekt, eine nicht zu unterschätzende Bürde.

Umsetzung

Die technische Umsetzung dieser Regel ist gewöhnlich kein Problem. Die Schwierigkeiten liegen vor allem im mentalen Bereich. Die meisten Menschen sind so geprägt, dass sie eine zusätzliche Leistung als etwas Gutes, einen Bonus sehen.

Es braucht Selbstdisziplin, aus diesem Muster auszubrechen und quasi als »Geizhals« zu arbeiten. Denken Sie daran, dass weniger manchmal mehr ist, und bezogen auf den Code können Sie das »manchmal« getrost durch »fast immer« ersetzen.

Es gibt eine scheinbare Ausnahme. Wenn Sie ohne Mehraufwand eine Funktion so abstrakt realisieren können, dass sie über die konkrete Anforderung hinausreicht, ist natürlich nichts dagegen einzuwenden. Das ist beispielsweise der Fall, wenn Sie in einer abgeleiteten Klasse eine Funktion benötigen, die logisch in die Basisklasse gehört. Dann sollten Sie diese auch in der Basisklasse einfügen. Dadurch entsteht kein zusätzlicher Aufwand, sodass kein Regelverstoß vorliegt.

Schwierigkeiten

Wenn Sie mit der richtigen Einstellung ans Werk gehen, bleibt letztlich nur eine ernsthafte Schwierigkeit zurück: Sie müssen den

Code so schreiben, dass er später, wenn es nötig sein sollte, leicht geändert werden kann.

Die Grenze dazwischen, was der Änderbarkeit dient, und dem, was schon zusätzliche Funktionalität ist, lässt sich allerdings gar nicht so leicht bestimmen.

Diese Regel wird manchmal überinterpretiert oder mit zu kleinem Fokus angewandt. Wenn klar absehbar ist, dass eine bestimmte Funktion für die Gesamtlösung unverzichtbar ist, dann ist diese Funktion letztlich auch gefordert, wenn auch möglicherweise noch nicht in der gerade anstehenden Teilaufgabe.

In dieser Lage müssen Sie abwägen, ob eine (vorfristige) vollständige Umsetzung oder nur deren Vorbereitung für das Gesamtprojekt günstiger ist.

Jedes für sich

Wenn vor Ihnen ein unübersichtlicher Berg von Arbeit liegt, ist es nicht die schlechteste Idee, die Aufgaben erst einmal zu sortieren und dann nach Gruppen getrennt zu erledigen. Das klappt auch beim Programmieren.

Die Regel

Die *Trennung der Anliegen*, englisch *Separation of Concerns*, kurz *SoC*, ist eines der wichtigsten Konzepte, um komplexe Sachverhalte in eine verständliche Form zu bringen.

Trennung der Anliegen: Die unterschiedlichen Anliegen, die bei der Programmierung einer Aufgabe zu beachten sind, müssen im Code erkennbar abgetrennt erscheinen.

Es ist gleichzeitig ein recht schwieriges Konzept, weil eine höhere Ebene der Abstraktion betroffen ist.

Tatsächlich handelt es sich um ein elementares Konzept der Wissenschaft, die seit jeher einzelne Aspekte von Objekten isoliert betrachtet und analysiert, weil anders der Komplexität der Realität nicht beizukommen ist. Und genau wie in der Wissenschaft besteht auch beim Programmieren die Kunst vorrangig darin, die am besten geeigneten Aspekte zu erkennen.

Motivation

Der übergreifende Sinn dieser Regel besteht darin, ein gewisses Maß von Ordnung in komplexe Probleme zu bringen, um diese dadurch verständlicher und besser beherrschbar zu machen.

In der Softwareentwicklung sind die folgenden Aspekte bedeutsam:

- ✔ Trennung von Anliegen, die sich unabhängig voneinander ändern können.
- ✔ Trennung von Anliegen, die auf verschiedenen logischen Ebenen angesiedelt sind, beispielsweise technische von solchen fachlicher Art.
- ✔ Aufteilung in Module, die separat realisiert werden können.

Durch die Trennung entstehen Schnittstellen oder Grenzen, die dem Ganzen eine gewisse Struktur geben und die interne Kopplung vermindern oder zumindest explizit sichtbar machen.

Umsetzung

Bevor Sie Ihre Anliegen trennen können, müssen Sie diese erst einmal erkennen. Das hört sich leichter an, als es ist, weil dabei allzu leicht Dinge übersehen werden. Außerdem können sich die Anliegen zwischen Projekten oder deren Teilen stark unterscheiden.

Einige Beispiele für technische Anliegen sind Logging, Fehlerbehandlung, Datenpersistenz, Monitoring,

Transaktionsverwaltung oder Kommunikation.

Auf der fachlichen Ebene könnte es sich um das Einholen von Angeboten, die Verfolgung von Aufträgen oder den Versand von Rechnungen handeln.

Es ist nicht möglich, eine auch nur annähernd komplette Übersicht potenzieller Anliegen zu geben. Sie müssen also zunächst die in Ihrem Fall wichtigen Anliegen sammeln.

Dabei werden Sie schnell feststellen, dass vieles eng miteinander verknüpft ist. Das müssen Sie akzeptieren. Versuche Sie dann, Gruppen von Anliegen zu finden, zwischen denen die Abhängigkeiten weniger stark ausgeprägt sind. Unter Umständen kann dieser Schritt mehrfach erfolgen, indem Sie ihn auf die jeweils gewonnenen Komponenten erneut anwenden.

Im Idealfall erhalten Sie eine hierarchische Struktur, an der Sie sich bei der Umsetzung orientieren sollten. Sie werden im Code nicht vermeiden können, dass unterschiedliche Anliegen in einer Klasse oder gar einer Funktion auftreten. Versuchen Sie trotzdem durch Benennungen und optisches Erscheinungsbild, die Trennung offensichtlich zu machen.

Schwierigkeiten

Die größte Schwierigkeit liegt darin, die Anliegen so zu definieren, dass sie auf das konkrete Projekt bezogen intellektuell eingängig und genügend trennscharf sind. Das braucht Übung und Erfahrung.

Zudem müssen Sie damit rechnen, dass sich im Laufe der Arbeit neue Erkenntnisse oder Anliegen ergeben. Da es um Abstraktionen geht, ist das (gemeinsame) Verständnis wichtig und formales Vorgehen kaum hilfreich.

Aspektorientierte Programmierung

Die aspektorientierte Programmierung ist ein Weg, *Separations of Concerns* auch syntaktisch zu unterstützen. Sie ermöglicht es, die Kernfunktionalität

einer Anwendung unabhängig von gewissen *Querschnittsfunktionen* – den *Aspekten* – zu modularisieren.

Potenzielle Beispiele für Aspekte sind unter anderem das Logging und die Transaktionsverwaltung. Derartige Aspekte werden unabhängig in eigenen Artefakten beschrieben.

Die Verbindung des Kerns mit den Aspekten erfolgt erst zur Laufzeit. Dafür werden sogenannte *Joinpoints* definiert, beispielsweise Methodenaufrufe, bei deren Erreichen zugeordnete Aspektfunktionen aufgerufen werden.

Der Code wird dadurch gewissermaßen mehrdimensional. Idealerweise sind Aspekte und Kerncode jeweils völlig unabhängig – *orthogonal*.

Im Alltag ist diese Unabhängigkeit selten total gegeben. Wenn mehrere Aspekte an einem Joinpoint aufeinandertreffen, können sich Probleme, zum Beispiel durch die nicht leicht steuerbare Reihenfolge, ergeben. Aus diesem und ähnlichen Gründen beschränkt man sich gewöhnlich auf sehr wenige Aspekte, häufig nur auf einen.

Die SOLID-Regeln

Dieser Name hat sich für fünf Regeln eingebürgert, die ursprünglich aus der objektorientierten Programmierung stammen, teilweise aber darüber hinaus helfen, besseren Code zu schreiben.

Der Name setzt sich aus den Anfangsbuchstaben der englischen Bezeichnungen zusammen, deshalb werde ich diese auch für die folgenden Abschnittsüberschriften wählen.

Single Responsibility Principle – SRP

Prinzip einer einzigen Verantwortung: Jede Klasse soll genau eine Verantwortung übernehmen, in dem Sinne, dass es nie mehr als einen Grund geben kann, diese Klasse zu ändern.

Dieses *Prinzip einer einzigen Verantwortung* für Klassen sieht auf den ersten Blick wie eine Konkretisierung des Prinzips der Trennung der Anliegen aus, und es wird auch an vielen Stellen so interpretiert.

Tatsächlich ist der Schwerpunkt jedoch ein anderer. Wie der Name schon sagt, geht es um »Verantwortung« und nicht um »Anliegen«. Verantwortung ist dabei eine außerhalb der Softwarewelt liegende Kategorie, die vor allem als Änderungserwartung verstanden wird. Einzig diese eine Änderungserwartung sollte einen Grund liefern, die Klasse zu ändern.

Dadurch soll eine hohe Kohäsion innerhalb der Klasse gefördert werden. Bei strikter Umsetzung wirken sich in der Folge Anforderungsänderungen nur auf einen sehr eng begrenzten Codebereich aus. Beispiele möglicher Verantwortungen sind Wissen (Daten, Informationen), Steuerung oder Erzeugung.

Dieses Prinzip muss mit Bedacht eingesetzt werden. Die Änderungserwartungen liegen außerhalb der Softwaresphäre und werden selten angemessen dokumentiert.

Wenn die später wirklich geforderten Änderungen stark von den ursprünglich erwarteten abweichen, kann das dazu führen, dass die Motivation für einzelne Klassendefinitionen nicht mehr intuitiv nachvollziehbar ist.

Möglicherweise sind es die genannten Nachteile, die dazu geführt haben, dass dieses Prinzip überwiegend im Sinne der Trennung von Anliegen ausgelegt wird, was nicht Bestandteil der SOLID-Regeln ist.

Der Nutzen des Hervorhebens der Verantwortung ist vorrangig darin zu sehen, dass eine andere Sicht auf die Frage, was Elemente zusammengehörig macht, gefördert wird.

Da Prognosen und damit Änderungserwartungen stets unsicher sind, sollten Sie dieses Prinzip konsequent nur dann anwenden, wenn ausnahmsweise hinreichende Klarheit über bevorstehende Änderungen besteht.

In allen anderen Fällen ist es eine nützliche Ergänzung, wenn Sie versuchen, Anliegen, die durchaus in Verbindung mit Verantwortungen stehen können, zu ermitteln.

Open Closed Principle – OCP

Offen-Geschlossen-Prinzip: Softwaremodule sollten offen für Erweiterungen und geschlossen gegenüber Änderungen sein.

Auch beim *Offen-Geschlossen-Prinzip* geht es um die Änderungen, denen die Software unterliegt. Im Gegensatz zum SRP, das darauf abstellte, erforderliche Änderungen im Code lokal zu begrenzen, zielt das Offen-Geschlossen-Prinzip darauf ab, Anforderungsänderungen durch Ergänzungen von Code zu realisieren – nicht durch Änderung im bestehenden Code.

Diese Zielsetzung resultiert aus der Erfahrung, dass jede Änderung im Code gefährlich ist. Umsetzen können Sie dieses Prinzip in erster Linie durch die Verwendung von Interfaces und Vererbung.

Alternative Implementationen eines Interfaces oder eine abgeleitete Klasse erweitern die Funktion, ohne dass der vorliegende Code geändert werden muss, abgesehen vielleicht von einer Erzeugungsmethode für die Instanziierung von Objekten der neuen Klasse.

Daneben existieren weitere Techniken, zum Beispiel *Extension Points* (Erweiterungspunkte), die eine einfache Erweiterbarkeit eines Programms unterstützen. Prinzipielle Schwierigkeiten gibt es dabei kaum. Allenfalls kann die Definition der Schnittstellen einiges Kopfzerbrechen verursachen, weil diese zum

»geschlossenen« Teil gehören und entsprechend allgemein gefasst sein müssen.

Ein angenehmer Nebeneffekt dieses Prinzips ist, dass bei Modifikationen keine Testfälle angepasst werden müssen. Es kommen lediglich einige für die neuen Funktionen hinzu.

Liskov Substitution Principle – LSP

Liskovsches Substitutionsprinzip: Eine Klasse soll in jedem Fall durch ihre Unterklassen ersetzbar sein.

Da das *liskovsche Substitutionsprinzip* syntaktisch stets erfüllt ist, handelt es sich um eine inhaltliche Anforderung. Damit beginnen die Schwierigkeiten, weil die implizit bei der Definition einer Klasse angenommenen Eigenschaften im Allgemeinen nicht ausreichend exakt und vollständig definiert werden.

Ein praktikabler Weg zur Umsetzung dieses Prinzips besteht darin, Testfälle bereitzustellen, die alle verwendeten Eigenschaften der Basisklasse verifizieren. Das bedeutet gleichzeitig, dass bei der Entwicklung einer Anwendung vor jeder Verwendung einer Funktion dieser Basisklasse geprüft werden muss, ob eben diese Art der Verwendung durch einen passenden Testfall abgesichert und damit garantiert ist.

Unterklassen, die alle Tests erfolgreich bestehen, erfüllen dann das LSP. Dieser Aufwand ist dann gerechtfertigt und notwendig, wenn eine größere Zahl von abgeleiteten Klassen zu erwarten ist. Ein typisches Beispiel sind die diversen Elemente einer grafischen Oberfläche, die alle von einer (abstrakten) Basisklasse erben.

Das Prinzip ist ausschließlich in dem eng umgrenzten Bereich der Bildung von abgeleiteten Klassen anwendbar. Weil inzwischen Delegation wegen der weniger engen Bindung häufig der Vererbung vorgezogen wird, hat das LSP etwas an Bedeutung verloren. Das ändert nichts daran, dass die Missachtung gravierende Folgen haben kann.

Interface Segregation Principle – ISP

Das *Prinzip der Abtrennung von Schnittstellen* wird hier nicht behandelt. Dieses Thema ist so grundlegend, dass es bereits ein eigenes Kapitel erhalten hat: 14 *Passend schneiden – Schnittstellen*.

Dependency Inversion Principle – DIP

Prinzip der Umkehr der Abhängigkeiten: Abhängigkeiten sind nur zu Abstraktionen erlaubt, nicht zu Spezialisierungen.

Das *Prinzip der Umkehr der Abhängigkeiten* ist schon lange Teil der etablierten Entwicklungspraxis geworden. Auch dieses Prinzip soll helfen, enge Kopplungen zu vermeiden. Deshalb muss die in einer naiven Struktur natürlicherweise vorhandene Abhängigkeit höher angeordneter von niedriger angeordneten Modulen – erstere realisieren ihre Funktionen unter Nutzung von Funktionen der tieferen Schichten – aufgebrochen werden.

Dazu wird zwischen den Schichten eine Abstraktions- oder Schnittstellenebene eingefügt. Diese Abstraktionen sollten weder Details der höheren noch der niedrigeren Schicht enthalten.

Beide Schichten hängen dann von der Abstraktion ab, die höhere als Nutzer, die niedrigere als Implementierer. Da die Schichten nur noch über die Abstraktionen verbunden sind, haben Änderungen auf der einen Ebene keine Auswirkungen auf die andere, solange die Schnittstellen unberührt bleiben. Sie sehen auch hier wieder die Bedeutung guter Schnittstellen oder Abstraktionen.

Dependency Injection – DI

Das *Einbringen von Abhängigkeiten* ist ein Muster zur Umsetzung des Prinzips der Umkehr der Abhängigkeiten. Das Prinzip der Umkehr der Abhängigkeiten selbst sagt nichts darüber aus, wie ein Objekt, das eine Funktion des

Interfaces nutzt, zu einem konkreten Objekt kommt, das dieses Interface implementiert.

Eine häufig benutzte Möglichkeit ist die Übergabe als Parameter im Konstruktor. Ebenfalls gebräuchlich ist der Zugriff auf ein Repository. Beiden Ansätzen gemeinsam ist, dass sich das nutzende Objekt aktiv um die benötigte Instanz bemühen muss.

Darüber hinaus bieten verschiedene *DI-Container* genannte Frameworks die Möglichkeit, das Einbringen der benötigten Objekte zu automatisieren.

Einfach besser

Zum Abschluss dieses Kapitels möchte ich Sie noch an eine Regel erinnern, die Sie bei allen Problemen nie vergessen sollten: Nichts unnötig kompliziert machen! Natürlich hat auch diese Regel Eingang in die Sammlung der anerkannten Programmiergrundsätze gefunden, sogar gleich mehrfach.

Halte es einfach

Diese Variante ist unter der Abkürzung *KISS* bekannt. Interessanterweise gibt es unterschiedliche Interpretationen, nämlich *Keep it simple, stupid* und *Keep it simple (and) stupid*. Es geht darum, ein Problem so einfach wie möglich zu lösen.

Einfache Lösungen sind nicht nur schneller umzusetzen, sondern weisen in der Regel weniger Fehler auf und sind leichter anzupassen.

Manchmal ist es allerdings gerade eine einfache Lösung, die schwer zu finden ist, weil sie umfangreiche Analysen und ein tiefes Verständnis des Problems erfordert.

Geringste Überraschung

Das *Prinzip der geringsten Überraschung*, englisch *Principle of Least Surprise*, *POLS*, konzentriert ebenfalls auf eine verständliche und eindeutige Codestruktur.

Dadurch sollen die Beteiligten vor unerwarteten Situationen bewahrt werden. Seiteneffekte und namenlose Konstanten

(»Magic Numbers«) sind typische Formen solcher Überraschungen. Code, der Überraschungen enthält, ist nicht leicht verständlich und entsprechend schwer weiterzuentwickeln.

Fazit

Außer den in diesem Kapitel besprochenen Regeln und Prinzipien gibt es noch viele weitere, die sich teilweise überschneiden oder im Widerspruch zueinander stehen können. Falls Sie Zweifel haben, welchem Prinzip der Vorrang zu geben ist, vergleichen Sie die Ergebnisse anhand eines Kriteriums, das als Ockhams Rasiermesser bekannt ist: »Wenn es mehrere Darstellungen für einen bestimmten Sachverhalt gibt, dann ist diejenige Darstellung zu bevorzugen, die am einfachsten ist, also mit den wenigsten Annahmen und Variablen auskommt.«

- Schreiben Sie einfach verständlichen Code. Erst wenn Sie bemerken, dass Ihnen das nicht recht gelingt, versuchen Sie, anhand der beschriebenen Regeln die Ursachen zu klären und es besser zu machen.
- Prüfen Sie, ob die Regeln Ihnen helfen, funktionierenden Code verständlicher zu machen.
- Denken Sie immer daran, dass die dargestellten Regeln Ihnen bei Problemen helfen sollen. Sie sind keine unbedingt einzuhaltenden Vorschriften.

Das Wichtigste in Kürze

- Regeln sind wichtige Hilfen, die in unübersichtlichen Lagen die Orientierung erleichtern. Ob eine Regel sinnvoll anwendbar ist, hängt stark von den jeweiligen Bedingungen ab.
- Wiederholung von Codeteilen sollte vermieden werden. Das spart Arbeit und verhindert Fehler durch Übersehen bei späteren Änderungen.

- ✔ Immer nur das liefern, was wirklich erforderlich ist. Zusätzliche Leistungen bringen oft keinen Nutzen, erhöhen aber die Codekomplexität.
- ✔ Unterschiedliche Anliegen sollten auch im Code als solche erkennbar bleiben.
- ✔ Jede Klasse sollte nur eine Verantwortung im Sinne einer Änderungserwartung haben.
- ✔ Softwaremodule sollten offen für Erweiterungen und geschlossen gegenüber Änderungen sein.
- ✔ Innerhalb der Anwendung sollte eine lose Kopplung durch Schnittstellendefinitionen erreicht werden.
- ✔ Verständlichkeit wird vor allem durch Einfachheit erreicht. Der Code sollte einen Leser an keiner Stelle überraschen.

Kapitel 17

Fehler passieren – Fehlerbehandlung

IN DIESEM KAPITEL

Fehler und Fehlerarten

Strategien, auf Fehler zu reagieren

Auch wenn es keiner gern zugibt, Software und Fehler scheinen untrennbar zusammenzugehören. In diesem Kapitel geht es einmal darum, ein fundiertes Verständnis darüber zu gewinnen, was Fehler überhaupt sind. Zum anderen wird erklärt, wie die verschiedenen Fehlerarten behandelt werden können, ohne die Verständlichkeit des Codes zu stark zu beeinträchtigen.

Ausgangslage

Nach verschiedenen Untersuchungen enthält aktuelle Software bei der Produktivnahme immer noch bis zu drei Fehler pro 1000 Zeilen Code. Das ist besorgniserregend angesichts der Tatsache, dass IT-Systeme immer tiefer in alle Bereiche des Daseins eindringen und damit die potenziellen Folgen und Kosten solcher Fehler enorm wachsen.

Es wäre deshalb unverantwortlich, wenn das Clean-Code-Konzept sich nicht auch um diese wichtige Frage kümmern würde. Natürlich erhöhen verständlicher Code und Tests die Chancen, dass mögliche Fehler frühzeitig erkannt werden. Das allein reicht jedoch nicht aus.

Notwendig ist eine ganz allgemein veränderte Einstellung zu Fehlern: Fehler sind keine lästige Randerscheinung, die bei völlig korrekter Arbeitsweise vermeidbar wäre – Fehler sind ein nicht ignorierbarer Teil der Realität. Entwickler müssen diese Tatsache akzeptieren und lernen, dass Fehler und der Umgang mit Fehlern ein unverzichtbarer Teil jeder Softwareentwicklung sein müssen, und zwar von Beginn an.

Leider sieht das in der Praxis gewöhnlich anders aus. Viel zu oft orientiert sich der gesamte Entwicklungsprozess am sogenannten »Gutfall«, das heißt der problemlosen Ausführung aller Funktionen. Abgesehen von einigen Exceptions, deren Behandlung nicht zu umgehen ist, enthält der Code keine Vorkehrungen zur Beherrschung von Fehlerzuständen. Diese werden erst nach und nach eingebaut, wodurch die ursprünglich (hoffentlich) klare Struktur langsam im Nebel der Fehlerbehandlung versinkt.

Es kommt vor, dass der Fehlerbehandlungscode umfangreicher wird als der Code für die eigentlichen Funktionen. Daher ist es nicht überraschend, dass gerade dieser zusätzliche Code überdurchschnittlich häufig verborgene Fehler enthält. Sie sehen daran, wie notwendig es ist, dem Umgang mit »erwarteten« Fehlern mehr Aufmerksamkeit zu schenken.

Fehlerarten

Es gibt verschieden Auslegungen für den Begriff Fehler in Programmen, die alle sehr allgemein gefasst sind. Ich verwende diese Definition:

Ein *Fehler* ist ein unerwarteter Zustand bei der Ausführung eines Programms.

Mit »unerwartet« ist dabei gemeint, dass das Erreichen dieses Zustands weder spezifiziert wurde noch der allgemeinen Erfahrung entspricht. Man kennt inzwischen einige dieser – normalerweise unerwarteten – Zustände und versucht, im Code

darauf zu reagieren. Software, die relativ stabil auf Fehler reagiert, wird *robust* oder *fehlertolerant* genannt.

Sauberer Code sollte stets auch möglichst robust sein. Um das zu erreichen, sind ganz unterschiedliche Strategien angebracht, und zwar je nach der Art und Wahrscheinlichkeit des erwarteten Fehlers.

- **Datenfehler**: Die gelieferten Daten entsprechen nicht den Erwartungen. Das kann sowohl das Format (Syntax) als auch den Inhalt (Semantik) betreffen.
- **Funktionsfehler**: Eine Programmfunktion liefert ein unerwartetes Ergebnis, beispielsweise einen `null`-Wert, oder wirft eine Exception.
- **Hardwarefehler**: Praktisch sind diese nur mit Bezug auf externe Datenquellen relevant, wenn beispielsweise eine Datenbank nicht erreichbar ist. Fehler in zentralen Komponenten (CPU, Speicher) haben praktisch immer so fatale Auswirkungen, dass sie im Code keiner Behandlung zugänglich sind.
- **Semantische Fehler**: Eine Programmfunktion liefert ein falsches Ergebnis, ohne dass dies sofort erkennbar ist.

Für jede genannte Fehlerart wird nun besprochen, wie Sie zweckmäßigerweise damit umgehen sollten.

Datenfehler

Datenfehler können abhängig von der jeweiligen Datenquelle mit sehr unterschiedlichen Häufigkeiten auftreten. Davon hängt die empfohlene Behandlung ab. Wo die Grenze zwischen »häufig« und »selten« liegt, ist wie so vieles eine Ermessensfrage, die stark vom jeweiligen Umfeld beeinflusst wird und nicht zuletzt anhand des Schadenspotenzials beurteilt werden muss.

Seltene Datenfehler

Wenn Sie Daten verarbeiten, die durch ein anderes Programm oder andere Funktionen erzeugt wurden, ist die Wahrscheinlichkeit groß, dass diese Daten korrekt sind. Ob Sie mit Fehlern rechnen müssen, hängt von den konkreten Umständen ab.

Bei fremder Software kann es angebracht sein, zu kontrollieren, sobald externe Datenquellen wie Datenbanken, Dateien oder Netzwerke beteiligt sind, muss kontrolliert werden.

Da fehlerhafte Daten in diesem Szenario sehr selten vorkommen, sollten Sie beim Auftreten eine Ausnahme oder Exception werfen. Was dabei zu bedenken ist, erfahren Sie in Kapitel 18 *Ausnahmen regeln – Exceptions*. Oft wird das zu einem Abbruch der Bearbeitung führen und die Beseitigung des Fehlers einen äußeren Eingriff erfordern.

Häufige Datenfehler

Von manchen Datenquellen ist bereits im Voraus bekannt, dass sie relativ unzuverlässig sind und Fehler deshalb keine seltenen Ereignisse sind, sondern gewissermaßen zum »Normalbetrieb« gehören.

Fehlerquellen

Dieser Fall ist zumindest immer dann gegeben, wenn Menschen die Daten produzieren, zum Beispiel in einem Eingabedialog. Denn Menschen machen nun einmal Fehler. Sie stehen damit aber nicht ganz allein. Es gibt ebenso technische Systeme, zum Beispiel Sensoren oder Bilderkennungssoftware, die unter bestimmten Bedingungen vermehrt fehlerhafte Daten liefern können.

Das bei seltenen Fehlern angemessene Abbrechen ist im Fall häufiger Fehler keine gangbare Option. Deshalb muss der Umgang mit dieser Art von Fehlern bereits in die Konzeption aufgenommen werden.

Zuerst muss geklärt werden, welche Arten von falschen Daten möglich sind. Danach können Sie überlegen, ob einige Fehler

dadurch ausgeschlossen werden können, dass Sie beispielsweise ein Textfeld durch eine Auswahlbox, die nur zulässige Werte enthält, ersetzen.

Zu verhindern, dass Fehler überhaupt erst möglich sind, ist zweifellos die beste Strategie, aber leider oft nicht praktikabel.

Streng genommen handelt es sich bei dem, was ich hier »häufige Datenfehler« nenne gar nicht um Fehler im Sinne der Definition. Denn wenn von vornherein klar ist, dass solche Fehler auftreten können, ist der durch sie hervorgerufene Zustand auch nicht »unerwartet«.

Syntaktische Datenfehler

Daten können syntaktisch oder semantisch falsch sein. Zum Beispiel ist eine Datumszeichenkette `30.02.1990` zwar syntaktisch richtig, semantisch aber falsch, weil es keinen 30. Februar gibt.

Das Beispiel bedarf eventuell einer kurzen Erläuterung, denn die Abgrenzung zwischen Syntax und Semantik ist im Wesentlichen pragmatisch. Sie können ohne Schwierigkeiten eine kontextfreie Grammatik definieren, die den 30.02. und andere nicht existierende Daten ausschließt. Wegen des damit verbundenen Aufwands wird das jedoch gewöhnlich nicht gemacht.

Die syntaktische Korrektheit ist vergleichsweise simpel beim Entgegennehmen der Daten zu prüfen. Das sollten Sie nach Möglichkeit auch tun, denn an dieser Stelle ist die Fehlerbehandlung noch einfach – zurückweisen oder ignorieren, und Sie können sich in der Folge zumindest darauf verlassen, dass das Format richtig ist. Das erspart Ihnen (ganz im Sinne der DRY-Regel) wiederholte Prüfungen.

Semantische Datenfehler

Das Vertrackte an semantischen Fehlern in den Daten ist, dass man sie nicht leicht erkennen kann. So einfach wie mit dem 30.

Februar werden Sie es selten haben. Trotzdem sollten Sie auch semantische Fehler so früh wie möglich zu erkennen versuchen. Denn dadurch wird die Behandlung einfacher und die Fehlerbeschreibung genauer.

Je tiefer in der Aufrufhierarchie Sie sind, wenn Sie einen Fehler entdecken, desto aufwendiger ist es, darauf zu reagieren. Im schlimmsten Fall ist die eigentliche Ursache gar nicht mehr zu ermitteln.

Es ist äußerst ärgerlich, wenn man nach dem Absenden eines längeren Webformulars nur die Antwort erhält, dass die Bearbeitung wegen eines falschen Werts nicht möglich war.

Derartiges ist inzwischen zwar seltener geworden, aber ich habe schon komplette Abstürze anstelle einer vernünftigen Reaktion erlebt.

Ebenso schlecht ist es, wenn eine größere Datenmenge wegen eines einzigen fehlerhaften Datensatzes nicht verarbeitet werden kann.

Um semantische Fehler feststellen zu können, benötigen Sie allerdings oft Informationen, die normalerweise erst im Laufe der Verarbeitung vorliegen. Es sieht dann so aus, als ob Sie für die semantische Prüfung Verschiedenes zweimal oder öfter erledigen müssten.

Das lässt sich jedoch vermeiden, wenn Sie die Fehlerprüfungen rechtzeitig einplanen. Eine Möglichkeit, mehrfachen Aufwand zielgerichtet zu vermeiden, ergibt sich durch die Analyse des Datenflusses.

Wenn Sie sich ansehen, wie ein bestimmtes Datum durch die Anwendung fließt, werden Sie erkennen, welche Abhängigkeiten zu anderen Daten bestehen und an welchen Stellen Sie bestimmte Prüfungen frühestmöglich ausführen können.

Entscheidend ist nun, dass Sie die für die Prüfung gewonnenen zusätzlichen Informationen anschließend nicht wieder verwerfen,

sondern mit dem ursprünglichen Datum zusammen weiterreichen.

Im Idealfall haben Sie am Ende eine Menge bereits angereicherter und korrekter Daten, die eventuell schon das benötigte Ergebnis enthalten. Selbst wenn das nicht so ist, wird die verbleibende Verarbeitung wesentlich übersichtlicher sein.

Das folgende Beispiel soll Ihnen das Prinzip der Datenanreicherung verdeutlichen. Wahrscheinlich kennen Sie ähnliche Konstellationen.

In einem Projekt, an dem ich zeitweise mitgearbeitet habe, wurden die Eingangsdaten in vorbildlicher Weise kontrolliert. In einer ersten Schicht erfolgte eine Plausibilitätsprüfung. Wurde diese bestanden, folgte in der nächsten Schicht die Validierung und danach die Übergabe an die eigentliche Verarbeitung.

Alle Daten wurden im Zeichenkettenformat angeliefert und auch weitergegeben. Im Falle eines Datums bedeutete das die dreimalige Konvertierung vom `String`- in das `Date`-Format. (Die Plausibilitätsprüfung hätte den 30. Februar bereits aussortiert, aber nicht berücksichtigt, ob das Datum unzulässigerweise in der Zukunft liegt. Das oblag der Verifikation.)

Datumskonvertierungen sind vergleichsweise aufwendige Operationen. Die bessere Lösung besteht deshalb darin, als Ergebnis der Plausibilitätsprüfung ein spezielles Objekt zu liefern, das die originalen und die konvertierten Daten enthält. Die Verifikation muss dann nur noch den konvertierten Wert prüfen, und die Verarbeitung kann diesen ohne jede weitere Vorbereitung verwenden.

Wenn Sie Daten durch eine Vorverarbeitung anreichern, ist es unter Umständen sinnvoll, das Ausgangsdatum ebenfalls weiterzugeben. Möglicherweise wird es in folgenden Anreicherungsschritten noch benötigt. Auf jeden Fall ist es nützlich, wenn Sie es für das Logging und eventuelle Fehlernachrichten verfügbar haben.

Bei der Übertragung sehr großer Datenobjekte sollten Sie allerdings überprüfen, ob die mehrfache Konvertierung nicht doch weniger Aufwand verursacht.

Funktionsfehler

Funktionsfehler sind solche Fehler, bei denen eine Funktion ein Ergebnis liefert, das im Unterschied zu den semantischen Fehlern sofort als fehlerhaft erkennbar ist. Sehr oft treten Funktionsfehler als Folge von Datenfehlern auf. Deshalb ist die frühzeitige Behandlung von Datenfehlern wichtig.

In Ihrem eigenen Code sollten Methoden nie derartige fehlerhafte Ergebnisse liefern, sondern stattdessen eine Exception werfen. Da es jedoch verbreitete Praxis ist, in solchen Fällen beispielsweise `null` zu liefern, müssen Sie bei Fremdsoftware darauf reagieren.

Wie in Kapitel 13 *Kleine Schritte – saubere Methoden* beschrieben, sollten Sie solche Methodenaufrufe kapseln. In Listing 17.1 sehen Sie drei Möglichkeiten.

```
public Object wrap1() {
  return Optional.ofNullable(methodeKannNullLiefern())
      .orElse(DEFAULTRESULTAT);
}
public Object wrap2(Object defaultResultat) {
  return Optional.ofNullable(methodeKannNullLiefern())
      .orElse(defaultResultat);
}
```

```
public Object wrap3() {
  Object result= methodeKannNullLiefern();
  if (result==null) {
    throw new NullPointerException(
                  "methodeKannNullLiefern returns
null");
  }
  return result;
}
```

Listing 17.1: Verpacken von Methoden

Auf keinen Fall sollte ein möglicherweise fehlerhafter Wert einfach weitergereicht werden.

Seien Sie vorsichtig beim Rückgriff auf Defaultwerte. Unter ungünstigen Umständen können dadurch Fehler, zumindest zeitweise, maskiert werden. Das behindert die frühzeitige Entdeckung und die Ursachensuche.

Ansonsten gilt wie bei allen Fehlerbehandlungen, dass es wichtig ist, die Frage, ob und wie nach einem Fehler die Verarbeitung fortgesetzt werden kann, bereits beim Entwurf der Umsetzung zu bedenken.

Hardwarefehler

Hardwarefehler treten heute praktisch nur noch in Verbindung mit peripheren Geräten auf. Fehler in CPU-Schaltkreisen, von denen Sie bisweilen in der Zeitung lesen, sind gewöhnlich so subtil, dass sie lediglich für Hacker interessant sind, auf den normalen Programmablauf aber keinen Einfluss haben. Ganz anders sieht es aus, wenn es um externe Geräte oder den Netzwerkanschluss geht.

Prinzipiell gilt auch für Hardwarefehler, dass es eine grundlegende Designentscheidung sein sollte, wie auf Anwendungsebene damit umzugehen ist.

Selbstverständlich sollten Sie jeden Hardwarezugriff so gekapselt haben, dass eine elementare Fehlerbehandlung, beispielsweise wiederholte Zugriffsversuche, unabhängig vom Rest des Codes möglich ist. Es hängt jedoch von Ihrer Anwendung ab, ob beispielsweise bei verloren gegangener Netzwerkverbindung Daten für das spätere Senden zwischengespeichert werden sollen oder die Verarbeitung abgebrochen werden muss.

Falls Ihr Code Funktionen enthält, die Netzwerkdienste und Ähnliches benötigen, sollten Sie den Umgang mit den dabei zu erwartenden Fehlern als ein gesondert zu behandelndes Anliegen im Sinne der SoC-Regel betrachten.

Semantische Fehler

Als semantische Fehler bezeichne ich hier diejenigen Fehler, die dazu führen, dass eine Funktion, unter Umständen nur in ganz besonderen Fällen, nicht das erwartete Ergebnis liefert.

Gegen semantische Fehler ist niemand gewappnet, und Sie können im Code auch nur wenig dagegen tun. Ihre Hauptwaffe im Kampf mit diesem Feind sind die Tests. Leider ist diese Waffe nicht besonders scharf, denn Tests können zwar Fehler finden, aber nie die Korrektheit nachweisen.

Plausibilitätsprüfung

Während es bei der Eingabe gute Praxis ist, Daten zumindest einer groben Gültigkeitsprüfung zu unterziehen, werden Sie das auf der Ergebnisseite nur selten finden. Offensichtlich sind trotz aller gegenteiligen Erfahrungen die meisten Entwickler immer noch davon überzeugt, dass ihre Software »im Prinzip« richtig ist.

Vielleicht macht es ja wirklich keinen guten Eindruck, wenn da noch ganz offensichtlich ein Auffangnetz eingebaut wird. Sie sollten es trotzdem machen.

Niemand käme auf die Idee, einem Auto- oder Flugzeughersteller mangelndes Vertrauen in seine Produkte zu unterstellen, nur weil wichtige Teile redundant ausgelegt sind.

Software ist so komplex, dass einzelne Fehlfunktionen einfach nicht ausgeschlossen werden können und deshalb vorbeugende Maßnahmen immer wichtiger werden.

Ergebnisse sind stets wieder Eingangsdaten für ein anderes System. Betrachten Sie deshalb die Daten aus der Sicht des Nachfolgers und stellen Sie dessen Plausibilitäts- und Verifikationskriterien zusammen. Dann haben Sie eine gute Grundlage für Ihre Ergebnisprüfung.

Viele grobe Fehlausgaben ließen sich schon durch einfache Kontrollen vermeiden.

Beispiele für unterlassene Ergebnisverifikationen gibt es viele. Ich möchte Ihnen zwei mit ganz unterschiedlichen Auswirkungen geben:

- ✔ Es ist höchst irritierend, wenn auf einer Informationstafel, wie man sie zum Beispiel in Zügen finden kann, für ein noch nicht eingetretenes Ereignis (Ankunft oder Abfahrt) eine Uhrzeit in der Vergangenheit prognostiziert wird – oft noch direkt neben der Anzeige der aktuellen Uhrzeit. Das lässt Zweifel an der Verlässlichkeit des Informationssystems aufkommen. Allerdings ist das nur ein ärgerlicher Schönheitsfehler.
- ✔ Ein Softwarefehler in der Steuerung der Bestrahlungsanlage Therac-25 hatte zur Folge, dass Krebspatienten teilweise zu hohe Dosen erhielten, woran einige in der Folge starben.

  In diesem Fall wäre eine Prüfung der Ausgangsdaten auf zulässige Höchstwerte sogar lebensrettend

gewesen.

Vermeiden Sie es aber, solche Prüfungen ad hoc, sozusagen »auf eigene Faust« einzufügen. Das Festlegen der zulässigen Wertebereiche ist oft nicht trivial und ganz klar eine fachliche Aufgabe. Deshalb müssen festgestellte Verstöße auch in einer Weise berichtet werden, die für die Fachseite verständlich ist.

Außerdem sollten Sie beim Implementieren berücksichtigen, dass Änderungen leicht umgesetzt werden können.

In vielen Bereichen ist die Welt sehr volatil geworden. Dinge, die gestern noch unvorstellbar waren, sind plötzlich alltäglich. Denken Sie beispielsweise an negative Zinsen. Die Wahrscheinlichkeit, dass Grenzwerte, die heute unumstößlich zu sein scheinen, morgen doch modifiziert werden müssen, ist hoch.

Ehe Sie jetzt daran gehen, diese Empfehlung mit aller Kraft umzusetzen, muss ich noch eine ernste Warnung aussprechen: Beachten Sie die Angemessenheit! Denn natürlich wird auf diese Art die Komplexität Ihres Codes steigen.

Es ist überdies nicht unwahrscheinlich, dass die fachlich vorgegebenen Grenzwerte selbst fehlerhaft sind, weil nicht alle Randbedingungen bedacht wurden. Diese zusätzlichen Risiken müssen Sie gegen die Konsequenzen falscher Ergebnisse bewerten.

Was ich hier vorschlage, ist das Einziehen einer dem konkreten Problem angemessenen letzten Verteidigungslinie im Sinne des defensiven Programmierens.

Defensive Programmierung

Unter defensivem Programmieren versteht man einen Programmierstil, bei dem möglichst viele Vorbedingungen überprüft werden, sodass unerwartete Fehler vermieden werden können. Das Konzept ist umstritten, weil der Code durch die Vielzahl der Testanweisungen, von denen die meisten überflüssig

sind, verschmutzt wird. Insbesondere in der Clean-Code-Bewegung wird bemängelt, dass der defensiven Programmierung fehlendes Vertrauen in die Arbeit anderer zugrunde liegt und dieses Manko durch bessere Kommunikation zu beheben wäre.

Dieser Aspekt ist nicht falsch, übersieht aber, dass daneben fehlendes Wissen über Fremdcode und dessen fachlichen Hintergrund ebenfalls wesentlich ist. Die Frage unvollständig definierter Schnittstellen spielt dabei eine zentrale Rolle.

Ein ernsterer Einwand ergibt sich schließlich daraus, dass die zahlreichen Tests zwar unmittelbare Fehlfunktionen verhindern können, aber oft den ursprünglichen Fehler nur maskieren und weitergeben und ihn damit schwerer lokalisierbar machen.

Grundsätzlich sollte defensives Programmieren nicht vollständig abgelehnt, sondern auf die Stellen konzentriert werden, an denen anderweitig unerkannt gebliebene Fehler erhebliche Auswirkungen haben können. Es ist einfach unrealistisch, zu erwarten, dass Software fehlerfrei ist.

Klären Sie bereits bei Projektbeginn, ob es für die von der Anwendung erwarteten Ergebnisse sinnvolle Beschränkungen gibt.

Selbst wenn Sie daraus später keine implementierten Kontrollen machen, trägt allein die Beantwortung der Frage erheblich zum Problemverständnis bei.

Wertebereichs-Überschreitungen

Lassen Sie mich mit einer kleinen Provokation beginnen. Ich behaupte, dass, wenn Sie überhaupt schon einmal Code geschrieben haben, auf dessen Funktionieren Sie stolz waren, Sie – mehr oder weniger bewusst – Code geliefert haben, der nicht vollständig korrekt war.

Wer kann von sich behaupten, dass er beim Codieren von mathematischen Ausdrücken stets daran denkt, dass die Zahlen-Datentypen der Programmiersprachen einen wichtigen Unterschied zu ihren mathematischen Vorbildern haben? Sie sind endlich. Das kann man leicht ignorieren, solange die Grenzen

weit genug entfernt sind. Nur wer sagt Ihnen, was weit genug entfernt ist und ob das in allen Anwendungsfällen garantiert ist?

Ein folgenschweres Beispiel dafür, wozu die Ignoranz von Zahlbereichsgrenzen führen kann, ist der Absturz einer Ariane-5-Rakete im Jahr 1996. Die Software enthielt eine Konvertierung einer 64-Bit-Gleitkommazahl in einen 16-Bit-Integerwert. Beim kleineren Vorgängermodell Ariane 4 waren die Beträge noch im zulässigen Bereich gewesen.

Daran sehen Sie auch sofort die Gefahr: Zum Zeitpunkt des Schreibens kann Code der Aufgabe angemessen sein. Aber bei der Wiederverwendung wird selten geprüft, ob die (stillschweigend) vorausgesetzten Einschränkungen weiterhin gelten.

Betrachten Sie bitte folgendes Java-Codeschnipsel:

```
int i= 0;
do; while (Math.abs(i--)>= 0);
```

Auf den ersten Blick könnte man meinen: Das ist eine unendliche Schleife. Ist es aber nicht.

Wenn Sie die Dokumentation von `Math.abs` nachschlagen, steht dort: »Note that if the argument is equal to the value of Integer.MIN_VALUE… the result is that same value, which is negative«. Das ist überraschend. Zum mathematischen Grundwissen gehört schließlich, dass der Absolutbetrag einer Zahl nie negativ ist.

Immerhin können Sie die Warnung noch finden. Aber ist Ihnen bewusst, dass die gleiche Aussage für den Ausdruck `(a < 0) ? -a : a` zutrifft, dass dessen Wert mitnichten immer positiv ist?

Gegen Fehler dieser Art hilft auch sauberer Code nur sehr bedingt. Einerseits ist das Bewusstsein dafür nur schwach ausgeprägt, andererseits spielen sie in der Projektpraxis bisher nur eine untergeordnete Rolle.

Wie leicht derartige Fehler übersehen werden, zeigt ein Problem aus dem Java Development Kit (JDK). Der dort implementierte Algorithmus für die binäre Suche wurde bereits 1986 erstmalig beschrieben und in den folgenden Jahren als formal richtig bewiesen.

Zwanzig Jahre nach der Veröffentlichung und neun Jahre nach der Freigabe des ersten JDK gab es einen Bugreport, der zur Korrektur eines bis dahin unbemerkten Fehlers führte. Ursache war die Codezeile:

```
int mid = (low + high) / 2;
```

Wenn die Summe der beiden Werte `low` und `high` größer als `Integer.MAX_VALUE` ist, wird das Ergebnis negativ, was bei Indizes unsinnig ist.

Diese Fehlfunktion war deshalb solange irrelevant, weil sie erst bei Feldgrößen im Milliardenbereich zutage tritt. Durch die technische Entwicklung und mit dem Aufkommen von Big-Data wurde der Mangel sichtbar.

Zwei Möglichkeiten, dieses Problem zu umgehen, sind:

```
int mid = low + ((high - low) / 2);
int mid = (low + high)>>> 1;
```

Eine durch die Beispiele illustrierte Konsequenz ist, dass im Allgemeinen mathematisch gleichwertige Ausdrücke in einer Programmiersprache nicht notwendig ebenfalls gleichwertig sein müssen.

Im Gegensatz zu Genauigkeitsproblemen haben Wertebereichsgrenzen bisher nicht die nötige Aufmerksamkeit gefunden. So lernt jeder Anfänger zwar, dass Gleitkommazahlen besser nicht auf Gleichheit geprüft werden sollten, aber dass Teilausdrücke von Berechnungen den darstellbaren Wertebereich überschreiten können und dadurch falsche Ergebnisse liefern, ist selten ein Thema.

Bedenken Sie beim Entwickeln, dass Software oft viel länger genutzt wird als ursprünglich geplant. (Was zuweilen an den Schwierigkeiten liegt, die beim Entwickeln einer Ablösung auftreten.)

Prüfen Sie deshalb bei Erweiterungen immer, ob die eingesetzten Module noch korrekt arbeiten. Aus den genannten Gründen skaliert Software nicht mit der Größe von Daten oder Werten, sondern versagt beim Überschreiten bestimmter Grenzen.

- Erinnern Sie sich an die Wertebereichsgrenzen und behalten Sie diese im Auge, auch wenn das scheinbar (noch) nicht notwendig ist.
- Bei der Implementierung mathematischer Verfahren sollten Sie immer versuchen, einen möglichst großen Argumentbereich abzudecken, weil das der allgemeinen Erwartungshaltung entspricht.

  Bauen Sie Begrenzungen (Exceptions) ein, wenn die vollständige Umsetzung nicht möglich ist.

Keine Panik

Fehler im Code werden sich auf absehbare Zeit nicht vermeiden lassen. Akzeptieren Sie das und versuchen Sie Ihr Bestes, die Anzahl zu minimieren und die Folgen so gering wie möglich zu halten. Vergessen Sie dabei nicht, dass verständlicher Code ein wichtiger Schritt zur Fehlervermeidung ist.

Der Umgang mit den in diesem Kapitel beschriebenen Herausforderungen ist kein Grund, Abstriche an der Klarheit zuzulassen. Verwischen Sie nicht die Strukturen. Kapseln Sie Fehlerbehandlungen und trennen Sie diese auch optisch erkennbar vom normalen Verarbeitungsfluss.

Wie Sie dabei Exceptions wirkungsvoll einsetzen, wird im folgenden Kapitel 18 *Ausnahmen regeln* beschrieben.

Das Wichtigste in Kürze

- ✔ Fehler und Fehlfunktionen sind Teil der Realität und dürfen nicht ignoriert werden.
- ✔ Der Umgang mit Fehlern muss von Anfang an in die Konzeption einbezogen werden.
- ✔ Fehler müssen nach Art und Häufigkeit speziell behandelt werden.
- ✔ Neben den Eingabedaten ist es bisweilen angezeigt, auch die Ergebnisdaten einer Plausibilitätsprüfung zu unterziehen.
- ✔ Nichtbeachtung der Wertebereichsgrenzen von Zahltypen kann schwer bemerkbare und subtile Fehler verursachen.

Kapitel 18

Ausnahmen regeln – Exceptions

IN DIESEM KAPITEL

- Umgang mit Ausnahmesituationen
- Wichtige Eigenschaften von Exceptions
- Exceptions richtig werfen
- Exceptions sinnvoll fangen

In diesem Kapitel erfahren Sie, wie Exceptions (oder Ausnahmen) den Umgang mit Fehlersituationen vereinfachen können und worauf Sie dabei ganz besonders achten sollten.

Sinn und Zweck

Exceptions sind das Mittel der Wahl für den sauberen Umgang mit Fehlern. Ihre Verwendung als generelle Fehlerbehandlungsmethode in imperativen Programmiersprachen ist vergleichsweise jung. Daher gibt es bei der Anwendung immer noch viele Unklarheiten und Missverständnisse.

In der Regel können Sie nicht erkennen, welche Anweisung nach dem Werfen (`throw`) einer Exception als nächste ausgeführt wird. Das ist ein entscheidender Unterschied zu allen anderen Anweisungen, die ebenfalls den Kontrollfluss unterbrechen wie etwa `return`, `break` oder ein Methodenaufruf.

Das Werfen eine Exception ist vergleichbar mit einem Notruf aus der Wildnis: Der Rufer übergibt die Verantwortung an einen (unbekannten) Empfänger, der ihn hoffentlich aus seiner misslichen Lage befreit.

Exceptions können Ihnen helfen, Ihren Code übersichtlicher zu strukturieren, indem der normale Ablauf nicht übermäßig durch Anweisungen zur Fehlerbehandlung belastet wird. Das ist allerdings nicht immer einfach und erfordert Übung.

Vergessen Sie nie: Exceptions sollen Ihren Code übersichtlicher und leichter lesbar machen!

Welche Fehler sollen Exceptions auslösen?

Sie könnten auch fragen: Wann ist ein Fehler ein Fehler? Wie bei so vielem gibt es darauf keine einfache Antwort. Deshalb gehe ich dieser Frage in Kapitel 17 *Fehler passieren – Fehlerbehandlung* genauer nach.

Nehmen Sie zum Beispiel ein Dialogprogramm. Menschen machen häufig Fehler. Daher wäre es wahrscheinlich wenig sinnvoll, bei jedem Tippfehler eine Exception auszulösen. Tippfehler gehören in diesem Fall einfach zur erwarteten Eingabe.

Es gibt aber Ausnahmen. Beim Log-in können Sie, obwohl das formal nichts anderes ist, sehr wohl eine Exception werfen, wenn die Eingabe des Passworts nicht korrekt ist. Erstens passieren Fehler beim Einloggen nicht so häufig, und zweitens, und noch wichtiger, kann ein solcher Fehler auf einen Einbruchsversuch hinweisen. Dokumentation und nachfolgende Auswertung sind also notwendig.

Die Entscheidung, ob Sie eine Exception werfen oder eine andere Fehlerbehandlung vorsehen, sollten Sie daher immer auch unter inhaltlichen Gesichtspunkten treffen.

Versuchen Sie, die Abgrenzung mittels folgender Kriterien vorzunehmen:

- ✔ Vereinfacht es den Code, wenn Sie eine Exception verwenden?
- ✔ Liefert Ihnen die Exception nützliche Informationen zum Auswerten?
- ✔ Wie häufig wird die Exception voraussichtlich auftreten?

Eine Vielzahl von trivialen Exceptions erhöht die Gefahr, dass Sie die wirklich wichtigen übersehen.

Checked und Unchecked Exceptions

Java unterscheidet zwischen *checked* und *unchecked* Exceptions. Der Unterschied besteht darin, dass Sie checked Exceptions in der Signatur jeder Methode, aus der sie möglicherweise geworfen wird, deklarieren müssen. Unchecked Exceptions werden im Gegensatz dazu implizit und quasi unsichtbar weitergereicht.

Auf den ersten Blick erscheint es, als ob das explizite Benennen möglicher Exceptions eine gute Idee wäre. Genaueres Hinsehen offenbart jedoch gravierende Schwächen. Bei jedem Aufruf einer entsprechenden Methode müssen Sie die checked Exceptions behandeln oder durch Aufnahme in die Signatur der rufenden Methode weiterreichen. Der Code wird dadurch schnell unübersichtlich, und Sie laufen Gefahr, den eigentlichen Ablauf aus den Augen zu verlieren.

Außerdem verletzen checked Exceptions fast immer das Prinzip der Datenkapselung, weil sie den Aufrufer zwingen, sich mit Implementierungsdetails der aufgerufenen Methode zu befassen.

Kosten

Das Werfen und Fangen einer Exception ist eine relativ teure Operation, weil der auffangende `catch`-Block erst zur Laufzeit durch sequenzielles Durchmustern der Aufrufhierarchie ermittelt werden kann.

Außerdem muss die gesamte Aufrufhierarchie gespeichert werden, damit Sie diese später beispielweise als *Stacktrace* in einem Log-Eintrag protokollieren können.

Verwenden Sie nie Exceptions, um normale Programmlogik auszudrücken.

Exceptions sind teure Operationen, die Sie – wie der Name bereits sagt – nur für Ausnahmesituationen verwenden sollten.

Werfen von Exceptions

Wann immer ein Zustand eintritt, von dem aus nicht mehr sinnvoll weitergearbeitet werden kann, werfen Sie eine Exception, und zwar eine unchecked Exception. Damit stehen Sie sofort vor der Frage: Welche? Leider kann ich Ihnen darauf nicht mit einer einfachen Empfehlung antworten.

Die Antwort hängt unter anderem davon ab, ob Sie ein Modul oder eine Bibliothek, die an vielen Stellen verwendet werden soll, schreiben oder ein einzelnes Programm. Für eine Batch-Anwendung ergeben sich andere Anforderungen als bei Nutzerinteraktionen. Außerdem spielt die Größe des Projekts eine wichtige Rolle. Lassen Sie mich deshalb die möglichen Varianten diskutieren.

Generische Exceptions verwenden

Unter *generischen Exceptions* verstehe ich hier alle in den Basisbibliotheken enthaltenen Exception-Klassen. Die Wiederverwendung hat für Sie zwei große Vorteile:

- ✔ Sie sparen sich die sonst notwendige eigene Definition.
- ✔ Sie verringern die Abhängigkeiten zwischen ihren einzelnen Packages, die daraus entstehen würden, dass Sie dort, wo die Exception gefangen wird, die Definition importieren müssen.

Einen Nachteil will ich Ihnen allerdings nicht verschweigen: Sie verlieren auf diesem Weg die Möglichkeit, Exceptions anhand ihres Typs gezielt zu fangen.

Wenn Sie eine wiederverwendbare Komponente schreiben, sollten Sie generische Exceptions daher mit Vorsicht und nur für ganz allgemeine Fehlersituationen verwenden.

Um die richtige Verwendung komponenteneigener Exceptions zu illustrieren, greife ich auf die in Kapitel 13 *Kleine Schritte – saubere Methoden* verwendete HSSF-Bibliothek zurück.

In ihr wird eine `OldExcelFileFormatException` definiert. Dadurch erhalten Sie im Fehlerfall die Chance, automatisch – zum Beispiel durch eine Konvertierung – zu reagieren. Bei einer generischen Exception müsste die in diesem Fall notwendige Information fehlerträchtig aus der Beschreibung ermittelt werden.

Im Unterschied dazu kann auf eine nicht gefundene Datei sehr wohl generisch mittels einer `FileNotFoundException` reagiert werden.

Spezielle Exceptions definieren

Eigene Exceptions definieren Sie wie im folgenden Beispiel gezeigt als Unterklasse von `RuntimeException`. Ein geeigneter Typ für die Kontextinformation, im Beispiel `XContext` genannt, hängt stark von den konkreten Umständen ab.

```
public class NoData extends RuntimeException {
  private final XContext contextInfo; //passender Typ!!
  public NoData(String message, XContext contextInfo,
                    Throwable cause) {
    super(message, cause);
    this.contextInfo = contextInfo;
  }
  … // evtl. weitere Konstruktoren
  public XContext getContextInfo() {
    return contextInfo;
```

```
    }
}
```

Notwendige Felder

Versetzen Sie sich bei der Definition einer eigenen Exception-Klasse in die Lage des Empfängers. Für die Auswertung muss das Exception-Objekt mindestens Antworten auf die folgenden Fragen liefern:

- ✔ Welche Art Fehler ist aufgetreten?
- ✔ Wo ist der Fehler aufgetreten?
- ✔ In welchem Kontext ist der Fehler aufgetreten?
- ✔ Wurde der Fehler durch eine (andere) Exception verursacht?

Die Beschreibung des Fehlers (Punkt eins) wird nicht nur hier gebraucht und nachfolgend in einem eigenen Abschnitt behandelt.

Um Punkt zwei brauchen Sie sich nicht zu kümmern, weil das durch den automatisch erstellten Stacktrace bereits erledigt wird.

Schwieriger steht es um den Kontext (Punkt drei). Versuchen Sie, ausreichend viele Informationen mitzugeben, sodass die unmittelbare Fehlerquelle eingegrenzt werden kann. Das ist bisweilen nicht einfach.

Versuchen Sie, bei aufgetretenen Exceptions deren Ursache zunächst nur mithilfe der geloggten Daten zu finden. Wenn Ihnen das nicht gelingt und Sie doch im Debugger nachschauen müssen, zeigt Ihnen das, welche Kontextinformationen Sie noch ergänzen sollten.

Frühe Versionen des Java-Laufzeitsystems lieferten die `ClassNotFoundException` ohne weitere Informationen, das heißt ohne den Namen der nicht gefundenen Klasse. Aus dem Stacktrace war zwar ersichtlich, in welcher Klasse der

Fehler auftrat, aber außer in trivialen Fällen handelte es sich meist um eine mühselig zu findende transitive Abhängigkeit in irgendeiner Bibliothek.

Global betrachtet hat diese kleine – inzwischen lange behobene Nachlässigkeit – wahrscheinlich Tausende Entwicklertage Kosten verursacht. Genau so etwas sollten Sie zu vermeiden trachten.

Der Kontext ist darüber hinaus die geeignete Stelle, um Daten unterzubringen, anhand derer Sie später beim Behandeln der Exception auf unterschiedliche Maßnahmen verzweigen können.

Und nun abschließend noch zum Punkt vier. Wenn der Fehler durch eine andere Exception verursacht wurde, muss diese natürlich als `cause` weitergereicht werden.

Benennung von Exceptions

In Java ist es üblich, Exceptions immer auch so zu benennen: `NullPointerException`, `ArrayIndexOutOfBoundsException` ... Entscheiden Sie bitte selbst, ob das die Lesbarkeit fördert. Dass es sich um Exceptions handeln muss, können Sie aus der Syntax leicht erkennen.

Der Zusatz `Exception` liefert Ihnen keine echte Information, sondern verlängert nur unnötigerweise den Code. Allerdings stehen Sie auch hier wieder vor der Frage, den etwaigen Vorteil kürzerer Namen gegen die resultierende Inkonsistenz zu vorhandenen Bibliotheken abzuwägen.

Exceptionbeschreibung

Im Idealfall beschreibt bereits der Name einer Exception präzise die auslösende Bedingung. Leider ist das Leben aber nicht immer so einfach wie bei einem Null-Pointer.

Nehmen Sie beispielsweise eine Datenbankanbindung. Dabei können so viele verschiedene Probleme auftreten, dass es praktisch unmöglich ist, für jeden Einzelfall eine eigene Exception-Klasse zu definieren. Ein Teil der Beschreibung wandert deshalb in das oben als `message` bezeichnete Feld.

Wenn Sie generische Exceptions verwenden, bleibt Ihnen ohnehin nur diese Möglichkeit.

Message-Texte

Darüber, wie dieser Message-Text aussehen sollte, herrscht wenig Einigkeit. Meistens werden fest programmierte Texte verwendet, selbst innerhalb eines Projekts bisweilen sogar in verschiedenen Sprachen.

Das ist aus Sicht des Empfängers der Exception unbefriedigend und gerade noch akzeptabel, wenn Sie diese lediglich loggen und keine speziellere Fehlerbehandlung vornehmen. Problematisch wird es, wenn Sie anhand solcher Texte Meldungen erzeugen müssen, die den Anwendern, eventuell sogar in verschiedenen Sprachen, angezeigt werden sollen.

Message-Schlüssel

Wenn Sie an einem größeren Projekt arbeiten, für das eine ausgefeilte Exception-Behandlung notwendig ist, sollten Sie deshalb das folgende Vorgehen in Betracht ziehen:

- ✔ Definieren Sie eine Syntax für Exception-Beschreibungen, vorzugsweise angelehnt an Property-Namen, also etwa `device.handler.busy`.
- ✔ Benutzen Sie ausschließlich die so definierten Beschreibungsschlüssel.
- ✔ Implementieren Sie – am besten pro Komponente – einen Service, der zu jedem Exception-Schlüssel einen aussagekräftigen Text liefert.

  Das können Sie unter Benutzung der für die Lokalisierung verfügbaren Mittel mit relativ wenig Aufwand tun.
- ✔ Denken Sie daran, dass bei der Auswertung aus dem vorigen Schritt auch wieder Exceptions auftreten können, die Sie dann einfacher behandeln müssen, um nicht in einer endlosen Rekursion zu enden.

Auf die Umsetzung in lesbare Texte können Sie unter Umständen verzichten, wenn die Meldungen nur von wenigen Personen

ausgewertet werden müssen, für die der Schlüssel bereits ausreichend informativ ist.

Fangen von Exceptions

Fangen Sie Exceptions nur an Stellen, an denen Sie sinnvoll darauf reagieren können. Eine Exception loggen und dann weiterwerfen ist keine sinnvolle Aktion.

Funktionsblöcke bestimmen

Zweckmäßigerweise sollten Sie jede Anwendung bereits bei der Konzeption in Funktionsblöcke zerlegen, für die eine Fehlerbehandlung möglich und sinnvoll ist. Stellen Sie sich darauf ein, dass Sie diese Struktur öfter werden überarbeiten müssen.

Schematisch vereinfacht sieht ein solcher Funktionsblock so aus:

```
try {
  //  Code-Block
} catch (SomeException e) {
  if (canHandle(e)) {
          handle(e);
  } else {
          throw e;
  }
}
```

Selbstverständlich sind auch mehrere `catch`-Blöcke und Exception-Listen möglich. Weil das auf das prinzipielle Vorgehen keinen Einfluss hat, beschränke ich mich hier zugunsten der Übersichtlichkeit auf den einfachsten Fall.

Durch `try-catch` wird eine Art Transaktionsklammer um den Codeblock gelegt. Wenn der Block nicht korrekt durchlaufen wird, ist es Ihre Aufgabe, im `catch`-Teil wieder einen konsistenten Programmzustand herzustellen.

Dabei sollten Sie nur solche Exceptions fangen, für die eine Stabilisierung möglich ist. Der dargestellte Fall mit der

zusätzlichen `if`-Anweisung zeigt die Situation, in der Sie allein auf Basis des Exception-Typs die Behandelbarkeit noch nicht entscheiden können, weil Sie beispielsweise generische Exceptions verwenden.

Fachliche und technische Exceptions

Wahrscheinlich wird Ihnen bei der Zerlegung in Funktionsblöcke auffallen, dass Sie sich um unterschiedliche Arten von Exceptions kümmern müssen.

- **Fachliche Exceptions** werden durch Fehler verursacht, die auf der fachlichen Ebene behoben werden müssen.

  Ein typisches Beispiel dafür ist eine Rechnungsposition, für die noch kein Preis in die Stammdaten eingepflegt wurde. Die Behandlung müssen Sie in der Regel mit einem fachlich Verantwortlichen klären. Stellen Sie sich darauf ein, dass es diesbezüglich im Laufe der Zeit sehr wahrscheinlich (häufig) Änderungswünsche geben wird.
- **Technische Exceptions** entstehen als Folge technischer Probleme, als da sind Programmfehler, Gerätefehler, Versionskonflikte und so weiter.

  Die Reaktion auf technische Exceptions kann innerhalb des Projekts geklärt werden. Allenfalls werden Sie sich dabei mit den später für den Betrieb Zuständigen abstimmen müssen.

Diese Unterscheidung hilft Ihnen vor allem beim Festlegen der Funktionsblöcke und der jeweils angemessenen Fehlerreaktionen. Bringen Sie den Unterschied in der Namensgebung deutlich zum Ausdruck.

Definieren Sie für fachliche Exceptions in jedem Fall spezielle Klassen! Das erleichtert Ihnen die Behandlung in von der technischen Umsetzung abgetrennten `catch`-Teilen und verbessert so die Codelesbarkeit.

Verpacken von Exceptions

Checked Exceptions haben sich zwar als nicht so gelungene Konstruktion herausgestellt, da sie aber unglücklicherweise in vielen Basispaketen verwendet werden, können Sie ihnen kaum aus dem Weg gehen. Was also tun?

Ich empfehle Ihnen, wo immer möglich, die Umwandlung in eine unchecked Exception nach dem folgenden Muster:

```
Appendable appendable;
…
try {
  appendable.append(" ");
} catch (IOException e) {
  throw new RuntimeException("append.unexpected", e);
}
```

Sie können dieses Modell auch in Adapterklassen verwenden, um unchecked Exceptions aus fremden Komponenten in Ihr eigenes Standardformat zu konvertieren.

Loggen von Exceptions

Exceptions sagen Ihnen, dass etwas schiefgelaufen ist. Üblicherweise werden Sie deshalb in irgendeiner Form geloggt. In vielen Programmen finden Sie dafür ein einfaches Muster. In jedem `catch`-Teil werden alle gefangenen Exceptions geloggt, unabhängig davon, ob sie behandelt oder weitergeworfen werden.

Versetzen Sie sich auch hier wieder in die Rolle des Empfängers. Dem nützt es überhaupt nichts, wenn ein und dieselbe Exception mehrmals im Log, mit immer längerem Stacktrace, erscheint – ganz im Gegenteil.

Deshalb halten Sie sich an die folgenden beiden Regeln:

- ✔ Loggen Sie Exceptions nur dann, wenn Sie sie wirklich abschließend behandeln können.

Jede Exception soll – soweit möglich – nur einmal erscheinen.

- ✔ Überlegen Sie, welche Informationen für die Auswertung wichtig sind, und loggen Sie nur diese.

Bei einer Null-Pointer-Exception ist der Stacktrace unbedingt erforderlich, weil es sich dabei praktisch immer um einen Programmierfehler handelt. Im Gegensatz dazu sind Sie bei einer verloren gegangenen Internetverbindung wahrscheinlich eher an der betroffenen Adresse und am Grund des Abbruchs interessiert.

Ebenso interessieren bei fachlichen Exceptions in erster Linie Informationen, die für die fachliche Fehlerbehebung wichtig sind.

Angemessenheit

Dieses Kapitel hat Ihnen hoffentlich ein paar interessante Vorschläge präsentiert, sodass Sie sich vielleicht die Frage stellen: Und was soll ich jetzt ganz konkret machen?

Meine einfache Antwort lautet: Wählen Sie eine angemessene Lösung. Das ist leicht dahingesagt, deshalb folgt eine Liste von Fragen, die Ihnen helfen sollen, die richtige Balance zwischen Aufwand und Nutzen zu finden.

- ✔ **Für welche Nutzungsdauer ist die Anwendung vorgesehen?**

  Handelt es sich um eine kurzlebige Aufgabe wie eine Datenmigration oder eine dauerhafte Funktion? Bedenken Sie dabei auch die Erfahrung, dass provisorische Lösungen ein langes Leben haben können.

- ✔ **Soll die Anwendung interaktiv oder im Batch-Modus betrieben werden?**

  Bei interaktiven Anwendungen müssen Sie aus Exceptions eventuell Fehlermeldungen in verschiedenen Sprachen

generieren, auf die die Anwender reagieren sollen.

- ✔ **Handelt es sich um eine Komponente, die mehrfach verwendet werden soll oder könnte?**

  Wiederverwendung erhalten Sie nicht zum Nulltarif. Mögliche Exceptions müssen Sie als Teil der öffentlichen Schnittstelle sorgfältig definieren.

- ✔ **Wie umfangreich ist die zu entwickelnde Anwendung?**

  Je umfangreicher Ihre Anwendung ist, desto eher machen sich konsequent durchdachte Lösungen bezahlt, auch wenn sie aufwendiger umzusetzen sind.

- ✔ **Wie ausgeprägt sind die Anforderungen an die Fehlerbehandlung?**

  Wenn es akzeptabel ist, dass Ihre Anwendung nach einer Exception einfach abbricht, reicht informatives Logging. Ganz anders sieht das aus, wenn Ihre Anwender erwarten, dass das Programm trotz Fehlern im Betrieb möglichst stabil bleibt.

- ✔ **Wie wahrscheinlich ist das Auftreten von Exceptions im Betrieb?**

  In Anwendungen mit externen Kommunikationsbeziehungen müssen Sie beispielsweise mit viel mehr Exceptions rechnen als bei einer einfachen Verarbeitung.

- ✔ **Gibt es fachliche Exceptions?**

  Bei Aufgaben eher technischer Art ist das oft nicht der Fall. Aber verneinen Sie diese Frage nicht zu früh. Manchmal werden Sie im Laufe der Entwicklung doch noch den einen oder anderen Kandidaten für eine fachliche Exception finden.

Wenn Sie dies Fragen durchgegangen sind, sollten Sie grob einschätzen können, wie viel Aufwand vertretbar ist. Scheuen Sie sich nicht, von Zeit zu Zeit Ihre Einschätzung zu überprüfen und – falls erforderlich – zu revidieren.

Das Wichtigste in Kürze

- ✔ Ermitteln Sie die Charakteristika Ihres Projekts oder Programms und wählen Sie ein angemessenes Vorgehen für den Umgang mit Fehlersituationen aus.
- ✔ Besprechen Sie die ausgewählte Variante im Team und versuchen Sie, ein möglichst einheitliches Verständnis zu erreichen. Das ist deshalb wichtig, weil Sie Exceptions komponentenübergreifend verwenden.
- ✔ Strukturieren Sie Ihren Code in Funktionsblöcke mit jeweils zugeordneten Exception-Behandlungen.
- ✔ Unterscheiden Sie gegebenenfalls zwischen fachlichen und technischen Exceptions.
- ✔ Loggen Sie Exceptions erst dann, wenn Sie sie abschließend behandeln können.

  Berücksichtigen Sie beim Loggen, welche Informationen für die Auswertung wichtig sind. Gestalten Sie die Darstellung so, dass Art, Schwere und Dringlichkeit der aufgetretenen Exception leicht erkennbar sind.
- ✔ Setzen Sie Exceptions so ein, dass Ihr Code verständlicher wird und der normale Ablauf nicht in einem Dickicht von Fehlerbehandlungen verschwindet.

Kapitel 19

Immer weiter – neue Sprachmittel

IN DIESEM KAPITEL

- Spracherweiterungen bewerten
- Gefahren durch Annotationen
- Funktionale Ausdrucksmittel integrieren
- Sprachmittel gezielt auswählen

Programmiersprachen, nicht nur Java, werden kontinuierlich weiterentwickelt. Die bisher betrachteten Regeln für sauberen Code haben sich im Wesentlichen auf lange bekannte Sprachelemente bezogen.

In diesem Kapitel werde ich Ihnen ein paar Hinweise geben, wie Sie an neue Sprachbestandteile herangehen sollten und diese Tipps dann für Annotationen, Lambda-Ausdrücke und Streams anwenden.

Wie beurteilen?

Die Maßstäbe für die Beurteilung eines neuen Elements leiten sich natürlich aus den Zielen sauberen Codes ab:

- ✔ Macht es den Code leichter verständlich?
- ✔ Wie stark erhöht es die kognitive Komplexität?
- ✔ Unterstützt es effektive Tests?
- ✔ Erleichtert es die Programmierung?

Besonders die Frage nach der Erhöhung der kognitiven Komplexität ist bei Erweiterungen wichtig. Neue Konstrukte machen eine Programmiersprache automatisch komplexer. Sie sind deshalb nur gerechtfertigt, wenn dadurch der Code weniger komplex wird, das heißt, wenn sie helfen, Kompliziertes einfacher zu auszudrücken.

Gleichzeitig muss bedacht werden, wie ein neu hinzukommendes Konzept mit den bereits vorhandenen harmoniert. Je umfangreicher eine Sprache wird, desto häufiger wird es dabei zu Überschneidungen kommen. Diese Überschneidungen machen das Entwickeln komplizierter, weil Sie Ihnen zusätzliche Entscheidungen abverlangen.

Es war ja gerade der ursprüngliche Charme von Java, dass die möglichen Varianten – ohne Beeinträchtigung der algorithmischen Ausdruckkraft – stark beschränkt waren. Im Gegensatz zu C++ konnte kein Java-Programmierer einen so eigenen Stil entwickeln, dass dessen Verständnis längere Einarbeitung erfordert. Ich sehe die Gefahr, dass dieser Vorteil – wenn auch recht langsam – schwindet.

Jede Erweiterung macht es gleichzeitig einfacher und schwieriger, gut verständlichen Code zu schreiben. Einfacher, weil Sie zielgenauere Mittel zur Verfügung haben, und schwieriger, weil Sie diese verantwortungsvoll einsetzen müssen.

Sie sind nun dafür verantwortlich, die Vielfalt der Konzepte zu begrenzen und darauf zu achten, dass an den Punkten, an denen Sie die freie Auswahl haben, immer konsistent nach den gleichen Kriterien entschieden wird.

Andernfalls verwirren Sie einen Leser unnötig, weil der sich schnell die Frage stellt: Warum wird das hier anders gelöst als dort? Das ist Denkaufwand, der besser in das Verständnis Ihrer Intentionen gesteckt werden sollte.

Die Janusköpfigkeit mancher Spracherweiterungen lässt sich gut am Beispiel der *Typinferenz für lokale Variable*, die

mit Java 10 eingeführt wurde, zeigen.

Der sofort erkennbare Vorteil ergibt sich aus dem Wegfall redundanten Codes. Statt

```
HashMap<Number, Product> productsByNumber = new
HashMap<>();
```

können Sie jetzt einfach schreiben

```
var productsByNumber = new HashMap<Number, Product>();
```

Das ist zweifellos eine begrüßenswerte Vereinfachung. Sie müssen weniger schreiben, und die Lesbarkeit gewinnt ebenfalls. Genau in dieser Weise sollten Sie derartige Konstrukte auch verwenden.

Wie steht es aber mit Anwendungen der Art

```
var wert = weitEntferntDefinierteMethode();
```

In diesem Fall ist der Typ der vereinbarten Variablen für einen Leser nicht mehr sofort offensichtlich. Das kann für das Verständnis problematisch sein, wenn Sie keine IDE zur Hand haben, die für Sie den konkreten Typ ermittelt.

Außerdem müssen Sie bei Deklarationen jetzt unterscheiden, ob ein Feld, ein Parameter oder eine lokale Variable vorliegt. Nur im letzteren Fall können Sie die Typinferenz nutzen.

Dabei sind jedoch noch einige Besonderheiten zu beachten. Beispielweise ist

```
var text; text = "TEXT";
```

nicht zulässig. Das Gleiche gilt für

```
var nullValue = null;
```

Allerdings dürfen Sie Folgendes schreiben:

```
var text = (String)null;
```

```
final var nullInt = (Integer)null;
```

```
var var = 1;    // Das sollte niemand so machen!
```

Die letzte Zeile ist deshalb erlaubt, weil `var` in Java kein Symbol, sondern lediglich eine reservierte Bezeichnung ist.

Sie sehen daran, dass in vielen Situationen eine bequemere Schreibweise zusätzliche Regeln erfordert, die zwar leicht zu begründen und einzusehen sind, aber die Sprache insgesamt doch unübersichtlicher machen.

Besonders kritisch ist es, wenn Sie älteren Code ändern müssen. Dann kann die Versuchung groß sein, Sprachmittel einzusetzen, die zur Entstehungszeit des Codes noch nicht verfügbar waren, aber jetzt vieles vereinfachen würden.

Solche Umstellungen bergen jedoch die Gefahr, dass sie die Verständlichkeit verschlechtern, wenn sie nicht vollständig und konsistent erfolgen. Beherzigen Sie daher die folgende Warnung!

Prüfen Sie bei der Arbeit an Bestandscode sorgfältig, ob der Einsatz neuer Sprachkonstrukte sinnvoll ist. Wenn Sie nicht eine Klasse – oder besser gleich eine Komponente – komplett überarbeiten, ist es meist besser, den vorliegenden Programmierstil beizubehalten.

Denken Sie daran, dass es sich sehr oft nicht um die letzte Veränderung handeln wird und Sie einem zukünftig daran arbeitenden Entwickler das Leben nicht unnötig schwer machen sollten.

Annotationen

Aus Sicht der Programmiersprache sind Annotationen zunächst nicht viel mehr als kleine Informationspakete, die je nach Art vom Compiler beim Laden oder zur Laufzeit ausgewertet werden können.

Ob und wie sie ein Programm beeinflussen, ist durch Konventionen oder Verabredungen – also nur semantisch –

festgelegt.

Funktion

Es gibt viele verschiedene Möglichkeiten, Annotationen zu nutzen. Das reicht von einfachen Markierungen, beispielsweise ermöglicht es `@Override` dem Compiler zu prüfen, ob wirklich eine Methode überschrieben wird, bis zu Werkzeugen, die ganze Methoden und Klassen generieren können.

Daran sehen Sie bereits den problematischsten Punkt für die Lesbarkeit: Die äußere Form – die Syntax – lässt keinerlei Schluss auf die Bedeutung zu. Letztere können Sie allein aus dem Namen erschließen, was nichts anderes heißt, als dass Sie die vorgefundene Annotation kennen müssen. Aber selbst dann sind Missverständnisse nicht völlig ausgeschlossen.

Annotationen sind aus Java-Sicht eine spezielle Form von Interfaces. Es können also zum einen gleichnamige Annotationen in verschiedenen Packages definiert werden und zum anderen für eine Definition verschiedene Implementierungen vorliegen. Wenn diese Fälle auch glücklicherweise nicht häufig sind, muss man sich ihrer doch bewusst sein.

Im Folgenden gehe ich kurz auf die verschiedenen Anwendungsarten und die damit verbundenen Problemfelder ein.

Anwendungsarten

Annotationen können auf ganz verschiedene Weise genutzt werden. Es ist sogar denkbar, dass es zukünftige Interpretationen geben wird, die zur Zeit des Schreibens noch völlig unbekannt sind.

Compilerhinweise

Annotationen wie `@Override`, `@SuppressWarnings` oder `@FunctionalInterface` sind reine Hinweise an den Compiler, die die Anwendung bestimmter Prüfungen steuern, aber letztlich keinen Einfluss auf den generierten Bytecode haben. Sie sind Bestandteil der Programmiersprache, und ihre Bedeutung ist

mithin klar. Für das Lesen und Verstehen des Codes sind sie nur teilweise nützlich.

Der Hinweis auf ein funktionales Interface bewahrt Sie eventuell vor dem unzulässigen irrtümlichen Hinzufügen einer Methode. Das Unterdrücken von Warnungen kann hingegen folgenlos überlesen werden.

Konfiguration

Bei dieser Form der Verwendung werden Angaben, die sich auf die Ausführungsumgebung des Programms beziehen, in den Code eingestreut. Häufig anzutreffende Beispiele beziehen sich auf die Persistenz oder die Serialisierung mittels XML oder JSON.

Der unmittelbare Vorteil von Annotationen gegenüber anderen Formen der Konfiguration liegt darin, dass die explizite Spezifikation des jeweils betroffenen Elements, wie zum Beispiel der vollständige Name einer Entitätenklasse, in einer zusätzlichen Datei entfällt, weil die Annotation im Code der bewussten Klasse steht.

Zudem kann durch den Compiler überprüft werden, ob Parameternamen richtig geschrieben und erforderliche Parameter angegeben wurden. Das macht es leichter, Konfiguration und Code konsistent zueinander zu halten.

Als nachteilig erweist es sich allerdings, dass Konfigurationsänderungen damit faktisch Programmcode-Änderungen mit allen daraus folgenden Konsequenzen sind.

Codegenerierung

Während die beiden vorigen Anwendungen den durch den Compiler generierten Bytecode nicht beeinflussen, ist das bei der Codegenerierung anders. Über eine offizielle Schnittstelle der Compiler-API können Prozessoren angeschlossen werden, die in den Übersetzungsprozess neu generierten Code einschleusen. Sie erhalten dann Bytecode, für den – zumindest teilweise – kein expliziter Quellcode vorliegt.

Beispielsweise können Setter- und Getter-Methoden automatisch ergänzt werden. Das hat den großen Vorteil, dass Sie beim Schreiben entlastet werden. Größtenteils schematischer *Boilerplate*-Code kann entfallen.

Dem steht für das Verständnis und die Weiterentwicklung allerdings gegenüber, dass derartige Annotationsprozessoren durch die tiefe Einbindung implizit zu Teilen des Projekts werden.

Laufzeitmodifikationen

Schließlich können Annotationen beim Laden oder gar erst zur Laufzeit ausgewertet werden. Das erlaubt beispielsweise auf elegante Weise dynamische Anpassungen für unterschiedliche Ausführungsumgebungen.

Andererseits hängt Ihre Software dann nicht mehr nur von den beim Bauen benutzen Werkzeugen ab, sondern in erheblichem Umfang von der speziellen Laufzeitumgebung.

Zudem gibt es keine Hinweise, wenn eine Annotation nicht, wie vom Entwickler geplant, erkannt wird. Gerade dieses Ignorieren kann zu sehr schwer zu lokalisierenden Fehlern führen.

Ein typischer Fall dieser Art der Modifikation ist die *Context and Dependency Injection* (CDI). Leicht vereinfacht versteht man darunter, dass ein Feld in der Form

```
@Inject private Context context;
```

deklariert wird. Eine explizite Wertzuweisung wird nicht gebraucht, weil diese durch die Ausführungsumgebung beim Initialisieren erfolgt.

Übliche Bedingung ist dabei, dass der Typ `Context` ein Interface ist und dass es unter allen ladbaren genau eine Klasse gibt, die dieses Interface implementiert. Durch eine später möglicherweise in einem ganz anderen Modul ergänzte zweite Implementation kann damit völlig beiläufig ein Laufzeitfehler verursacht werden.

Risiken minimieren

Annotationen können die Entwicklung von Software spürbar vereinfachen. Allerdings entsteht durch sie eine zweite Dimension für Schnittstellen, und für diese neue Dimension existieren kaum formal prüfbare Regeln.

Die semantischen Abhängigkeiten sind schwerer erkennbar und Fehler oft erst zur Laufzeit zu bemerken.

- Verwenden Sie Annotationen zurückhaltend und mit Bedacht. Behalten Sie dabei stets das Verhältnis von Nutzen und Risiko im Auge.
- Bedenken Sie die durch Annotationen entstehenden Abhängigkeiten. Beschränken Sie sich auf etablierte und stabil unterstützte Annotationen.
- Weil durch Annotationen viele Dinge nicht mehr explizit aus dem Code ersichtlich sind, wird oft zusätzliche Dokumentation mit all ihren bekannten Problemen nicht zu vermeiden sein.

Lambda-Ausdrücke

Mit der Version 8 hat die *funktionale Programmierung* in Form der *Lambda-Ausdrücke* – das sind anonyme Klassen, die funktionale Interfaces implementieren in einer stark verkürzten Schreibweise – auch in Java Einzug gehalten. Die neuen Ausdrucksmittel können Ihnen an vielen Stellen die Arbeit erleichtern, erfordern aber ein gewisses Umdenken.

Während imperativer Code im Wesentlichen als schrittweise Veränderung eines Zustands interpretiert werden kann, steht bei funktionalem Code die Anwendung von Funktionen (im mathematischen Sinn) auf Werte, die ihrerseits Werte als Ergebnisse liefern, im Zentrum.

Diese neue Sicht müssen Sie als Ausgangspunkt für den Einsatz von Lambda-Ausdrücken einnehmen und dann versuchen, so

klaren Code zu schreiben, dass sich diese Sicht dem Leser erschließt. Das ist nicht einfach, und es gilt, einigen Klippen auszuweichen.

Klippen

Um Klippen erfolgreich umschiffen zu können, müssen sie diese kennen. Die Gefahr liegt dabei oft unter einer unschuldig aussehenden Oberfläche.

Syntax

Die erste Klippe für eine gut lesbare Verwendung von Lambda-Ausdrücken stellt bereits die Syntax dar. Schauen Sie sich den folgenden Code an und bewerten Sie selbst, wie unauffällig das Pfeilsymbol `->` in einem größeren Codeblock ist.

Funktionale Programmierung

Unter funktionaler Programmierung wird ein Programmiermodell verstanden, das sich stark am mathematischen Begriff der Funktion orientiert. Eine Berechnung ist dabei die Ausführung einer Funktion, deren Ergebnis ausschließlich von den Argumenten abhängt.

Begriffe wie Zustand oder der veränderliche Inhalt einer Speicherzelle existieren in diesem Modell nicht. Reine Funktionen haben daher keine Seiteneffekte, was das Verständnis fördert. Außerdem erleichtert dieses Konzept formale Korrektheitsbeweise und automatische Parallelisierung.

Der größere Abstand zwischen dem abstrakten Ausführungsmodell und dessen konkreter Realisierung bereitet in der Praxis jedoch Probleme. Im Allgemeinen sind deshalb pragmatische Abweichungen von der »reinen Lehre« notwendig, beispielsweise für Ein- und Ausgabeoperationen.

Außerdem werden für wichtige, aber umständlich zu implementierende Operationen vordefinierte Standardfunktionen angeboten. Funktionale Programme sind häufig weniger effizient, weil Sie beispielsweise keine Schleifen kennen und stattdessen Rekursion verwenden.

Ein nicht zu unterschätzendes Problem stellt auch die für viele Entwickler ungewohnte Art der Lösungsbeschreibung dar.

```
JButton taste= new JButton("Taste drücken");
taste.addActionListener(e -> {
```

```
    eingabe.setText("Hier Text eingeben");
    ergebnis.setText(eingabe.getText());
  });
```

Ein Lambda-Ausdruck ist leider nichts, was einem sofort ins Auge springt. Die Kürze, die beim Schreiben positiv ist, erschwert das Lesen.

Vereinfachung und Typerschließung

Auch die Vereinfachungsregeln, die Ihnen möglichst kurze Formulierungen erlauben, erhöhen den kognitiven Aufwand beim Lesen. Folgende sechs Varianten einer Inkrementierungsfunktion können völlig gleichwertig sein:

```
(int x) -> {return x+1;}
(int x) -> x+1
(x) -> {return x+1;}
(x) -> x+1
x -> {return x+1;}
x -> x+1
```

Ich habe bewusst »können« geschrieben, weil der Compiler weggelassene Typinformationen zu ermitteln versucht und dadurch der konkrete Typ gegebenenfalls aus der Umgebung »erschlossen« wird.

Letzteres ist ebenfalls sehr praktisch beim Entwickeln und beim Betrachten dann kein besonderes Problem, wenn Sie eine IDE benutzen, die Ihnen die Schlussfolgerungen des Compilers sichtbar macht. Das Lesen ohne derartige Hilfen kann allerdings schwierig werden.

Effektiv konstant

Wie innere Klassen können Lambdas auf Variablen ihrer Umgebung zugreifen, wenn deren Werte konstant sind. Auch dabei hat man versucht, den Entwicklern Schreibarbeit zu ersparen, und dazu die Eigenschaft *effective final* eingeführt. Diese ist gegeben, wenn Variablen nur einmalig, und zwar sofort bei ihrer Deklaration ein Wert zugewiesen wird. Die Angabe `final`, wie sie bei expliziter Klassenschreibweise erforderlich ist,

kann dann entfallen. Das ist eine weitere implizite Festlegung, die beim Lesen zu beachten ist.

Nicht alles geht

Lambdas können an vielen Stellen helfen, die explizite Definition anonymer innerer Klassen zu vermeiden, aber nicht immer. Nur wenn das zu implementierende Interface ein funktionales ist, also genau eine abstrakte Methode hat, ist dieser Weg möglich.

Das kann zu der unschönen Situation führen, dass Sie wie im obigen Beispiel einen `ActionListener` als Lambda-Ausdruck schreiben können, aber für einen vielleicht gleichfalls gebrauchten `KeyListener`, der drei abstrakte Methoden umfasst, die überkommene Schreibweise zwingend ist.

Ich empfehle Ihnen in diesem Fall, sich an der Konsistenz zu orientieren: Wenn der Lambda-Ausdruck eine Ausnahme darstellte, sollten Sie auf ihn verzichten. Umgekehrt ist eine anonyme Klasse als Ausnahme akzeptabel. Wo die Grenze zwischen beiden Fällen liegt, müssen Sie entscheiden.

So vielleicht

Lambdas sind vorrangig mit dem Ziel eingeführt worden, funktionale Schreibweisen unterstützen zu können. Ihre Verwendung erleichtert in vielen Fällen die Entwicklung und verkürzt den entstehenden Code.

Es ist aber voreilig, wenn Sie daraus schließen, dass funktionaler Code immer besser lesbar und leichter verständlich ist. Ohne bewusste Gestaltung trifft eher das Gegenteil zu, weil der Anteil der erschlossenen, impliziten Informationen größer ist.

Es ist also Ihre Aufgabe, funktionalen Code verständlich zu gestalten. Selbstverständlich gebe ich Ihnen nun ein paar Anregungen, wie das gelingen könnte.

Methoden statt Anweisungen

Den weiter oben beispielhaft gezeigten `ActionListener` sollten Sie nicht als Vorbild nehmen. Besser ist es, den Anweisungsteil des

Lambda-Ausdrucks durch einen Methodenaufruf zu ersetzen:

```
void tasteGedrueckt(ActionEvent e) {
  eingabe.setText("Hier Text eingeben");
  ergebnis.setText(eingabe.getText());
}
…
taste.addActionListener(e -> tasteGedrueckt(e));
```

Das reduziert, wie Sie erkennen können, das enorme Ungleichgewicht zwischen linker und rechter Seite des Pfeil-Operators und ist durch den zusätzlichen sprechenden Methodennamen auch noch instruktiver.


Halten Sie Ihre Lambda-Ausdrücke so einfach wie möglich und nutzen Sie dazu Methoden mit aussagekräftigen Namen.

Benennen

Lambda-Ausdrücke müssen nicht anonym bleiben, in vielen Fällen sollten sie das auch nicht. Vergessen Sie nie, dass erklärende Namen eine Kerneigenschaft sauberen Codes sind. Geben Sie Ihren Lambdas Bezeichnungen:

```
Predicate<String> isLongWord = word -> word.length()> 15;
```

Wenn Sie das gut machen, helfen Sie dem Leser gleich zweifach:

- ✔ An der Stelle der Definition erschließt sich die Aufgabe.
- ✔ An der Stelle der Anwendung wird der Zweck sofort erkennbar und muss nicht aus dem Ausdruck ermittelt werden.


Benennen Sie Ihre Lambda-Ausdrücke! Das erfordert zwar etwas mehr Schreibarbeit, aber dieser zusätzliche Aufwand ist unbedeutend im Vergleich zu der gewonnenen Klarheit.

Streams

Das ebenfalls mit Java 8 eingeführte Konzept der *Streams* dient wie die gerade besprochenen Lambda-Ausdrücke der Unterstützung des funktionalen Codeschreibstils.

Die Idee

Funktionale Programmierung kennt keine Iteration. Mathematische Funktionen benutzen stattdessen rekursive Strukturen, die sich aber meist nicht sonderlich effizient auf realer Hardware umsetzen lassen.

Einen Ausweg aus dieser misslichen Lage bieten die Streams. Ein Stream ist eine Folge von Elementen, auf die Funktionen angewendet werden können.

Es gibt drei Arten von Stream-Operationen:

- *Erzeugungsoperationen* erzeugen einen Stream.
- *Zwischenoperationen* erzeugen aus einem Eingabestream einen Ausgabestream. Sie werden immer erst dann aktiv, wenn auf der Ausgabeseite das nächste Element angefordert wird.
- *Beendigungsoperationen* erzeugen das Ergebnis und steuern durch die Anforderung von Elementen gewissermaßen die gesamte Verarbeitungskette.

Jeder Stream muss genau eine Erzeugungs- und eine Beendigungsoperation haben. Dazwischen können beliebig viele – auch keine – Zwischenoperationen liegen. Für die Operationen wird Ihnen eine Reihe von vordefinierten Methoden angeboten, die zum Teil durch Lambda-Ausdrücke genauer spezifiziert werden müssen.

Die obige Beschreibung ist nicht vollständig. Sie können die Elemente eines Streams nicht nur in definierter oder undefinierter Folge, sondern auch parallel bearbeiten lassen. Besonders der letzte Punkt ermöglicht erhebliche Codevereinfachungen.

Anwendung

Die Verkettung der Zwischenoperationen gestattet es Ihnen, Verarbeitungsfolgen so zu formulieren, dass sie leicht verständlich sind. Der folgende Code extrahiert aus einer Log-Datei die ersten 100 Fehlermeldungen, die einer durch ihren Namen spezifizierten Klasse zuzuordnen sind.

```
List<String> errorsInClass =
    Files.lines(inputFile.toPath())
            .filter(line -> line.contains("ERROR"))
            .filter(line -> line.contains("de.dummies.Name"))
            .limit(100)
            .collect(Collectors.toList());
```

Wenn Sie die Lambda-Ausdrücke, wie empfohlen, passend benennen, wird das Ganze noch verständlicher:

```
    …
            .filter(lineIsErrorMessage)
            .filter(messageContainsClassnameName)
    …
```

Es sind genau solche schrittweisen Verarbeitungen von allen Objekten einer Menge, die durch Stream-Anweisungen gut lesbar und modifizierbar geschrieben werden können.

Verwenden Sie die Stream-Schreibweise immer dann, wenn mehrere Elemente einer Folge von Verarbeitungsschritten unterworfen werden müssen.

Versuchen Sie dabei, die Lambdas so zu benennen, dass die Aufrufkette als Folge von problemspezifischen Anweisungen gelesen werden kann.

Aber Vorsicht

Der beim funktionalen Programmieren anzutreffende größere Abstand zwischen Beschreibung und tatsächlicher Ausführung

macht sich vor allem bei Streams bemerkbar. Deshalb sollten Sie das Folgende wissen und im Auge behalten.

Ausführungsfolge

Das naive Ausführungsmodell eines Streams lässt ein Element nach dem anderen durch die definierten Zwischenoperationen wandern, wobei die Elemente modifiziert, durch neue ersetzt oder entfernt werden können.

Oft ist das so, aber nicht immer. Sehen Sie sich das folgende Beispiel, dessen Sinnhaftigkeit hier nicht von Bedeutung ist, an:

```
IntStream.iterate( 0, i -> i + 1 )
  .map(i->i+i)
  .forEach(System.out::println);
```

Erzeugt wird eine nicht endende Ausgabe von unendlich vielen geraden Zahlen. Der Punkt ist, dass das beobachtbare Verhalten des Streams genau dem beschriebenen Modell entspricht. Sie können diesen Code stundenlang auf Ihrem Computer laufen lassen.

Nun nehmen Sie eine kleine Änderung vor:

```
IntStream.iterate( 0, i -> i + 1 )
  .map(i->i+i)
  .sorted()
  .forEach(System.out::println);
```

Der Effekt ist deutlich: `OutOfMemoryError: Java heap space`. Was ist passiert? Mit `sorted()` wurde eine Zwischenoperation eingefügt, die nicht dem einfachen Ausführungsmodell entspricht. Das ist leicht zu verstehen, denn um eine Menge sortieren zu können, werden alle Elemente benötigt.

Die Sortieroperation braucht daher immer einen Zwischenspeicher, der alle verfügbaren Eingangselemente aufnehmen kann. Wenn Sie in nicht funktionalem Code Daten sortieren wollten, haben Sie den Umfang der zu sortierenden Daten wahrscheinlich bedacht, weil das ein wichtiger Faktor für

die Auswahl eines Sortierverfahrens war und Sie den benötigten Platz explizit bereitstellen mussten.

Die Einfachheit, mit der Operationen wie `sorted()` oder `distinct()` in Stream-Verarbeitungen eingefügt werden können, birgt die Gefahr, dass schwerwiegende Folgen übersehen werden.

Aufwand

Auch wenn Streams nur die Schleifenanweisungen zu ersetzen scheinen – ganz so einfach ist die Sache nicht. Für die Erzeugungs- und die Zwischenoperationen wird zunächst eine Objektstruktur aufgebaut, deren Methoden sich, veranlasst durch die Beendigungsoperation, wechselseitig aufrufen.

Daraus können Sie unmittelbar schließen, dass Streams einen höheren Initialisierungsaufwand verursachen als einfache Schleifen, was vor allem bei wenigen Elementen ins Gewicht fallen kann.

Eine zweite Konsequenz, die nicht so direkt auf der Hand liegt, ist, dass Stream-Anweisungen für einen Just-In-Time-Compiler (JIT) erheblich schwieriger zu optimieren sind.

Beides dürfte in den allermeisten Fällen unerheblich sein, sollte von Ihnen aber bedacht werden, wenn Probleme auftreten.

Vergessen Sie nie, dass die funktionalen Ausdrucksmittel in der Programmiersprache ein höheres Abstraktionsniveau verkörpern. Die Abbildung auf die operationale Ebene ist weniger direkt. Das kann unter Umständen auch das schrittweise Debuggen schwieriger machen.

Fazit

Auch aus dieser unvollständigen Betrachtung lassen sich schon Schlussfolgerungen ziehen, die Ihnen helfen können, neue

Sprachmittel – die es immer wieder geben wird – gewinnbringend in sauberen Code einzubauen.

Spezialisierung

> Jedes Werkzeug erhält nämlich dadurch seine Vollendung, dass es nicht mehreren, sondern nur einem Zwecke dient. (Aristoteles)

Die Weiterentwicklung von Programmiersprachen dient nicht der Erhöhung ihrer Ausdruckskraft – die ist im Allgemeinen von Anfang an vollständig. Es geht vielmehr darum, für bestimmte Aufgabenstellungen bessere oder einfachere Darstellungen zu ermöglichen. Dessen sollten Sie sich bei jeder Erweiterung bewusst sein.

Die neuen Mittel ersetzen die alten nur in einem Teil der Fälle. Nur dort bieten sie wirklich Vorteile. Ihre Aufgabe besteht deshalb zunächst vor allem darin, herauszufinden, welcher Teil das ist.

Die Sprache wird auf diese Art schrittweise um spezielle Werkzeuge bereichert, die nicht mehr den Anspruch erheben, für alle Anwendungsfälle wirklich geeignet zu sein. Das hat Vor- und Nachteile.

Zur Illustration können Sie an eine Küche oder Werkstatt denken. Mit einer soliden Grundausstattung sind Sie jeder Aufgabe gewachsen.

Allerdings kann es passieren, dass Sie mit der Zeit bemerken, dass für eine häufig vorkommende Aufgabe ein spezielles Werkzeug existiert. Damit lässt sich diese Aufgabe leichter und schneller erledigen.

Gleichzeitig entsteht damit für Sie ein neues Problem. Es wird nämlich Situationen geben, wo Sie entscheiden müssen, ob das Spezialwerkzeug wirklich die beste Wahl ist. Wenn sich derartige Spezialwerkzeuge ansammeln, kann noch die

Frage dazukommen, welches der vorhandenen denn nun am besten passt.

Der Vorteil eines speziell angepassten Werkzeugs ist es, Ihnen in geeigneten Konstellationen die Arbeit zu erleichtern. Der Nachteil ist die steigende Komplexität.

Sie müssen für jedes neue Werkzeug den richtigen Umgang und die Grenzen seiner Anwendbarkeit erlernen und diese Kenntnisse aktuell halten. Außerdem ist es sehr wahrscheinlich, dass mit zunehmender Anzahl immer mehr Überschneidungen auftreten, sodass Sie zusätzliche Abwägungen vornehmen müssen.


Setzen Sie neue Sprachmittel nicht unbedacht ein. Prüfen Sie vorher stets, ob dadurch wirklich ein Sie bedrängendes Problem gelöst wird.

Beschränkung

In der Beschränkung zeigt sich erst der Meister. (Goethe)

Sie haben einen umfangreichen Werkzeugkasten zur Verfügung. Doch niemand verpflichtet Sie, immer alles zu benutzen. Ganz im Gegenteil. Bemühen Sie sich, die für die anstehende Aufgabe am besten geeigneten Werkzeuge herauszufinden und sich dann konsequent auf diese zu beschränken.

Das macht es nicht nur Ihnen und Ihrem Team einfacher. Weniger Werkzeuge heißt auch weniger Konzepte und damit weniger Komplexität. Dieser Vorteil vermindert die Wahrscheinlichkeit von Fehlern und sorgt für besser verständlichen Code.

Verstehen Sie mich aber bitte nicht falsch. Die Einschränkung ist kein Selbstzweck. Verwenden Sie alles, was Sie wirklich brauchen. Wenn das allerdings unübersichtlich viel sein sollte, stellt sich die Frage, ob nicht ein Entwurfsfehler vorliegt und die anstehende Aufgabe weiter zerlegt werden sollte.

Wenn Sie sauberen Code produzieren wollen, müssen Sie darauf achten, dass nicht nur die Funktion Ihrer Komponenten einfach und klar ist, sondern auch bei der Umsetzung nur möglichst wenige Konzepte konsequent benutzt werden.

Das Wichtigste in Kürze

- ✔ Neue Sprachmittel bringen Verbesserungen, machen die Sprache insgesamt jedoch komplexer.
- ✔ Annotationen vergrößern die Abhängigkeit von Fremdsoftware, ohne dass das immer sofort erkennbar ist.
- ✔ Funktionale Ausdrücke definieren eine neue, höhere Abstraktionsebene in der Sprache.
- ✔ Sauberer Code sollte sich jeweils mit einem Teil der verfügbaren Sprachmittel begnügen.

Teil IV

Wege zum Ziel

IN DIESEM TEIL …

- ✔ Erfahren Sie, wie man Code Reviews organisiert und durchführt
- ✔ Lernen Sie Fehler als Erkenntnisquelle schätzen
- ✔ Verbessern Sie Code durch Refactoring

Kapitel 20

Miteinander lernen – Code Reviews

IN DIESEM KAPITEL

Ziele von Code Reviews

Voraussetzungen

Durchführung

Checklisten

Sie kennen jetzt wichtige Grundsätze für das Schreiben von Clean Code. In diesem Kapitel geht es darum, wie Sie das Gelernte effektiv in die tägliche Arbeit einbringen können. Sie erfahren, wie Code Reviews Ihnen und Ihren Kollegen helfen, den Code besser zu verstehen und die Codequalität zu erhöhen.

Der Begriff *Code Review* wird in unterschiedlichen Ausprägungen benutzt. Einige Autoren verstehen darunter, jede Form Code zu analysieren, zum Beispiel auch *Pair Programming* oder die Verwendung von Analysewerkzeugen. Hier verwende ich die folgende Definition:

Ein Code Review ist die organisierte gemeinsame Begutachtung von Code mit dem Ziel, in Ihrem Team oder Projekt eine Clean-Code-Kultur zu entwickeln.

Diese bewusste Einschränkung sollten Sie beim Lesen des vorliegenden Abschnitts nicht unbeachtet lassen. Denn das ist keine Anleitung dafür, wie Sie in großen Projekten solche Reviews administrativ einführen sollten. Mir geht es mehr darum,

wie Sie aus Ihrem Team heraus, sozusagen »von unten«, ohne übermäßigen Aufwand diese Art des voneinander Lernens nutzen können, um Clean Code zum festen Bestandteil Ihrer Arbeitsweise werden zu lassen.

Selbstverständlich gibt es noch weitere Arten, die individuellen handwerklichen Programmierfertigkeiten zu verbessern. Recht beliebt sind beispielsweise *Code-Katas* und *Coding-Dojos*.

Code-Katas und Coding-Dojos

In fernöstlichen Kampfsportarten trainiert man vorgegebene Bewegungsabläufe, die Katas, in einem Dojo genannten Trainingsraum. Dementsprechend ist ein Code-Kata eine kleinere Programmieraufgabe, die ein Entwickler viele Male hintereinander löst. Wobei er versucht, immer bessere und sauberere Lösungen zu finden. Durch die Wiederholung gehen – hoffentlich – bestimmte Techniken in gewohnheitsmäßiges Handeln über.

Bei einem Coding-Dojo treffen sich mehrere Entwickler, um eine Code-Kata gemeinsam durchzuführen und dabei voneinander zu lernen. Zwei Entwickler arbeiten an einem Rechner, während die anderen zusehen. Nach einem festgelegten Intervall, zum Beispiel alle acht Minuten, wechseln jeweils ein aktiver und ein passiver Teilnehmer die Rollen.

Zweck

Kenntnisse sind das eine – ihre korrekte Anwendung ist das schwierige andere. Sauberen Code schreiben lernen Sie nur beim Entwickeln und bei anschließender kritischer Reflexion. Für Letzteres sind Code Reviews ein hervorragendes Mittel.

Denken Sie an Wissenschaftler, die ihre neuen Ideen auch immer wieder einander vorstellen und kritisch diskutieren. Selbst harte Auseinandersetzungen dienen – abseits persönlicher Eitelkeiten – dabei letztlich immer dem Ziel, eine möglichst überzeugende Theorie zu entwickeln. Um zu überzeugen, muss man verständlich argumentieren.

Code Reviews sind in diesem Sinne die Symposien der Softwareentwickler.

Wie beim Verfassen einer wissenschaftlichen Arbeit sollten Sie sich auch beim Programmieren um möglichst weitgehende Korrektheit und Verständlichkeit bemühen. Die Verständlichkeit Ihres Codes können Sie jedoch nur sehr eingeschränkt selbst beurteilen.

Sie wissen (hoffentlich), warum Sie eine bestimmte Anweisung gerade so geschrieben haben. Aber ob sich Ihre Absicht einem Leser erschließt, können Sie allein niemals mit Gewissheit erkennen.

Deshalb müssen Sie Ihren Code gemeinsam mit anderen durchsehen und diskutieren. Das ist der Zweck von Code Reviews.

Was nicht geht

Man kann bei Code Reviews viel falsch machen, und einige Fehler lassen sich im Nachhinein nur schwer korrigieren. Deshalb beginne ich mit den Dingen, die Sie auf keinen Fall zulassen dürfen.

-  Ein Code Review ist weder ein Tribunal noch ein Wettbewerb. Es geht nicht um Entwickler – es geht um den Code.
- Vermeiden Sie – leicht ausufernde – Rechtfertigungsdiskussionen. Es geht nicht darum, warum etwas so ist, wie es ist, sondern um das Verbessern des Codes.
- Vorgesetzte (falls sie nicht selbst mitentwickeln) dürfen an Code Reviews nicht teilnehmen.
- Erlauben Sie externen Zuhörern nur ausnahmsweise und bei gut eingespielten Teams die Teilnahme.

- ✔ Widerstehen Sie der Verlockung, auf der Basis von Code Reviews Leistungsbewertungen vorzunehmen.
- ✔ Sarkastische und beleidigende Bemerkungen sind verboten. Schließen Sie uneinsichtige Teilnehmer im Wiederholungsfall aus.
- ✔ Code Reviews sind keine Bühne für Selbstdarsteller und Besserwisser. Das zu gewährleisten, kann für Sie zu einer schwierigen Aufgabe werden.
- ✔ Verlassen Sie sich nicht zu stark auf Tools und Metriken. Die können helfen, aber Code Reviews lassen sich nicht automatisieren.
- ✔ Vermeiden Sie starken Zeitdruck. Verstehen ist (für die hier besprochenen Reviews) wichtiger, als ein bestimmtes Pensum erledigt zu haben.

Code Reviews dürfen von den Beteiligten auf keinen Fall als ein Mittel zur verdeckten Kontrolle oder Reglementierung wahrgenommen werden.

Das Potenzial

Code Reviews sind bereits seit Längerem ein bewährter Weg, um die Softwarequalität zu verbessern. Meistens steht dabei das Finden von Fehlern im Vordergrund.

Das ist zweifellos wichtig, aber für Clean Code nicht genug. Gemeinsame – auch kontroverse – Diskussion von Code ist der einzige Weg, auf dem Sie wirklich eine respektable Fertigkeit im Schreiben von sauberem Code erwerben können.

Das ist vergleichbar mit dem Verfassen eines Romans. Nur wenige Schriftsteller haben ihre Werke wirklich ganz allein »im stillen Kämmerlein« geschrieben. Viele Meisterwerke der Literatur verdanken ihre Vollkommenheit ganz erheblich dem Einsatz engagierter Lektoren.

Code Reviews bieten darüber hinaus weitere Möglichkeiten, Ihre Arbeit und die des gesamten Teams zu verbessern:

- ✔ **Erfahrungsaustausch**

  Durch die Diskussion lernen Sie andere Sichtweisen und daraus ableitbare Umsetzungen kennen. Sie und Ihre Partner sind gezwungen, Vor- und Nachteile zu bewerten und argumentativ zu vertreten.

  Gemeinsames Lernen können Sie dabei gar nicht vermeiden.

- ✔ **Entwicklung eines gemeinsamen Verständnisses**

  Die Betrachtung des Codes schließt automatisch die Erörterung des fachlichen Hintergrunds ein. Erst das gemeinsame Verständnis erlaubt es Ihnen, über die Formalisierung durch Code zu reden.

- ✔ **Angleichung der Programmierstile**

  Der geteilte Erfahrungsschatz und das gemeinsame Verständnis führen im Laufe der Zeit zu einer Annäherung der Programmierstile. Das macht es zunächst Ihnen im Team leichter, fremden Code zu verstehen. Auf längere Sicht vereinfacht es in späteren Phasen die Einarbeitung neuer Entwickler.

- ✔ **Teamgeist**

  Code, den Sie im Team durchgesprochen und eventuell anschließend verbessert haben, ist nicht mehr ausschließlich Ihr Code. Der Code gehört dem Team.

- ✔ **Qualitätsbewusstsein**

  Ganz absichtlich an letzter Stelle, weil es möglicherweise mit dem Prinzip kollidiert, keine Leistungsbewertung vorzunehmen, es andererseits aber gar nicht vermieden werden kann: Wenn Code diskutiert und damit wertgeschätzt wird, werden Sie als Entwickler ganz automatisch Ihre Anstrengungen erhöhen und ordentliche Arbeit leisten wollen.

Code Reviews helfen Ihnen, Code mit weniger Fehlern zu liefern. Das gilt unabhängig davon, wie erfolgreich Sie Clean Code

bereits praktizieren.

Außerdem können Code Reviews sehr effektiv sein, wenn Sie weniger erfahrene Mitarbeiter in ihrer persönlichen Entwicklung unterstützen wollen.

Durchführung

Sie haben jetzt gehört, wie wichtig Code Reviews für das Etablieren einer an Clean Code ausgerichteten Entwicklungskultur sind. Entsprechend sorgfältig müssen Sie diese Veranstaltungen vorbereiten und durchführen.

Erfolgsvoraussetzungen

In einem vorangegangenen Abschnitt habe ich Ihnen bereits die unbedingt zu vermeidenden Fallstricke dargestellt. Das reicht jedoch nicht. Im besten Fall haben Sie einen professionellen Moderator für Ihre Reviews, dann können Sie das Folgende gern überspringen. Gerade bei kleinen Projekten wird Ihnen aber gar nichts anderes übrig bleiben, als Ihre Reviews selbst zu organisieren.

Folglich müssen Sie oder ein anderer Entwickler das Ganz in die Hand nehmen, und dann sind die folgenden Hinweise hoffentlich hilfreich.

Sie müssen den Erfolg organisieren.

- ✔ **Schaffen Sie eine vertrauensvolle Atmosphäre**

  Wenn Ihr Team noch relativ neu ist, können Sie jeweils am Beginn nochmals die Grundsätze wiederholen, um so mögliches Misstrauen zu abzubauen.

- ✔ **Vermeiden Sie Störungen**

  Machen Sie dem Projektumfeld klar, dass Code Reviews wichtig sind und nicht durch Herausrufen von Teilnehmern oder kurzfristige Absagen gestört werden dürfen.

- ✔ **Regelmäßigkeit**

Um Code Reviews als festen Bestandteil des Entwicklungsprozesses zu etablieren, müssen sie regelmäßig und zu einem festen Zeitpunkt stattfinden.

Sie sollten dafür alle ein bis zwei Wochen zwischen 60 und 90 Minuten einplanen.

- ✔ **Checklisten**

  Verwenden Sie eine Liste derjenigen Punkte, die beim Review besprochen werden müssen. Das hilft Ihnen besonders im Anfang. Später können Sie kontrollieren, ob sich möglicherweise Unterlassungen eingeschlichen haben.

- ✔ **Lernwille**

  Sie und die anderen Teilnehmer müssen gewillt sein, an der Entwicklung Ihrer Fähigkeiten zu arbeiten und sich aktiv zu beteiligen.

- ✔ **Reviews müssen Spaß machen**

  Wenn Sie das erreicht haben, steht dem Erfolg nichts mehr im Wege.

Vorbereitung

Code Reviews müssen sorgfältig vorbereitet werden. Der zu besprechende Code muss allen Teilnehmern so zeitig bekannt sein, dass sie sich vorbereiten können. Als angemessener Umfang wird allgemein ein Wert von 300–500 Zeilen (LOC) für eine Stunde angesehen.

Code-Auswahl

Zweckmäßigerweise führen Sie eine Liste (*Backlog*) von Quelltexten, die für ein Review anstehen. Da Sie es wahrscheinlich nicht schaffen werden, den gesamten Code Ihres Projekts zu begutachten, sollten Sie für diese Liste Code nach bestimmten Gesichtspunkten auswählen.

Im Folgenden einige Vorschläge für die Auswahl. Suchen Sie Code aus,

- ✔ der besonders wichtig ist (häufig durchlaufen, zentrale Geschäftslogik),
- ✔ in dem auffällig viele Fehler gefunden wurden,
- ✔ der von Entwicklern geschrieben wurde, die neu im Team oder unerfahren sind,
- ✔ der als besonders komplex oder schwierig zu entwickeln gilt,
- ✔ der demnächst erweitert oder angepasst werden muss oder
- ✔ dessen Pflege von einem anderen Entwickler übernommen werden soll.

Hin und wieder sollten Sie auch solchen Code erneut durchsehen, bei dem in einem früheren Review größerer Überarbeitungsbedarf gefunden wurde. Auf diese Weise erhalten Sie nicht nur eine Rückkopplung, ob die Review-Ergebnisse korrekt eingearbeitet wurden, sondern Sie lernen gleichzeitig, wie gut und angemessen Ihre damaligen Schlussfolgerungen wirklich waren.

Review-Fokus

Wenn Ihr Team eine gewisse Reife erreicht hat, kann es passieren, dass sich Routine breitmacht und das Engagement nachlässt. Spätestens dann ist es an der Zeit, die Reviews gezielter auf bestimmte Aspekte zu konzentrieren.

Sammeln Sie deshalb von Anfang an diejenigen Themen, die eine nähere Betrachtung wert sind.

Einige Kriterien für die Auswahl:

- ✔ Probleme, die immer wieder kontrovers diskutiert werden
- ✔ Bibliotheken und Module, die neu hinzugekommen sind
- ✔ Umgang mit externen Schnittstellen
- ✔ Programmteile, bei denen die Laufzeit oder der Speicherbedarf als kritisch erkannt wurde

Außerdem gibt es einige Themen, die immer eine gezielte Analyse verdient haben, dazu gehören:

- ✔ Logging
- ✔ Fehler- und Exception-Behandlung
- ✔ Konfiguration der Anwendung
- ✔ Abhängigkeiten zwischen Packages und Modulen
- ✔ Sicherheitsaspekte

Den gewählten Schwerpunkt des Reviews geben Sie zusammen mit dem ausgewählten Code so rechtzeitig bekannt, dass sich alle Teilnehmer darauf vorbereiten können.

Externe Unterstützung

Es wird vorkommen, dass Sie auf Themen stoßen, zu denen in Ihrem Team nicht ausreichend Kompetenz vorhanden ist. Zögern Sie nicht, wenn irgend möglich, in einem solchen Fall externe Experten einzubeziehen.

Das bietet sich vor allem dann an, wenn erstmals Pakete eingebunden oder Schnittstellen bedient werden, für die an anderer Stelle bereits Erfahrungen vorliegen. Optimal ist es, wenn der externe Reviewer als Ergänzung eigenen Code präsentieren kann.

Ganz allgemein empfehle ich Ihnen, zumindest ab und zu, einen Entwickler von außerhalb zu Ihren Reviews einzuladen, insbesondere wenn Ihr Team relativ stabil ist. Dadurch bekommen Ihre Reviews – hoffentlich – neue Impulse. Wichtig ist nur, dass die externen Teilnehmer die Grundregeln erfolgreicher Reviews kennen und einhalten.

Schließlich möchte ich Sie noch bitten: Nehmen Sie nicht nur externe Hilfe in Anspruch, seien Sie auch bereit, andere Teams mit Ihrer Erfahrung zu unterstützen, wenn Sie darum gebeten werden.

Review-Rollen

Um ein Review erfolgreich durchzuführen, müssen Sie die folgenden Rollen besetzen. Ohne klare Struktur laufen Sie immer Gefahr, dass Ihnen die Veranstaltung in ein wenig strukturiertes Geschwätz entgleitet.

Es sei hier noch einmal daran erinnert, dass es bei dieser Diskussion um teaminterne Reviews geht, bei denen die gegenseitige Weiterbildung das zentrale Ziel ist. Alle Rollen sollten daher im Wechsel von den Teammitgliedern wahrgenommen werden.

Moderator

Code Reviews müssen moderiert werden. Besonders in ungeübten Teams ist die Rolle des Moderators extrem wichtig. Zunächst sind vor allem soziale Fähigkeiten gefordert. Er muss mit Geduld und Fingerspitzengefühl die vertrauensvolle und konstruktive Atmosphäre entwickeln, die gute Code Reviews auszeichnet.

Gleichzeitig muss er aber auch ein von den anderen Teammietgliedern akzeptierter Entwickler sein, um seiner Rolle gerecht werden zu können. Sie sehen jetzt sicher schon ein Problem: Gute Entwickler mit sozialen und kommunikativen Fähigkeiten werden für viele Aufgaben gebraucht. Mit etwas Glück aber haben Sie bereits geeignete Kandidaten in Ihrem Team.

Es ist nicht zwingend notwendig, dass die Moderation immer von der gleichen Person wahrgenommen wird. Sie sollten sogar anstreben, dass auf Dauer jeder im Team als Moderator fungieren kann.

Präsentator

Der Präsentator stellt den ausgewählten Code in Form eines *Walkthrough* vor und macht dabei bereits erste Vorschläge für Veränderungen oder zu diskutierende Punkte.

Präsentieren Sie den Code aus einer Entwicklungsumgebung (IDE) heraus. Das erlaubt Ihnen, die gleiche Unterstützung wie

beim Entwickeln (Infoboxen, Ergänzungsvorschläge) zu benutzen und kleinere Änderungen direkt zu erproben.

Diese Rolle wird von pro Code-Artefakt wechselnden Teilnehmern ausgeübt.

Protokollant

Ein pro Review wechselnder Teilnehmer wird damit beauftragt, die Ergebnisse des Reviews festzuhalten. In welcher Form das geschieht, muss im Einzelfall entschieden werden.

Reine Protokolle sind eher nicht sinnvoll, weil das darin enthalten Wissen häufig in einer Ablage verkümmert. Besser ist es, wenn Sie die Erkenntnisse dort dokumentieren, wo sie später auch mit hoher Wahrscheinlichkeit gelesen werden, also zum Beispiel in einem Wiki oder im Programmierhandbuch.

Die Rolle des Protokollführers ist zwar bedeutsam, aber insofern schwierig, als gewissenhafte Protokollierung und engagierte Diskussionsteilnahme kaum gleichzeitig möglich sind.

Review-Werkzeuge und Metriken

Es gibt eine Reihe von Werkzeugen, die versprechen, Code Reviews automatisieren zu können. Da es bei Clean Code um die Qualität von Code und Software geht, gebe ich Folgendes zu bedenken. Qualität ist prinzipiell nicht messbar. Man kann sie daher auch mit formalen Methoden nicht analysieren.

Was man messen oder automatisch analysieren kann, sind Indikatoren, die mehr oder weniger mit der Qualität korrelieren. In Kapitel 2 *Dimensionen von Codequalität* finden Sie eine umfassendere Diskussion dieser Frage.

Wenn Sie sich nur auf derartige Werkzeuge verlassen, besteht die Gefahr, dass die Entwickler ihr Augenmerk zu stark auf die Kennziffern lenken und dabei das ursprüngliche Qualitätsziel aus den Augen verlieren.

Solche Werkzeuge sind jedoch sehr nützliche Hilfsmittel bei der Vorbereitung von Reviews. Sie können Ihnen helfen, die

Programmteile zu finden, die eine eingehendere Prüfung verdient haben.

Diskutieren Sie im Review stets die vorliegenden Markierungen (neudeutsch auch *Findings* genannt). Versuchen Sie, sich zu einigen, in welchen Fällen Markierungen als *fälschlich positiv* (*false positive*) ignoriert werden können. Möglicherweise kommen Sie auch zu dem Schluss, einige Regeln vollständig zu deaktivieren.

Review-Bewertung

Mit den Code Reviews verfolgen Sie ein Ziel: Der von Ihnen und Ihrem Team produzierte Code soll in wachsendem Maße den Idealen von Clean Code entsprechen. Dafür investieren Sie einiges an Zeit und Mühe.

Vergewissern Sie sich deshalb regelmäßig, ob der eingesetzte Aufwand gerechtfertigt ist. Die wichtigsten Maßstäbe sind allerdings qualitativer Art und daher nicht in Zahlen auszudrücken.

Codequalität

Die hoffentlich steigende Codequalität ist das beste Maß für den Erfolg der Code Reviews. Beginnen Sie dabei mit folgenden Indikatoren:

- ✔ **Dichte der kritischen Punkte**

  Je weniger Mängel im Review gefunden werden, umso näher sind Sie Ihrem Ziel bereits gekommen.

- ✔ **Diskussionsbedarf**

  Je sauberer Ihr Code formuliert ist, desto weniger Verständnisfragen sind zu erwarten. (Wenn Sie allerdings wenige Mängel finden, dann aber viele unverständliche Dinge sehen, war die Analyse nicht gründlich genug.)

- ✔ **Fehlerhäufigkeit**

Sauberer Code und wachsende Fähigkeiten drücken sich nicht zuletzt in einer sinkenden Fehlerhäufigkeit aus.

Review-Qualität

Wie fühlen Sie sich am Ende eines Reviews? Ein wenig erschöpft, aber gleichzeitig mit dem befriedigenden Gefühl, etwas erreicht zu haben? Wenn es den anderen Teilnehmern ähnlich geht, ist alles in Ordnung. Sicher klappt das nicht jedes Mal, aber die vorherrschende Tendenz sollte schon in diese Richtung weisen.

Lassen Sie sich nicht irritieren, wenn ein Review ergibt, dass nichts oder nur sehr wenig am Code zu ändern ist. Erinnern Sie sich dann daran, das Ziel eines Reviews ist vor allem, dass alle Teilnehmer dabei lernen. (Fragen Sie sich trotzdem selbstkritisch, ob Sie nicht zu nachlässig und bequem geworden sind.)

Nichts ist so gut, dass es nicht noch besser werden könnte. Organisieren Sie deshalb regelmäßig, vielleicht vierteljährlich, eine Retrospektive, wie man das aus agilen Projekten kennt.

Diskutieren Sie, was besonders gut lief und was nicht. Überlegen Sie, was man ändern könnte, und überarbeiten Sie bei dieser Gelegenheit die Review-Checkliste.

Das Wichtigste in Kürze

- ✔ Code Reviews sind der beste Weg, um in einem Team oder einem Projekt die Clean-Code-Kultur zu verbreiten und fest zu etablieren.
- ✔ Gute Code Reviews zeichnen sich durch eine offene, konstruktive und sachliche Atmosphäre aus. Es geht um Code, nicht um Personen.
- ✔ Code Reviews dürfen nicht zur Bewertung von Entwicklern benutzt werden. Die Teilnahme von Vorgesetzten ist deshalb tabu.

- ✔ Führen Sie die Reviews regelmäßig als Teil des Entwicklungsprozesses durch.
- ✔ Code Reviews müssen vorbereitet und Rollen festgelegt werden. Eine Schlüsselposition nimmt der Moderator ein.
- ✔ Beziehen Sie bei Bedarf Experten ein, um die Substanz der Diskussion zu sichern.
- ✔ Die Ergebnisse müssen so festgehalten werden, dass sie für den weiteren Entwicklungsprozess wirksam werden.
- ✔ Code Reviews müssen Spaß machen.

Kapitel 21

Aus Fehlern lernen

IN DIESEM KAPITEL

Auch unangenehme Ereignisse, die man, wie Fehler, gewöhnlich zu vermeiden trachtet, können ihre positiven Seiten haben. Ausgerechnet Fehler sind eine wichtige, vielleicht sogar die wichtigste Quelle neuer Erkenntnisse.

Fehler macht jeder

> Wer aufhört, Fehler zu machen, lernt nichts mehr dazu. (Th. Fontane)

Alle wissen, dass Fehler beim Softwareentwickeln unvermeidlich sind, aber trotzdem wird weithin so getan, als wären sie vermeidbar, wenn man nur *alles richtig macht*. Wahrscheinlich denken Sie in diesem Zusammenhang auch zuerst ans Testen.

Diese Einstellung verhindert jedoch, dass das in Fehlern steckende Potenzial erschlossen wird. Natürlich sollten Fehler möglichst vermieden werden. Akzeptieren Sie aber ganz einfach die Tatsache, dass Sie (oder andere) fehlerhaften Code geschrieben haben und schreiben werden.

Vertuschen und verbergen Sie nichts! Aus jedem Fehler können Sie neues Wissen und neue Fertigkeiten erwerben. Nutzen Sie

diese Chance!

Fehler sind die wichtigste Möglichkeit, wesentliche Einflussfaktoren zu isolieren. Ein Faktor, der einen Fehler verursacht, hat sich damit ganz automatisch als nicht vernachlässigbare Größe identifiziert. Im Erfolgsfall, das heißt ohne das Auftreten von Fehlern, ist es in der Regel deutlich aufwendiger, die entscheidenden Einflussfaktoren zu finden.

Fehlerkultur

Ein eindrucksvolles Beispiel für den Umgang mit Fehlern liefert die Luftfahrt. Flugzeuge sind in ihrer Komplexität mit IT-Systemen vergleichbar und verfügen zudem über zahlreiche softwaregesteuerte Funktionen. Vor ihrer Freigabe müssen sie unzählige Prüfungen und Zertifizierungsprozesse durchlaufen. Trotzdem lassen sich Fehler nicht vollständig verhindern, und es passieren Abstürze und andere Unfälle.

Bemerkenswerterweise und in deutlicher Diskrepanz zur IT-Welt nimmt die Anzahl der erheblichen technikbedingten Vorkommnisse bei Flugzeugen seit Jahren ab, und das trotz wachsender Komplexität der Systeme, Zunahme des Verkehrs und Anstieg des Durchschnittsalters der Flotten.

Diese erfolgreiche Entwicklung beruht vor allem darauf, dass es weitgehend gelingt, die Wiederholung von Fehlern zu verhindern. Erreicht wird das durch die sorgfältige Untersuchung aller relevanten Vorfälle durch unabhängige Inspektoren.

In den meisten Ländern gibt es dazu Institutionen wie die Bundesstelle für Flugunfalluntersuchung (BFU) in Deutschland, die im Ergebnis ihrer Auswertungen dann auch Empfehlungen für das Beseitigen von Risiken aussprechen können.

Die Erarbeitung solcher Auswertungsberichte ist meist langwierig, schwierig und erfordert ausgeprägte Expertise – verursacht mithin erhebliche Kosten. Die erreichte hohe Sicherheit im Luftverkehr rechtfertigt diesen Aufwand und beweist die Richtigkeit einer solchen Strategie. In Bereichen wie Bahnverkehr oder Reaktorsicherheit oder Bauaufsicht gibt es ähnliche Verfahren mit mehr oder weniger ausgeprägter Stringenz.

In der IT fehlt es leider noch an einer vergleichbaren Fehlerkultur. Eine begrüßenswerte Ausnahme sind die Listen erkannter Sicherheitslücken. Angesichts der zunehmenden Durchdringung aller Lebensbereiche und der damit einhergehenden Risiken kann das aber nur ein erster Schritt sein.

Fehler analysieren

Wenn ein Programm sich nicht so wie erwartet verhält, ist sehr oft ein Fehler die Ursache. Ich schreibe hier ganz bewusst *sehr oft*, denn bisweilen ist auch einfach die Erwartung falsch.

Deshalb sollten Sie – selbst wenn es schwerfällt – nach einer Fehlermeldung nicht sofort in hektische Betriebsamkeit verfallen. Jeder Praktiker kann von überstürzt eingespielten *Hotfixes* berichten, die dann durch weitere Hotfixes korrigiert oder ganz zurückgenommen werden mussten. Also Ruhe bewahren und die folgenden Schritte ausführen:

1. **Klären Sie, ob tatsächlich ein Fehler vorliegt.**

   Prüfen Sie dazu das beanstandete Verhalten gegen die fachlichen Anforderungen.

2. **Versuchen Sie, das Fehlverhalten zu reproduzieren.**

   Optimal ist es, wenn Sie einen passenden Testfall anlegen können.

3. **Lokalisieren Sie den Fehler im Code.**

   Das Ergebnis kann eine einzelne Codezeile, aber auch (bei konzeptionellen Fehlern) ein ganzer Bereich sein.

4. **Analysieren Sie die Fehlfunktion und entwickeln Sie ein Konzept für die Behebung.**

   In diesem Schritt geht es um die Frage: Was läuft falsch?

   Besonders unter Zeitdruck (»Die Produktion wartet!«) wird dieser Schritt oft nur unvollständig ausgeführt. Wenn die Zeit nicht reicht, um einen Fehler in der gebotenen Gründlichkeit zu beheben, müssen Sie Mittel und Wege finden, das später (in einer Refactoring-Phase) nachzuholen.

   Auf jeden Fall merken Sie sich solche kritischen Punkte in einer Liste.

5. **Nehmen Sie die konzipierte Korrektur vor, testen Sie das Ergebnis und geben Sie es frei.**
6. **Ermitteln Sie die Fehlerursachen.**

   Im Gegensatz zu Schritt 4 geht es jetzt nicht darum, was falsch war, sondern darum, wie es zum fehlerhaften Code kommen konnte.
7. **Leiten Sie notwendige Schlussfolgerungen ab.**

Die beiden letzten Punkte werden in den nächsten Abschnitten genauer behandelt.

 Vergessen Sie nie: Es geht um Fehler und deren Zustandekommen – nicht um Personen!

Persönliche Kritik, spitze Bemerkungen und Vorwürfe sind strikt zu unterlassen. Suchen Sie keine Sündenböcke, suchen Sie Ursachen! Nur so lässt sich die notwendige offene Atmosphäre herstellen und erhalten.

Fehlerursachen ermitteln

Die Stelle im Code, die Sie ändern müssen, um das monierte Fehlverhalten zu korrigieren, hat zwar das falsche Verhalten verursacht, ist aber nicht unbedingt die wirkliche Ursache des Fehlers.

Bei der Suche nach den Ursachen geht es darum, herauszubekommen, warum Sie (oder andere) diesen Code genau in der Form geschrieben haben, die später das Problem herbeigeführt hat.

Fehlerarten

Bei der Suche nach den Ursachen ist es vorteilhaft, die Fehler in verschiedene Kategorien einzuteilen:

- ✔ **Fehlerhafte fachliche Vorgaben**

In diesem Fall sind Sie aus dem Schneider. Vielleicht empfehlen Sie der betroffenen Fachabteilung, etwas aus Ihrem produktiven Umgang mit Fehlern zu lernen.

- **Konzeptionelle Fehler**

  Das sind in der Regel die am aufwendigsten zu behebenden Fehler. Bei der Analyse muss der gesamte Entwicklungsprozess einbezogen werden. Die Konsequenzen gehen möglicherweise über die hier betrachteten Themen hinaus.

- **Codierfehler**

  Das sind die Fehler, bei denen das Modell nicht korrekt formalisiert wurde. Solche Fehler sind durch ausreichende Tests weitgehend vermeidbar. Das darf Sie aber nicht davon abhalten, im Fall des Falls konsequent nach den Ursachen zu suchen.

- **Schreibfehler (»Typo«)**

  Auch vermeintlich einfache Tippfehler können durch die Umstände begünstigt werden und müssen deshalb analysiert werden.

Priorisierung

Natürlich können Sie nicht jeden Fehler derart ausführlich untersuchen. Deshalb empfiehlt es sich, zusätzlich eine Priorisierung nach den folgenden Kriterien vorzunehmen:

- **Auswirkung**

  Das betrifft die »Schwere« des Fehlers und die Dringlichkeit seiner Behebung. Je gravierender die Auswirkungen sind, desto wichtiger ist die Analyse.

- **Anzahl ähnlicher Fehler**

  Wenn ein bestimmter Fehlertyp gehäuft auftritt, deutet das auf ein strukturelles Problem hin, dessen Behebung möglicherweise für den gesamten Code positive Auswirkungen haben wird.

- **Lokale Häufung**

  Wenn bestimmte Codeteile überproportional viele Fehler aufweisen, müssen diese Fehler mit Vorrang untersucht werden.

- **Alter des Fehlers**

  Manche Fehler treten erst lange, nachdem ein Programm in Produktion genommen wurde, in Erscheinung. Widmen Sie solchen Exemplaren besondere Aufmerksamkeit. Oft verstecken sich in der Nähe weitere unerwünschte Überraschungen.

Die endgültige Gewichtung hängt stark von der Situation und der Anzahl der aktuell zu bearbeitenden Fehler ab. Versuchen Sie so zu entscheiden, dass für Ihre Software der größtmögliche Nutzen entsteht.

Denken Sie an …

Nach diesen ganzen Vorbereitungen kommt nun der schwierigste Teil der Analyse: die Suche nach den wirklichen Ursachen.

Die Gründe, aus denen Fehler gemacht werden, sind so vielfältig wie die Fehler selbst. Darum kann ich Ihnen hier auch kein einfach abzuarbeitendes Rezept anbieten, sondern lediglich anhand von Beispielen einige Denkanstöße geben.

Vermeintliche Tippfehler

Auf den ersten Blick denken Sie vielleicht: Tippfehler kommen eben vor. – Richtig und trotzdem falsch. Stellen Sie sich einfach die folgenden Fragen:

- Warum hat der Compiler das nicht bemerkt?
- Sind die verwendeten Namen zu ähnlich?

  Wenn beispielsweise, wie ich schon erwähnt habe, `record` und `rekord` als Feldnamen in einer Methode vorkommen, sind Verwechslungen nur eine Frage der Zeit.

- ✔ Gibt oder gab es äußere Umstände, die die Konzentration bei der Arbeit beeinträchtigt haben?

  Die Spitzenreiter in dieser Kategorie sind häufige Unterbrechungen und zu lautes Umfeld. Denken Sie bitte daran, dass überlange Arbeitstage ebenfalls derartige Fehler provozieren können.

Codierfehler

Das ist die größte und heterogenste Gruppe von Fehlern. Sie umfasst alles, was sich nicht durch die Abänderung von ein paar Zeichen in Ordnung bringen lässt und andererseits nicht völlig falsch konzipiert ist.

Gerade Fehler dieser Art sollen durch sauberen Code vermieden werden. Deshalb verweise ich hier nur auf Teil III dieses Buchs und verzichte darauf, die Grundsätze zu wiederholen.

Prüfen Sie, ob sich der Fehler auf die Missachtung oder falsche Interpretation einer Regel zurückführen lässt. Schließen Sie nicht grundsätzlich aus, dass auch das Einhalten einer Regel manchmal falsch sein kann.

Da selbst ein umfangreiches Regelwerk nicht alle Eventualitäten berücksichtigen kann, beherzigen Sie bitte die nachstehende Regel.

Clean Code ist kein abgeschlossenes Regelwerk und will das auch nicht sein. Seien Sie kreativ und versuchen Sie, selbst Ergänzungen zu finden.

Diskutieren Sie mit anderen darüber und seien Sie nicht enttäuscht, wenn sich später herausstellt, dass Ihre Regeln doch nicht so gut waren. Sie werden auf jeden Fall mehr dabei gelernt haben, als es durch das bloßes Befolgen vorgegebener Regeln jemals möglich gewesen wäre.

Konzeptionelle Fehler

Konzeptionelle Fehler lassen sich fast immer auf zwei grundlegende Defizite zurückführen:

- ✔ Fehlendes technisches Know-how des Entwicklers

  Ein typisches Beispiel dafür ist das Verwenden von Methoden, die nicht thread-safe sind, in einer Anwendung, die das jedoch erfordert.
- ✔ Kommunikationsprobleme beim Vermitteln der fachlichen Anforderungen

Beide Probleme lassen sich nicht mit den hier betrachteten Mitteln lösen. Das bedeutet allerdings nicht, dass man sie ignorieren darf!

Fehler durch vorhergehende Fehlerkorrekturen

Das ist streng genommen keine eigene Fehlerkategorie, denn derartige Fehler können in jede der zuvor betrachteten Kategorien fallen. Trotzdem erscheint mir diese Gruppe so wichtig, dass ich ihr einen eigenen Abschnitt widmen möchte.

Fehlerkorrekturen, die – unbeabsichtigterweise – neue Fehlfunktionen verursachen, sind in umfangreichen Systemen leider eher die Regel als die Ausnahme.

Vielleicht lesen Sie dieses Buch ja gerade deshalb, um an diesem betrüblichen Zustand etwas zu ändern. Sehr gut. Wenn Sie das hier Diskutierte beherzigen, haben Sie schon einen großen Schritt in die richtige Richtung gemacht.

Fehlerkorrekturen sind deshalb so heimtückisch, weil sie meist unter Zeitdruck erfolgen. Das eigentlich notwendige Eindenken in die betroffene Funktion erfolgt nur ungenügend. Insbesondere nur selten auftretende Konstellationen können so übersehen werden und bei nicht vollständiger Testabdeckung als neue Fehler unbemerkt in Produktion gehen.

Fehler, die bei Korrekturen entstehen, sind ein klares Zeichen, dass Sie entweder keine ausreichende Analyse gemacht haben oder dass Sie bei der Ursachensuche nicht tief genug vorgedrungen sind. Manchmal kann es darüber hinaus daran

liegen, dass durch den beseitigten Fehler andere Fehler maskiert wurden oder dass die nicht im Code liegenden Fehlerursachen nicht so schnell behoben werden konnten.

Wenn die oben erwähnte Liste der nicht aufgearbeiteten Fehlerkorrekturen, also der Fehler, die im Code zwar behoben, deren wirkliche Ursache aber noch nicht ermittelt wurde, schon einigen Umfang hat, sollten Sie versuchen, jedes neue Problem als Argument für ein dringend notwendiges Refactoring zu verwenden.

Lassen Sie sich auf keinen Fall entmutigen und fahren Sie konsequent mit der Fehleraufarbeitung fort. Sie lernen dabei viel darüber, wie Fehler zustande kommen. Das hilft Ihnen dann, bessere Voraussetzungen für eine fehlerarme Arbeitsweise zu schaffen.

Der Erfolg wird auf die Dauer nicht ausbleiben, und die damit einhergehende Abnahme der Anzahl von Fehlern dieser Art ist ein sicheres Indiz, dass Sie Clean Code erfolgreich umsetzen.

Erkenntnisse nutzen

Ich hoffe, es ist Ihnen klar geworden, auf welche Weise Sie aus Fehlern neues Wissen gewinnen können. Bleibt noch die Frage, wie Sie mit dem erworbenen Wissensschatz umgehen. Auch dazu möchte ich Ihnen einige Anregungen geben.

Code Reviews nutzen

Code Reviews sind eine ideale Gelegenheit, um die Erkenntnisse aus der Fehleranalyse zu vertiefen und zu kommunizieren. Wie in Kapitel 20 *Miteinander Lernen – Code-Reviews* ausgeführt, herrscht dabei bereits die erforderliche konstruktive und von Lernwillen geprägte Atmosphäre.

Durch die Arbeit am konkreten Fall gewinnt die Diskussion ganz sicher an Tiefe und Engagement. Und bei der Vorstellung Ihrer Analyse werden Sie mit großer Wahrscheinlichkeit auf zusätzliche Aspekte aufmerksam gemacht werden. Die Diskussion wird

automatisch engagierter, weil es nicht mehr darum geht, was zu Fehlern führen *könnte*, sondern ganz klar darum, was zu echten Fehlern geführt hat.

Ergebnisse dokumentieren

Nun haben Sie bereits so viel Zeit und Energie in die Fehleranalyse gesteckt. Halten Sie durch und sorgen Sie dafür, dass etwas davon bleibt. Ja, es geht um die Dokumentation der Ergebnisse.

Ihre Präsentation im Code Review und die damit verbundene Diskussion mögen noch so eindrucksvoll gewesen sein; unter den wechselnden Anforderungen des Tagesgeschäfts wird ein großer Teil schon bald vergessen werden.

Also dokumentieren Sie! Ganz gleich, ob in einem Wiki, einer Tabelle oder einer Zettelsammlung an der Wand – verhindern Sie, dass die frisch gewonnenen Erkenntnisse verloren gehen.

Wiederholen: Erkenntnisse erneut erörtern

Aufschreiben allein ist gut, reicht jedoch nicht, wenn es gilt, Erkenntnisse in die Tat umzusetzen. So wie Sie eine Vokabel nicht vom einmaligen Aussprechen lernen, ist es auch mit dem Schreiben sauberen Codes.

Wiederholen Sie deshalb häufig, was Sie aus analysierten Fehlern gelernt haben.

Das kann beispielsweise am Beginn eines Code Reviews passieren, indem Sie die Erkenntnisse des vorangegangenen Reviews nochmals kurz erörtern. Außerdem ist es wahrscheinlich, dass Sie im Laufe der Zeit auf Fehler stoßen werden, die zu Erkenntnissen führen, die früheren sehr ähnlich sind.

Nutzen Sie solche Gelegenheiten, um die alten Erfahrungen aufzufrischen.

Das Wichtigste in Kürze

- ✔ Fehler sind eine wichtige Erkenntnisquelle.
- ✔ Gehen Sie offen mit Fehlern um, lernen Sie daraus.
- ✔ Fragen Sie nach der Ursache, keinesfalls nach einem Schuldigen. Letzteres führt dazu, dass Fehler verschwiegen werden, und ist daher absolut kontraproduktiv.
- ✔ Analysieren Sie auch scheinbar triviale Fehler sorgfältig. Möglicherweise hat deren wirkliche Ursache nur durch Zufall noch keine größeren Auswirkungen gehabt.
- ✔ Besprechen Sie Fehler und Fehleranalysen im Code Review.
- ✔ Halten Sie die gewonnenen Erkenntnisse fest und versuchen Sie, diese in die tägliche Arbeit einfließen zu lassen.

Kapitel 22

Es gibt immer was zu tun – Refactoring

IN DIESEM KAPITEL

Sinn und Zweck von Code-Refactoring

Refactoring vorbereiten und durchführen

Code-Refactoring ist ein eigenes Thema, das selbst in einem ganzen Buch schwerlich umfassend abgehandelt werden kann. In diesem Kapitel geht es mir deshalb darum, Ihnen die Technik der schrittweisen Überarbeitung nahezubringen.

Hoffentlich kann ich Sie davon überzeugen, dass es sich lohnt, sich intensiver mit Refactoring zu beschäftigen.

Die Idee

Dem Refactoring-Prinzip liegt die einfache – nicht nur beim Programmieren gültige – Erkenntnis zugrunde, dass es fast unmöglich ist, bereits beim ersten Versuch eine perfekte Lösung zu gestalten. Je komplexer die Anforderungen an eine Anwendung sind, desto höher ist die Wahrscheinlichkeit, dass sich nach der Umsetzung Defizite herausstellen.

Jeder kennt das und weiß, was gemeint ist, wenn bei neuen Produkten von »Kinderkrankheiten« die Rede ist. Die gibt es genauso auch bei neuem Code.

Das wird Ihnen in Ihrer täglichen Arbeit nicht anders gehen. Meistens bringen die Anwender in dieser Situation Verständnis

auf, erwarten aber – mit Recht – trotzdem, dass diese Defizite möglichst schnell behoben werden.

Sie müssen also alles daransetzen, Verbesserungen vorzunehmen und Fehler zu beheben. Dabei lauert jedoch die Gefahr, dass Sie sich zu viel auf einmal vornehmen. Zu verlockend kann es sein, da Sie den Code ja ohnehin schon umschreiben, gleich noch dies und das mit zu erledigen.

Wenn niemand Sie aufhält und Sie selbst nicht bremsen, finden Sie sich plötzlich inmitten offener Baustellen, und nichts funktioniert mehr. Deshalb müssen Sie den Code systematisch überarbeiten.

Refactoring ist ein solches systematisches Vorgehen, das Ihnen hilft, nicht in die beschriebene Falle zu tappen.

Never change a running system?

Wahrscheinlich haben Sie diesen Spruch schon zu hören bekommen, wenn es darum ging, Software zu ändern. Steht das nicht im Widerspruch zum Refactoring?

Ja und nein. Einerseits ist es vernünftig, keine unnötigen Risiken einzugehen. Jede Codeänderung birgt nun einmal die Gefahr, dass danach etwas nicht mehr funktioniert, und sei es nur, dass dadurch ein vorher maskierter Fehler plötzlich zuschlägt. Andererseits entwickelt sich die Welt ständig weiter.

Deshalb ist es besser, wenn Sie sich an die im englischsprachigen Raum übliche Variante des Spruchs halten, die da lautet: »If it ain't broke, don't fix it«. Das meint nichts anderes, als dass Sie einen vernünftigen Grund für die Überarbeitung haben sollten. Allerdings müssen Sie nicht immer warten, bis wirklich etwas nicht mehr geht, und Sie dann unter Zeitdruck geraten.

In der Zuverlässigkeitsanalyse nennt man es *vorbeugende Wartung*, wenn Komponenten kurz vor ihrem statistisch erwartbaren Ausfall ersetzt werden. Bei Software ist so ein erwartbarer Ausfall zum Beispiel ein angekündigtes Update für eine Komponente. Also beginnen Sie das Refactoring ebenfalls rechtzeitig, nicht erst wenn sich zeigt, dass nach dem Update Ihre Anwendung »broken« ist.

Die Praxis

Weil es sehr schwer ist, sauberen Code direkt zu schreiben, können Sie dieses Ziel ohne Refactoring praktisch nicht erreichen. Selbst wenn Sie von der Notwendigkeit zutiefst überzeugt sind, heißt das noch lange nicht, dass alle anderen das auch so sehen.

Vorbereitung

Refactoring birgt erhebliches Potenzial, aber immer auch ein Risiko. Deshalb können Sie auf eine sorgfältige Vorbereitung nicht verzichten.

Überzeugen

Aus Sicht des Auftraggebers bedeutet Refactoring Aufwand ohne unmittelbaren Nutzen. Sie wollen dabei keine neuen Funktionen realisieren. Ganz im Gegenteil besteht das Ziel gerade darin, die Umbauten so vorzunehmen, dass sie im Verhalten nach außen keine Spuren hinterlassen.

Um Ihren Auftraggeber davon zu überzeugen, dass diese Arbeit sinnvoll ist, muss sie gut begründet werden. Der mögliche Nutzen tritt erst längerfristig ein. Er besteht im Wesentlichen aus all den Vorteilen, um derentwillen Sie sich überhaupt mit sauberem Code befassen.

Wenn Sie Code überarbeiten wollen, sollte es trotzdem immer einen konkreten Grund dafür geben. Zumindest in Projekten, die wirtschaftliche Ziele verfolgen, ist die Verschönerung des Codes allein kein ausreichender Grund.

Refactoring ist hingegen gerechtfertigt oder sogar unbedingt erforderlich, wenn

- ✔ Erweiterungen geplant sind,
- ✔ das Entwicklungstempo nachlässt oder die Fehlerhäufigkeit steigt, weil die Komplexität nicht mehr sicher beherrschbar ist,
- ✔ der Code zur Weiterbearbeitung übergeben oder das Team vergrößert werden soll oder

- es sich nur um kleinere Änderungen handelt, die zum Beispiel aus einem Code Review resultieren.

Versetzen Sie sich in Gedanken selbst in die Rolle des Budgetverantwortlichen. Versuchen Sie aus dieser Sicht, den Nutzen zu bewerten, und stellen Sie diesen Nutzen dem Aufwand gegenüber. Wenn dieser Vergleich nicht überzeugend gelingt, sollten Sie das Refactoring eventuell zurückstellen.

Testfälle vorbereiten

Bevor Sie überhaupt einen Gedanken an Refactoring verschwenden können, müssen Sie sich darum kümmern, wie Sie garantieren, dass die Umbauten die Funktion nicht verändern.

Sie brauchen also einen ausreichenden Satz an Testfällen. Und diese Testfälle müssen wirklich alle wichtigen Funktionalitäten nicht nur formal, sondern auch inhaltlich abdecken, insbesondere Grenzfälle. Ganz gleich wie sorgfältig Sie vorgehen, Fehler können Ihnen immer unterlaufen, einfach weil wir alle Menschen sind.

Ohne hinreichende Testabdeckung wird Refactoring zum Glücksspiel. Falls Sie die benötigten Tests noch nicht haben, verteuert sich dadurch das Refactoring. Allerdings geht der Nutzen der Tests deutlich über den aktuellen Anlass hinaus, da sie dauerhaft eine höhere Softwarequalität sichern.

Lassen Sie sich nicht dadurch abschrecken, dass Sie eventuell keine vollständige Testabdeckung erreichen können. Testen Sie so viel wie möglich. Es ist besser, Sie lassen Ihre unvollständigen Tests laufen, als wenn Sie gar nichts unternehmen.

Schritt für Schritt

Funktionierenden Code ohne Schaden zu verändern, ist schwierig. Beherzigen Sie deshalb diese Regeln:

- ✔ Überarbeiten Sie den Code in kleinen Schritten. Nehmen Sie stets nur eine Änderung vor und überprüfen Sie das Ergebnis durch einen Lauf all Ihrer Testfälle. Sollte sich dabei ein Fehler herausstellen, können Sie ihn sehr viel schneller finden und beheben als nach großen Umbauten.
- ✔ Verlieren Sie inmitten der kleinen Schritte Ihr Gesamtziel nicht aus den Augen. Treten Sie von Zeit zu Zeit, im übertragenen Sinne, ein paar Schritte zurück und überprüfen Sie, ob die Richtung noch stimmt.
- ✔ Vergessen Sie nicht, am Ende soll der Code leichter verständlich sein. Wenn Sie irgendwann unsicher sind, diskutieren Sie mit Kollegen.
- ✔ Manchmal werden Sie feststellen müssen, dass Ihr geplantes Refactoring nicht wie gedacht funktioniert. Scheuen Sie sich nicht, notfalls alles zurückzudrehen, wenn keiner der Zwischenstände besser ist als die Ausgangslage.
- ✔ Refactoring ist kein Selbstzweck. Prüfen Sie immer, ob das erreichbare Ziel den Einsatz rechtfertigt. Nicht jeder fehlerhafte Entwurf lässt sich nachträglich korrigieren.

Die Methoden zum Refactoring sind keine Einbahnstraßen. Sie werden zu jeder Umbauregel stets auch die entgegengesetzte Regel finden. Es ist durchaus typisch, bei einem Umbau eine Regel sowohl in die eine als auch in die andere Richtung zu verwenden.

Wenn Sie erst einige Übung erlangt haben und das Refactoring beginnt, Ihnen Spaß zu machen, hüten Sie sich vor Übertreibung. Jeder Umbau eröffnet schließlich neue Möglichkeiten für weitere Veränderungen.

Hören Sie rechtzeitig auf. Es gibt meist andere Stellen im Code, die es nötiger haben.

Im Großen und im Kleinen

Sehen Sie Refactoring nicht nur als Methode für größere Überarbeitungen an. Es gibt nichts, was Sie hindert, stets und überall Ihren Code – mit Augenmaß! – zu verbessern.

Gewöhnen Sie sich an, jedes Mal, wenn Sie denken, eine Aufgabe abgeschlossen zu haben, innezuhalten. Machen Sie eine Kaffeepause oder besprechen Sie ein anderes Problem.

Danach werfen Sie durch die »Refactoring-Brille« einen kritischen Blick auf das Erreichte. Wenn Sie etwas finden, das sich zu verbessern lohnt, machen Sie es gleich.

Durch so ein *Refactoring im Kleinen* trainieren Sie das dabei notwendige Gefühl für die richtigen Entscheidungen. Außerdem üben Sie die diversen Muster, sodass Sie sich später beim *Refactoring im Großen* besser auf die inhaltlichen Aspekte konzentrieren können.

Ein Beispiel

Wenn Sie bereits Code überarbeitet haben und die Grundzüge des Refactoring kennen, können Sie diesen Abschnitt gefahrlos überspringen. Das kleine Beispiel erhebt keinen Anspruch auf praktische Relevanz. Es soll Ihnen das prinzipielle Vorgehen demonstrieren, nicht mehr.

Listing 22.1 zeigt Ihnen eine Methode, die einen Text wahlweise auf der Standard- oder der Fehlerkonsole ausgibt. Die Unterscheidung wird durch einen booleschen Parameter gesteuert.

```
public void output(String text, boolean isError) {
  if (isError) {
    System.err.println(text);
  } else {
    System.out.println(text);
```

```
  }
}
```

Listing 22.1: Ausgangssituation: Methode mit booleschem Parameter.

Diese Methode wird nun durch zwei Methoden ersetzt. Das Ergebnis sehen Sie in Listing 22.2. Diese Art des Refactoring heißt *Ersetzen von Parametern durch explizite Methoden*. Ein wichtiger Vorteil besteht darin, dass danach beide Methoden ein klares, sofort verständliches Interface haben.

```
public void outputError(String text) {
  System.err.println(text);
}
public void outputDefault(String text) {
  System.out.println(text);
}
```

Listing 22.2: Der erste Schritt: Die Ausgangsmethode wird durch zwei Methoden ersetzt.

In der Ursprungsversion muss die Bedeutung des zweiten Parameters (`isError`) eventuell erläutert werden. Außerdem sind die beiden neuen Methoden einfacher und damit weniger fehleranfällig.

Nachteilig ist allerdings, dass sie beide fast identischen Code enthalten. Deshalb sollten Sie bei dieser Lösung nicht stehen bleiben.

In einem nächsten Schritt kann jetzt die Vorschrift *Parametrisiere Methode* angewandt werden. Dabei wird aus Methoden, die das Gleiche nur mit verschiedenen Werten tun, eine Methode mit einem entsprechenden Parameter extrahiert. Listing 22.3 zeigt, wie in diesem Fall das Resultat aussieht.

```
public void outputText(PrintStream stream, String text) {
  stream.println(text);
}
```

Listing 22.3: Der zweite Schritt: Aus zwei Methoden wird wieder eine – aber mit anderem Parameter.

Sie könnten jetzt fragen, wo der Vorteil gegenüber der Ausgangslösung liegt. Der ist tatsächlich genauso klein wie dieses Beispiel. Aber wenn Sie die beiden Aufrufe

```
bsp.output(STATICTEXT, true);
```

und

```
bsp.outputText(System.err, STATICTEXT);
```

vergleichen, sollten Sie erkennen, dass der letztere deutlicher erkennen lässt, was seine Funktion ist.

Ein weiterer Vorteil besteht darin, dass diese Version allgemeiner ist und dadurch möglicherweise noch andere Methoden ersetzen kann. Es ist immer gut, wenn Sie nach der Überarbeitung weniger Code haben als vorher.

Die beiden Umformungen sind überdies ein Beispiel für die Verwendung genau entgegengesetzter Regeln und damit geradezu prototypisch. Zuerst wird ein Parameter dadurch beseitigt, dass man aus einer Methode mehrere bildet. Anschließende wird umgekehrt aus zwei Methoden durch Hinzunahme eines Parameters wieder eine erzeugt.

Das Wichtigste in Kürze

- ✔ Code-Refactoring ist das schrittweise systematische Verbessern bestehender Software. Da niemand aus dem Stand idealen Code schreiben kann, ist es die wichtigste Methode, um sauberen Code zu produzieren.
- ✔ Beim Refactoring werden keine neuen Funktionen realisiert. Nur die Codestruktur wird verbessert. Es muss deshalb einen Anlass geben, der geeignet ist, aus der verbesserten Struktur einen Nutzen zu generieren.
- ✔ Ohne gute Testabdeckung ist Refactoring ein schwer kalkulierbares Risiko. Im Zweifel sollten Sie ein verfügbares Budget eher in eine Erhöhung der Testabdeckung als in Umbauten investieren.

- ✔ Refactoring erfolgt in kleinen Schritten. Pro Schritt wird nur ein Umbau vorgenommen, sodass entstandene Fehler leichter gefunden und behoben werden können.
- ✔ Refactoring muss die Lesbarkeit des Codes verbessern. Wenn das nicht gelingt, ist es besser, alle Änderungen zurückzunehmen, als verschlechterten Code zu hinterlassen.
- ✔ Auch Refactoring unterliegt einer Aufwand-Nutzen-Bewertung. Führen Sie nur solche Veränderungen aus, die einen erkennbaren Wert liefern.

Teil V

Der Top-Ten-Teil

Besuchen Sie uns auf www.facebook.de/fuerdummies

IN DIESEM TEIL …

- ✔ Fehler, die es zu vermeiden gilt
- ✔ Nützliche Quellen für die intensivere Beschäftigung mit sauberem Code

Kapitel 23

10 Fehler, die Sie vermeiden sollten

IN DIESEM KAPITEL

Ratschläge für die praktische Tätigkeit

Häufige Fehler vermeiden

Mit dem Kauf dieses Buchs haben Sie bereits Ihr Interesse gezeigt, zukünftig besser programmieren zu wollen. Das ist löblich, denn die tägliche Erfahrung zeigt ja, dass es in Bezug auf die Qualität von Software noch viel zu verbessern gibt.

Damit Sie mit Ihren Bemühungen nicht schon auf den ersten Metern stecken bleiben, habe ich hier ein paar Dinge zusammengestellt, die Sie unbedingt *nicht* tun sollten.

Wenn Ihnen dieses Buch geschenkt wurde, freue ich mich als Autor natürlich ebenso. Für Sie als Beschenkter stellt sich allerdings eine knifflige Frage: Warum gerade dieses Buch?

Doch lassen Sie sich dadurch nicht die Laune verderben – lesen Sie es einfach!

Buch in Schrank stellen

Dieses Buch ist zum Gebrauch und nicht für den Bücherschrank bestimmt. Wie ein Reiseführer sollte es immer zur Hand sein. Markieren Sie, was Ihnen wichtig erscheint, machen Sie sich Notizen und ergänzen Sie Hinweise.

Legen Sie das Buch als Erinnerung neben Ihren PC, ganz so, wie manche – zur Erinnerung an ihre Diätvorsätze – ein unvorteilhaftes Foto an die Kühlschranktür kleben. Und natürlich können Sie das alles auch gern digital machen.

Nicht sofort anfangen

Die größte Gefahr für gute Vorsätze besteht darin, nicht gleich mit der Umsetzung zu beginnen. Irgendwo habe ich einmal den Spruch gelesen: »Morgen – der Tag, an dem die meisten Diäten anfangen.» Genau so verhält es sich mit dem besseren Programmieren.

Fangen Sie klein an, aber sofort. Haben Sie erst einen Grund fürs Aufschieben gefunden, werden Sie ständig noch viel schwerwiegendere finden und schließlich nicht mehr daran denken.

Aufgeben

Aller Anfang ist schwer und die Gefahr groß, dass Ihre Ergebnisse zunächst eher kümmerlich ausfallen. Lassen Sie sich davon nicht entmutigen. Vom Schriftsteller Leonhard Frank wird berichtet, dass er länger als ein halbes Jahr brauchte, um die erste Seite seines ersten Romans »Die Räuberbande« zu schreiben. So viel Geduld werden Sie bestimmt nicht benötigen.

Denken Sie stets daran, sauberer Code ist ein langfristiges Ziel. Mancher hat bessere Voraussetzungen, manche Aufgaben sind besser geeignet. Nicht jeder kann alles erreichen, aber jeder kann sein eigenes Potenzial voll entwickeln und ausnutzen.

Nicht streiten

Streit wird meist nicht als angenehm empfunden, aber er ist notwendig. Nichts ist auf die Dauer belastender als unterdrückte

Widersprüche. Ohne Streit gibt es keine Entwicklung. Unterschiedliche Ansichten sind völlig natürlich.

Wenn der Streit vernünftig ausgetragen wird, fördert das Wechselspiel von Argument und Gegenargument das Verständnis ungemein. Wichtig ist es, zu einem Ergebnis zu kommen, dass für unterschiedliche Menschen akzeptabel ist.

Schematisch anwenden

Einige Clean-Code-Darstellungen könnten den Eindruck erwecken, dass es sich um einen Satz strikt zu befolgender Regeln handelt. Nichts wäre verkehrter. Die Regeln sind Handreichungen und Hilfestellungen. Die Wirklichkeit ist so vielfältig, dass Ihnen die Entscheidung, was wo am besten geeignet ist, niemand abnehmen kann. Die Ausgewogenheit ist entscheidend und mit Regeln allein nicht erreichbar.

Kompromisse verweigern

Wo Menschen aufeinandertreffen, gibt es unterschiedliche Ansichten und Interessen. Nicht alle werden Ihre Sicht der Dinge teilen. Tragen Sie Ihre Argumente überzeugend und geduldig vor. Verweigern Sie sich jedoch nicht etwaigen Kompromissen, selbst wenn Sie fest von deren Fehlerhaftigkeit überzeugt sind.

Manchmal lassen sich Fehler leider nicht verhindern. Unduldsamkeit oder Verweigerung nimmt Ihnen jedoch jede Wirkungsmöglichkeit, um aus einer schlechten Entscheidung wenigstens noch das Beste zu machen. Das ist allerdings keine Rechtfertigung, immer oder zu oft nachzugeben.

Unrealistische Terminzusagen

Viele Entwickler sind hilfsbereite Menschen und sehen den Termindruck, dem ihre Auftraggeber ausgesetzt sind. Diese durchaus positive Haltung kann sich aber schnell gegen Sie

wenden, falls Sie leichtfertig nicht einzuhaltende Termine zusagen.

Sicher, wer fühlt sich nicht geschmeichelt, wenn eine kaum einzuhaltende Zusage unter dem Kompliment „das zu schaffen, traue ich nur Ihnen zu“ herausgekitzelt wird. Trotzdem, bleiben Sie hart und vergessen Sie nie: Zu starker Zeitdruck führt zu schlechtem Code – immer!

Überheblichkeit

Hüten Sie sich vor übereilten Urteilen. Nicht jeder Code, der nicht so aufgeräumt und klar aussieht, wie Sie sich das wünschen, stammt von unfähigen oder leichtfertigen Entwicklern.

Es kann viele Ursachen geben, beispielsweise großer Termindruck oder fehlende Erfahrung. Bisweilen ist die Komplexität der Aufgabe schuld, und manches, was auf den ersten Blick schlecht aussieht, haben Sie nur noch nicht vollständig verstanden.

Denken, fertig zu sein

So wie Code nie fertig überarbeitet ist, werden auch Sie nie damit fertig sein, die Kunst des sauberen Programmierens zu lernen. Die Programmiersprachen entwickeln sich ebenso wie Werkzeuge und Erwartungen weiter. Was heute gut und verständlich ist, lässt sich morgen vielleicht schon ganz anders und besser beschreiben.

Der Volksmund sagt zu Recht: Wer rastet, der rostet. Bleiben Sie am Ball. Sie können immer noch etwas lernen.

Alles tierisch ernst nehmen

Nichts ist so motivierend wie Spaß bei der Arbeit. Clean Code ist keine Spaßbremse – ganz im Gegenteil.

Vergessen Sie nie, das Ziel besteht darin, dass Ihnen die Softwareentwicklung noch mehr Spaß machen soll. Und lachen Sie auch mal herzhaft, wenn Ihnen etwas so richtig danebengegangen ist.

Kapitel 24

(Mehr als) 10 nützliche Quellen zum Auffrischen und Vertiefen

IN DIESEM KAPITEL

Weitere Quellen

Nützliche Ergänzungen

In diesem Kapitel finden Sie Webseiten, die den Inhalt dieses Buchs vertiefen und ergänzen. Ich beschränke mich dabei auf solche Seiten, die bereits einige Zeit existieren und den Eindruck vermitteln, auch weiterhin verfügbar zu sein. Trotzdem ist es nicht auszuschließen, dass die eine oder andere Adresse im Laufe der Zeit ungültig wird.

Die Suche nach `Clean Code` liefert zwar eine große Anzahl von Treffern. Leider sind darunter jedoch sehr viele, die sich inhaltlich nicht stark unterscheiden und häufig bei den einfachen Regeln stehen bleiben. Wirklich nützliche Seiten zu finden, erfordert daher etwas Ausdauer und Geschick.

Clean Code – das Buch und der Blog

https://www.oreilly.com/library/view/clean-code/9780136083238/

Das ist Robert Martins Buch, das wesentlich zur Entstehung der Clean-Code-Bewegung beitrug, in der englischen

Originalfassung. Das kostenpflichtige Angebot erlaubt einen zehntägigen freien Probezugriff.

Sie können dieses Werk auch auf Deutsch erwerben.

Einen äußerst kurzen Überblick finden Sie zudem unter

https://gist.github.com/wojteklu/73c6914cc446146b8b533c0988cf8d29

Auf seinem seit 2011 geführten Blog behandelt Robert Martin immer wieder einzelne Themen aus dem Clean-Code-Bereich:

https://blog.cleancoder.com/

Clean Code Developer

https://clean-code-developer.de/

Eine kurze Zusammenfassung des Clean-Code-Konzepts und der wichtigsten Regeln in deutscher und englischer Sprache. Enthält Verweise auf weitere Websites zum Thema. Als Einstiegspunkt gut geeignet.

Software Craftsmanship

http://manifesto.softwarecraftsmanship.org

Das sogenannte *Manifesto for Software Craftsmanship* in mehreren Sprachen kann hier unterzeichnet werden. Nützlich ist auch die verlinkte Liste weiterführender Literatur.

Versäumen Sie aber nicht, sich auch die wohlbegründete Kritik an der Idee, dass Softwareentwicklung (nur) eine Handwerkskunst ist, anzusehen:

https://dannorth.net/2011/01/11/programming-is-not-a-craft/

Dieser lesenswerte Beitrag ist wie viele informative Artikel leider nur auf Englisch vorhanden.

Java Code Conventions

https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Diese bereits 1997 noch von Sun Microsystems veröffentlichten Richtlinien werden nach wie vor gern als Basis für eigene Festlegungen genommen.

Inzwischen haben sie in dieser Rolle jedoch Konkurrenz durch Googles Pendant bekommen:

https://google.github.io/styleguide/javaguide.html

Wenn Sie die Wahl haben, sollten Sie letzteren Richtlinien den Vorzug geben.

97 Dinge, die jeder Programmierer wissen sollte

http://freecomputerbooks.com/97-Things-Every-Programmer-Should-Know.html

Eine nützliche Sammlung von Tipps für Softwareentwickler, nicht nur auf sauberen Code bezogen. Sehr anregend zu lesen, selbst wenn Sie nicht alle Ansichten teilen.

The Pragmatic Bookshelf

http://pragprog.com/articles

Eine weitere Sammlung von Artikeln und Präsentationen rund ums Programmieren – immer gut durchdacht und von echten Praktikern verfasst.

Prinzipien der Softwaretechnik

http://prinzipien-der-softwaretechnik.blogspot.com/

Das ist eine gute und recht vollständige Auflistung von Prinzipien der Softwaretechnik mit Quellenangaben. Die Seite ist nicht mehr ganz neu, aber deutsch und hilfreich für eine schnelle Übersicht.

Refactoring

https://refactoring.com/catalog/

Dieser von Martin Fowler zusammengestellte Katalog von Refactorings ist vor allem deshalb bemerkenswert, weil er die möglichen Umbauten stets bidirektional darstellt.

Es gibt keine Richtung, die automatisch zu besserem Code führt. Während an einer Stelle beispielsweise das Herausziehen einer Funktion sinnvoll ist, kann an anderer Stelle das Gegenteil, nämlich das Einfügen angebracht sein.

Deutsche Übertragung der Webseite:

http://www.tutego.de/java/refactoring/catalog/

Code Reviews

http://www.slideshare.net/mattiask/code-reviews-devoxx-france-2012

Diese Folien fassen die elementaren Grundlagen erfolgreicher Code Reviews in einer kompakten Übersicht zusammen.

https://entwickler.de/online/development/code-review-best-practice-295369.html

Dieser Artikel (auf Deutsch) beschäftigt sich mit Vorschlägen, durch die Code Reviews so verbessert werden können, dass alle Entwickler davon profitieren.

https://smartbear.de/learn/code-review/best-practices-for-peer-code-review/?l=ua

Diese Website zeigt, dass manchmal auch Werbung nützliche Tipps, in diesem Fall für die Durchführung effektiver Code

Reviews, enthalten kann.

Codeanalyse

https://jqassistant.org/

JQassistant ist ein sehr interessantes und mächtiges Werkzeug, um die Struktur und Architektur von Softwareprojekten zu analysieren. Die Verwendung ist nicht auf eine bestimmte Programmiersprache beschränkt.

Als Kehrseite der Mächtigkeit müssen Sie allerdings eine nicht ganz niedrige Einstiegshürde überwinden. Für die Bewertung oder Weiterentwicklung größerer Codebestände trotzdem sehr empfehlenswert.

https://blog.codecentric.de/2017/09/kontinuierliche-architekturvalidierung-mit-jqassistant/

Das ist eine auf Deutsch geschriebene und gut lesbare Einführung in JQassistant einschließlich herunterladbaren Democodes.

Verzögerungskosten

http://blackswanfarming.com/experience-report-maersk-line/

Wenn es in Ihrem Projekt einmal wieder überhaupt nicht vorwärtsgeht, weil dringend notwendige Entscheidungen nicht gefällt werden, versorgen Sie sich vielleicht aus den Erkenntnissen dieser Studie mit überzeugenden Argumenten.

Project Oberon

https://inf.ethz.ch/personal/wirth/ProjectOberon/index.html

Am Ende dieses Buchs noch etwas für diejenigen, die schon immer wissen wollten, was die Software *im Innersten zusammenhält*. Niklaus Wirth hat sein »Projekt Oberon« bis 2013

überarbeitet und präsentiert jetzt vom Design der Hardware bis zur grafischen Oberfläche ein vollständiges PC-System, das nicht nur für Spielzeuganwendungen geeignet ist. Dazu gehören alle Teile des Betriebssystems und natürlich auch ein Compiler.

Als Programmiersprache wird zwar das aus der Pascal-Familie stammende Oberon verwendet, aber ich kenne kein vergleichbar umfassendes Werk. All das, was sich beispielsweise in Java hinter `native`-Methoden versteckt, können Sie hier sehen. Im Text werden ergänzend die grundlegenden Entscheidungen erläutert und kommentiert. Allein das ist überaus informativ.

Project Oberon ist nichts für einen verregneten Sonntagnachmittag, eher für den Aufenthalt auf einer einsamen Insel. Aber wenn Sie der Typ sind, der am liebsten alles selbst aus Einzelteilen zusammenbaut, weil Sie ganz genau wissen wollen, wie es im Detail wirklich funktioniert, dann ist das genau das Richtige für Sie.

Dummies Junior – die frechen »... für Dummies« für interessierte Kids und Jugendliche

- » Projekte zum Ausprobieren, Programmieren und Experimentieren
- » Mit pädagogischem Konzept
- » Viele Abbildungen
- » Verständliche Texte
- » In Workshops erprobt

C. Ermel, N. Rosenfeld

Spaß mit Elektronik für Dummies Junior

1. Auflage 2020 **ISBN:** 978-3-527-71705-7

Ca. 192 Seiten

Format: 176 mm x 240 mm

Ladenpreis: ca. 15,– €*

In diesem Buch lernst du, Schaltungen für coole Gadgets aufzubauen: eine Glückwunschkarte, die leuchtet, eine blinkende Weihnachtsbaumkugel, einen klingenden Draht und anderes mehr.

M. Ponce Kärgel

Hörspiel und Podcast selber machen für Dummies Junior

1. Auflage 2020 **ISBN:** 978-3-527-71704-0

Ca. 176 Seiten

Format: 176 mm x 240 mm

Ladenpreis: ca. 15,– €*

Sein eigener Produzent sein, wer will das nicht? Steig ein in die Welt der Hörspiele und Podcasts. Dieses Buch zeigt dir Schritt für Schritt, wie du deine eigenen Audiobeiträge produzieren und veröffentlichen kannst.

U. Schmid, K. Weitz, M. Siebers

Künstliche Intelligenz selber programmieren für Dummies Junior

1. Auflage 2019 **ISBN:** 978-3-527-71573-2

134 Seiten

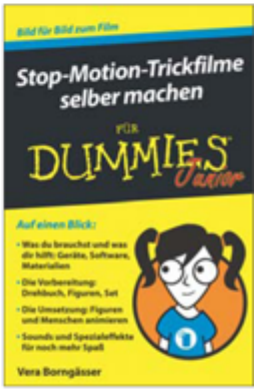
Format: 140 mm x 216 mm

Ladenpreis: 12,99 €*

Mit diesem Buch verstehst du, was Künstliche Intelligenz ist. Du findest heraus, ob es Künstliche Intelligenz bereits gibt. Und das Beste: Mit Hilfe kleiner Programme erkennst du, wie durch Computerprogramme Künstliche Intelligenz entsteht

V. Borngässer

Stop-Motion-Trickfilme selber machen für Dummies Junior



1. Auflage 2018 **ISBN:** 978-3-527-71484-1

144 Seiten

Format: 140 mm x 216 mm

Ladenpreis: 11,99 €*

Schritt für Schritt zum eigenen Stop-Motion-Video mit der richtigen Beleuchtung, krassen Geräuschen und Spezialeffekten. Hier erfährst du, wie es geht.

N. Bergner, Th. Leonhardt

Eigene Apps programmieren für Dummies Junior

2. Auflage 2019 **ISBN:** 978-3-527-71596-1

136 Seiten

Format: 140 mm x 216 mm

Ladenpreis: 11,99 €*

Hast du Lust, deine eigene App zu programmieren, die dann auch wirklich auf einem Android-Smartphone läuft? Mit dem kostenlosen App Inventor ist das gar nicht schwer.

* Der €-Preis gilt nur für Deutschland. Preisänderungen und Irrtümer vorbehalten.

Stichwortverzeichnis

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

V

W

Y

Diese Bücher könnten Sie auch interessieren

M. Schlich

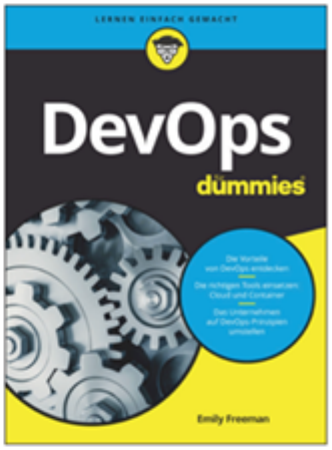
Softwaretesten nach ISTQB für Dummies

1. Auflage 2019 **ISBN:** 978-3-527-71518-3
360 Seiten
Format: 176 mm x 240 mm
Ladenpreis: 26,99 €*

Dieses Buch liefert Ihnen das notwendige Handwerkszeug zum Softwaretesten und die notwendige Sicherheit zum Bestehen des ISTQB® Certified Tester Foundation Level.

E. Freeman

DevOps für Dummies



1. Auflage 2019 **ISBN:** 978-3-527-71624-1
328 Seiten
Format: 176 mm x 240 mm
Ladenpreis: 26,99 €*

Finden Sie mit diesem Buch den Einstieg in DevOps: Wie Sie DevOps-Arbeitsweisen praktisch implementieren, wo die Hürden sind und wie Sie sie überwinden können.

M. C. Layton und S. J. Ostermiller

Agiles Projektmanagement für Dummies

1. Auflage 2018 **ISBN:** 978-3-527-71476-6
419 Seiten
Format: 176 mm x 240 mm
Ladenpreis: 24,99 €*

Dieses Buch ermöglicht es Ihnen, Projekte flexibel zu planen, auf Veränderungen angemessen zu reagieren und schnellere und bessere Ergebnisse zu erzielen.

M. C. Layton

Scrum für Dummies

2. Auflage 2019 **ISBN:** 978-3-527-71598-5
408 Seiten
Format: 176 mm x 240 mm
Ladenpreis: 24,99 €*

Verstehen Sie zunächst die grundlegenden Konzepte von Scrum und die dazugehörige Terminologie. Lernen Sie dann das gesamte Framework kennen. Erfahren Sie, wie Sie Scrum in den unterschiedlichsten Situationen anwenden können.

B. Burd

Java für Dummies

7. Auflage 2017 **ISBN:** 978-3-527-71364-6
468 Seiten
Format: 176 mm x 240 mm
Ladenpreis: 19,99 €*

Auf *Java für Dummies* können Sie sich jederzeit verlassen: Hier finden Java-Programmierer – Einsteiger und solche mit Erfahrung – alles, was sie wissen müssen.

A. Willemer

Java Alles-in-einem-Band für Dummies

1. Auflage 2018 **ISBN:** 978-3-527-71450-6
837 Seiten
Format: 176 mm x 240 mm
Ladenpreis: 29,99 €*

Java für Dummies Alles-in-einem-Band ist ein umfasssender Lernbegleiter und ein vollständiges Nachschlagewerk: Dieses Buch ist für alle, die tief in die Java-Programmierung einsteigen möchten.

*Europreise nur gültig für Deutschland. Preisänderungen und Irrtümer vorbehalten.

WILEY END USER LICENSE AGREEMENT

Besuchen Sie www.wiley.com/go/eula, um Wiley's E-Book-EULA einzusehen.