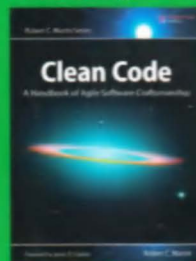


Robert C. Martin

Clean Code

Refactoring, Patterns, Testen
und Techniken für sauberen Code

Deutsche Ausgabe von:



Kommentare, Formatierung,
Strukturierung

Fehler-Handling und Unit-Tests

Zahlreiche Fallstudien,
Best Practices, Heuristiken
und Code Smells

Robert C. Martin

Clean Code

**Refactoring, Patterns, Testen und Techniken
für sauberen Code**

Unter Mitarbeit von:

Michael C. Feathers, Timothy R. Ottinger,
Jeffrey J. Langr, Brett L. Schuchert,
James W. Grenning, Kevin Dean Wampler
Object Mentor, Inc.

Übersetzt aus dem Amerikanischen
von Reinhard Engel



mitp

Bibliografische Information der Deutschen Nationalbibliothek

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

ISBN 978-3-8266-5548-7

1. Auflage 2009

Alle Rechte, auch die der Übersetzung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder einem anderen Verfahren) ohne schriftliche Genehmigung des Verlages reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden. Der Verlag übernimmt keine Gewähr für die Funktion einzelner Programme oder von Teilen derselben. Insbesondere übernimmt er keinerlei Haftung für eventuelle aus dem Gebrauch resultierende Folgeschäden.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Authorized translation from the English language edition, entitled CLEAN CODE: A HANDBOOK OF AGILE SOFTWARE CRAFTSMANSHIP, 1st Edition, 0132350882 by MARTIN, ROBERT C., published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2009 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. GERMAN language edition published by mitp-Verlag, VERLAGSGRUPPE HÜTHIG JEHLE REHM GMBH, Copyright © 2009.

Printed in Austria

© Copyright 2009 by mitp-Verlag

Verlagsgruppe Hüthig Jehle Rehm GmbH

Heidelberg, München, Landsberg, Frechen, Hamburg

www.it-fachportal.de

Lektorat: Sabine Schulz

Korrektur: Petra Heubach-Erdmann

Satz: III-satz, Husby, www.drei-satz.de

Inhaltsverzeichnis

	Vorwort.	15
	Einführung.	21
I	Sauberer Code	25
I.1	Code, Code und nochmals Code	26
I.2	Schlechter Code	27
I.3	Die Lebenszykluskosten eines Chaos	28
	Das große Redesign in den Wolken	29
	Einstellung	30
	Das grundlegende Problem	31
	Sauberen Code schreiben – eine Kunst?	31
	Was ist sauberer Code?	32
I.4	Denkschulen	40
I.5	Wir sind Autoren	41
I.6	Die Pfadfinder-Regel	43
I.7	Vorläufer und Prinzipien	43
I.8	Zusammenfassung	43
2	Aussagekräftige Namen	45
2.1	Einführung.	45
2.2	Zweckbeschreibende Namen wählen	45
2.3	Fehlinformationen vermeiden	47
2.4	Unterschiede deutlich machen.	49
2.5	Aussprechbare Namen verwenden	50
2.6	Suchbare Namen verwenden	51
2.7	Codierungen vermeiden	52
	Ungarische Notation	52
	Member-Präfixe	53
	Interfaces und Implementierungen.	54
2.8	Mentale Mappings vermeiden	54
2.9	Klassennamen	55
2.10	Methodennamen	55

2.11	Vermeiden Sie humorige Namen	55
2.12	Wählen Sie ein Wort pro Konzept	56
2.13	Keine Wortspiele	56
2.14	Namen der Lösungsdomäne verwenden	57
2.15	Namen der Problemdomäne verwenden	57
2.16	Bedeutungsvollen Kontext hinzufügen	57
2.17	Keinen überflüssigen Kontext hinzufügen	60
2.18	Abschließende Worte	60
3	Funktionen	61
3.1	Klein!	64
	Blöcke und Einrückungen	65
3.2	Eine Aufgabe erfüllen	65
	Abschnitte innerhalb von Funktionen	66
3.3	Eine Abstraktionsebene pro Funktion	67
	Code Top-down lesen: die Stepdwn-Regel	67
3.4	Switch-Anweisungen	68
3.5	Beschreibende Namen verwenden	70
3.6	Funktionsargumente	71
	Gebräuchliche monadische Formen	72
	Flag-Argumente	72
	Dyadische Funktionen	73
	Triaden	73
	Argument-Objekte	74
	Argument-Listen	74
	Verben und Schlüsselwörter	75
3.7	Nebeneffekte vermeiden	75
	Output-Argumente	76
3.8	Anweisung und Abfrage trennen	77
3.9	Ausnahmen sind besser als Fehler-Codes	77
	Try/Catch-Blöcke extrahieren	78
	Fehler-Verarbeitung ist eine Aufgabe	79
	Der Abhängigkeitsmagnet Error.java	79
3.10	Don't Repeat Yourself	80
3.11	Strukturierte Programmierung	80
3.12	Wie schreibt man solche Funktionen?	81
3.13	Zusammenfassung	81
3.14	SetupTeardownIncluder	82

4	Kommentare	85
4.1	Kommentare sind kein Ersatz für schlechten Code	86
4.2	Erklären Sie im und durch den Code	87
4.3	Gute Kommentare	87
	Juristische Kommentare	87
	Informierende Kommentare	88
	Erklärung der Absicht	88
	Klarstellungen	89
	Warnungen vor Konsequenzen	90
	TODO-Kommentare	91
	Verstärkung	91
	Javadocs in öffentlichen APIs	92
4.4	Schlechte Kommentare	92
	Geraune	92
	Redundante Kommentare	93
	Irreführende Kommentare	95
	Vorgeschriebene Kommentare	96
	Tagebuch-Kommentare	96
	Geschwätz	97
	Beängstigendes Geschwätz	99
	Verwenden Sie keinen Kommentar, wenn Sie eine Funktion oder eine Variable verwenden können	100
	Positionsbezeichner	100
	Kommentare hinter schließenden Klammern	101
	Zuschreibungen und Nebenbemerkungen	101
	Auskommentierter Code	102
	HTML-Kommentare	102
	Nicht-lokale Informationen	103
	Zu viele Informationen	104
	Unklarer Zusammenhang	104
	Funktions-Header	104
	Javadocs in nicht-öffentlichem Code	105
	Beispiel	105
5	Formatierung	109
5.1	Der Zweck der Formatierung	109
5.2	Vertikale Formatierung	110
	Die Zeitungs-Metapher	111
	Vertikale Offenheit zwischen Konzepten	112

	Vertikale Dichte	113
	Vertikaler Abstand	114
	Vertikale Anordnung	119
5.3	Horizontale Formatierung	119
	Horizontale Offenheit und Dichte	120
	Horizontale Ausrichtung	121
	Einrückung	123
	Dummy-Bereiche	124
5.4	Team-Regeln	125
5.5	Uncle Bobs Formatierungsregeln	125
6	Objekte und Datenstrukturen	129
6.1	Datenabstraktion	129
6.2	Daten/Objekt-Anti-Symmetrie	131
6.3	Das Law of Demeter	133
	Zugkatastrophe	134
	Hybride	135
	Struktur verbergen	135
6.4	Datentransfer-Objekte	136
	Active Record	137
6.5	Zusammenfassung	138
7	Fehler-Handling	139
7.1	Ausnahmen statt Rückgabe-Codes	139
7.2	Try-Catch-Finally-Anweisungen zuerst schreiben	141
7.3	Unchecked Exceptions	143
7.4	Ausnahmen mit Kontext auslösen	144
7.5	Definieren Sie Exception-Klassen mit Blick auf die Anforderungen des Aufrufers	144
7.6	Den normalen Ablauf definieren	146
7.7	Keine Null zurückgeben	147
7.8	Keine Null übergeben	148
7.9	Zusammenfassung	150
8	Grenzen	151
8.1	Mit Drittanbieter-Code arbeiten	151
8.2	Grenzen erforschen und kennen lernen	154
8.3	log4j kennen lernen	154
8.4	Lern-Tests sind besser als kostenlos	156

8.5	Code verwenden, der noch nicht existiert	157
8.6	Saubere Grenzen	158
9	Unit-Tests	159
9.1	Die drei Gesetze der TDD	160
9.2	Tests sauber halten	161
	Tests ermöglichen die -heiten und -keiten	162
9.3	Saubere Tests	163
	Domänenspezifische Testsprache	166
	Ein Doppelstandard	166
9.4	Ein assert pro Test	168
	Ein Konzept pro Test	170
9.5	F.I.R.S.T.	171
9.6	Zusammenfassung	172
10	Klassen	173
10.1	Klassenaufbau	173
	Einkapselung	174
10.2	Klassen sollten klein sein!	174
	Fünf Methoden sind nicht zu viel, oder?	176
	Das Single-Responsibility-Prinzip	176
	Kohäsion	178
	Kohäsion zu erhalten, führt zu vielen kleinen Klassen	179
10.3	Änderungen einplanen	185
	Änderungen isolieren	188
11	Systeme	191
11.1	Wie baut man eine Stadt?	191
11.2	Konstruktion und Anwendung eines Systems trennen	192
	Trennung in main	193
	Factories	194
	Dependency Injection	195
11.3	Aufwärtsskalierung	196
	Cross-Cutting Concerns	199
11.4	Java-Proxies	200
11.5	Reine Java-AOP-Frameworks	202
11.6	AspectJ-Aspekte	205
11.7	Die Systemarchitektur testen	205
11.8	Die Entscheidungsfindung optimieren	207

II.9	Standards weise anwenden, wenn sie nachweisbar einen Mehrwert bieten	207
II.10	Systeme brauchen domänenspezifische Sprachen	208
II.11	Zusammenfassung	208
12	Emergenz	209
12.1	Saubere Software durch emergentes Design.	209
12.2	Einfache Design-Regel 1: Alle Tests bestehen	210
12.3	Einfache Design-Regeln 2–4: Refactoring	211
12.4	Keine Duplizierung	211
12.5	Ausdrucksstärke.	214
12.6	Minimale Klassen und Methoden	215
12.7	Zusammenfassung	215
13	Nebenläufigkeit	217
13.1	Warum Nebenläufigkeit?	218
	Mythen und falsche Vorstellungen	219
13.2	Herausforderungen	220
13.3	Prinzipien einer defensiven Nebenläufigkeitsprogrammierung	221
	Single-Responsibility-Prinzip	221
	Korollar: Beschränken Sie den Gültigkeitsbereich von Daten	221
	Korollar: Arbeiten Sie mit Kopien der Daten.	222
	Korollar: Threads sollten voneinander so unabhängig wie möglich sein	222
13.4	Lernen Sie Ihre Library kennen	223
	Thread-sichere Collections	223
13.5	Lernen Sie Ihre Ausführungsmodelle kennen	224
	Erzeuger-Verbraucher	224
	Leser-Schreiber	225
	Philosophenproblem	225
13.6	Achten Sie auf Abhängigkeiten zwischen synchronisierten Methoden	226
13.7	Halten Sie synchronisierte Abschnitte klein	226
13.8	Korrekten Shutdown-Code zu schreiben, ist schwer	227
13.9	Threaded-Code testen	227
	Behandeln Sie gelegentlich auftretende Fehler als potenzielle Threading-Probleme	228
	Bringen Sie erst den Nonthreaded-Code zum Laufen	228

	Machen Sie Ihren Threaded-Code pluggable	229
	Schreiben Sie anpassbaren Threaded-Code	229
	Den Code mit mehr Threads als Prozessoren ausführen	229
	Den Code auf verschiedenen Plattformen ausführen	229
	Code-Scheitern durch Instrumentierung provozieren	230
	Manuelle Codierung	230
	Automatisiert	231
13.10	Zusammenfassung	232
14	Schrittweise Verfeinerung	235
14.1	Args-Implementierung	236
	Wie habe ich dies gemacht?	243
14.2	Args: der Rohentwurf	243
	Deshalb hörte ich auf	256
	Über inkrementelle Entwicklung	256
14.3	String-Argumente	258
14.4	Zusammenfassung	300
15	JUnit im Detail	301
15.1	Das JUnit-Framework	301
15.2	Zusammenfassung	316
16	Refactoring von SerialDate	317
16.1	Zunächst bring es zum Laufen!	318
16.2	Dann mach es richtig!	320
16.3	Zusammenfassung	336
17	Smells und Heuristiken	337
17.1	Kommentare	337
	C1: Ungeeignete Informationen	337
	C2: Überholte Kommentare	338
	C3: Redundante Kommentare	338
	C4: Schlecht geschriebene Kommentare	338
	C5: Auskommentierter Code	339
17.2	Umgebung	339
	E1: Ein Build erfordert mehr als einen Schritt	339
	E2: Tests erfordern mehr als einen Schritt	339
17.3	Funktionen	340
	F1: Zu viele Argumente	340
	F2: Output-Argumente	340

	F3: Flag-Argumente	340
	F4: Tote Funktionen	340
17.4	Allgemein	340
	G1: Mehrere Sprachen in einer Quelldatei	340
	G2: Offensichtliches Verhalten ist nicht implementiert.	341
	G3: Falsches Verhalten an den Grenzen	341
	G4: Übergangene Sicherungen	341
	G5: Duplizierung	342
	G6: Auf der falschen Abstraktionsebene codieren	342
	G7: Basisklasse hängt von abgeleiteten Klassen ab.	344
	G8: Zu viele Informationen	344
	G9: Toter Code.	345
	G10: Vertikale Trennung.	345
	G11: Inkonsistenz.	345
	G12: Müll	346
	G13: Künstliche Kopplung.	346
	G14: Funktionsneid	346
	G15: Selektor-Argumente	347
	G16: Verdeckte Absicht.	348
	G17: Falsche Zuständigkeit.	349
	G18: Fälschlich als statisch deklarierte Methoden	350
	G19: Aussagekräftige Variablen verwenden	350
	G20: Funktionsname sollte die Aktion ausdrücken	351
	G21: Den Algorithmus verstehen	351
	G22: Logische Abhängigkeiten in physische umwandeln	352
	G23: Polymorphismus statt If/Else oder Switch/Case verwenden	353
	G24: Konventionen beachten	354
	G25: Magische Zahlen durch benannte Konstanten ersetzen	354
	G26: Präzise sein.	355
	G27: Struktur ist wichtiger als Konvention	356
	G28: Bedingungen einkapseln	356
	G29: Negative Bedingungen vermeiden	356
	G30: Eine Aufgabe pro Funktion!.	356
	G31: Verborgene zeitliche Kopplungen	357
	G32: Keine Willkür	358
	G33: Grenzbedingungen einkapseln	359
	G34: In Funktionen nur eine Abstraktionsebene tiefer gehen	359

	G35: Konfigurierbare Daten hoch ansiedeln	361
	G36: Transitive Navigation vermeiden	362
17.5	Java	362
	J1: Lange Importlisten durch Platzhalter vermeiden	362
	J2: Keine Konstanten vererben	363
	J3: Konstanten im Gegensatz zu Enums	364
17.6	Namen	366
	N1: Deskriptive Namen wählen	366
	N2: Namen sollten der Abstraktionsebene entsprechen	367
	N3: Möglichst die Standardnomenklatur verwenden	368
	N4: Eindeutige Namen	368
	N5: Lange Namen für große Geltungsbereiche	369
	N6: Codierungen vermeiden	369
	N7: Namen sollten Nebeneffekte beschreiben	370
17.7	Tests	370
	T1: Unzureichende Tests	370
	T2: Ein Coverage-Tool verwenden	370
	T3: Triviale Tests nicht überspringen	370
	T4: Ein ignorierte Test zeigt eine Mehrdeutigkeit auf	371
	T5: Grenzbedingungen testen	371
	T6: Bei Bugs die Nachbarschaft gründlich testen.	371
	T7: Das Muster des Scheiterns zur Diagnose nutzen.	371
	T8: Hinweise durch Coverage-Patterns	371
	T9: Tests sollten schnell sein	371
17.8	Zusammenfassung	372
A	Nebenläufigkeit II	373
A.1	Client/Server-Beispiel	373
	Der Server	373
	Threading hinzufügen	375
	Server-Beobachtungen	375
	Zusammenfassung	377
A.2	Mögliche Ausführungspfade	377
	Anzahl der Pfade	378
	Tiefer graben	380
	Zusammenfassung	383
A.3	Lernen Sie Ihre Library kennen	383
	Executor Framework	383

	Nicht blockierende Lösungen	384
	Nicht thread-sichere Klassen	385
A.4	Abhängigkeiten zwischen Methoden können nebenläufigen Code beschädigen	387
	Das Scheitern tolerieren	388
	Clientbasiertes Locking	388
	Serverbasiertes Locking	390
A.5	Den Durchsatz verbessern	391
	Single-Thread-Berechnung des Durchsatzes	392
	Multithread-Berechnung des Durchsatzes	393
A.6	Deadlock	393
	Gegenseitiger Ausschluss	395
	Sperren & warten	395
	Keine präemptive Aktion	395
	Zirkuläres Warten	395
	Den gegenseitigen Ausschluss aufheben	396
	Das Sperren & Warten aufheben	396
	Die Präemption umgehen	397
	Das zirkuläre Warten umgehen	397
A.7	Multithreaded-Code testen	398
A.8	Threadbasierten Code mit Tools testen	401
A.9	Zusammenfassung	402
A.10	Tutorial: kompletter Beispielcode	402
	Client/Server ohne Threads	402
	Client/Server mit Threads	406
B	org.jfree.date.SerialDate	407
C	Literaturverweise	463
	Epilog	465
	Stichwortverzeichnis	467

Vorwort

Bei uns in Dänemark zählt Ga-Jol zu den beliebtesten Süßigkeiten. Sein starker Lakritzduft ist ein perfektes Mittel gegen unser feuchtes und oft kühles Wetter. Der Charme, den Ga-Jol für uns Dänen entfaltet, liegt auch an den weisen oder geistreichen Sprüchen, die auf dem Deckel jeder Packung abgedruckt sind. Ich habe mir heute Morgen eine Doppelpackung dieser Köstlichkeit gekauft und fand darauf das folgende alte dänische Sprichwort:

Ærlighed i små ting er ikke nogen lille ting.

Ehrlichkeit in kleinen Dingen ist kein kleines Ding. Dies war ein gutes Omen. Es passte zu dem, was ich hier sagen wollte. Kleine Dinge spielen eine Rolle. In diesem Buch geht es um bescheidene Belange, deren Wert dennoch beträchtlich ist.

Gott steckt in den Details, sagte der Architekt Ludwig Mies van der Rohe. Dieses Zitat erinnert an zeitgenössische Auseinandersetzungen über die Rolle der Architektur bei der Software-Entwicklung und insbesondere in der Agilen Welt. Bob und ich führen gelegentlich heiße Diskussionen über dieses Thema. Und ja, Mies van der Rohe war sehr an der Nützlichkeit und der Zeitlosigkeit der Formen des Bauens interessiert, auf denen großartige Architektur basiert. Andererseits wählte er auch persönlich jeden Türknopf für jedes Haus aus, das er entworfen hatte. Warum? Weil kleine Aufgaben eine Rolle spielen.

Bei unserer laufenden »Debatte« über TDD haben Bob und ich festgestellt, dass wir beide der Auffassung sind, dass die Software-Architektur eine wichtige Rolle bei der Entwicklung spielt, obwohl wir wahrscheinlich verschiedene Vorstellungen davon haben, was genau das bedeutet. Doch solche Differenzen sind relativ unwichtig, weil wir davon ausgehen können, dass verantwortungsbewusste Profis am Anfang eines Projekts eine gewisse Zeit über seinen Ablauf und seine Planung nachdenken. Die Vorstellungen der späten 1990er-Jahre, dass Design nur durch Tests und den Code vorangetrieben werden könnte, sind längst passé. Doch die Aufmerksamkeit im Detail ist ein noch wesentlicherer Aspekt der Professionalität als Visionen im Großen. Erstens: Es ist die Übung im Kleinen, mit der Profis ihr Können und ihr Selbstvertrauen entwickeln, sich an Größeres heranzuwagen. Zweitens: Die kleinste Nachlässigkeit bei der Konstruktion, die Tür, die nicht richtig schließt, die missratene Kachel auf dem Fußboden oder sogar ein unordentlicher Schreibtisch können den Charme des größeren Ganzen mit einem Schlag ruinieren. Darum geht es bei sauberem Code.

Dennoch bleibt die Architektur nur eine Metapher für die Software-Entwicklung und insbesondere die Phase der Software, in der das anfängliche Produkt etwa in derselben Weise entsteht, wie ein Architekt ein neues Gebäude hervorbringt. In der heutigen Zeit von *Scrum* und *Agile* geht es hauptsächlich darum, ein Produkt schnell auf den Markt zu bringen. Die Fabrik soll mit höchster Kapazität Software produzieren. Nur: Hier geht es um menschliche Fabriken, denkende und führende Programmierer, die eine Aufgabenliste abarbeiten oder sich bemühen, anhand von Benutzer-Stories ein brauchbares Produkt zu erstellen. Ein solches Denken wird von der Metapher der Fabrik dominiert. Die Produktionsaspekte der Herstellung von Autos in Japan, also einer von Fließbändern dominierten Welt, haben viele Ideen von *Scrum* inspiriert.

Doch selbst in der Automobilindustrie wird der Hauptteil der Arbeit nicht bei der Produktion, sondern bei der Wartung geleistet – oder bei dem Bemühen, sie zu vermeiden. Bei der Software werden die 80 oder mehr Prozent unserer Tätigkeit anheimelnd als »Wartung« bezeichnet, ein beschönigendes Wort für »Reparatur«. Statt uns also auf typische westliche Weise auf die *Produktion* guter Software zu konzentrieren, sollten wir anfangen, eher wie ein Hausmeister bei der Gebäudeinstandhaltung oder wie ein Kfz-Mechaniker bei der Reparatur von Autos zu denken. Was haben japanische Managementweisheiten dazu zu sagen?

Etwa 1951 erschien ein Qualitätsansatz namens Total Productive Maintenance (TPM; »Umfassendes Management des Produktionsprozesses«, der Ausdruck wird nicht ins Deutsche übersetzt) in der japanischen Szene. Er konzentrierte sich weniger auf die Produktion, sondern mehr auf die Wartung. Eine der fünf Säulen des TPM ist der Satz der so genannten 5S-Prinzipien. 5S bezieht sich auf einen Satz von Disziplinen oder Tätigkeitsbereichen. Tatsächlich verkörpern diese 5S-Prinzipien die Bedeutung von *Lean* – einem anderen Modewort in westlichen Produktionskreisen, das zunehmend auch in der Software-Entwicklung verwendet wird. Diese Prinzipien sind nicht optional. Wie Uncle Bob auf der Titelseite sagt, erfordert die Software-Praxis eine gewisse Disziplin: Fokus, Aufmerksamkeit und Denken. Es geht nicht immer nur darum, aktiv zu sein und die Fabrikausrüstung mit der optimalen Geschwindigkeit zu betreiben. Die 5S-Philosophie umfasst die folgenden Konzepte:

- *Seiri* oder Organisation (Eselsbrücke: »sortieren«). Zu wissen, wo sich Dinge befinden, ist erfolgsentscheidend. Dazu zählen zum Beispiel Ansätze wie das Vergeben geeigneter Namen. Wenn Sie meinen, die Benennung Ihrer Bezeichner sei nicht wichtig, sollten Sie auf jeden Fall die folgenden Kapitel lesen.
- *Seiton* oder Ordentlichkeit (Eselsbrücke: »aufräumen«). Ein altes Sprichwort sagt: *Ein Platz für alles, und alles an seinem Platz*. Ein Code-Abschnitt sollte da stehen, wo Sie ihn zu finden erwarten; und falls er nicht da steht, sollten Sie ein Refactoring von Ihrem Code vornehmen, so dass er danach an der erwarteten Stelle steht.

- *Seiso* oder Sauberkeit (Eselsbrücke: »wienern«). Sorgen Sie dafür, dass der Arbeitsplatz frei von herumhängenden Drähten, Müllspuren, Einzelteilen und Abfall ist. Was sagen die Autoren hier über die Vermüllung Ihres Codes mit Kommentaren und auskommentierten Codezeilen, die den Verlauf der Entwicklung oder Wünsche für die Zukunft wiedergeben? Weg damit.
- *Seiketsu* oder Standardisierung. Die kommt zu einem Konsens darüber, wie der Arbeitsplatz sauber gehalten werden soll. Hat dieses Buch etwas über einen konsistenten Codierstil und gemeinsam in der Gruppe befolgte Praktiken zu sagen? Wo kommen diese Standards her? Lesen Sie weiter.
- *Shutsuke* oder Disziplin (*Selbst*-disziplin). Dies bedeutet, die Disziplin aufzubringen, den Praktiken zu folgen, seine Arbeit regelmäßig zu überdenken und bereit zu sein, sich zu ändern.

Wenn Sie die Herausforderung – jawohl, die Herausforderung – annehmen, dieses Buch zu lesen und anzuwenden, werden Sie den letzten Punkt verstehen und schätzen lernen. Hier kommen wir schließlich zu den Wurzeln einer verantwortlichen professionellen Einstellung in einem Beruf, der sich um den Lebenszyklus eines Produktes kümmern sollte. So wie Automobile und andere Maschinen unter TPM vorbeugend gewartet werden, sollte eine Wartung im Falle eines Zusammenbruchs – also darauf zu warten, dass die Bugs sich zeigen – die Ausnahme sein. Stattdessen sollten Sie eine Ebene höher ansetzen: Inspizieren Sie die Maschinen jeden Tag und tauschen Sie Verschleißteile aus, bevor sie kaputtgehen, oder lassen Sie den sprichwörtlichen Ölwechsel vornehmen, um die Abnutzung des Motors möglichst zu verringern. Für Ihren Code bedeutet das: Nehmen Sie ein gnadenloses Refactoring vor. Sie können mit der Verbesserung noch eine Ebene höher ansetzen, wie es die TPM-Bewegung innovativ vor über 50 Jahren vorgemacht hat: Bauen Sie Maschinen, die von vornherein wartungsfreundlicher sind. Code lesbar zu machen, ist genauso wichtig, wie ihn ausführbar zu machen. Die ultimative Praxis, die in TPM-Kreisen etwa 1960 eingeführt wurde, besteht darin, die alten Maschinen vorbeugend durch vollkommen neue zu ersetzen. Wie Fred Brooks anmahnt, sollten wir wahrscheinlich Hauptteile unserer Software etwa alle sieben Jahre von Grund auf erneuern, um die schleichende Ansammlung von *Cruft* (Jargon: Müll, Staub, Unrat) zu beseitigen. Vielleicht sollten wir die von Brooks genannte Zeitspanne drastisch reduzieren und nicht von Jahren, sondern von Wochen, Tagen oder gar Stunden sprechen. Denn dort finden sich die Details.

Details haben eine mächtige Wirkungskraft. Dennoch hat dieser Ansatz etwas Bescheidenes und Grundsätzliches, so wie wir es vielleicht stereotyp von einem Ansatz erwarten, der japanische Wurzeln für sich in Anspruch nimmt. Aber dies ist nicht nur eine köstliche Art, das Leben zu sehen. Auch die westliche Volksweisheit enthält zahlreiche ähnliche Sprichwörter. Das Seiton-Zitat von oben floss auch aus der Feder eines Priesters in Ohio, der Ordentlichkeit buchstäblich »als Gegenmittel für jede Art von Bösem« sah. Was ist mit *Seiso*? *Sauberkeit kommt gleich nach Gött-*

lichkeit. So schön ein Haus auch sein mag, ein unordentlicher Tisch raubt ihm seinen ganzen Glanz. Was bedeutet Shutsuke in diesen kleinen Dingen? *Wer an wenig glaubt, glaubt an viel*. Wie wär's damit, das Refactoring des Codes rechtzeitig durchzuführen, um seine Position für folgende »große« Entscheidungen zu stärken, als die Aufgabe aufzuschieben? *Ein Stich zur rechten Zeit erspart dir neun weitere*. *Wer früh kommt, mahlt zuerst*. *Was du heute kannst besorgen, das verschiebe nicht auf morgen*. (Dies war die ursprüngliche Bedeutung des Ausdrucks »der letzte tragbare Moment« bei Lean, bevor er in die Hände von Software-Beratern fiel.) Was ist mit der Ordnung eines Arbeitsplatzes im Kleinen im Vergleich zum großen Ganzen? *Auch die größte Eiche wächst aus einer kleinen Eichel*. Oder wie ist es damit, einfache vorbeugende Arbeiten in den Alltag einzubauen? *Vorsicht ist besser als Nachsicht*. *Ein Apfel am Tag hält den Doktor fern*. Sauberer Code anerkennt die verwurzelte Weisheit unserer Kultur im Allgemeinen, oder wie sie einmal war, oder wie sie sein sollte, oder wie sie sein *könnte*, indem er den Details die schuldige Aufmerksamkeit schenkt.

Selbst in der Literatur über große Architektur greifen Autoren auf Sprichwörter über die Bedeutung von Details zurück. Denken Sie an die Türgriffe von Mies van der Rohe. Das ist Seiri. Das bedeutet, sich um jeden Variablennamen zu kümmern. Sie sollten Variablennamen mit derselben Sorgfalt auswählen wie den Namen eines erstgeborenen Kindes.

Jeder Hausbesitzer weiß, dass eine solche Pflege und fortwährende Verbesserung niemals zu Ende ist. Der Architekt Christopher Alexander – Vater der Patterns und Pattern-Sprachen – betrachtet jeden Design-Akt selbst als einen kleinen, lokalen Akt der Reparatur. Und er betrachtet die Anwendung des Könnens auf feine Strukturen als den einzigen legitimen Arbeitsbereich des Architekten; die größeren Formen können den Patterns und deren Anwendung durch die Bewohner überlassen werden. Design geht immer weiter. Es betrifft nicht nur den Anbau neuer Räumlichkeiten, sondern auch profanere Aufgaben wie ein neuer Anstrich, das Ersetzen eines abgelaufenen Teppichs oder die Modernisierung der Spüle in der Küche. In den meisten Künsten werden ähnliche Überzeugungen vertreten. Bei unserer Suche nach anderen, die das Haus Gottes in den Details sehen, stießen wir beispielsweise auf den französischen Autor Gustav Flaubert aus dem 19. Jahrhundert. Der französische Dichter Paul Valéry sagt uns, ein Gedicht wäre niemals fertig und müsse laufend überarbeitet werden; mit dieser Arbeit aufzuhören wäre vergleichbar damit, dieses Gedicht zu verwerfen. Eine solche Besessenheit von Details ist allen Bemühungen gemeinsam, die auf Exzellenz, also auf herausragende Leistungen gerichtet sind. Also: Vielleicht gibt es hier wenig Neues, aber wenn Sie dieses Buch lesen, werden Sie herausgefordert, die guten Disziplinen wieder aufzunehmen, die Sie vor langer Zeit aus Apathie, dem Wunsch nach Spontaneität oder einfach deswegen aufgegeben haben, weil Sie »etwas anderes« machen wollten.

Leider betrachten wir solche Überlegungen normalerweise nicht als Grundbausteine der Kunst der Programmierung. Wir entlassen unseren Code früh aus unse-

rer Obhut, nicht, weil er fertig ist, sondern weil unser Wertesystem mehr auf die äußere Erscheinung als auf die Substanz des gelieferten Produkts gerichtet ist. Diese fehlende Aufmerksamkeit kommt uns letztendlich teuer zu stehen: Irgendwann machen sich die Defekte immer bemerkbar. Weder in akademischen Kreisen noch in der Industrie begibt sich die Forschung in die niedrigen Ebenen hinunter, Code sauber zu halten. Früher, als ich noch bei der Bell Labs Software Production Research Organization (also wirklich Produktion!) beschäftigt war, machten wir die beiläufige Entdeckung, dass ein konsistenter Stil beim Einrücken von Klammern der statistisch signifikanteste Indikator für eine geringe Fehlerhäufigkeit war. Wir wollen, dass die Architektur, die Programmiersprache oder irgendein anderes hoch angesiedeltes Konzept die Ursache für Qualität sein soll. Als Entwickler, deren vorgebliche Professionalität auf der Meisterung von Werkzeugen und abgehobenen Design-Methoden basiert, fühlen wir uns von dem Mehrwert beleidigt, den diese Maschinen aus der Fabrikhalle, pardon!, Codierer, erzielen, indem sie einfach einen bestimmten Stil der Einrückung von Klammern konsistent anwenden. Um mein eigenes Buch zu zitieren, das ich vor 17 Jahren geschrieben habe: Ein solcher Stil unterscheidet Exzellenz von bloßer Kompetenz. Die japanische Weltsicht versteht den entscheidenden Beitrag jedes gewöhnlichen Arbeiters und mehr noch, wie Entwicklungssysteme von den einfachen gewöhnlichen Aktionen dieser Arbeiter abhängen. Qualität ist das Ergebnis eine Million selbstloser Akte der Sorgfalt – nicht nur das einer großartigen Methode, die vom Himmel gefallen ist. Dass diese Akte einfach sind, bedeutet nicht, dass sie simplizistisch (einfältig) sind. Es bedeutet auch nicht, dass sie leicht sind. Dennoch sind sie der Stoff, aus dem die Größe und mehr noch die Schönheit menschlicher Anstrengungen gemacht ist. Wer diese Akte ignoriert, hat noch nicht sein volles menschliches Potenzial realisiert.

Natürlich befürworte ich immer noch, auch den größeren Rahmen zu betrachten und besonders den Wert von architektonischen Ansätzen zu berücksichtigen, die in einem tiefen Wissen sowohl des Problembereiches als auch der Software-Usability verwurzelt sind. Doch darum geht es in diesem Buch nicht – zumindest nicht vordergründig. Dieses Buch will eine subtilere Botschaft verbreiten, deren Wichtigkeit nicht unterschätzt werden sollte. Sie passt zu dem gegenwärtigen Credo wirklich codebasierter Entwickler wie Peter Sommerlad, Kevlin Henney und Giovanni Asproni. Ihre Mantras lauten: »Der Code ist das Design« und »Einfacher Code«. Während wir darauf achten müssen, dass die Schnittstelle das Programm ist und dass ihre Struktur viel über die Struktur unseres Programmes aussagt, ist es von erfolgsentscheidender Bedeutung, dass wir uns laufend mit der bescheidenen Einstellung identifizieren, dass das Design tatsächlich nur im Code existiert. Und während eine Überarbeitung in der Metapher der fabrikmäßigen Produktion zu höheren Kosten führt, führt sie beim Design zu einem Mehrwert. Wir sollten unseren Code als den wunderschönen Ausdruck edler Design-Anstrengungen betrachten – Design als Prozess, nicht als statischer Endpunkt. Nur im Code lassen sich letztlich die architektonischen Metriken der Kopplung und Kohäsion nachweisen. Wenn Larry Constantine Kopplung und Kohäsion beschreibt, spricht er von Code

– nicht von abgehobenen abstrakten Konzepten, die man vielleicht in UML findet. Richard Gabriel rät uns in seinem Essay *Abstraction Descant*, Abstraktion sei böse. Code ist anti-böse, und sauberer Code ist vielleicht göttlich.

Zurück zu meiner kleinen Packung Ga-Jol: Ich glaube, dass es wichtig ist, anzumerken, dass uns die dänische Weisheit nicht nur rät, unsere Aufmerksamkeit auf die kleinen Aufgaben zu lenken, sondern auch, bei kleinen Aufgaben ehrlich zu sein. Dies bedeutet, wir müssen dem Code gegenüber ehrlich sein, wir müssen unseren Kollegen gegenüber über den Zustand unseres Codes ehrlich sein und vor allem, wir müssen uns selbst gegenüber über den Zustand unseres Codes ehrlich sein. Haben wir unser Bestes gegeben, um den »Campingplatz sauberer zu hinterlassen, als wir ihn gefunden haben«? Haben wir das Refactoring des Codes erledigt, bevor wir ihn eingeeckelt haben? Dies sind keine nebensächlichen Belange, sondern Belange, die zum Kern agiler Werte gehören. Zum Beispiel empfiehlt *Scrum*, das Refactoring zu einem Bestandteil des Konzepts »Done« (»Fertig«) zu machen. Weder Architektur noch sauberer Code verlangt nach Perfektion, sondern nur nach Ehrlichkeit und dem Bemühen, unser Bestes zu geben. Irren ist menschlich, vergeben göttlich. Bei *Scrum* machen wir alles sichtbar. Wir zeigen unsere schmutzige Wäsche. Wir sind ehrlich über den Zustand unseres Codes, weil Code niemals perfekt ist. Wir werden menschlicher, werden würdiger, das Göttliche zu empfangen, und nähern uns der Größe in den Details.

In unserem Beruf brauchen wir verzweifelt alle Hilfe, die wir bekommen können. Wenn ein sauberer Fabrikboden die Anzahl der Unfälle reduziert und wohlgeordnete Werkzeugkästen die Produktivität verbessern, dann kann man dies nur befürworten. Was dieses Buch angeht: Es ist die beste pragmatische Anwendung von Lean-Prinzipien auf Software, die ich je in Druckform gesehen habe. Aber weniger hatte ich von dieser praktischen kleinen Gruppe denkender Individuen auch nicht erwartet, die sich nicht nur seit Jahren darum bemühen, immer besser zu werden, sondern auch ihr Wissen an die Branche weiterzugeben. Dieses Buch ist ein Ausdruck dieses Bemühens. Es hinterlässt die Welt ein wenig besser, als ich sie vorfand, bevor Uncle Bob mir das Manuskript schickte.

Doch genug von meinen abgehobenen Einsichten. Ich muss meinen Schreibtisch aufräumen.

James O. Coplien

Mørdrup, Dänemark

Einführung

The ONLY valid MEASUREMENT
OF CODE QUALITY: WTFs/minute

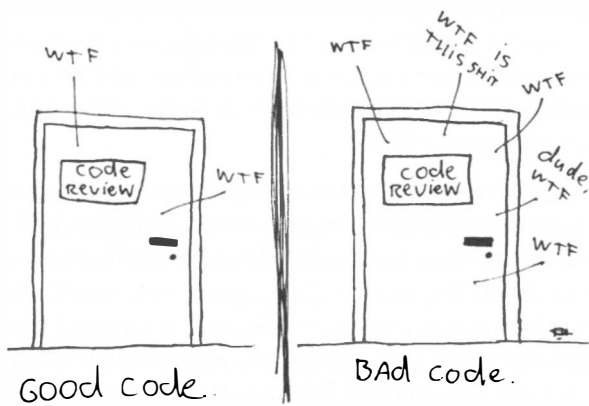


Abb. 1: Abdruck mit freundlicher Genehmigung von Thom Holwerda
(http://www.osnews.com/story/19266/WTFs_m)

Welche Tür repräsentiert Ihren Code? Welche Tür repräsentiert Ihr Team oder Ihr Unternehmen? Warum sind wir in diesem Raum? Geht es nur um einen normalen Code-Review oder haben wir eine Reihe schrecklicher Probleme gefunden, die kaum geringer als das Leben sind? Debuggen wir in Panik; brüten wir über Code, der unserer Meinung nach funktionieren sollte? Laufen uns die Kunden in Scharen davon und hängen uns die Manager im Nacken? Wie können wir dafür sorgen, dass wir hinter der *richtigen* Tür landen, wenn es heiß hergeht? Die Antwort ist: *Könnerschaft*.

Können zu erwerben, hat zwei Aspekte: Wissen und Arbeit. Sie müssen sich Prinzipien, Patterns, Techniken und Heuristiken aneignen, die jeder Fachmann kennt, und Sie müssen dieses Wissen in Ihre Finger, Ihre Augen und Ihren Bauch einprägen, indem Sie hart arbeiten und üben.

Ich kann Ihnen die physikalischen Gesetzmäßigkeiten des Fahrradfahrens vermitteln. Tatsächlich ist die entsprechende klassische Mathematik relativ einfach. Schwerkraft, Reibung, Drehmoment, Massenschwerpunkt usw. können auf weniger als einer Seite mit Gleichungen demonstriert werden. Mit diesen Formeln:

könnte ich Ihnen beweisen, dass Fahrradfahren praktisch ist, und Ihnen das gesamte Wissen vermitteln, das Sie brauchen, um es auszuüben. Doch was passiert beim ersten Mal, wenn Sie ein Fahrrad besteigen? Sie fallen runter.

Beim Codieren ist es genauso. Wir könnten alle »So-geht's«-Prinzipien von sauberem Code niederschreiben und dann darauf vertrauen, dass Sie die Arbeit erledigen. (Anders ausgedrückt: Wir könnten Sie einfach auf die Nase fallen lassen, wenn Sie das Fahrrad besteigen.) Doch was für eine Art von Lehrer wären wir dann, und welche Art von Schüler würde dann aus Ihnen werden?

So also nicht. So soll dieses Buch nicht funktionieren.

Sauberen Code schreiben zu lernen, ist harte Arbeit. Es erfordert mehr, als nur die Prinzipien und Patterns zu kennen. Sie müssen sich an dem Code abarbeiten. Sie müssen es selbst üben und aus Ihren Fehlern lernen. Sie müssen andere dabei beobachten, wie sie es ausprobieren, welche Fehler sie machen. Sie müssen erkennen, wo sie stolpern, und ihre Schritte nachvollziehen. Sie müssen sehen, welche Entscheidungen ihnen besonders schwerfallen und welchen Preis sie bezahlen müssen, wenn sie die falschen Entscheidungen treffen.

Sie sollten sich auf harte Arbeit einstellen, wenn Sie dieses Buch lesen. Dies ist kein Buch, das Ihnen »angenehme« Unterhaltung bietet und das Sie im Flugzeug lesen und vor der Landung beenden können. Dieses Buch fordert Sie auf, hart zu arbeiten. Um welche Art von Arbeit geht es dabei? Sie werden Code lesen – sehr viel Code. Und Sie werden aufgefordert, darüber nachzudenken, wo dieser Code richtig und wo er falsch ist. Sie werden aufgefordert, uns dabei zu folgen, wie wir Module zerpfücken und wieder zusammensetzen. Dies braucht Zeit und Mühe; aber wir sind der Ansicht, dass es sich für Sie lohnt.

Wir haben dieses Buch in drei Teile aufgeteilt. Die ersten Kapitel beschreiben die Prinzipien, Patterns und Techniken für das Schreiben von sauberem Code. Diese Kapitel enthalten eine ganze Menge Code, und es wird nicht einfach sein, ihn zu lesen. Sie bereiten Sie auf den zweiten Teil vor. Wenn Sie das Buch nach dem Lesen des ersten Abschnitts aus der Hand legen, alles Gute für Sie!

Der zweite Teil des Buches enthält die schwerere Arbeit. Er besteht aus mehreren Fallstudien, die zunehmend komplexer werden. Jede Fallstudie ist ein Beispiel für die Bereinigung von Code – also von der Umwandlung von Code, der gewisse Probleme enthält, in Code, der weniger Probleme enthält. In diesen Teil geht es sehr ins Detail. Sie müssen ständig zwischen dem beschreibenden Text und den Code-Listings hin- und herblättern. Sie müssen den Code, mit dem wir arbeiten, analysieren und verstehen und unsere Überlegungen für die durchgeführten Änderungen nachvollziehen. Sie müssen dafür schon einige Zeit reservieren, denn Sie werden dafür einige Tage benötigen.

Im dritten Teil des Buches erhalten Sie Ihren Lohn. Er besteht aus einem einzigen Kapitel mit einer Liste von Heuristiken und Smells, die während der Erstellung der

Fallstudien zusammengetragen wurden. Während wir den Code in den Fallstudien analysierten und bereinigten, haben wir alle Gründe für unsere Aktionen als Heuristikoder Smell dokumentiert. Wir versuchten, unsere eigenen Reaktionen auf den Code zu verstehen, den wir gelesen und geändert hatten, und mühten uns wirklich ab, herauszufinden und festzuhalten, warum wir fühlten, was für fühlten. und warum wir taten, was wir taten. Das Ergebnis ist eine Wissensbasis, die unsere Art zu denken beschreibt, wenn wir sauberen Code schreiben und lesen.

Diese Wissensbasis hat für Sie nur einen beschränkten Wert, wenn Sie die Fallstudien im zweiten Teil dieses Buches nicht sorgfältig lesen und nachvollziehen. In diesen Fallstudien haben wir sorgfältig alle durchgeführten Änderungen mit Referenzen auf die Wissensbasis versehen. Diese Referenzen werden wie folgt in eckigen Klammern angegeben: [H22]. Dies ermöglicht es Ihnen, den Kontext zu sehen, in dem diese Heuristiken angewendet und geschrieben wurden! Es sind nicht die Heuristiken selbst, die so wertvoll sind, sondern die Beziehungen zwischen diesen Heuristiken und den einzelnen Entscheidungen, die wir beim Bereinigen des Codes in den Fallstudien getroffen haben.

Um Ihnen mit diesen Beziehungen noch weiterzuhelfen, haben wir am Anfang des Indexes Querverweise auf die Seitenzahlen eingefügt, unter denen Sie die jeweilige Referenz finden können. So können Sie leicht alle Stellen lokalisieren, an denen eine bestimmte Heuristik angewendet wurde.

Wenn Sie nur den ersten und dritten Teil lesen und die Fallstudien überspringen, dann haben Sie nur ein weiteres unterhaltsames Buch über das Schreiben von Software gelesen. Doch wenn Sie sich die Zeit nehmen, die Fallstudien durcharbeiten und jedem winzigen Schritt und jeder kleinen Entscheidung zu folgen – wenn Sie sich also an unsere Stelle versetzt und sich gezwungen haben, in denselben Bahnen zu denken wie wir, dann werden Sie ein viel tieferes Verständnis dieser Prinzipien, Patterns, Techniken und Heuristiken gewonnen haben. Diese werden dann nicht mehr nur gewisse »ganz nützliche« Techniken unter anderen sein, sondern werden Ihnen in Fleisch und Blut übergegangen sein. Sie werden ein Teil von Ihnen geworden sein, ähnlich wie ein Fahrrad eine Erweiterung Ihres Willens wird, wenn Sie das Fahren erlernt haben.

Danksagungen

Grafiken

Danke an meine beiden Künstlerinnen Jenniffer Kohnke und Angela Brooks. Jennifer ist für die beeindruckenden und kreativen Bilder am Anfang jedes Kapitels und die Porträts von Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers und mir verantwortlich.

Für Ann Marie: die Liebe meines Lebens.

Sauberer Code



Sie lesen das Buch aus zwei Gründen. Erstens: Sie sind Programmierer. Zweitens: Sie wollen ein besserer Programmierer werden. Gut. Wir brauchen bessere Programmierer.

Dieses Buch handelt von guter Programmierung. Es ist voller Code. Wir werden uns Code aus allen möglichen Richtungen anschauen. Wir werden ihn von oben und unten und von außen und innen betrachten. Wenn wir fertig sind, werden Sie sehr viel über Code wissen. Darüber hinaus werden Sie guten Code von schlechtem Code unterscheiden können. Sie werden wissen, wie Sie guten Code schreiben können. Und Sie werden wissen, wie Sie schlechten Code in guten Code transformieren können.

1.1 Code, Code und nochmals Code

Vielleicht könnte man einwenden, ein Buch über Code wäre doch etwas altmodisch – Code wäre doch längst kein Thema mehr; stattdessen sollte man sich mit Modellen und Anforderungen befassen. Tatsächlich vertreten einige Leute die Auffassung, die Ära des Codes ginge zu Ende. Bald werde aller Code nicht mehr geschrieben, sondern generiert werden. Programmierer würden einfach überflüssig werden, weil Geschäftsentwickler Programme einfach aus Spezifikationen generieren würden.

Unsinn! Wir werden niemals ohne Code arbeiten können, weil der Code die Details der Anforderungen repräsentiert. Auf einer gewissen Ebene können diese Details nicht ignoriert oder abstrahiert werden; sie müssen spezifiziert werden. Und die Spezifikation von Anforderungen in einer Detailgenauigkeit, dass sie von einer Maschine ausgeführt werden können, ist *Programmierung*. Und eine solche Spezifikation ist *Code*.

Ich rechne damit, dass die Abstraktionsebene unserer Sprachen noch höher gehen wird. Ich erwarte auch, dass die Anzahl der domänenspezifischen Sprachen weiterhin wachsen wird. Diese Entwicklung bringt Vorteile mit sich, aber sie wird den Code nicht eliminieren. Tatsächlich werden alle Spezifikationen, die auf diesen höheren Ebenen und in den domänenspezifischen Sprachen geschrieben werden, Code *sein*! Sie müssen immer noch stringent, genau und so formal und detailliert sein, dass sie von einer Maschine verstanden und ausgeführt werden können.

Leute, die denken, Code werde eines Tages verschwinden, ähneln Mathematikern, die hoffen, eines Tages eine Mathematik zu entdecken, die nicht formal sein muss. Sie hoffen, dass wir eines Tages eine Methode entdecken werden, Maschinen zu erschaffen, die tun, was wir wollen, und nicht, was wir sagen. Diese Maschinen müssen in der Lage sein, uns so gut zu verstehen, dass sie unsere unscharf formulierten Bedürfnisse in perfekt ausgeführte Programme übersetzen können, die genau diese Bedürfnisse erfüllen.

Dies wird nie passieren. Nicht einmal Menschen mit all ihrer Intuition und Kreativität sind bis jetzt in der Lage gewesen, aus den unscharfen Gefühlen ihrer Kunden erfolgreiche Systeme abzuleiten. Wenn wir überhaupt etwas aus der Disziplin der Anforderungsspezifikation gelernt haben, ist es Folgendes: Wohlspezifizierte Anforderungen sind genauso formal wie Code und können als ausführbare Tests dieses Codes verwendet werden!

Vergessen Sie nicht, dass Code letztlich die Sprache ist, in der wir die Anforderungen ausdrücken. Wir können Sprachen konstruieren, die näher bei den Anforderungen angesiedelt sind. Wir können Werkzeuge schaffen, die uns helfen, diese Anforderungen zu parsen und zu formalen Strukturen zusammenzusetzen. Aber wir werden niemals die erforderliche Präzision eliminieren können – und deshalb wird es immer Code geben.

1.2 Schlechter Code



Abb. 1.1: Kent Beck

Neulich las ich das Vorwort zu dem Buch *Implementation Patterns* von Kent Beck [Beck07]. Darin schreibt er: »... dieses Buch basiert auf einer recht fragilen Prämisse: dass guter Code eine Rolle spiele ...«. Eine *fragile* Prämisse? Dem kann ich nicht zustimmen! Ich glaube, dass diese Prämisse zu den robustesten, am besten unterstützten und meistdiskutierten Prämissen unserer Zunft gehört (und ich glaube, das weiß Kent Beck auch). Wir wissen, dass guter Code eine Rolle spielt, weil wir uns so lange mit seiner mangelnden Qualität auseinandersetzen mussten.

Ich kenne ein Unternehmen, das in den späten 80er-Jahren eine *Killer*-Applikation herausbrachte. Sie war sehr beliebt, und zahlreiche professionelle Anwender kauften und nutzten sie. Aber dann wurden die Release-Zyklen immer länger. Bugs wurden von einem Release zum nächsten nicht mehr repariert. Die Startzeiten wurden länger und die Abstürze häufiger. Ich erinnere mich an den Tag, an dem ich das Produkt frustriert abschaltete und niemals wieder benutzte. Kurz danach verschwand das Unternehmen vom Markt.

Zwei Jahrzehnte später traf ich einen früheren Mitarbeiter dieses Unternehmens und fragte ihn, was damals passiert wäre. Die Antwort bestätigte meine Befürchtungen. Das Unternehmen hatte das Produkt zu schnell auf den Markt gebracht und im Code ein riesiges Chaos angerichtet. Je mehr Funktionen zu dem Code hinzugefügt wurden, desto schlechter wurde er, bis das Unternehmen ihn einfach nicht mehr verwalten konnte. *Es war der schlechte Code, der das Unternehmen in den Abgrund trieb.*

Sind Sie jemals erheblich von schlechtem Code beeinträchtigt worden? Wenn Sie als Programmierer auch nur ein bisschen Erfahrung haben, dann haben Sie eine solche Behinderung viele Male erlebt. Tatsächlich haben wir eine spezielle Bezeich-

nung dafür: *Wading* (Waten). Wir waten durch schlechten Code. Wir kämpfen uns durch einen Morast verschlungener Schlingpflanzen und verborgener Fallgruben. Wir mühen uns ab, den richtigen Weg zu finden, und hoffen auf irgendwelche Hinweise, die uns zeigen, was passiert; aber alles, was wir sehen, ist ein schier endloses Meer von sinnlosem Code.

Natürlich sind Sie von schlechtem Code behindert worden. Also – warum haben Sie ihn geschrieben?

Haben Sie zu schnell gearbeitet? Waren Sie unter Druck? Wahrscheinlich. Vielleicht hatten Sie das Gefühl, keine Zeit für gute Arbeit zu haben, meinten, Ihr Chef würde ärgerlich werden, wenn Sie sich die Zeit nehmen würden, Ihren Code aufzuräumen. Vielleicht waren Sie es einfach leid, an diesem Programm zu arbeiten, und wollten endlich damit fertig werden. Oder vielleicht haben Sie Ihren Stapel unerledigter Arbeit angeschaut, die Sie längst hätten erledigen müssen, und sind zu dem Schluss gekommen, Sie müssen dieses Modul zusammenschustern, um mit dem nächsten weitermachen zu können. Wir alle kennen diese Erfahrung.

Wir alle haben uns das Chaos angeschaut, das wir gerade angerichtet hatten, und dann beschlossen, es an einem anderen Tag zu beseitigen. Wir alle haben die Erleichterung gefühlt, zu sehen, dass unser chaotisches Programm lief, und beschlossen, dass ein laufendes Chaos besser wäre als nichts. Wir alle haben uns vorgenommen, später zurückzukommen und das Chaos zu beseitigen. Natürlich kannten wir damals das Gesetz von LeBlanc nicht: *Später gleich niemals*.

1.3 Die Lebenszykluskosten eines Chaos

Wenn Sie schon länger als zwei bis drei Jahre programmieren, haben Sie wahrscheinlich die Erfahrung gemacht, dass Ihre Arbeit von dem chaotischen Code eines anderen Entwicklers erheblich verlangsamt worden ist. Die Verlangsamung kann beträchtlich sein. Im Laufe von einem oder zwei Jahren kann es passieren, dass Teams, die am Anfang eines Projekts sehr schnell vorangekommen waren, sich plötzlich nur noch im Schneckentempo vorwärtsbewegen. Jede Änderung des Codes führt zur Defekten an zwei oder drei anderen Stellen des Codes. Keine Änderung ist trivial. Für jede zusätzliche Funktion oder Modifikation des Systems müssen alle Verzweigungen, Varianten und Knoten »verstanden« werden, damit weitere Verzweigungen, Varianten und Knoten hinzugefügt werden können. Im Laufe der Zeit wird das Chaos so groß und so verfilzt, dass Sie es nicht mehr bereinigen können. Sie sind am Ende Ihres Weges angelangt.

Während das Chaos immer größer wird, nimmt die Produktivität des Teams laufend ab und geht asymptotisch gegen null. Während die Produktivität sinkt, tut das Management das Einzige, was es kann: Es weist dem Projekt mehr Personal zu in der Hoffnung, die Produktivität zu steigern. Aber das neue Personal versteht das Design des Systems nicht. Es kennt nicht den Unterschied zwischen einer Ände-

rung, die zum Zweck des Designs passt, und einer Änderung, die dem zuwiderläuft. Darüber hinaus stehen Sie und die anderen Teammitglieder unter schrecklichem Druck, die Produktivität zu verbessern. Deshalb produzieren alle immer mehr Chaos und senken damit die Produktivität immer weiter gegen null (siehe Abbildung 1.2).

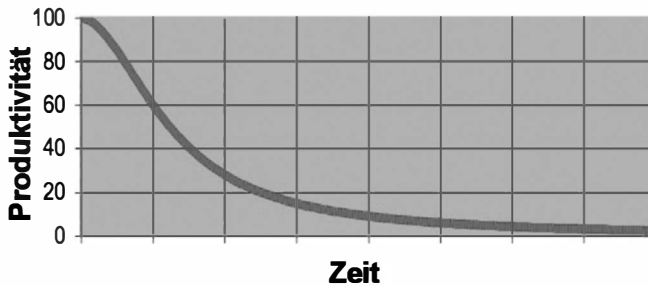


Abb. 1.2: Produktivität und Zeit

Das große Redesign in den Wolken

Schließlich rebelliert das Team. Das Management wird darüber informiert, dass man mit dieser zweifelhaften Code-Basis nicht weiterarbeiten könne. Es wird ein Redesign gefordert. Das Management will aber nicht die Ressourcen für ein komplett neues Redesign des Projekts aufwenden, kann sich aber auch nicht der Erkenntnis verschließen, dass die Produktivität nicht akzeptabel ist. Schließlich beugt es sich den Forderungen der Entwickler und autorisiert das große Redesign in den Wolken.

Es wird ein neues Tiger-Team zusammengestellt. Jeder möchte zu diesem Team gehören, weil es ein frisches neues Projekt ist. Man darf neu anfangen und etwas wirklich Schönes erstellen. Aber nur die Besten und Hellsten werden für das Tiger-Team ausgewählt. Alle anderen müssen sich um die Wartung des gegenwärtigen Systems kümmern.

Jetzt gibt es ein Wettrennen zwischen den beiden Teams. Das Tiger-Team muss ein neues System erstellen, das alle Funktionen des alten erfüllt. Und nicht nur das: Es muss auch mit den Änderungen Schritt halten, die laufend an dem alten System vorgenommen werden. Das Management wird das alte System nicht ersetzen, bevor nicht das neue alle Funktionen des alten erfüllt.

Dieser Wettlauf kann sich sehr lange hinziehen. Ich habe Zeitspannen von bis zu zehn Jahren erlebt. Und wenn er beendet wird, sind die ursprünglichen Mitglieder des Tiger-Teams längst nicht mehr da, und die gegenwärtigen Mitglieder verlangen nach einem Redesign des neuen Systems, weil es ein solches Chaos sei.

Wenn nur ein kleiner Teil dieser Geschichte Ihrer Erfahrung entspricht, dann wissen Sie bereits, dass die Zeit, die Sie für das Sauberhalten Ihres Codes verwenden, nicht nur kosteneffizient, sondern eine Frage des beruflichen Überlebens ist.

Einstellung

Sind Sie jemals durch einen Morast gewatet, der so dicht war, dass es Wochen dauerte, um zu tun, was nur einige Stunden hätte dauern sollen? Haben Sie erlebt, dass eine Änderung, die nur eine Zeile hätte erfordern sollen, in Hunderten verschiedener Module durchgeführt werden musste? Diese Symptome kommen leider allzu oft vor.

Warum passiert das mit Code? Warum verrottet guter Code so schnell zu schlechtem Code? Dafür haben wir viele Erklärungen. Wir beklagen uns, dass die Anforderungen in einer Weise geändert wurden, die dem ursprünglichen Design zuwiderläuft. Sie jammern, dass der Zeitplan zu eng bemessen war, um die Aufgaben richtig zu erledigen. Geben dummen Managern und den toleranten Kunden und nutzlosen Marketing-Typen und einem unzureichenden Telefon-Support die Schuld. Aber der Fehler, lieber Dilbert, liegt nicht in unseren Sternen, sondern in uns selbst. Wir sind unprofessionell.

Diese Pille zu schlucken, mag etwas bitter sein. Wie könnte dieses Chaos *unsere* Schuld sein? Was ist mit den Anforderungen? Was ist mit dem Zeitplan? Gibt es etwa keine dummen Manager und nutzlose Marketing-Typen? Tragen sie nicht einen Teil der Schuld?

Nein. Die Manager und Marketing-Leute fragen *uns* nach den Informationen, die sie benötigen, um Versprechungen und Zusagen zu machen; und selbst wenn sie uns nicht fragen, sollten wir keine Hemmungen haben, ihnen zu sagen, was wir denken. Die Benutzer wenden sich an uns, damit wir ihnen zeigen, wie das System ihre Anforderungen erfüllt. Die Projektmanager benutzen unsere Informationen, um ihre Zeitpläne aufzustellen. Wir sind eng in die Planung des Projekts eingebunden und tragen einen großen Teil der Verantwortung für auftretende Fehler, insbesondere wenn diese Fehler mit schlechtem Code zu tun haben!

»Doch halt!«, sagen Sie. »Wenn ich nicht tue, was mein Manager sagt, werde ich gefeuert.« Wahrscheinlich nicht. Die meisten Manager wollen die Wahrheit wissen, selbst wenn sie sich nicht immer entsprechend verhalten. Die meisten Manager wollen guten Code haben, selbst wenn sie von ihrem Zeitplan besessen sind. Vielleicht verteidigen sie leidenschaftlich den Zeitplan und die Anforderungen; aber das ist ihr Job. Dagegen ist es *Ihr* Job, den Code mit gleicher Leidenschaft zu verteidigen.

Betrachten wir eine Analogie: Was würden Sie als Arzt machen, wenn ein Patient Sie auffordern würde, dieses blödsinnige Händewaschen bei der Vorbereitung auf einen chirurgischen Eingriff zu lassen, weil es zu viel Zeit kostet? (Als das Händewaschen 1847 den Ärzten erstmals von Ignaz Semmelweis empfohlen wurde, wurde es mit der Begründung zurückgewiesen, die Ärzte wären zu beschäftigt und

hätten keine Zeit, sich die Hände zwischen ihren Patientenbesuchen zu waschen.) Natürlich hat der Patient Vorrang. Dennoch sollte der Arzt in diesem Fall die Forderung kompromisslos zurückweisen. Warum? Weil der Arzt mehr über die Risiken einer Erkrankung und Infektion weiß als der Patient. Es wäre unprofessionell (und in diesem Fall sogar kriminell), wenn der Arzt der Forderung des Patienten nachgeben würde.

Deshalb ist es auch unprofessional, dass sich Programmierer dem Willen von Managern beugen, die die Risiken nicht verstehen, die mit dem Erzeugen von Chaos im Code verbunden sind.

Das grundlegende Problem

Programmierer werden mit einem grundlegenden Wertekonflikt konfrontiert. Erfahrene Entwickler wissen, dass ihre Arbeit durch alten chaotischen Code erheblich behindert wird. Dennoch fühlen alle Entwickler den Druck, chaotischen Code zu schreiben, um Termine einzuhalten. Kurz gesagt: Sie nehmen sich nicht die Zeit, es richtig zu machen!

Echte Profis wissen, dass der zweite Teil dieses Konflikts falsch ist. Man erfüllt einen Termin eben *nicht*, indem man chaotischen Code produziert. Tatsächlich verlangsamt chaotischer Code Ihre Arbeit sofort und führt dazu, dass Sie Ihren Termin nicht einhalten können. Die *einzige* Methode, den Termin einzuhalten, besteht darin, den Code jederzeit so sauber wie möglich zu halten.

Sauberen Code schreiben – eine Kunst?

Angenommen, Sie glaubten, chaotischer Code wäre eine beträchtliche Behinderung Ihrer Arbeit. Wenn Sie jetzt akzeptieren, dass die einzige Möglichkeit, schneller zu arbeiten, darin besteht, den eigenen Code sauber zu halten, müssen Sie sich zwangsläufig fragen: »Wie schreibe ich sauberen Code?« Es hat keinen Sinn, zu versuchen, sauberen Code zu schreiben, wenn Sie nicht wissen, wie sauberer Code aussieht!

Leider haben wir hier eine schlechte Nachricht: Sauberen Code zu schreiben, hat sehr viel mit dem Malen eines Bildes zu tun. Die meisten können erkennen, wann ein Bild gut oder schlecht gemalt ist. Aber dies erkennen zu können, bedeutet nicht, dass wir auch malen können. Wenn Sie also in der Lage sind, sauberen von schlechtem Code zu unterscheiden, bedeutet dies nicht automatisch, dass Sie sauberen Code schreiben können!

Sauberen Code zu schreiben, erfordert den disziplinierten Einsatz zahlreicher kleiner Techniken, die mit einem sorgfältig erworbenen Gefühl für »Sauberkeit« angewendet werden. Dieses »Gefühl für den Code« ist der Schlüssel. Einige sind damit geboren. Einige müssen es sich mehr oder weniger mühsam erarbeiten. Dieses Gefühl für den Code hilft uns nicht nur, guten von schlechtem Code zu unterschei-

den; sondern zeigt uns die Strategie, wie wir unser Arsenal von erworbenen Techniken anwenden müssen, um schlechten Code in guten zu transformieren.

Ein Programmierer ohne dieses »Gefühl für den Code« kann sich ein chaotisches Modul anschauen und das Chaos erkennen, hat aber absolut keine Vorstellung davon, was er dagegen tun könnte. Ein Programmierer *mit* »Gefühl für den Code« schaut sich das chaotische Modul an und erkennt seine Optionen und Änderungsmöglichkeiten. Sein »Gefühl für den Code« hilft ihm dabei, die beste Option auszuwählen und eine Reihe von Änderungsschritten festzulegen, die ihn zum Ziel bringen und zugleich nach jedem Teilschritt die volle Funktionsfähigkeit des Codes erhalten.

Kurz gesagt: Ein Programmierer, der sauberen Code schreibt, ist ein Künstler, der einen leeren Bildschirm mit einer Reihe von Transformationen in ein elegant codiertes System umwandelt.

Was ist sauberer Code?

Es gibt wahrscheinlich so viele Definitionen wie Programmierer. Deshalb fragte ich einige sehr bekannte und sehr erfahrene Programmierer nach ihrer Meinung.

Bjarne Stroustrup



Abb. 1.3: Bjarne Stroustrup

Bjarne Stroustrup, Erfinder von C++ und Autor von *The C++ Programming Language*

Mein Code sollte möglichst elegant und effizient sein. Die Logik sollte gradlinig sein, damit sich Bugs nur schwer verstecken können, die Abhängigkeiten sollten minimal sein, um die Wartung zu vereinfachen, das Fehler-Handling sollte vollständig gemäß einer vordefinierten Strategie erfolgen, und das Leistungsverhalten sollte dem Optimum so nah wie möglich kommen, damit der Entwickler nicht versucht ist, den Code durch Ad-hoc-Optimierungen zu verunstalten. Sauberer Code erledigt eine Aufgabe gut.

Bjarne verwendet das Wort »elegant«. Was für ein Wort! Im Wörterbuch findet man auch folgende Synonyme: *ansprechende Anmut und Kunstfertigkeit in Aussehen oder Verhalten; ansprechende Sinnlichkeit und Einfachheit*. Wichtig ist dabei die Betonung von »ansprechend«. Offensichtlich glaubt Bjarne, dass sauberer Code *angenehm* zu lesen sein soll. Solchen Code zu lesen, sollte einen Ausdruck des Wohlgefallens auf Ihr Gesicht zaubern, den Sie auch vom Anschauen eines rassigen Automobils kennen.

Bjarne erwähnt auch die Effizienz des Codes. Vielleicht sollte uns dies bei dem Erfinder von C++ weniger überraschen; aber ich glaube, dass er damit mehr als seinen Wunsch nach einer Geschwindigkeit meint. Verschwendete Zyklen sind unelegant, sie vermitteln kein angenehmes Gefühl. Und jetzt beachten Sie, wie Bjarne die Folgen dieser Uneleganz beschreibt. Er verwendet den Ausdruck »versucht sein«. Darin ist eine tiefe Weisheit verborgen. Schlechter Code *verleitet dazu*, das Chaos zu vergrößern! Wenn andere schlechten Code ändern, neigen sie dazu, ihn noch schlechter zu machen.

Die Pragmatischen Programmierer Dave Thomas und Andy Hunt drückten dasselbe auf andere Weise aus. Sie verwendeten die Metapher der zerbrochenen Fenster (<http://www.pragprog.com/the-pragmatic-programmer/extracts/software-entropy>). Ein Gebäude mit zerbrochenen Fenstern sieht so aus, als würde sich niemand darum kümmern. Deshalb kümmern sich auch andere Entwickler nicht um den Code. Sie lassen es gewissermaßen zu, dass weitere Fenster zerbrochen werden. Schließlich helfen sie aktiv dabei. Sie verschmieren die Vorderfront mit Graffiti und lassen zu, dass sich Müll ansammelt. Der Prozess des Zerfalls beginnt mit einem zerbrochenen Fenster.

Bjarne erwähnt auch, dass das Fehler-Handling vollständig sein sollte. Dies gehört zur Disziplin, aufmerksam in den Details zu sein. Ein verkürztes Fehler-Handling ist nur eine Methode, wie Programmierer Details vernachlässigen. Speicherlecks und Race-Bedingungen sind weitere Beispiele dafür. Eine inkonsistente Namensgebung zählt ebenfalls zu. Das Fazit ist: Sauberer Code bedeutet auch große Sorgfalt im Detail.

Bjarne schließt mit der Zusicherung, dass sauberer Code eine Aufgabe gut erledigt. Es ist kein Zufall, dass so viele Prinzipien des Software-Designs auf diese einfache Mahnung zurückgeführt werden können. Autor für Autor hat versucht, diesen einen Gedanken zu kommunizieren. Schlechter Code tut zu viel; seine Absicht ist nicht klar zu erkennen und er versucht, mehrere Zwecke auf einmal zu erfüllen. Sauberer Code ist *fokussiert*. Jede Funktion, jede Klasse, jedes Modul ist eindeutig auf einen einzigen Zweck ausgerichtet und lässt sich von den umgebenden Details weder ablenken noch verunreinigen.

Grady Booch



Abb. 1.4: Grady Booch

Grady Booch, Autor von *Object-Oriented Analysis and Design with Applications*

Sauberer Code ist einfach und direkt. Sauberer Code liest sich wie wohlgeschriebene Prosa. Sauberer Code verdunkelt niemals die Absicht des Designers, sondern ist voller griffiger (engl. crisp) Abstraktionen und geradliniger Kontrollstrukturen.

Einige Punkte von Grady decken sich mit denen von Bjarne, aber er betrachtet das Ganze aus der Perspektive der *Lesbarkeit*. Mir gefällt besonders seine Auffassung, dass sich sauberer Code wie wohlgeschriebene Prosa lesen lassen soll. Denken Sie zurück an ein wirklich gutes Buch, das Sie gelesen haben. Erinnern Sie sich, wie die Wörter verschwanden und durch Bilder ersetzt wurden? Es war, als würden Sie einen Film sehen, nicht wahr? Besser! Sie sahen die Zeichen, Sie hörten die Geräusche, Sie erlebten die Leidenschaften und den Humor.

Sauberen Code zu lesen, wird natürlich niemals dasselbe sein, wie *Der Herr der Ringe* zu lesen. Dennoch ist die literarische Metapher brauchbar. Wie ein guter Roman sollte sauberer Code die Spannung in den zu lösenden Problemen sauber herausarbeiten. Diese Spannung sollte einem Höhepunkt zutreiben und dann dem Leser dieses »Aha! Ja natürlich!« vermitteln, wenn die Probleme und Spannungen bei der Enthüllung einer offensichtlichen Lösung aufgelöst werden.

Für mich ist der Ausdruck »griffige Abstraktion« (im Original: »crisp abstraction«, wörtl. »knackige oder frische Abstraktion«, unübersetzbar) ein faszinierendes Oxymoron (ein Widerspruch in sich)! Schließlich bedeutet das Wort »griffig« eher etwas Handgreifliches, Konkretes, praktisch Nutzbares. Trotz dieses scheinbaren Widerspruchs der Wörter vermitteln sie eine klare Botschaft. Unser Code sollte nicht spekulativ, sondern nüchtern und sachbezogen sein. Es sollte nur das Erforderliche enthalten. Unsere Leser sollten unsere Bestimmtheit erkennen können.

Dave Thomas



Abb. 1.5: Dave Thomas

»Big« Dave Thomas, Gründer der OTI, der Pate (Godfather) der Eclipse-Strategie

Sauberer Code kann von anderen Entwicklern gelesen und verbessert werden. Er verfügt über Unit- und Acceptance-Tests. Er enthält bedeutungsvolle Namen. Er stellt zur Lösung einer Aufgabe nicht mehrere, sondern eine Lösung zur Verfügung. Er enthält minimale Abhängigkeiten, die ausdrücklich definiert sind, und stellt ein klares und minimales API zur Verfügung. Code sollte »literate« sein, da je nach Sprache nicht alle erforderlichen Informationen allein im Code klar ausgedrückt werden können. (A.d. Ü.: »Literate Programming« ist eine Unterströmung, bei der die Einheit von Kommentaren und Code betont und gefördert wird.)

Auch Big Dave strebt wie Grady Lesbarkeit an, fordert aber eine wichtige Ergänzung. Für Dave ist es wichtig, dass sauberer Code es *anderen* Entwicklern erleichtert, ihn zu verbessern. Dies mag offensichtlich scheinen, aber es kann nicht genug betont werden. Schließlich gibt es einen Unterschied zwischen Code, der einfach zu lesen ist, und Code, der einfach zu ändern ist.

Dave macht Sauberkeit von Tests abhängig! Vor zehn Jahren hätten viele bei dieser Forderung die Stirn gerunzelt. Aber die Disziplin der Test Driven Development (TDD) hat einen wesentlichen Einfluss auf die Software-Branche gehabt und hat sich zu einer der grundlegenden Disziplinen entwickelt. Dave hat recht. Code ohne Tests ist nicht sauber. Egal wie elegant er ist, egal wie lesbar und änderungsfreundlich er ist, ohne Tests ist er unsauber.

Dave verwendet das Wort *minimal* zweimal. Offensichtlich schätzt er Code, der einen geringen Umfang hat. Tatsächlich hat sich im Laufe der letzten Jahre ein Konsens in der Software-Literatur gebildet: Kleiner ist besser.

Dave sagt auch, Code solle *literate* sein. Dies ist ein sanfter Hinweis auf das *Literate Programming* von Donald Knuth [Knuth92]. Die Quintessenz lautet: Code sollte so aufbereitet werden, dass er von Menschen gelesen werden kann.

Michael Feathers



Abb. 1.6: Michael Feathers

Michael Feathers, Autor von *Working Effectively with Legacy Code*

Ich könnte alle Eigenschaften auflisten, die mir bei sauberem Code auffallen; aber es gibt eine übergreifende Qualität, die alle anderen überragt: Sauberer Code sieht immer so aus, als wäre er von jemandem geschrieben worden, dem dies wirklich wichtig war. Es fällt nichts ins Auge, wie man den Code verbessern könnte. Alle diese Dinge hat der Autor des Codes bereits selbst durchdacht; und wenn Sie versuchen, sich Verbesserungen vorzustellen, landen Sie wieder an der Stelle, an der Sie gerade sind: Sie sitzen einfach da und bewundern den Code, den Ihnen jemand hinterlassen hat – jemand, der sein ganzes Können in sorgfältige Arbeit gesteckt hat.

Ein Wort: Sorgfalt. Das ist das eigentliche Thema dieses Buches. Vielleicht hätte ein passender Untertitel gelautet: *Wie man seinem Code Sorgfalt angedeihen lässt.*

Michael trifft den Nagel auf den Kopf. Sauberer Code ist Code, der sorgfältig erstellt worden ist. Jemand hat sich die Zeit genommen, den Code sauber zu strukturieren und zu schreiben. Jemand hat den Details die erforderliche Sorgfalt angedeihen lassen. Jemand war es nicht egal, wie sein Arbeitsergebnis aussah.

Ron Jeffries



Abb. 1.7: Ron Jeffries

Ron Jeffries, Autor von *Extreme Programming Installed* und *Extreme Programming Adventures in C#*

Ron begann seine Karriere als Programmierer in Fortran bei dem Strategic Air Command und hat Code in fast jeder Sprache und auf fast jeder Maschine geschrieben. Es zählt sich aus, seine Worte sorgfältig zu überdenken.

In den letzten Jahren beginne ich (und ende ich fast immer) mit den Regeln von Beck für einfachen Code. In der Reihenfolge der Priorität erfüllt einfacher Code die folgenden Bedingungen:

- *Er besteht alle Tests.*
- *Er enthält keine Duplizierung.*
- *Er drückt alle Design-Ideen aus, die in dem System enthalten sind.*
- *Er minimiert die Anzahl der Entities, also der Klassen, Methoden, Funktionen usw.*

Unter diesen Bedingungen konzentriere ich mich hauptsächlich auf die Duplizierung. Wenn dieselbe Sache immer wieder gemacht wird, ist dies ein Zeichen dafür, dass wir einen bestimmten Gedanken in unserem Code nicht gut genug repräsentiert haben. Dann versuche ich herauszufinden, diesen Gedanken zu fassen und klarer auszudrücken.

Ausdruckskraft umfasst für mich bedeutungsvolle Namen. Häufig ändere ich die Namen der Dinge mehrfach, bis ich die endgültige Version gefunden habe. Mit modernen Entwicklungswerkzeugen wie etwa Eclipse ist die Umbenennung recht einfach, weshalb ich mir darüber keine Gedanken mache. Doch die Ausdruckskraft geht über Namen hinaus. Ich schaue mir auch an, ob ein Objekt oder eine Methode mehr als eine Aufgabe erfüllt. Ist dies der Fall, zer-

lege ich ein Objekt in zwei oder mehr kleinere Objekte oder führe ein Refactoring einer Methode mit der Extract-Methode so durch, dass die neue Methode klarer zum Ausdruck bringt, was sie tut, und einige Untermethoden sagen, wie dies getan wird.

Duplizierungen zu eliminieren und die Ausdruckskraft zu steigern, bringen mich meinem Ideal von sauberem Code schon sehr viel näher. Chaotischen Code allein unter zwei dieser Gesichtspunkte zuvor zu verbessern, kann bereits zu einem erheblich besseren Ergebnis führen. Es gibt jedoch noch einen anderen Aspekt meiner Bemühungen, der etwas schwerer zu erklären ist.

Aufgrund meiner langen Erfahrung beim Programmieren scheint es mir, dass alle Programme aus sehr ähnlichen Elementen aufgebaut sind. Ein Beispiel: »Suche Dinge in einer Collection.« Egal was wir durchsuchen wollen, eine Datenbank mit Mitarbeiter-Datensätzen, eine Hash-Map mit Schlüsseln und Werten oder ein Array mit Elementen bestimmter Art, immer wollen wir ein spezielles Element aus dieser Collection abrufen. Wenn ich eine solche Aufgabe identifiziere, verpacke ich oft ihre spezielle Implementierung in eine abstraktere Methode oder Klasse. Dadurch erziele ich eine Reihe interessanter Vorteile.

Ich kann die Funktionalität jetzt mit etwas Einfachem, etwa einer Hash-Map, implementieren. Doch da jetzt alle Referenzen auf diese Suche in meiner kleinen Abstraktion eingekapselt sind, kann ich die Implementierung jederzeit ändern. Ich komme schneller voran und bewahre mir trotzdem meine Fähigkeit, den Code später zu ändern.

Außerdem lenkt die Collection-Abstraktion oft meine Aufmerksamkeit auf das, was »wirklich« passiert, und hält mich davon ab, Implementierungspfade zu verfolgen, auf denen ich alle möglichen Collection-Verhaltensweisen realisiere, auch wenn ich wirklich nur eine ziemlich einfache Methode brauche, um etwas Gesuchtes zu finden.

Reduzierung der Duplizierung, Steigerung der Ausdruckskraft und frühe Formulierung einfacher Abstraktionen machen für mich sauberen Code aus.

Hier hat Ron in einigen kurzen Absätzen den Inhalt dieses Buches zusammengefasst: keine Duplizierung, eine Aufgabe, Ausdruckskraft, kleine Abstraktionen. Es ist alles da.

Ward Cunningham

Ward Cunningham, Erfinder des Wiki, Erfinder von Fit, Miterfinder des eXtreme Programming. Treibende Kraft hinter den Design Patterns. Smalltalk- und OO-Vordenker. Der Pate aller Leute, denen ihr Code nicht egal ist.



Abb. 1.8: Ward Cunningham

Sie wissen, dass Sie an sauberem Code arbeiten, wenn jede Routine, die Sie lesen, ziemlich genau so funktioniert, wie Sie es erwartet haben. Sie können den Code auch »schön« nennen, wenn er die Sprache so aussehen lässt, als wäre sie für das Problem geschaffen worden.

Solche Aussagen sind für Ward charakteristisch. Sie lesen sie, nicken mit dem Kopf und wenden sich dann dem nächsten Thema zu. Die Aussage hört sich so vernünftig, so offensichtlich an, dass sie kaum als etwas Grundlegendes wahrgenommen wird. Sie glauben, sie drücke recht genau das aus, was Sie erwartet haben. Doch schauen wir etwas genauer hin.

»... ziemlich genau so, wie Sie es erwartet haben.« Wann haben Sie zum letzten Mal ein Modul gesehen, das ziemlich genau dem entsprach, was Sie erwartet haben? Ist es nicht wahrscheinlicher, dass die Module, die Sie sich anschauen, rätselhaft, kompliziert und verworren aussehen? Ist es nicht die Regel, dass Sie in die falsche Richtung gelockt werden? Sind Sie es nicht gewohnt, verzweifelt und frustriert die Denkfäden aufzuspüren und durch das ganze System zu verfolgen, aus denen letztlich das Modul gewebt ist? Wann haben Sie zum letzten Mal Code gelesen und dabei mit dem Kopf genickt, wie Sie eben die Aussage von Ward aufgenommen haben?

Ward erwartet, dass Sie beim Lesen von sauberem Code überhaupt keine Überraschungen erleben. Tatsächlich sollte das Ganze fast mühelos sein. Sie lesen den Code, und er entspricht ziemlich genau dem, was Sie erwartet haben. Er ist offensichtlich, einfach und überzeugend. Jedes Modul ist eine Stufe für das nächste. Jedes gibt vor, wie das nächste geschrieben sein wird. Programme, die so sauber sind, sind so außerordentlich gut geschrieben, dass Sie es gar nicht mal bemerken. Der Designer lässt das Problem lächerlich einfach aussehen – ein herausragendes Merkmal aller außerordentlichen Designs.

Und was ist mit Wards Vorstellung von Schönheit? Wir haben alle schon darüber geschimpft, dass unsere Sprachen nicht für unsere Probleme konzipiert worden wären. Aber Wards Aussage gibt uns den Schwarzen Peter zurück. Er sagt, schöner Code lasse die Sprache aussehen, als wäre sie für das Problem gemacht worden! Es liegt

also in *unserer* Verantwortung, die Sprache einfach aussehen zu lassen! Dies sollte Spracheifern jeder Couleur zu denken geben! Es ist nicht die Sprache, die ein Programm einfach aussehen lässt. Es ist der Programmierer, der die Sprache einfach aussehen lässt!

1.4 Denkschulen



Abb. 1.9: Uncle Bob (Robert C. Martin)

Was ist mit mir (Uncle Bob)? Was ist für mich sauberer Code? Dieses Buch wird Ihnen bis ins kleinste Detail sagen, was meine Mitstreiter und ich über sauberen Code denken. Wir werden Ihnen sagen, was unserer Meinung nach einen sauberen Variablennamen, eine saubere Funktion, eine saubere Klasse usw. ausmacht. Wir werden diese Meinungen als Absoluta formulieren, und wir werden uns nicht für unsere Schärfe entschuldigen. Für uns sind sie, an diesem Punkt unserer Karriere, absolute Postulate. Sie sind *unsere Denkschule* für sauberen Code.

Kampfsportkünstler sind sich über die beste Kampfsportart oder die beste Technik innerhalb einer Kampfsportart überhaupt nicht einig. Oft gründen Meister einer Kampfsportart ihre eigene Denkschule und sammeln Schüler um sich, denen sie ihren speziellen Kampfstil vermitteln. So gibt es etwa ein *Gracie Jiu Jistu*, das von der Gracie-Familie in Brasilien begründet wurde und gelehrt wird. Es gibt ein *Hak-koryu Jiu Jistu*, das von Okuyama Ryuho in Tokyo begründet wurde und gelehrt wird. Es gibt ein *Jeet Kune Do*, das von Bruce Lee in den Vereinigten Staaten begründet wurde und von seinen Nachfolgern gelehrt wird.

Schüler dieser Ansätze unterziehen sich einem intensiven Studium der Lehren der Gründer. Sie widmen ihre Zeit dem Erlernen des speziellen Kampfstils des jeweiligen Meisters. Oft blenden sie dabei die Lehren aller anderen Meister aus. Später, wenn die Schüler in ihrem Kampfstil eine gewisse Reife erreicht haben, können sie auch bei einem anderen Meister studieren, um ihr Wissen und ihre Kampftechniken auf eine breitere Basis zu stellen. Einige entwickeln und verfeinern ihre Fähig-

keiten schließlich so weit, dass sie neue Techniken entdecken und eigene Schulen gründen.

Keine dieser verschiedenen Schulen ist die absolut *richtige*. Doch innerhalb einer speziellen Schule *verhalten* wir uns so, als *wären* die Lehren und Techniken richtig. Denn schließlich gibt es eine korrekte Methode, Hakkoryu Jiu Jitsu oder Jeet Kune Do auszuüben. Aber diese Selbstgerechtigkeit innerhalb einer Schule entwertet nicht die Lehren einer anderen Schule.

Betrachten Sie dieses Buch als eine Beschreibung der *Object Mentor School of Clean Code* (Object-Mentor-Schule des sauberen Codes). Die Techniken und Lehren in diesem Buch sind die Methoden, wie *wir unsere* Kunst praktizieren. Wir gehen so weit zu behaupten, dass Sie, wenn Sie diese Lehren befolgen, die Vorteile erlangen werden, die wir erlangt haben, und dass Sie lernen werden, sauberen und professionellen Code zu schreiben. Sie sollten aber nicht den Fehler machen zu glauben, dass wir in irgendeinem absoluten Sinne »recht« hätten. Es gibt andere Schulen und andere Meister, die mit demselben Nachdruck Professionalität für sich in Anspruch nehmen wie wir. Und auch von ihnen zu lernen, würde Ihrem Können nur zugutekommen.

Tatsächlich werden viele Empfehlungen in diesem Buch kontrovers diskutiert. Sie werden wahrscheinlich nicht mit allem einverstanden sein. Vielleicht werden Sie einige sogar heftig ablehnen. Das ist in Ordnung. Wir können keinen Anspruch auf die endgültige Autorität erheben. Andererseits sind die Empfehlungen in diesem Buch Dinge, über die wir lange und gründlich nachgedacht haben. Sie basieren auf jahrzehntelanger Erfahrung und haben sich in wiederholten Versuch-und-Irrtum-Zyklen herauskristallisiert. Also: Egal, ob Sie mit uns übereinstimmen oder nicht, es wäre eine Schande, wenn Sie unseren Standpunkt nicht kennen lernen und respektieren würden.

1.5 Wir sind Autoren

Das `@author`-Feld einer Javadoc sagt, wer wir sind. Wir sind Autoren. Ein Merkmal von Autoren ist es, dass sie Leser haben. Tatsächlich sind Autoren dafür *verantwortlich*, erfolgreich mit ihren Lesern zu kommunizieren. Wenn Sie Ihre nächste Codezeile schreiben, sollten Sie daran denken, dass Sie ein Autor sind, der für Leser schreibt, die Ihre Anstrengung beurteilen.

Vielleicht fragen Sie, wie viel Code wirklich gelesen wird. Steckt der größte Aufwand nicht darin, den Code zu schreiben?

Haben Sie jemals ein Play-back einer Edit-Sitzung durchgeführt? In den 80ern und 90ern hatten wir Editoren wie Emacs, die jeden Tastenanschlag speichern konnten. Sie konnten eine Stunde lang arbeiten und dann ein Play-back Ihrer gesamten Edit-Sitzung wie im Zeitraffer ablaufen lassen. Als ich dies tat, waren die Ergebnisse faszinierend.

Der bei Weitem größte Anteil des Play-backs bestand darin, dass ich herumschrollte und zu anderen Modulen navigierte!

Bob kommt zu dem Modul.

Er scrollt zu der Funktion herunter, die geändert werden muss.

Er macht eine Pause und denkt über seine Optionen nach.

Oh, er scrollt wieder nach oben an den Anfang des Moduls, um die Initialisierung einer Variablen zu überprüfen.

Jetzt scrollt er zurück nach unten und beginnt zu tippen.

Ups, er löscht, was er getippt hat!

Er tippt erneut.

Er löscht erneut!

Er tippt die Hälfte von etwas anderem, aber löscht es dann wieder!

Er scrollt runter zu einer anderen Funktion, die die Funktion aufruft, die er ändert, um zu sehen, wie sie aufgerufen wird.

Er scrollt zurück und tippt denselben Code ein, den er gerade gelöscht hat.

Er macht eine Pause.

Er löscht den Code wieder!

Er öffnet ein anderes Fenster und schaut sich eine Unterklasse an. Wird die Funktion überschrieben?

...

Sie verstehen, was abgeht. Tatsächlich beträgt das Verhältnis der Zeit, die mit Lesen verbracht wird, zu der Zeit, die mit Schreiben verbracht wird, weit über 10:1. Wir lesen *permanent* alten Code als Teil unseres Bemühens, neuen Code zu schreiben.

Weil dieses Verhältnis so hoch ist, sollte das Lesen von Code leicht sein, selbst wenn dadurch das Schreiben schwerer wird. Natürlich ist es unmöglich, Code zu schreiben, ohne ihn zu lesen. Deshalb ist das Bemühen, *das Lesen von Code zu erleichtern*, zugleich ein Bemühen, *das Schreiben von Code leichter zu machen*.

Dieser Logik kann man sich nicht entziehen. Man kann keinen Code schreiben, wenn man den umgebenden Code nicht lesen kann. Der Code, den Sie heute zu schreiben versuchen, wird schwer oder leicht zu schreiben sein, und zwar abhängig davon, wie schwer oder leicht Sie den umgebenden Code lesen können. Wenn Sie also schnell vorwärtskommen wollen, wenn Sie schnell fertig werden wollen, wenn Sie Ihren Code leicht schreiben wollen, machen Sie es leicht, ihn zu lesen.

1.6 Die Pfadfinder-Regel

Es reicht nicht aus, guten Code zu schreiben. Der Code muss auch im Zeitablauf *sauber gehalten* werden. Wir haben alle erlebt, wie Code im Laufe der Zeit verrottet und immer schlechter geworden ist. Deshalb müssen wir aktiv tätig werden, um diesem schleichenden Verfall vorzubeugen.

Bei den Pfadfindern gibt es eine einfache Regel, die wir auch auf unseren Beruf anwenden können. (Sie wurde aus der Abschiedsbotschaft, *Versuche die Welt ein wenig besser zu hinterlassen, als du sie gefunden hast*, ..., von Robert Stephenson Smyth Baden-Powell an die Pfadfinder, abgeleitet.) Die Botschaft lautet:

Hinterlasse den Campingplatz sauberer, als du ihn gefunden hast.

Wenn wir alle unseren Code ein wenig sauberer einchecken, als wir ihn ausgecheckt haben, kann der Code einfach nicht verrotten. Es muss kein Großreinemachen sein. Verbessern Sie hier einen Variablennamen, zerlegen Sie dort eine etwas zu große Funktion, eliminieren Sie doppelte Codezeilen oder bauen Sie eine zusammengesetzte `if`-Anweisung um.

Können Sie sich vorstellen, an einem Projekt zu arbeiten, bei dem der Code im Laufe der Zeit *einfach besser wurde*? Glauben Sie, dass ein anderes Verhalten professionell wäre? Oder ist die laufende Verbesserung vielleicht ein wesentliches Merkmal von Professionalität?

1.7 Vorläufer und Prinzipien

In vieler Hinsicht ist dieses Buch ein »Vorläufer« zu meinem Buch *Agile Software Development: Principles, Patterns, and Practices* (PPP) aus dem Jahre 2002. Das PPP-Buch behandelt die Prinzipien des Objekt-orientierten Design (OOD) und viele Techniken, die von professionellen Entwicklern eingesetzt werden. Falls Sie PPP nicht gelesen haben, werden Sie meinen, dass es eine Fortsetzung der Geschichte ist, die in diesem Buch erzählt wird. Wenn Sie es bereits gelesen haben, werden Sie in dem vorliegenden Buch einen Widerhall vieler Aussagen aus dem PPP-Buch finden, und zwar diesmal auf der Ebene des Codes.

In diesem Buch werden gelegentlich verschiedene Design-Prinzipien referenziert: das Single-Responsibility-Prinzip (SRP), das Open-Closed-Prinzip (OCP) und das Dependency-Inversion-Prinzip (DIP) und andere. Diese Prinzipien werden in PPP ausführlich beschrieben.

1.8 Zusammenfassung

Bücher über Kunst versprechen Ihnen nicht, aus Ihnen einen Künstler zu machen. Sie können Ihnen nur einige Werkzeuge, Techniken und Gedankenprozesse ver-

mitteln, die von anderen Künstlern verwendet worden sind. Deshalb verspricht Ihnen dieses Buch nicht, aus Ihnen einen guten Programmierer zu machen. Es kann Ihnen nicht versprechen, Ihnen ein Gefühl für den Code zu vermitteln. Es kann Ihnen nur die Gedankenprozesse von guten Programmierern aufzeigen und die Tricks, Techniken und Werkzeuge mitteilen, die sie verwenden.

Ähnlich wie ein Kunstbuch enthält dieses Buch zahlreiche Details. Es enthält umfangreichen Code. Sie sehen guten Code, und Sie sehen schlechten Code. Es wird demonstriert, wie schlechter Code in guten Code transformiert werden kann. Sie finden Listen mit Heuristiken, Prinzipien und Techniken. Und es wird Ihnen ein Beispiel nach dem anderen gezeigt. Danach sind Sie auf sich gestellt.

Kennen Sie die Geschichte von dem Konzert-Violinenspieler, der sich auf dem Weg zur Vorführung verlaufen hatte? Er fragte einen alten Mann an der Straßenecke nach dem Weg zur Carnegie Hall. Der alte Mann schaute den Violinenspieler an, sah die Geige unter seinem Arm und sagte: »Übung, mein Sohn. Übung!«

Aussagekräftige Namen

von Tim Ottinger



2.1 Einführung

Namen kommen in Software überall vor. Wir benennen unsere Variablen, unsere Funktionen, unsere Argumente, Klassen und Packages. Wir benennen unsere Quelldateien und die Verzeichnisse, in denen sie enthalten sind. Wir benennen unsere jar-Dateien und war-Dateien und ear-Dateien. Wir benennen und benennen und benennen. Was wir so häufig tun, sollten wir besser gut tun. In den folgenden Abschnitten finden Sie einige einfache Regeln für die Bildung guter Namen.

2.2 Zweckbeschreibende Namen wählen

Es ist einfach zu sagen, Namen sollten den Zweck beschreiben. Wir möchten Ihnen vermitteln, dass wir dies *ernst* meinen. Gute Namen zu wählen, braucht Zeit. *spart* aber letztlich mehr Zeit ein. Deshalb sollten Sie Ihre Namen sorgfältig auswählen.

und ändern, wenn Sie bessere finden. Jeder Leser Ihres Codes (Sie eingeschlossen) profitiert davon.

Der Name einer Variablen, Funktion oder Klasse sollte alle großen Fragen beantworten. Er sollte Ihnen sagen, warum er existiert, was er tut und wie er benutzt wird. Wenn ein Name einen Kommentar erfordert, dann drückt er seinen Zweck nicht aus.

```
int d; // abgelaufene Zeit in Tagen
```

Der Name `d` enthüllt nichts. Er ruft weder das Bild einer Zeitspanne noch einen Gedanken an Tage hervor. Wir sollten einen Namen wählen, der angibt, was gemessen wird und in welcher Einheit es gemessen wird:

```
int elapsedTimeInDays;  
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Namen zu wählen, die den Zweck ausdrücken, macht es viel leichter, den Code zu verstehen und zu ändern. Welchen Zweck erfüllt der folgende Code?

```
public List<int[]> getThem() {  
    List<int[]> list1 = new ArrayList<int[]>();  
    for (int[] x : theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

Warum ist es so schwierig zu erkennen, was dieser Code tut? Es enthält keine komplexen Ausdrücke. Die Zeichenabstände und Einrückungen sind vernünftig. Es gibt nur drei Variablen und zwei Konstanten. Es gibt keine exotischen Klassen oder polymorphe Methoden. nur eine Liste von Arrays (so scheint es jedenfalls).

Das Problem ist nicht die Einfachheit des Codes, sondern seine *Implizitheit* (um einen Begriff zu prägen): ein Maß dafür, wie weit der Kontext explizit aus dem Code selbst hervorgeht oder nicht. Der Code erfordert von uns implizit, dass wir die Antworten auf die folgenden Fragen kennen:

1. Welche Dinge sind in `theList` gespeichert?
2. Welche Bedeutung hat das Subskript `null` eines Elements von `theList`?
3. Welche Bedeutung hat der Wert `4`?
4. Wie wird die zurückgegebene Liste verwendet?

Die Antworten auf diese Fragen gehen aus dem Code-Beispiel nicht hervor, aber sie hätten daraus hervorgehen können. Angenommen, Sie arbeiteten an einem Mine-

Sweeper-Spiel. Sie stellen fest, dass Sie das Spielfeld als eine Liste von Zellen repräsentieren können, die Sie als `theList` bezeichnen. Wir wollen diese Liste in `gameBoard` umbenennen.

Jede Zelle des Spielfelds wird durch ein einfaches Array repräsentiert. Sie stellen weiter fest, dass das Subskript `null` einen Statuswert enthält und dass der Statuswert 4 »flagged (markiert)« bedeutet. Einfach, indem Sie diese entsprechenden Konzepte benennen, können Sie den Code erheblich verbessern:

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard)
        if (cell[STATUS_VALUE] == FLAGGED)
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Beachten Sie, dass sich die Einfachheit des Codes nicht geändert hat. Er enthält immer noch genau dieselbe Zahl von Operatoren und Konstanten, mit genau derselben Zahl von Verschachtelungsebenen. Aber der Code ist jetzt sehr viel expliziter, drückt also seinen Zweck sehr viel klarer aus.

Wir können einen Schritt weitergehen und eine einfache Klasse für Zellen schreiben, anstatt ein Array von `ints` zu benutzen. Die Klasse kann eine den Zweck ausdrückende Funktion (nennen wir sie `isFlagged`) enthalten, um die magischen Zahlen zu verbergen. Hier ist die neue Version der Funktion:

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard)
        if (cell.isFlagged())
            flaggedCells.add(cell);
    return flaggedCells;
}
```

Nach diesen einfachen Namensänderungen ist es nicht mehr schwierig zu verstehen, was passiert. Dies ist ein Beispiel für die Kraft, die gut gewählten Namen inne-
liegt.

2.3 Fehlinformationen vermeiden

Programmierer sollten keine irreführenden Hinweise hinterlassen, die die Bedeutung des Codes verdunkeln. Wir sollten Wörter vermeiden, deren etablierte Bedeutungen von unserer beabsichtigten Bedeutung abweichen. Beispielsweise wären `hp`, `aix` oder `sco` als Variablennamen ungeeignet, weil es sich um die Namen von Unix-Plattformen oder -Varianten handelt. Selbst wenn Sie eine Hypotenuse codieren und `hp` für eine geeignete Abkürzung halten, könnte dieser Name irreführend sein.

Bezeichnen Sie eine Gruppe von Konten nur dann als `accountList`, wenn es sich wirklich um eine `List` handelt. Das Wort `Liste` bedeutet für einen Programmierer etwas ganz Spezielles. Wenn der Container, der die Konten enthält, nicht tatsächlich eine `List` ist, könnte der Name zu falschen Schlussfolgerungen führen. Deshalb wäre `accountGroup` oder `bunchOfAccounts` oder einfach `accounts` besser. Und selbst wenn der Container eine Liste ist, wäre es wahrscheinlich besser, den Container-Typ nicht im Namen zu codieren. Mehr darüber später.

Achten Sie auf Namen, die sich geringfügig unterscheiden. Wie lange dauert es, den subtilen Unterschied zwischen `XYZControllerForEfficientHandlingOfStrings` in einem Modul und `XYZControllerForEfficientStorageOfStrings` an etwas entfernterer Stelle zu entdecken? Die äußere Form der Wörter ist erschreckend ähnlich.

Ähnliche Konzepte ähnlich zu schreiben, vermittelt Informationen. Eine inkonsistente Schreibweise ist Desinformation. Moderne Java-Umgebungen bieten uns den Luxus der automatischen Code-Ergänzung (*code completion*). Wir schreiben einige Zeichen eines Namens und drücken eine Hotkey-Kombination aus (`if that`) und werden mit einer Liste möglicher Ergänzungen für diesen Namen verwöhnt. Es ist sehr hilfreich, wenn Namen für sehr ähnliche Aufgaben alphabetisch benachbart stehen und wenn die Unterschiede klar erkennbar sind, weil der Entwickler wahrscheinlich ein Objekt anhand seines Namens auswählt, ohne Ihre umfangreichen Kommentare oder sogar die Liste der Methoden dieser Klasse zu studieren.

Wirklich abschreckende Beispiele für irreführende Namen sind der Kleinbuchstabe `l` oder der Großbuchstabe `O` als Variablennamen, besonders wenn sie kombiniert werden. Das Problem liegt hier natürlich darin, dass sie fast genau wie die Konstanten `eins` bzw. `null` aussehen.

```
int a = 1;
if ( 0 == 1 )
    a = 01;
else
    1 = 01;
```

Wenn Sie meinen, dieses Beispiel wäre getürkt, können wir nur sagen, dass wir Code untersucht haben, in dem derartige Dinge in Hülle und Fülle vorkamen. In einem Fall schlug der Autor des Codes vor, eine andere Schriftart zu verwenden, damit die Unterschiede besser sichtbar wären. Eine solche Lösung müsste dann natürlich an alle künftigen Entwickler weitergegeben werden, entweder als mündliche Überlieferung oder in einem schriftlichen Dokument. Durch eine einfache Umbenennung wird das Problem endgültig und ohne zusätzlichen Aufwand erledigt.

2.4 Unterschiede deutlich machen

Programmierer schaffen sich ihre eigenen Probleme, wenn sie Code schreiben, nur um einem Compiler oder Interpreter gerecht zu werden. Ein Beispiel: Weil man im selben Geltungsbereich nicht denselben Namen zur Bezeichnung verschiedener Aufgaben verwenden darf, könnte man versucht sein, einen Namen willkürlich zu ändern. Manchmal wird für diesen Zweck einfach der Name falsch geschrieben, was zu einer überraschenden Situation führt, dass eine Korrektur des vorgeblichen »Schreibfehlers« zu Fehlern beim Kompilieren führt. Betrachten Sie beispielsweise die wirklich abscheuliche Praxis, eine Variable namens `klass` zu erstellen, einfach weil der Name `class` für etwas anderes verwendet wird.

Es reicht nicht aus, eine Zahlenfolge oder leere Wörter anzuhängen, auch wenn der Compiler damit zufrieden sein sollte. Wenn die Namen unterschiedlich sein müssen, dann sollten sie auch etwas Verschiedenes bezeichnen.

Namen mit Zahlenserien (`a1`, `a2`, .. `aN`) sind das Gegenteil einer zweckvollen Benennung. Solche Namen sind nicht irreführend – sie sind informationsleer; sie enthalten keinen Hinweis auf die Absicht des Autors. Ein Beispiel:

```
public static void copyChars(char a1[], char a2[]) {  
    for (int i = 0; i < a1.length; i++) {  
        a2[i] = a1[i];  
    }  
}
```

Diese Funktion liest sich viel besser, wenn *source* und *destination* als Argument-Namen benutzt werden.

Leere Wörter sind eine andere Form der bedeutungsleeren Unterscheidung. Angenommen, Sie hätten eine `Product`-Klasse. Wenn Sie eine andere Klasse namens `ProductInfo` oder `ProductData` haben, haben Sie zwar einen anderen Namen, aber keine andere Bedeutung. `Info` und `Data` sind unbestimmte Leerwörter wie `a`, `an` und `the`.

Beachten Sie, dass nichts dagegen einzuwenden ist, wenn Sie Präfix-Konventionen wie `a` und `the` verwenden, solange Sie damit eine sinnvolle Unterscheidung ausdrücken. Beispielsweise könnten Sie `a` für alle lokalen Variablen und `the` für alle Funktionsargumente verwenden. (Uncle Bob hat diese Technik früher in C++ eingesetzt, aber dann aufgegeben, weil sie durch moderne IDEs überflüssig wurde.) Das Problem tritt auf, wenn Sie beschließen, eine Variable `theZork` zu nennen, weil Sie bereits eine andere Variable namens `zork` haben.

Leerwörter sind redundant. Das Wort `variable` sollte niemals in einem Variablen-namen erscheinen. Das Wort `table` sollte niemals in einem Tabellennamen vorkommen. Wieso ist `NameString` besser als `Name`? Könnte ein `Name` jemals eine Fließkommazahl sein? Wäre dies der Fall, würden Sie gegen eine frühere Regel über

Fehlinformationen verstoßen. Stellen Sie sich vor, Sie stießen auf eine Klasse namens `Customer` und eine andere namens `CustomerObject`. Wodurch unterscheiden sich die Klassen? Welche repräsentiert den besten Pfad zu der Zahlungshistorie eines Kunden?

Wir kennen eine Anwendung, in der dies illustriert wird. Wir haben den Namen geändert, um die Schuldigen zu schützen; doch hier ist die genaue Form des Fehlers:

```
getActiveAccount();  
getActiveAccounts();  
getActiveAccountInfo();
```

Woher sollen die Programmierer in diesem Projekt wissen, welche Funktion sie aufrufen müssen?

Wenn keine speziellen Konventionen vereinbart sind, ist die Variable `moneyAmount` von `money` nicht unterscheidbar; dasselbe gilt für `customerInfo` und `customer`, `accountData` und `account` sowie `theMessage` und `message`. Namen sollten so unterschieden werden, dass der Leser weiß, was der Unterschied bedeutet.

2.5 Aussprechbare Namen verwenden

Menschen können gut mit Wörtern umgehen. Ein großer Teil unseres Gehirns dient dem Hervorbringen und Verarbeiten von Wörtern. Und Wörter sind per Definition aussprechbar. Es wäre eine Schande, diesen riesigen Teil unseres Gehirns, der für den Umgang mit gesprochener Sprache entwickelt worden ist, nicht zu unserem Vorteil zu nutzen. Deshalb sollten Ihre Namen aussprechbar sein.

Wenn Sie einen Namen nicht aussprechen können, können Sie nicht darüber diskutieren, ohne sich wie ein Idiot anzuhören. »Na ja, hier bei dem be ce er drei ce en te haben wir pe es ze qu int, nicht wahr?« Dies spielt eine Rolle; denn Programmieren ist eine soziale Aktivität.

Ein mir bekanntes Unternehmen hat `genymdhms` (generation date, year, month, day, hour, minute und second; Erstellungsdatum, Jahr, Monat, Tag, Stunde, Minute und Sekunde). Deshalb laufen sie rum und reden von »gen why emm dee aich emm ess« (in Englisch!). Ich habe die nervige Angewohnheit, alles so auszusprechen, wie es geschrieben ist. Deshalb fing ich an mit: »gen-yah-mudda-hims«. Später haben mehrere Designer und Analysten meine Sprechweise übernommen, sie hört sich trotzdem immer noch albern an. Aber dann war es für uns halt ein Insider-Witz. Spaß oder nicht, wir tolerierten einen schlechten Namen. Neue Entwickler brauchten eine Erklärung und fingen dann ebenfalls an, alberne erfundene Wörter anstelle verständlicher umgangs- oder fachsprachlicher Wörter zu benutzen. Vergleichen Sie

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
};
```

mit

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;;
    private final String recordId = "102";
    /* ... */
};
```

Jetzt kann man sich intelligent darüber unterhalten: »Hey, Mikey, schau dir diesen Datensatz an! Der generationTimestamp wird auf das morgige Datum gesetzt! Wie kann das sein?«

2.6 Suchbare Namen verwenden

Bei Namen aus einzelnen Buchstaben und numerischen Konstanten gibt es ein spezielles Problem: Sie sind in einem Textabschnitt nur schwer zu finden.

Während es leicht ist, per `grep` nach `MAX_CLASSES_PER_STUDENT` zu suchen, bereitet die Zahl 7 wohl mehr Schwierigkeiten. Die Suche weist Ergebnisse aus, die die Ziffer 7 als Bestandteil eines Dateinamens, in anderen konstanten Definitionen und in verschiedenen Ausdrücken enthalten, wo sie jeweils unterschiedliche Zwecke erfüllt. Noch schlimmer ist es bei einer konstanten langen Zahl: Jemand könnte aus Versehen Ziffern vertauschen und damit einen Bug verursachen, während die Variable dadurch gleichzeitig durch den Suchfilter des Programmierers fällt.

In diesem Sinne ist auch der Name `e` als Variablenname ungeeignet. Weil dieser Buchstabe zu den häufigsten der normalen Sprache gehört, führt eine Suche nach diesem Namen zu zahlreichen Nieten. In dieser Hinsicht sind längere besser als kürzere, und suchbare Name sind besser als Konstanten im Code.

Persönlich ziehe ich es vor, Variablennamen aus einem einzigen Buchstaben NUR als lokale Variablen in kurzen Methoden zu verwenden. *Die Länge eines Namens sollte der Größe seines Geltungsbereiches entsprechen* [N5]. Wenn eine Variable oder Konstante an mehreren Stellen des Codes erscheint oder benutzt wird, muss sie einen suchfreundlichen Namen haben. Vergleichen Sie wieder

```
for (int j=0; j<34; j++) {
    s += (t[j]*4)/5;
}
```

mit

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
    int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
    int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
    sum += realTaskWeeks;
}
```

Beachten Sie, dass `sum` in dem obigen Beispiel kein besonders nützlicher Name ist, aber wenigstens ist er suchbar. Der Code mit den zweckvollen Namen ist länger. Doch bedenken Sie, wie viel leichter es sein wird, `WORK_DAYS_PER_WEEK` zu suchen, als alle Stellen zu prüfen, an denen 5 verwendet wurde und die Liste auf die Instanzen mit der beabsichtigten Bedeutung zu reduzieren.

2.7 Codierungen vermeiden

Wir müssen uns schon mit genug Codierungen herumschlagen und brauchen uns keine weiteren aufzuladen. Informationen über den Typ oder den Geltungsbereich per Codierung in Namen aufzunehmen, erschwert einfach nur zusätzlich die Last der Entschlüsselung. Es gibt selten einen vernünftigen Grund dafür, dass neue Mitarbeiter zusätzlich zu dem (normalerweise beträchtlichen) vorhandenen Code, mit dem sie arbeiten sollen, noch eine weitere Verschlüsselungs-»Sprache« lernen müssen. Es ist eine unnötige mentale Belastung, wenn wir versuchen, ein Problem zu lösen. Codierte Namen lassen sich selten aussprechen und können leicht falsch getippt werden.

Ungarische Notation

In der guten alten Zeit, als die Länge der Namen in den Sprachen allzu beschränkt war, mussten wir, leider, zwangsläufig gegen diese Regel verstoßen. In Fortran musste der erste Buchstabe eines Codes den Typ angeben. In frühen Versionen von BASIC durften Namen nur aus einem Buchstaben plus einer Ziffer bestehen. Und die Hungarian Notation (HN; ungarische Notation) hob diese Kunst auf eine ganz neue Ebene.

Zu Zeiten des C-API von Windows, als alles ein Integer-Handle oder ein Long-Pointer oder ein void-Pointer oder eine von mehreren Implementierungen von »String« (mit verschiedenen Anwendungszwecken und Attributen) war, galt die HN als hohe Kunst. Damals führten Compiler keine Typ-Prüfungen durch, deshalb brauchten Programmierer eine Krücke, die ihnen half, die Typen zu erkennen und auseinanderzuhalten.

In modernen Sprachen verfügen wir über viel reichhaltigere Typensysteme, und die Compiler prüfen die Typen und erzwingen ihre Einhaltung. Darüber hinaus gibt es einen Trend zu kleineren Klassen und kürzeren Funktionen, wodurch Entwickler normalerweise die Stelle der Deklaration aller Variablen sehen können, mit denen sie arbeiten.

Java-Programmierer brauchen keine Typ-Codierung. Objekte sind stark typisiert, und die Entwicklungsumgebungen sind so weit fortgeschritten, dass sie Typenfehler entdecken, lange bevor Sie das Programm kompilieren können! Deshalb sind heute die HN und anderen Formen der Typ-Codierung einfach nur hinderlich. Sie erschweren das Ändern des Namens und/oder Typs einer Variablen, Funktion oder Klasse. Sie erschweren das Lesen des Codes. Und sie schaffen das Risiko, dass das Codierungssystem den Leser irreführt.

```
PhoneNumber phoneString;  
// Name wird nicht geändert, wenn sich der Typ ändert!
```

Member-Präfixe

Außerdem braucht man heute keine Member-Variablen mehr mit dem Präfix `m_` zu versehen. Ihre Klassen und Funktionen sollten so klein sein, dass Sie es einfach nicht benötigen. Und Sie sollten eine Entwicklungsumgebung benutzen, die Member-Variablen durch eine geeignete Farbe hervorhebt, um sie von anderen zu unterscheiden. Also nicht

```
public class Part {  
    private String m_dsc; // die textliche Beschreibung  
    void setName(String name) {  
        m_dsc = name;  
    }  
}
```

sondern

```
public class Part {  
    String description;  
    void setDescription(String description) {  
        this.description = description;  
    }  
}
```

Außerdem lernen Entwickler schnell, das Präfix (oder Suffix) zu ignorieren, und achten nur auf den bedeutungsvollen Teil des Namens. Je häufiger wir den Code lesen, desto weniger bemerken wir die Präfixe. Schließlich werden die Präfixe unmerkter Müll und ein Zeichen für älteren Code.

Interfaces und Implementierungen

Diese sind manchmal spezielle Fälle von Codierungen. Angenommen, Sie wollten eine *Abstract Factory* für die Erstellung von geometrischen Formen entwickeln. Diese Factory soll ein Interface haben und durch eine konkrete Klasse implementiert werden. Wie sollte sie heißen? *IShapeFactory* und *ShapeFactory*? Ich ziehe es vor, Interfaces nicht mit einem dekorierten Namen zu benennen. Das vorangehende I, das in den heutigen Legacy-Bibliotheken so häufig anzutreffen ist, ist bestenfalls eine Ablenkung und liefert schlimmstenfalls zu viele Informationen. Ich möchte nicht, dass meine Benutzer wissen, dass ich Ihnen ein Interface übergebe. Ich will nur, dass sie wissen, dass es sich um eine *ShapeFactory* handelt. Wenn ich also entscheiden muss, ob ich entweder das Interface oder die Implementierung codiere, wähle ich die Implementierung. Ein Name wie *ShapeFactoryImp* oder selbst der schreckliche Name *CShapeFactory* sind einer Codierung des Interface-Namens vorzuziehen.

2.8 Mentale Mappings vermeiden

Die Leser sollten Ihren Namen nicht mental in einen anderen Namen übersetzen müssen, den sie bereits kennen. Dieses Problem tritt im Allgemeinen auf, wenn man weder die Termini der Problemdomäne noch die der Lösungsdomäne verwendet.

Bei Variablennamen aus einem einzigen Buchstaben kann es Probleme geben. Sicher ist ein Schleifenzähler namens *i* oder *j* oder *k* (obwohl niemals *l*!) akzeptabel, wenn sein Geltungsbereich sehr klein ist und keine Konflikte mit anderen Namen auftreten können. Derartige Namen für Schleifenzähler haben eine lange Tradition. Doch in den meisten anderen Kontexten ist ein Name aus einem einzigen Buchstaben schlecht gewählt; er ist nur ein Platzhalter, den der Leser mental in das tatsächliche Konzept übersetzen muss. Es gibt keinen schlimmeren Grund dafür, den Namen *c* nur deshalb zu verwenden, weil *a* und *b* bereits vergeben waren.

Im Allgemeinen sind Programmierer ziemlich schlau. Manche schlaue Leute geben gerne mit ihrer Schlauheit an und zeigen, wie gut sie mental jonglieren können. Denn schließlich müssen sie, wenn sie sich zuverlässig daran erinnern können, dass *r* die Kleinbuchstabenversion eines URLs ohne Host und Schema ist, sicher sehr schlau sein.

Der Unterschied zwischen einem schlaunen Programmierer und einem professionellen Programmierer besteht darin, dass der professionelle Programmierer weiß, dass die *Klarheit absoluten Vorrang hat*. Profis nutzen ihre Fähigkeiten zum Guten und schreiben Code, den andere verstehen.

2.9 Klassennamen

Klassen und Objekte sollten Namen haben, die aus einem Substantiv oder einem substantivischen Ausdruck bestehen: `Customer`, `WikiPage`, `Account` oder `AddressParser`. Vermeiden Sie Wörter wie `Manager`, `Processor`, `Data` oder `Info` im Namen einer Klasse. Ein Klassenname sollte kein Verb sein.

2.10 Methodennamen

Methoden sollten Namen haben, die aus einem Verb oder einem Ausdruck mit einem Verb bestehen: `postPayment`, `deletePage` oder `save`. Accessoren, Mutatoren und Prädikate sollten nach ihrem Wert benannt werden und, dem `JavaBean`-Standard folgend, ein Präfix wie `get`, `set` und `is` haben (<http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp>).

```
string name = employee.getName();  
customer.setName("mike");  
if (paycheck.isPosted())...
```

Wenn Konstruktoren überladen werden, sollten Sie statische `Factory`-Methoden mit Namen verwenden, die die Argumente beschreiben. Ein Beispiel,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

ist im Allgemeinen besser als

```
Complex fulcrumPoint = new Complex(23.0);
```

Um die Anwendung der entsprechenden Konstruktoren zu erzwingen, können Sie sie als `private` deklarieren.

2.11 Vermeiden Sie humorige Namen

Wenn Namen zu humorig gewählt sind, erinnern sich nur Entwickler daran, die denselben Sinn für Humor wie der Autor haben, und auch nur so lange, wie sie sich an den Witz erinnern. Werden sie später noch wissen, was die Funktion namens `HolyHandGrenade` tun soll? Sicher, der Name ist humorig, aber vielleicht wäre in diesem Falle ein nüchternes `DeleteItems` vielleicht doch der bessere Name. Klarheit ist wichtiger als der Unterhaltungswert.

Humorigkeit im Code zeigt sich oft auch in Form von umgangssprachlichen Ausdrücken oder Slang. Beispielsweise sollten Sie nicht den Namen `whack()` für `kill()` verwenden. Erzählen Sie keine kleinen kulturspezifischen Witze wie `eatMyShorts()`, wenn Sie `abort()` meinen.

Sagen Sie, was Sie meinen. Meinen Sie, was Sie sagen.

2.12 Wählen Sie ein Wort pro Konzept

Wählen Sie ein Wort für ein abstraktes Konzept aus und bleiben Sie dabei. Beispielsweise ist es verwirrend, wenn Sie `fetch`, `retrieve` und `get` als Namen für gleichwertige Methoden verschiedener Klassen verwenden. Wie erinnern Sie sich, welcher Methodenname zu welcher Klasse gehört? Traurigerweise müssen Sie sich oft merken, welches Unternehmen, welche Gruppe oder welche Person die Library oder Klasse schrieb, um sich daran zu erinnern, welche Bezeichnung benutzt wurde. Andernfalls verbringen Sie schrecklich viel Zeit damit, Header und vorherige Code-Beispiele zu durchsuchen.

Moderne Entwicklungsumgebungen wie Eclipse und IntelliJ stellen Ihnen kontext-sensitive Hinweise zur Verfügung, etwa die Liste der Methoden, die Sie bei einem gegebenen Objekt aufrufen können. Allerdings zeigt die Liste normalerweise nicht die Kommentare mit an, die Sie zu Ihren Funktionsnamen und Parameterlisten geschrieben haben. Wenn Sie Glück haben, zeigt sie die Parameter-Namen aus den Funktionsdeklarationen an. Die Funktionsnamen müssen für sich selbst sprechen, und sie müssen konsistent sein, damit Sie die korrekte Methode ohne zusätzliche Sucharbeit auswählen können.

Ähnlich ist es auch verwirrend, wenn Sie in einer Code-Basis einen `controller` und einen `manager` und einen `driver` verwenden. Was ist der wesentliche Unterschied zwischen einem `DeviceManager` und einem `ProtocolController`? Warum sind nicht beide `controllers` oder beide `managers`? Sind beide wirklich `Drivers`? Der Name verleitet Sie dazu, zwei Objekte zu erwarten, die einem sehr verschiedenen Typ angehören und ganz verschiedene Klassen haben.

Ein konsistentes Lexikon ist ein großer Vorteil für die Programmierer, die Ihren Code verwenden müssen.

2.13 Keine Wortspiele

Verwenden Sie nicht dasselbe Wort für zwei Zwecke. Wenn Sie dieselbe Bezeichnung für zwei verschiedene Konzepte verwenden, treiben Sie ein Wortspiel.

Wenn Sie die Regel »ein Wort pro Konzept« beachten, erhalten Sie möglicherweise viele Klassen, die beispielsweise eine `add`-Methode enthalten. Solange die Parameterlisten und Rückgabewerte der verschiedenen `add`-Methoden semantisch gleichwertig sind, ist alles in Ordnung.

Doch möglicherweise beschließen Sie, das Wort `add` aus Gründen der »Konsistenz« für einen Zweck zu verwenden, bei dem keine Addition in demselben Sinne erfolgt. Angenommen, Sie hätten viele Klassen, in denen mit `add` ein neuer Wert erstellt wird, indem zwei vorhandene Werte addiert oder verkettet werden. Doch jetzt wollten Sie eine neue Klasse schreiben, die über eine Methode verfügt, die ihren einzigen Parameter in eine Collection einfügt. Sollten Sie diese Methode `add` nennen?

Vielleicht halten Sie dies für konsistent, weil Sie so viele andere `add`-Methoden haben. Doch in diesem Fall ist die Bedeutung eine andere; deshalb sollten Sie stattdessen einen Namen wie `insert` oder `append` verwenden. Die neue Methode ebenfalls `add` zu nennen, wäre ein Wortspiel.

Als Autoren wollen wir Code schreiben, der so leicht lesbar wie möglich ist. Der Leser soll den Code schnell überfliegen können und nicht intensiv studieren müssen. Sie sollten dem beliebten Taschenbuch-Modell folgen und sich als Autor dafür verantwortlich fühlen, sich klar auszudrücken. Dagegen ist es beim akademischen Modell die Aufgabe des Gelehrten, die Bedeutung aus dem Papier auszugraben.

2.14 Namen der Lösungsdomäne verwenden

Denken Sie daran, dass die Entwickler, die Ihren Code lesen, Programmierer sind. Deshalb sollten Sie Fachbegriffe der Informatik, Algorithmenamen, Pattern-Namen, mathematische Begriffe usw. verwenden. Es ist nicht sinnvoll, jeden Namen aus der Problemdomäne zu entlehnen, weil unsere Kollegen nicht jedes Mal gezwungen sein sollten, beim Kunden rückzufragen, um die Bedeutung eines Namens zu erfahren, wenn sie das Konzept bereits unter einem anderen Namen kennen.

Der Name `AccountVisitor` sagt einem Programmierer sehr viel, der mit dem *Visitor*-Pattern vertraut ist. Dagegen würde ein Programmierer wahrscheinlich nicht wissen, was eine `JobQueue` ist. Programmierer müssen zahlreiche sehr technische Aufgaben erledigen. Technische Namen für diese Aufgaben zu wählen, ist normalerweise die angemessenste Vorgehensweise.

2.15 Namen der Problemdomäne verwenden

Wenn es keine Termini aus der Informatik für Ihre Aufgaben gibt, sollten Sie die Namen aus der Problemdomäne entlehnen. Dann können Programmierer, die Ihren Code warten, wenigstens einen Bereichsexperten nach der Bedeutung fragen.

Das Konzept der Lösungs- und der Problemdomäne zu trennen, gehört zu den Aufgaben eines guten Programmierers und Designers. Der Code, der mehr mit Konzepten der Problemdomäne zu tun hat, sollte entsprechend mehr Namen aus der Problemdomäne enthalten.

2.16 Bedeutungsvollen Kontext hinzufügen

Es gibt einige Namen, die an sich schon bedeutungsvoll sind – die meisten sind es nicht. Stattdessen müssen Sie die Namen für Ihre Leser in einen Kontext stellen, indem Sie die Namen in wohlbenannte Klassen, Funktionen oder Namespaces ein-

fügen. Wenn alles andere keinen Erfolg hat, ist möglicherweise ein Präfix als letzter Ausweg erforderlich.

Stellen Sie sich vor, Sie hätten Variablen namens `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` und `zipcode`. Zusammengenommen ist es ziemlich klar, dass sie eine Adresse bilden. Aber was wäre, wenn Sie nur die `state`-Variable allein in einer Methode sehen würden? Würden Sie automatisch schließen, dass es sich um einen Teil einer Adresse handelte?

Sie können den Kontext mithilfe von Präfixen hinzufügen: `addrFirstName`, `addrLastName`, `addrState` usw. Wenigstens versteht dann der Leser, dass diese Variablen zu einer größeren Struktur gehören. Natürlich wäre es besser, eine Klasse namens `Address` zu erstellen. Dann wüsste sogar der Compiler, dass die Variablen zu einem größeren Konzept gehörten.

Betrachten Sie die Methode in Listing 2.1. Brauchen die Variablen einen bedeutungsvolleren Kontext? Der Funktionsname stellt nur einen Teil des Kontextes zur Verfügung; der Algorithmus liefert den Rest. Wenn Sie die Funktion lesen, sehen Sie, dass die drei Variablen, `number`, `verb` und `pluralModifier` zu der Nachricht »guess statistics« (etwa: »statistische Daten erraten«) gehören. Leider muss der Kontext erschlossen werden. Wenn Sie sich die Methode zum ersten Mal anschauen, sind die Bedeutungen der Variablen nicht zu erkennen.

Listing 2.1: Variablen mit unklarem Kontext

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    } else if (count == 1) {
        number = "1";
        verb = "is";
        pluralModifier = "";
    } else {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }
    String guessMessage = String.format(
        "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
}
```

Die Funktion ist ein wenig zu lang und die Variablen werden in der ganzen Funktion benutzt. Um die Funktion in kleinere Teile zu zerlegen, müssen wir eine `GuessStatisticsMessage`-Klasse erstellen und die drei Variablen als Felder dieser Klasse deklarieren. Dadurch erstellen wir einen klaren Kontext für die drei Variablen. Sie gehören *definitiv* zu der `GuessStatisticsMessage`. Wegen der Verbesserung des Kontextes wird auch der Algorithmus viel sauberer, indem wir ihn in viele kleinere Funktionen zerlegen (siehe Listing 2.2).

Listing 2.2: Variablen mit Kontext

```
public class GuessStatisticsMessage {
    private String number;
    private String verb;
    private String pluralModifier;

    public String make(char candidate, int count) {
        createPluralDependentMessageParts(count);
        return String.format(
            "There %s %s %s%s",
            verb, number, candidate, pluralModifier );
    }

    private void createPluralDependentMessageParts(int count) {
        if (count == 0) {
            thereAreNoLetters();
        } else if (count == 1) {
            thereIsOneLetter();
        } else {
            thereAreManyLetters(count);
        }
    }

    private void thereAreManyLetters(int count) {
        number = Integer.toString(count);
        verb = "are";
        pluralModifier = "s";
    }

    private void thereIsOneLetter() {
        number = "1";
        verb = "is";
        pluralModifier = "";
    }

    private void thereAreNoLetters() {
        number = "no";
        verb = "are";
        pluralModifier = "s";
    }
}
```

2.17 Keinen überflüssigen Kontext hinzufügen

Angenommen, Sie erstellen eine Anwendung namens »Gas Station Deluxe«. Dann wäre es keine gute Idee, jede Klasse mit dem Präfix `GSD` zu beginnen. Offen gesagt: Sie würden dann gegen Ihre Tools arbeiten. Sie tippen `G` ein, drücken auf Completion Key und werden mit einer ellenlangen Liste aller Klassen des Systems belohnt. Ist das intelligent? Warum sollten Sie es der IDE schwermachen, Ihnen zu helfen?

Ein ähnlicher Fall: Sie haben eine `MailingAddress`-Klasse in dem Buchhaltungsmodul von `GSD` entwickelt und nennen Sie `GSDAccountAddress`. Später brauchen Sie eine Postanschrift für Ihre Kundenkontakt-Anwendung. Verwenden Sie `GSDAccountAddress`? Hört sich das wie der richtige Name an? Zehn von 17 Zeichen sind redundant oder irrelevant.

Kürzere Namen sind im Allgemeinen besser als längere, solange sie klar sind. Fügen Sie nicht mehr Kontext zu einem Namen hinzu, als erforderlich ist.

Die Namen `accountAddress` und `customerAddress` sind geeignete Namen für Instanzen der Klasse `Address`, könnten aber für Klassen ungeeignet sein. `Address` ist ein geeigneter Name für eine Klasse. Wenn ich zwischen MAC-Adressen, Port-Adressen und Web-Adressen unterscheiden muss, würde ich `PostalAddress`, `MAC` und `URI` in Erwägung ziehen. Die fertigen Namen sind präziser, und darum geht es bei der Benennung.

2.18 Abschließende Worte

Die Schwierigkeit, gute Namen zu wählen, geht auf zwei Faktoren zurück. Erstens muss man über gute Beschreibungsfähigkeiten verfügen. Zweitens braucht man einen gemeinsamen kulturellen Hintergrund. Dies ist ein Problem des Lehrens und Lernens und kein technisches, geschäftliches oder organisatorisches Problem. Deshalb haben viele Entwickler auf diesem Teilgebiet nicht sehr viel gelernt.

Viele haben auch Angst, Dinge umzubenennen, weil sie Proteste ihrer Kollegen fürchten. Wir teilen diese Angst nicht und sind eigentlich dankbar, wenn Namen verbessert werden. Meistens lernen wir die Namen von Klassen und Methoden nicht auswendig. Wir verwenden unsere modernen Werkzeuge, um die Details zu handhaben, damit wir uns darauf konzentrieren können, ob sich der Code wie Absätze und Sätze oder wenigstens wie Tabellen und Datenstrukturen lesen lässt. (Ein Satz ist nicht immer die beste Form, Daten anzuzeigen.) Wahrscheinlich werden Sie jeden überraschen, wenn Sie etwas umbenennen; aber das gilt für andere Code-Verbesserungen auch. Lassen Sie sich davon nicht abhalten.

Befolgen Sie einige dieser Regeln und prüfen Sie, ob Sie damit die Lesbarkeit Ihres Codes verbessern. Wenn Sie den Code eines anderen Entwicklers warten, sollten Sie Refactoring-Werkzeuge heranziehen, die Ihnen helfen, diese Probleme zu lösen. Es zahlt sich schon kurzfristig aus, und auch langfristig werden Sie immer weiteren Nutzen ernten.

Funktionen



In den Anfangstagen der Programmierung setzen wir unsere Systeme aus Routinen und Subroutinen zusammen. Dann, in der Ära von Fortran und PL/I, setzen wir unsere Systeme aus Programmen, Unterprogrammen und Funktionen zusammen. Heutzutage haben nur die Funktionen aus den Anfangstagen überlebt. Funktionen sind die ersten Organisationseinheiten jedes Programms. Wie man gute Funktionen schreibt, ist Thema dieses Kapitels.

Betrachten Sie den Code in Listing 3.1. Es ist schwer, eine lange Funktion in FitNesse, einem Open-Source-Testwerkzeug (<http://fitnesse.org>), zu finden; aber nachdem ich ein wenig gesucht hatte, stieß ich auf die hier gezeigte Funktion. Sie ist nicht nur lang, sondern sie enthält duplizierten Code, zahlreiche komische Strings und viele seltsame und nicht offensichtliche Datentypen und APIs. Versuchen Sie, ob Sie in den nächsten drei Minuten daraus schlau werden.

Listing 3.1: HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
    PageData pageData,
    boolean includeSuiteSetup
) throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup =
                PageCrawlerImpl.getInheritedPage(
                    SuiteResponder.SUITE_SETUP_NAME, wikiPage
                );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
                    suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
                    .append(pagePathName)
                    .append("\n");
            }
        }
        WikiPage setup =
            PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
                wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .")
                .append(setupPathName)
                .append("\n");
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown =
            PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath teardownPath =
                wikiPage.getPageCrawler().getFullPath(teardown);
            String teardownPathName = PathParser.render(teardownPath);
            buffer.append("\n")
                .append("!include -teardown .")
                .append(teardownPathName)
                .append("\n");
        }
    }
    if (includeSuiteSetup) {
        WikiPage suiteTeardown =
            PageCrawlerImpl.getInheritedPage(
```

```

        SuiteResponder.SUITE_TEARDOWN_NAME,
        wikiPage
    );
    if (suiteTeardown != null) {
        WikiPagePath pagePath =
            suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
            .append(pagePathName)
            .append("\n");
    }
}
}
pageData.setContent(buffer.toString());
return pageData.getHtml();
}

```

Verstehen Sie die Funktion nach drei Minuten Studium? Wahrscheinlich nicht. Es passiert zu viel auf zu vielen verschiedenen Abstraktionsebenen. Es gibt seltsame Strings und gelegentliche Funktionsaufrufe, die mit doppelt verschachtelten `if`-Anweisungen vermengt sind, die durch Flags gesteuert werden.

Doch durch Extrahieren einiger einfacher Methoden, einige Umbenennungen und ein wenig Umstrukturierung konnte ich den Zweck der Funktion in den neun Zeilen von Listing 3.2 zum Ausdruck bringen. Schauen Sie, ob Sie *das* in den nächsten drei Minuten verstehen können.

Listing 3.2: HtmlUtil.java (nach Refactoring)

```

public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }

    return pageData.getHtml();
}

```

Wenn Sie FitNesse nicht näher studiert haben, verstehen Sie wahrscheinlich nicht alle Details. Dennoch verstehen Sie wahrscheinlich, dass diese Funktion einige Setup- und Teardown-Seiten in eine Test-Seite einfügt und dann diese Seite in HTML darstellt. Wenn Sie JUnit, ein Open-Source-Testwerkzeug für Java

(www.junit.org), kennen, erkennen Sie wahrscheinlich, dass diese Funktion zu einem webbasierten Test-Framework gehört. Das stimmt natürlich. Diese Informationen lassen sich aus Listing 3.2 recht leicht erschließen, dagegen sind sie in Listing 3.1 ziemlich gut verborgen.

Also: Wodurch wird eine Funktion wie Listing 3.2 leicht lesbar und verstehbar? Wie können wir den Zweck einer Funktion klar kommunizieren? Welche Eigenschaften müssen unsere Funktionen haben, damit ein zufälliger Leser die Art von Programm erschließen kann, zu der die Funktion gehört?

3.1 Klein!

Die erste Regel für Funktionen lautet: Funktionen sollten klein sein. Die zweite Regel für Funktionen lautet: *Funktionen sollten noch kleiner sein*. Diese Aussage kann ich nicht weiter begründen. Ich kann keine professionellen Forschungsarbeiten zitieren, die gezeigt haben, dass sehr kleine Funktionen besser sind. Ich kann Ihnen allerdings sagen, dass ich in fast vier Jahrzehnten Funktionen aller Größen geschrieben habe: sehr hässliche 3.000 Zeilen lange Monster: tonnenweise Funktionen im Bereich von 100 bis 300 Zeilen; zahlreiche Funktionen, die 20 bis 30 Zeilen lang waren. Aus dieser Erfahrung habe ich durch Versuch und Irrtum gelernt, dass Funktionen sehr klein sein sollten.

In den 80er-Jahren pflegten wir zu sagen, dass eine Funktion nicht länger als eine Bildschirmseite sein sollte. Allerdings waren damals unsere VT100-Bildschirme 24 Zeilen hoch und 80 Spalten breit; und unsere Editoren brauchten vier Zeilen für administrative Zwecke. Heute könnten Sie mit einem kleinen Font und einem schönen großen Monitor 150 Zeichen in einer Zeile und etwa 100 Zeilen oder mehr auf einem Bildschirm unterbringen. Zeilen sollten nicht länger als 150 Zeichen sein. Funktionen sollten nicht länger als 100 Zeilen sein. Funktionen sollten kaum jemals länger als 20 Zeilen sein.

Wie kurz sollte eine Funktion sein? 1999 besuchte ich Kent Beck zu Hause in Oregon. Wir setzten uns zusammen und programmierten ein wenig. Irgendwann zeigte er mir ein hübsches kleines Java/Swing-Programm, das er *Sparkle* nannte. Es produzierte auf dem Bildschirm einen visuellen Effekt, der dem Feenstaub ähnelte, den der Zauberstab der Feenmutter aus dem Märchen-Zeichentrickfilm Aschenputtel von Walt Disney versprühte. Wenn man die Maus bewegte, lösten sich die Staubkörnchen mit einem befriedigenden Funkeln vom Cursor und fielen dann durch ein simuliertes Gravitationsfeld auf den »Boden« des Fensters. Als Kent mir den Code zeigte, war ich überrascht, wie viele kleine Funktionen der Code enthielt. Ich war an die Funktionen in Swing-Programmen gewöhnt, die vertikal sehr viel Platz beanspruchten. Jede Funktion in Becks Programm war nur zwei oder drei oder vier Zeilen lang. Jede hat einen klar erkennbaren Zweck. Jede erzählte eine Geschichte. Und eine Funktion führte zwangsläufig zur nächsten. Das Programm zeigt eindringlich, wie kurz Ihre Funktionen sein sollten! Ich habe Kent gefragt, ob

er noch eine Kopie dieses Programms hätte; leider konnte er keine finden. Ich habe auch alle meine alten Computer durchsucht. Leider ebenfalls vergeblich. Ich habe nur noch meine Erinnerung an dieses Programm.

Wie kurz sollten Ihre Funktionen sein? Normalerweise sollten sie kürzer sein als Listing 3.2! Tatsächlich sollte Listing 3.2 zu Listing 3.3 verkürzt werden.

Listing 3.3: HtmlUtil.java (nach erneutem Refactoring)

```
public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite) throws Exception {
    if (isTestPage(pageData))
        includeSetupAndTeardownPages(pageData, isSuite);
    return pageData.getHtml();
}
```

Blöcke und Einrückungen

Dies bedeutet auch, dass die Blöcke innerhalb von `if`-, `else`-, `while`- und ähnlichen Anweisungen eine Zeile lang sein sollten. Wahrscheinlich sollte diese Zeile einen Funktionsaufruf enthalten. Dadurch wird nicht nur der Umfang der einschließenden Funktion kleiner gehalten, sondern auch ihr dokumentarischer Wert erhöht, weil die Funktionsaufrufe innerhalb der Blöcke aussagestarke beschreibende Namen haben können.

Dies bedeutet auch, dass Funktionen nicht groß genug sind, um verschachtelte Strukturen aufzunehmen. Deshalb sollte die Einrückungstiefe einer Funktion nicht größer als eine oder zwei Ebenen sein. Dadurch wird es natürlich leichter, die Funktionen zu lesen und zu verstehen.

3.2 Eine Aufgabe erfüllen

Aus Listing 3.1 sollte ganz klar hervorgehen, dass die Funktion sehr viel mehr als eine Aufgabe erfüllt. Sie erstellt Puffer, ruft Seiten ab, sucht nach geerbten Seiten, setzt Pfade zusammen, hängt geheime Strings an, generiert HTML u.a. Die Funktion in Listing 3.1 ist stark damit beschäftigt, zahlreiche verschiedene Aufgaben zu erfüllen. Andererseits erfüllt die Funktion in Listing 3.3 eine einfache Aufgabe. Sie fügt Setups und Teardowns in Testseiten ein.

Der folgende Rat wird in der ein oder anderen Form seit über 30 Jahren ausgesprochen:

Funktionen sollten eine Aufgabe erledigen. Sie sollten sie gut erledigen. Sie sollten nur diese Aufgabe erledigen.

Dieser Rat ist in einer Hinsicht problematisch: Es ist schwer zu erkennen, was »eine Aufgabe« ist. Erledigt Listing 3.3 eine Aufgabe? Man könnte leicht begründen, dass sie drei Aufgaben erfüllt:

1. Sie bestimmt, ob die Seite eine Testseite ist.
2. Falls ja, schließt sie Setups und Teardowns ein.
3. Sie stellt die Seite in HTML dar.

Was ist denn nun richtig? Erledigt die Funktion eine Aufgabe oder drei Aufgaben? Beachten Sie, dass die drei Schritte der Funktion eine Abstraktionsebene unter dem Zweck liegen, der durch den Namen der Funktion ausgedrückt wird. Wir können die Funktion beschreiben, indem wir sie mit einem kurzen *TO*-Absatz (UM-ZU-Absatz) beschreiben:

TO RenderPageWithSetupsAndTeardowns, prüfen wir, ob die Seite eine Testseite ist, und wenn dies der Fall ist, schließen wir die Setups und Teardowns ein. In beiden Fällen stellen wir die Seite in HTML dar.

Die Technik ist der Programmiersprache LOGO entlehnt. In LOGO wurde das Schlüsselwort *T0* in derselben Art und Weise verwendet wie *def* in Ruby oder Python. Deshalb begann jede Funktion mit dem Schlüsselwort *T0*. Dies hatte eine interessante Auswirkung auf das Design von Funktionen.

Wenn eine Funktion nur solche Schritte ausführt, die eine Abstraktionsebene unter dem im Namen der Funktion ausgedrückten Zweck liegen, dann erledigt die Funktion eine Aufgabe. Schließlich besteht der Grund, warum wir Funktionen schreiben, darin, dass wir ein größeres Konzept (anders ausgedrückt: den Namen der Funktion) in einen Satz von Schritten auf der nächsttieferen Abstraktionsebene zerlegen.

Es sollte klar sein, dass Listing 3.1 Schritte auf vielen verschiedenen Abstraktionsebenen enthält. Deshalb erfüllt die Funktion zweifellos mehr als eine Aufgabe. Sogar Listing 3.2 enthält zwei Abstraktionsebenen. Wir konnten dies durch unsere Fähigkeit beweisen, die Funktion zu verkürzen. Aber es wäre sehr schwer, Listing 3.3 sinnvoll zu verkleinern. Wir könnten die *if*-Anweisung in eine Funktion namens *includeSetupsAndTeardownsIfTestPage* extrahieren, aber damit würde nur der Code unter einem neuen Namen an eine andere Stelle verlagert, ohne die Abstraktionsebene zu ändern.

Damit haben Sie eine andere Methode, zu erkennen, dass eine Funktion mehr als »eine Aufgabe« erfüllt: wenn Sie aus ihr eine andere Funktion mit einem Namen extrahieren können, der nicht nur eine Neuformulierung ihrer Implementierung ist [G34].

Abschnitte innerhalb von Funktionen

Blättern Sie vor zu Listing 4.7 im folgenden Kapitel. Beachten Sie, dass die Funktion *generatePrimes* in Abschnitte unterteilt ist, wie etwa *declarations*, *initializations* und *sieve*. Dies ist offensichtlich ein Symptom dafür, dass sie mehr als eine Aufgabe erfüllt. Funktionen, die eine Aufgabe erledigen, können nicht vernünftigerweise in Abschnitte zerlegt werden.

3.3 Eine Abstraktionsebene pro Funktion

Wenn unsere Funktionen »(nur) eine Aufgabe« erledigen sollen, müssen sich die Anweisungen innerhalb unserer Funktion alle auf derselben Abstraktionsebene befinden. Es ist einfach zu sehen, wie Listing 3.1 gegen diese Regel verstößt. Die Funktion enthält Konzepte, die auf einer sehr hohen Abstraktionsebene angesiedelt sind, wie etwa `getHtml()`; andere befinden sich auf mittleren Abstraktionsebenen, wie etwa `String pagePathName = PathParser.render(pagePath);` und wieder andere sind auf einer bemerkenswert tiefen Ebene angesiedelt, wie etwa `.append("\n")`.

Abstraktionsebenen innerhalb einer Funktion zu vermischen, ist immer verwirrend. Möglicherweise können die Leser nicht erkennen, ob ein spezieller Ausdruck ein wesentliches Konzept oder ein Detail ist. Noch schlimmer: Ähnlich wie ein zerbrochenes Fenster den Verfall einleitet, führen Details, die mit wesentlichen Konzepten vermischt sind, dazu, dass die Funktion im Laufe der Zeit immer mehr Details anzieht.

Code Top-down lesen: die Stepdown-Regel

Code sollte wie eine Erzählung von oben nach unten gelesen werden können ([KP78], S. 37). Danach sollten hinter jeder Funktion andere Funktionen auf der nächsttieferen Abstraktionsebene stehen, damit wir das Programm lesen können, indem wir jeweils eine Abstraktionsebene tiefer gehen, wenn wir die Liste der Funktionen von oben nach unten lesen. Ich bezeichne dies als die *Stepdown-Regel* (»eine Treppe runtergehen«).

Anders ausgedrückt: Wir sollten das Programm wie eine Folge von *UM-ZU*-Absätzen lesen können, die jeweils die gegenwärtige Abstraktionsebene beschreiben und die eine Abstraktionsebene tiefer liegenden *UM-ZU*-Absätze referenzieren.

Um die Setups und Teardowns einzuschließen, schließen wir Setups ein, dann schließen wir den Inhalt der Testseite ein, und dann schließen wir die Teardowns ein.

Um die Setups einzuschließen, schließen wir das Suite-Setup ein, wenn dies eine Suite ist; dann schließen wir das normale Setup ein.

Um das Suite-Setup einzuschließen, durchsuchen wir die Parent-Hierarchie nach der »SuiteSetUp«-Seite und fügen eine »include«-Anweisung mit dem Pfad dieser Seite hinzu.

Um die Parent-Hierarchie ...

Es hat sich gezeigt, dass Programmierer nur sehr schwer lernen, diese Regel zu befolgen und Funktionen zu schreiben, die auf einer einzigen Abstraktionsebene bleiben. Doch diese Technik beherrschen zu lernen, ist ebenfalls sehr wichtig. Sie ist der Schlüssel dazu, kurze Funktionen zu schreiben, die »eine Aufgabe« erfüllen.

Den Code so zu strukturieren, dass er wie ein Top-down-Satz von *UM-ZU*-Absätzen gelesen werden kann, ist eine wirksame Technik, um das Abstraktionsniveau konsistent zu halten.

Betrachten Sie Listing 3.7 am Ende dieses Kapitels. Es zeigt, wie ein Refactoring der kompletten `testableHtml`-Funktion nach den hier beschriebenen Prinzipien durchgeführt wurde. Beachten Sie, wie jede Funktion die nächste einführt, und jede Funktion auf einer konsistenten Abstraktionsebene bleibt.

3.4 Switch-Anweisungen

Es ist schwer, kleine `switch`-Anweisungen zu schreiben. (Das gilt natürlich auch für `if/else`-Ketten.) Selbst eine `switch`-Anweisung mit nur zwei Fällen ist größer, als ein einziger Block oder eine Funktion für meinen Geschmack sein sollte. Es ist ebenfalls schwer, eine `switch`-Anweisung zu schreiben, die nur eine Aufgabe erfüllt. Von Natur aus sind `switch`-Anweisungen immer auf *N* Aufgaben ausgelegt. Leider können wir `switch`-Anweisungen nicht immer vermeiden, aber wir können dafür sorgen, dass jede `switch`-Anweisung tief in einer niedrig angesiedelten Klasse vergraben ist und niemals wiederholt wird. Wir verwenden zu diesem Zweck natürlich den Polymorphismus.

Betrachten Sie Listing 3.4. Es zeigt nur eine der Operationen, die vom Typ des Mitarbeiters abhängen könnten.

Listing 3.4: `Payroll.java`

```
public Money calculatePay(Employee e)
    throws InvalidEmployeeType {
    switch (e.type) {
        case COMMISSIONED:
            return calculateCommissionedPay(e);
        case HOURLY:
            return calculateHourlyPay(e);
        case SALARIED:
            return calculateSalariedPay(e);
        default:
            throw new InvalidEmployeeType(e.type);
    }
}
```

Bei dieser Funktion gibt es mehrere Probleme:

- Erstens: Sie ist groß; und wenn neue Mitarbeiter-Typen hinzugefügt werden, wird sie noch größer.
- Zweitens: Sie erfüllt mehr als eine Aufgabe.
- Drittens: Sie verstößt gegen das Single-Responsibility-Prinzip (SRP), weil es mehr als einen Grund für eine Änderung gibt; siehe:

http://en.wikipedia.org/wiki/Single_responsibility_principle,
<http://www.objectmentor.com/resources/articles/srp.pdf>

- Viertens: Sie verstößt gegen das Open-Closed-Prinzip (OCP), weil sie geändert werden muss, wenn ein neuer Typ hinzugefügt wird; siehe:
http://en.wikipedia.org/wiki/Open_closed_principle,
<http://www.objectmentor.com/resources/articles/ocp.pdf>
- Fünftens: Doch möglicherweise am schlimmsten ist es, dass es eine unbegrenzte Anzahl anderer Funktionen mit derselben Struktur gibt. So könnten wir beispielsweise die Funktion

```
isPayday(Employee e, Date date),
```

oder

```
deliverPay(Employee e, Money pay),
```

oder zahlreiche andere haben, die alle dieselbe schädliche Struktur hätten.

Die Lösung für dieses Problem (siehe Listing 3.5) besteht darin, die `switch`-Anweisung in den Keller einer *Abstract Factory* [GOF] zu verbannen und niemals von jemandem sehen zu lassen. Die Factory erstellt anhand der `switch`-Anweisung die entsprechenden Instanzen der abgeleiteten Klassen von `Employee`. Die verschiedenen Funktionen, wie etwa `calculatePay`, `isPayday` und `deliverPay`, werden polymorph von dem `Employee`-Interface an die richtige Stelle dirigiert.

Listing 3.5: Employee und Factory

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}

-----

public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}

-----

public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
        switch (r.type) {
            case COMMISSIONED:
                return new CommissionedEmployee(r);
            case HOURLY:
                return new HourlyEmployee(r);
            case SALARIED:
                return new SalariedEmployee(r);
            default:
                throw new InvalidEmployeeType(r.type);
        }
    }
}
```



```
}  
}  
}
```

Meine allgemeine Regel für `swi tch`-Anweisungen lautet: Sie können toleriert werden, wenn sie nur einmal auftauchen, zur Erstellung polymorpher Objekte verwendet werden und hinter einer Vererbungsbeziehung verborgen werden, damit sie für den Rest des Systems unsichtbar sind [G23]. Natürlich ist jeder Fall einzigartig; und es gibt Gelegenheiten, bei denen ich gegen einen oder mehrere Teile dieser Regel verstoße.

3.5 Beschreibende Namen verwenden

In Listing 3.7 habe ich den Namen unserer Beispielfunktion von `testableHtml` in `SetupTeardownIncluder.render` geändert. Dieser Name ist erheblich besser, weil er die Aktion der Funktion besser beschreibt. Ich habe auch alle privaten Methoden mit gleichermaßen beschreibenden Namen versehen, wie etwa `isTestable` oder `includeSetupAndTeardownPages`. Es ist schwer, den Wert guter Namen zu überschätzen. Erinnern Sie sich an Wards Prinzip: *»Sie erkennen, dass Sie mit sauberem Code arbeiten, wenn jede Routine im Wesentlichen das tut, was Sie erwartet haben.«* Die halbe Schlacht im Kampf um die Realisierung dieses Prinzips besteht darin, gute Namen für kleine Funktionen zu finden, die eine Aufgabe erledigen. Je kleiner und fokussierter eine Funktion ist, desto leichter finden Sie einen beschreibenden Namen.

Haben Sie keine Angst vor langen Namen. Ein langer beschreibender Name ist besser als ein kurzer geheimnisvoller Name. Ein langer beschreibender Name ist besser als ein langer beschreibender Kommentar. Verwenden Sie eine Namenskonvention, die die Unterscheidung mehrerer Wörter in Funktionsnamen leicht macht. Nutzen Sie dann diese mehreren Wörter, um der Funktion einen Namen zu geben, der beschreibt, was sie tut.

Haben Sie keine Angst davor, sich Zeit für die Auswahl eines Namens zu nehmen. Tatsächlich sollten Sie mehrere Namen ausprobieren und den Code probenhalber mit ihnen lesen. Bei modernen IDEs wie Eclipse oder IntelliJ ist es trivial, Namen zu ändern. Arbeiten Sie mit einer dieser IDEs und probieren Sie verschiedene Namen aus, bis Sie einen finden, der so beschreibend wie möglich ist.

Beschreibende Namen verhelfen Ihnen in Ihrer Vorstellung zu einem klareren Bild des Designs des Moduls und helfen Ihnen, es zu verbessern. Es ist nicht ungewöhnlich, dass die Suche nach einem guten Namen zu einer vorteilhaften Umstrukturierung des Codes führt.

Vergeben Sie Namen konsistent. Verwenden Sie dieselben Ausdrücke, Substantive und Verben in den Funktionsnamen, die Sie für Ihre Module wählen. Betrachten Sie beispielsweise die Namen `includeSetupAndTeardownPages`, `includeSetup-`

Pages, includeSuiteSetupPage und includeSetupPage. Wegen der ähnlichen Ausdrücke in diesen Namen kann diese Aufreihung eine Geschichte erzählen. Tatsächlich könnten Sie, wenn ich Ihnen nur die obige Folge von Namen zeigen würde, sich fragen: »Was ist mit includeTeardownPages, includeSuiteTeardownPage und includeTeardownPage passiert?« Wäre das nicht ein Beispiel für »... im Wesentlichen das, was Sie erwartet haben«?

3.6 Funktionsargumente

Die ideale Anzahl von Argumenten für eine Funktion ist null (*niladisch*). Als Nächstes kommt eins (*monadisch*), dicht gefolgt von zwei (*dyadisch*). Drei Argumente (*triadisch*) sollten, wenn möglich, vermieden werden. Mehr als drei (*polyadisch*) erfordert eine sehr spezielle Begründung – und sollte dann trotzdem nicht benutzt werden.

Argumente sind schwer. Sie erfordern eine beträchtliche konzeptionelle Kraft. Deshalb habe ich in dem Beispiel fast keine Argumente verwendet. Betrachten Sie etwa den `StringBuffer` in dem Beispiel. Wir hätten ihn als Argument herumreichen können, anstatt ihn zu einer Instanzvariablen zu machen, aber dann hätte der Leser ihn jedes Mal interpretieren müssen, wenn er ihn gesehen hätte. Wenn Sie die Geschichte lesen, die von dem Modul erzählt wird, ist `includeSetupPage()` leichter zu verstehen als `includeSetupPageInto(newPageContent)`. Das Argument befindet sich auf einer anderen Abstraktionsebene als der Funktionsname und zwingt Sie, ein Detail (anders ausgedrückt: `StringBuffer`) zu kennen, das an diesem Punkt nicht besonders wichtig ist.

Vom Gesichtspunkt des Testens aus sind Argumente noch schwieriger. Stellen Sie sich vor, wie schwierig es ist, alle Testfälle zu schreiben, um zu gewährleisten, dass alle verschiedenen Kombinationen von Argumenten korrekt funktionieren. Gibt es keine Argumente, ist diese Aufgabe trivial. Bei einem Argument ist es nicht zu schwer. Bei zwei Argumenten wird die Lösung des Problems etwas schwieriger. Bei mehr als zwei Argumenten kann das Testen aller Kombinationen von entsprechenden Werten eine Riesenaufgabe sein.

Output-Argumente sind schwerer zu verstehen als Input-Argumente. Wenn wir eine Funktion lesen, sind wir an die Idee gewöhnt, dass Informationen als Argumente *in* die Funktion hineingehen und als Rückgabewert *aus* der Funktion herauskommen. Normalerweise erwarten wir nicht, dass Informationen durch die Argumente herauskommen. Deshalb müssen wir bei Output-Argumenten oft doppelt hinschauen.

Ein Input-Argument ist nach null Argumenten die nächstbeste Variante. `SetupTeardownIncluder.render(pageData)` ist ziemlich leicht zu verstehen. Daraus geht klar hervor, dass wir die Daten in dem `pageData`-Objekt darstellen werden.

Gebräuchliche monadische Formen

Es gibt zwei sehr verbreitete Gründe, ein einziges Argument an eine Funktion zu übergeben. Sie können eine Frage über das Argument stellen, wie etwa in `boolean fileExists("MyFile")`. Oder Sie möchten das Argument manipulieren, etwa indem Sie es in etwas anderes umwandeln und *dann zurückgeben*. Beispielsweise wandelt `InputStream fileOpen("MyFile")` einen Dateinamens-String in einen `InputStream`-Rückgabewert um. Diese beiden Anwendungen werden von dem Leser erwartet, wenn sie eine Funktion sehen. Sie sollten Namen wählen, die diesen Unterschied deutlich machen, und die beiden Formen immer in einem konsistenten Kontext verwenden (siehe den Abschnitt *Anweisung und Abfrage trennen* etwas später).

Eine etwas weniger gebräuchliche, aber immer noch sehr nützliche Form einer Funktion mit einem einzigen Argument ist ein *Event* (Ereignis). Diese Form hat ein Input-Argument, aber kein Output-Argument. Das übergreifende Programm soll die Funktionsaufrufe als Events interpretieren und mit dem Argument den Zustand des Systems verändern, beispielsweise `void passwordAttemptFailedN-times(int attempts)`. Verwenden Sie diese Form mit Bedacht. Es sollte für den Leser ganz deutlich sein, dass dies ein Event ist. Wählen Sie Namen und Kontexte sorgfältig.

Verwenden Sie möglichst keine monadische Funktionen, die nicht eine dieser Formen verwenden, beispielsweise `void includeSetupPageInto(StringBuffer pageText)`. Ein Output-Argument anstelle eines Rückgabewerts für eine Transformation zu verwenden, ist verwirrend. Wenn eine Funktion ihr Input-Argument transformiert, sollte die Transformation als Rückgabewert zurückgegeben werden. Tatsächlich ist `StringBuffer transform(StringBuffer in)` besser als `void transform-(StringBuffer out)`, selbst wenn die Implementierung im ersten Fall einfach das Input-Argument zurückgibt. Wenigstens folgt sie immer noch der Form einer Transformation.

Flag-Argumente

Flag-Argumente sind hässlich. Ein boolesches Argument an eine Funktion zu übergeben, ist eine wirklich schreckliche Technik. Es verkompliziert sofort die Signatur der Methode und gibt laut und deutlich zu verstehen, dass diese Funktion mehr als eine Aufgabe erfüllt: eine, wenn das Flag wahr ist, und eine andere, wenn es falsch ist!

In Listing 3.7 hatten wir keine Wahl, weil die Aufrufer bereits dieses Flag übergeben hatten und ich den Umfang des Refactorings auf die Funktion und tiefer begrenzen wollte. Dennoch ist der Methodenaufruf `render(true)` für einen armen Leser einfach nur verwirrend. Mit der Maus über den Aufruf zu fahren und `render(boolean isSuite)` zu sehen, hilft nicht allzu viel. Wir hätten die Funktion in zwei zerlegen sollen: `renderForSuite()` und `renderForSingleTest()`.

Dyadische Funktionen

Eine Funktion mit zwei Argumenten ist schwerer zu verstehen als eine monadische Funktion. Beispielsweise ist `writeField(name)` leichter zu verstehen als `writeField(outputStream, name)`. (Ich habe gerade das Refactoring eines Moduls durchgeführt, das eine dyadische Form verwendete. Ich konnte den `outputStream` zu einem Feld der Klasse machen und alle `writeField`-Aufrufe in eine monadische Form umwandeln. Das Ergebnis war viel sauberer.) Obwohl die Bedeutung der beiden Argumente klar ist, liest man leicht über das erste hinweg und verpasst seine Bedeutung. Das zweite Argument erfordert eine kurze Pause, bis wir gelernt haben, den ersten Parameter zu ignorieren. Und *das* führt natürlich zu Problemen, weil wir niemals irgendeinen Teil des Codes ignorieren sollten. Die Teile, die wir ignorieren, sind die Stellen, an denen sich die Bugs verbergen.

Manchmal sind natürlich zwei Argumente die passende Lösung. Beispielsweise ist an `Point p = new Point(0,0)`; überhaupt nichts auszusetzen. Kartesische Koordinaten erfordern von Natur aus zwei Argumente. Tatsächlich wären wir sehr überrascht, `new Point(0)` zu sehen. Doch die beiden Argumente sind in diesem Fall *geordnete Komponenten eines einzigen Werts!* Dagegen haben `outputStream` und `name` weder eine natürliche Kohäsion noch eine natürliche Reihenfolge.

Selbst offensichtlich dyadische Funktionen wie `assertEquals(expected, actual)` sind problematisch. Wie oft haben Sie das `actual` an die Stelle gesetzt, an der `expected` stehen sollte? Die beiden Argumente haben keine natürliche Reihenfolge. Die Reihenfolge `expected, actual` ist eine Konvention, die durch Übung gelernt werden muss.

Dyaden sind kein Übel, und Sie werden sie sicher schreiben müssen. Doch Sie sollten wissen, dass sie mit gewissen Kosten verbunden sind und Sie sollten jede Möglichkeit nutzen, sie in Monaden umzuwandeln. Beispielsweise könnten Sie die `writeField`-Methode zu einem Element von `outputStream` machen, damit Sie `outputStream.writeField(name)` sagen können. Oder Sie könnten `outputStream` zu einer Member-Variablen der gegenwärtigen Klasse machen, damit Sie ihn nicht übergeben müssen. Oder Sie könnten eine neue Klasse wie `FieldWriter` extrahieren, die den `outputStream` in ihrem Konstruktor übernimmt und über eine `write`-Methode verfügt.

Triaden

Funktionen, die drei Argumente übernehmen, sind erheblich schwerer zu verstehen als Dyaden. Die Probleme, zum Beispiel der Reihenfolge, des mehrmaligen Lesens und des Übersehens, werden mehr als verdoppelt. Ich rate Ihnen, sehr sorgfältig nachzudenken, bevor Sie eine Triade erstellen.

Betrachten Sie beispielsweise die gebräuchliche Überladung der Funktion von `assertEquals`, die drei Argumente übernimmt: `assertEquals(message, expected, actual)`. Wie oft haben Sie `message` gelesen und gedacht, es wäre

expected? Ich bin sehr oft über diese Triade gestolpert und musste mehrfach nachlesen. Tatsächlich schaue ich *jedes Mal, wenn ich sie sehe*, doppelt nach und ignoriere dann die Nachricht.

Andererseits gibt es auch Triaden, die nicht ganz so tückisch sind: `assertEquals(1.0, amount, .001)`. Obwohl Sie auch hier doppelt hinschauen müssen, lohnt es in diesem Fall. Es ist immer gut, daran erinnert zu werden, dass die Gleichheit von Fließkommawerten eine relative Sache ist.

Argument-Objekte

Wenn eine Funktion anscheinend mehr als zwei oder drei Argumente benötigt, ist es wahrscheinlich, dass einige dieser Argumente in eine separate Klasse eingehüllt werden sollten. Betrachten Sie beispielsweise den Unterschied zwischen den beiden folgenden Deklarationen:

```
Circle makeCircle(double x, double y, double radius);  
Circle makeCircle(Point center, double radius);
```

Die Anzahl der Argumente zu reduzieren, indem daraus Objekte erstellt werden, mag wie Schummelei aussehen, ist es aber nicht. Wenn Gruppen von Variablen zusammen übergeben werden, wie etwa `x` und `y` in dem obigen Beispiel, gehören sie wahrscheinlich zu einem Konzept, das einen eigenen Namen haben sollte.

Argument-Listen

Manchmal wollen wir eine variable Anzahl von Argumenten an eine Funktion übergeben. Betrachten Sie beispielsweise die Methode `String.format`:

```
String.format("%s worked %.2f hours.", name, hours);
```

Wenn die variablen Argumente alle identisch behandelt werden, wie in dem obigen Beispiel, dann entsprechen sie einem einzigen Argument vom Typ `List`. Folgt man dieser Auffassung, ist `String.format` faktisch dyadisch. Tatsächlich ist die folgende Deklaration von `String.format` deutlich dyadisch.

```
public String format(String format, Object... args)
```

Deshalb gelten alle entsprechenden Regeln. Funktionen, die eine variable Anzahl von Argumenten übernehmen, können Monaden, Dyaden oder sogar Triaden sein. Aber es wäre ein Fehler, ihnen noch mehr Argumente zu übergeben.

```
void monad(Integer... args);  
void dyad(String name, Integer... args);  
void triad(String name, int count, Integer... args);
```

Verben und Schlüsselwörter

Ein guter Name für eine Funktion kann sehr viel dazu beitragen, den Zweck der Funktion und die Reihenfolge und den Zweck der Argumente zu erklären. Bei einer Monade sollten Funktion und Argument ein aussagestarkes Verb/Substantiv-Paar bilden. Beispielsweise erklärt sich `write(name)` von selbst. Was immer »name« sein mag, es wird »geschrieben«. Möglicherweise wäre ein Name wie `writeField(name)` noch besser, weil er uns zugleich sagt, dass der »name« ein »Feld« ist.

Diese letzte Variante ist ein Beispiel für die *Schlüsselwort*-Form eines Funktionsnamens. Bei dieser Form codieren wir die Namen der Argumente in den Funktionsnamen. Beispielsweise könnte `assertEquals` besser als `assertExpectedEqualsActual(expected, actual)` geschrieben werden. Dadurch wird das Problem, sich die Reihenfolge der Argumente merken zu müssen, erheblich geringer.

3.7 Nebeneffekte vermeiden

Nebeneffekte sind Lügen. Ihre Funktion verspricht, eine Aufgabe zu erfüllen, aber sie erledigt auch andere *verborgene* Aufgaben. Manchmal führt sie unerwartete Änderungen an Variablen ihrer eigenen Klasse durch. Manchmal macht sie daraus Parameter, die an die Funktion übergeben werden, oder System-Globals. In jedem Fall sind sie unaufrichtige und schädigende Falschheiten, die oft zu seltsamen zeitlichen Kopplungen und anderen Abhängigkeiten führen.

Betrachten Sie beispielsweise die anscheinend unverfängliche Funktion in Listing 3.6. Diese Funktion verwendet einen Standardalgorithmus, um einen `userName` mit einem `password` abzugleichen. Bei einem Treffer gibt sie `true` zurück, andernfalls `false`. Aber sie hat auch einen Nebeneffekt. Können Sie ihn entdecken?

Listing 3.6: `UserValidator.java`

```
public class UserValidator {
    private Cryptographer cryptographer;

    public boolean checkPassword(String userName, String password) {
        User user = UserGateway.findByName(userName);
        if (user != User.NULL) {
            String codedPhrase = user.getPhraseEncodedByPassword();
            String phrase = cryptographer.decrypt(codedPhrase, password);
            if ("Valid Password".equals(phrase)) {
                Session.initialize();
                return true;
            }
        }
        return false;
    }
}
```

Der Nebeneffekt ist natürlich der Aufruf von `Session.initialize()`. Die `checkPassword`-Funktion sagt laut Name, dass sie Passwörter prüft. Der Name impliziert nicht, dass sie die Sitzung initialisiert. Deshalb läuft ein Aufrufer, der glaubt, was der Name der Funktion sagt, Gefahr, die vorhandenen Sitzungsdaten zu löschen, wenn er die Validität des Benutzers prüfen will.

Dieser Nebeneffekt erzeugt eine zeitliche Kopplung. Das heißt, `checkPassword` kann nur zu bestimmten Zeiten aufgerufen werden (anders ausgedrückt: wenn es sicher ist, die Sitzung zu initialisieren). Wenn die Funktion nicht zum normalen Zeitpunkt aufgerufen wird, können Sitzungsdaten versehentlich verloren gehen. Zeitliche Kopplungen sind verwirrend, besonders wenn sie als Nebeneffekt verborgen sind. Wenn Sie eine zeitliche Kopplung brauchen, sollten Sie dies im Namen der Funktion klar zum Ausdruck bringen. In diesem Fall könnten wir die Funktion in `checkPasswordAndInitializeSession` umbenennen, obwohl dies sicherlich gegen »Tue eine Aufgabe« verstößt.

Output-Argumente

Argumente werden auf natürlichste Weise als *Inputs* einer Funktion interpretiert. Wenn Sie schon länger programmieren, haben Sie sicher schon Argumente näher analysiert, die tatsächlich nicht als Input, sondern als *Output* verwendet wurden. Ein Beispiel:

```
appendFooter(s);
```

Hängt diese Funktion `s` als Fußzeile an etwas an? Oder hängt sie eine Fußzeile an `s` an? Ist `s` ein Input oder ein Output? Es dauert nicht lange, sich die Funktionssignatur anzuschauen:

```
public void appendFooter(StringBuffer report)
```

Damit wird die Frage geklärt, aber nur auf Kosten der Prüfung der Funktionsdeklaration. Alles, was Sie zwingt, die Funktionssignatur zu prüfen, ist gleichwertig mit einem zweimaligen Nachschauen. Es ist eine kognitive Unterbrechung und sollte vermieden werden.

Vor dem Aufkommen der objektorientierten Programmierung musste man manchmal Output-Argumente verwenden. Doch in OO-Sprachen werden Output-Argumente weitgehend überflüssig, weil es eine *Aufgabe von this* ist, als Output-Argument zu agieren. Anders ausgedrückt: Es wäre besser, wenn `appendFooter` wie folgt aufgerufen werden würde:

```
report.appendFooter();
```

Im Allgemeinen sollten Sie keine Output-Argumente verwenden. Wenn Ihre Funktion den Status einer Komponente ändern muss, sollte sie den Status des Eigentümerobjekts ändern.

3.8 Anweisung und Abfrage trennen

Funktionen sollten entweder etwas tun oder etwas antworten, aber nicht beides. Entweder sollte Ihre Funktion den Status eines Objekts ändern oder sie sollte Informationen über das Objekt zurückgeben. Beides zu tun, führt oft zu Verwirrung. Betrachten Sie beispielsweise die folgende Funktion:

```
public boolean set(String attribute, String value);
```

Diese Funktion setzt den Wert eines benannten Attributs und gibt `true` zurück, wenn dies erfolgreich ist, und `false`, wenn kein solches Attribut existiert. Dies führt zu seltsamen Anweisungen wie dieser:

```
if (set("username", "unclebob"))...
```

Betrachten Sie dies aus der Sicht des Lesers. Was bedeutet dies? Fragt die Anweisung, ob das »username«-Attribut vorher auf »unclebob« gesetzt war? Oder fragt sie, ob das »username«-Attribut erfolgreich auf »unclebob« gesetzt worden ist? Es ist schwer, die Bedeutung aus dem Aufruf abzuleiten, weil nicht klar ist, ob das Wort »set« ein Verb oder ein Adjektiv ist.

Für den Autor sollte `set` ein Verb sein, aber im Kontext der `if`-Anweisung *fühlt* sich das Wort wie ein Adjektiv an. Deshalb bedeutet sie: »Wenn das `username`-Attribut vorher auf `unclebob` gesetzt war ...« und nicht »setze das `username`-Attribut auf `unclebob` und falls dies funktioniert, dann ...«. Wir könnten versuchen, dieses Problem zu beseitigen, indem wir die `set`-Funktion in `setAndCheckIfExists` umbenennen, aber das hilft dem Leser der `if`-Anweisung auch nicht viel weiter. Die richtige Lösung besteht darin, die Anweisung von der Abfrage zu trennen, damit die Mehrdeutigkeit nicht auftreten kann.

```
if (attributeExists("username")) {  
    setAttribute("username", "unclebob");  
    ...  
}
```

3.9 Ausnahmen sind besser als Fehler-Codes

Fehler-Codes von Befehlen zurückzugeben, ist eine subtile Verletzung der Anweisung-Abfrage-Trennung. Diese Technik fördert den Gebrauch von Anweisungen als Ausdrücke in den Prädikaten von `if`-Anweisungen.

```
if (deletePage(page) == E_OK)
```


Diese Anweisung leidet nicht unter der Verb/Adjektiv-Verwechslung, führt aber zu tief verschachtelten Strukturen. Wenn Sie einen Fehler-Code zurückgeben, schaffen Sie das Problem, dass der Aufrufer den Fehler sofort behandeln muss.

```
if (deletePage(page) == E_OK) {
    if (registry.deleteReference(page.name) == E_OK) {
        if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
            logger.log("page deleted");
        } else {
            logger.log("configKey not deleted");
        }
    } else {
        logger.log("deleteReference from registry failed");
    }
} else {
    logger.log("delete failed");
    return E_ERROR;
}
```

Wenn Sie dagegen Ausnahmen anstelle von Fehler-Codes verwenden, dann können Sie den Code für die Fehler-Verarbeitung von dem Code des Normalverlaufs trennen und sehr vereinfachen:

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
    logger.log(e.getMessage());
}
```

Try/Catch-Blöcke extrahieren

Try/catch-Blöcke sind von Natur aus hässlich. Sie verdunkeln die Struktur des Codes und vermengen die Fehler-Verarbeitung mit der normalen Verarbeitung. Deshalb ist es besser, die Körper der try- und catch-Blöcke in separate Funktionen zu extrahieren.

```
public void delete(Page page) {
    try {
        deletePageAndAllReferences(page);
    }
    catch (Exception e) {
        logError(e);
    }
}

private void deletePageAndAllReferences(Page page)
```

```
        throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

In dem obigen Code hat die `delete`-Funktion die Aufgabe der Fehler-Verarbeitung. Man kann sie einfach verstehen und dann ignorieren. Die Funktion `deletePageAndAllReferences` hat nur die Aufgabe, eine `page` komplett zu löschen. Die Fehler-Verarbeitung kann ignoriert werden. Damit haben wir eine saubere Trennung, die es erleichtert, den Code zu verstehen und zu ändern.

Fehler-Verarbeitung ist eine Aufgabe

Funktionen sollten eine Aufgabe erfüllen. Fehler-Verarbeitung ist eine Aufgabe. Deshalb sollte eine Funktion, die Fehler verarbeitet, nichts anderes tun. Dies impliziert (wie in dem obigen Beispiel) Folgendes: Wenn eine Funktion das Schlüsselwort `try` enthält, sollte es das allererste Wort in der Funktion sein und nach den `catch/finally`-Blöcken sollte nichts anderes stehen.

Der Abhängigkeitsmagnet Error.java

Fehler-Codes zurückzugeben, bedeutet normalerweise auch, dass es eine Klasse oder `enum` gibt, in der alle Fehler-Codes definiert sind.

```
public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Klassen wie diese sind ein *Abhängigkeitsmagnet*; viele andere Klassen müssen sie importieren und verwenden. Wird `enum Error` geändert, ist eine Neukompilierung und ein Redeployment all dieser anderen Klassen erforderlich. Dies übt einen negativen Druck auf die `Error`-Klasse aus. Programmierer wollen keine neuen Fehler hinzufügen, weil sie dann alles neu erstellen und ausliefern müssen. Deshalb wiederverwenden sie alte Fehler-Codes, anstatt neue hinzuzufügen.

Wenn Sie nicht mit Fehler-Codes, sondern mit Ausnahmen arbeiten, dann sind neue Ausnahmen *abgeleitete Klassen* der `Exception`-Klasse (Ausnahme-Klasse). Sie

können sie hinzufügen, ohne dass eine Neukompilierung und ein Redeployment erforderlich sind. Dies ist ein Beispiel für das Open-Closed-Prinzip (OCP) [PPP02].

3.10 Don't Repeat Yourself

Don't-Repeat-Yourself-Prinzip (DRY; »Wiederhole dich nicht!«; [PRAG]). Wenn Sie sich Listing 3.1 weiter vorne noch einmal sorgfältig anschauen, stellen Sie fest, dass ein Algorithmus vier Mal wiederholt wird, einmal für jeden Fall von `SetUp`, `Suite-SetUp`, `TearDown` und `SuiteTearDown`. Es ist nicht leicht, diese Duplizierung zu erkennen, weil die vier Instanzen mit anderem Code vermischt sind und nicht einheitlich dupliziert werden. Dennoch ist die Duplizierung ein Problem, weil sie den Code aufbläht und eine vierfache Änderung erfordert, sollte der Algorithmus jemals geändert werden müssen. Außerdem bietet er vier Mal die Gelegenheit für Auslassungsfehler.

Diese Duplizierung wurde durch die `include`-Methode in Listing 3.7 behoben. Lesen Sie diesen Code noch einmal und achten Sie darauf, wie die Lesbarkeit des gesamten Moduls durch die Verringerung dieser Duplizierung verbessert wird.

Möglicherweise ist die Duplizierung die Wurzel allen Übels bei der Software-Entwicklung. Viele Prinzipien und Techniken wurden zu dem Zweck entwickelt, sie zu kontrollieren oder zu eliminieren. Beispielsweise dienen alle Datenbank-Normalformen von Codd dazu, die Duplizierung von Daten zu eliminieren. Bei der objekt-orientierten Programmierung bemüht man sich darum, andernfalls redundanten Code in Basisklassen zu konzentrieren. Bei der Strukturierten Programmierung, der Aspektorientierten Programmierung, der Komponentenorientierten Programmierung geht es immer auch um Strategien, Duplizierung zu eliminieren. Man könnte sagen, dass alle Innovationen in Software-Entwicklung seit der Erfindung der Subroutine ein fortlaufender Versuch sind, Duplizierung in unserem Sourcecode zu eliminieren.

3.11 Strukturierte Programmierung

Einige Programmierer befolgen die Regeln der Strukturierten Programmierung von Edsger Dijkstra [SP72]. Dijkstra sagte, dass jede Funktion und jeder Block innerhalb einer Funktion einen Eingang und einen Ausgang haben solle. Diesen Regeln entsprechend sollte eine Funktion nur eine `return`-Anweisung, keine `break`- oder `continue`-Anweisungen in einer Schleife und niemals, wirklich *niemals*, `goto`-Anweisungen enthalten.

Obwohl wir den Zielen und Techniken der Strukturierten Programmierung wohlwollend gegenüberstehen, bieten diese Regeln wenig Vorteile, wenn Funktionen sehr klein sind. Nur bei größeren Funktionen können Sie durch solche Regeln beträchtliche Vorteile erzielen.

Wenn Sie also Ihre Funktionen klein halten, dann richtet eine gelegentliche Anwendung von mehreren `return`-, `break`- oder `continue`-Anweisungen keinen Schaden an und kann den Code manchmal ausdrucksstärker machen als die Ein-Eingang-ein-Ausgang-Regel. Andererseits lässt sich `goto` nur bei großen Funktionen sinnvoll einsetzen und sollte deshalb vermieden werden.

3.12 Wie schreibt man solche Funktionen?

Software zu schreiben, ist wie jede andere Art des Schreibens. Wenn Sie einen Aufsatz oder einen Artikel schreiben, notieren Sie zunächst Ihre Gedanken, dann ordnen Sie sie, bis sie sich gut lesen lassen. Der erste Entwurf mag unbeholfen und unstrukturiert wirken, weshalb Sie ihn umformulieren und umstrukturieren werden, bis er Ihren Vorstellungen entspricht.

Wenn ich Funktionen schreibe, sind sie zunächst lang und kompliziert. Sie haben viele Einrückungen und verschachtelte Schleifen. Sie haben lange Argumentenlisten. Die Namen sind willkürlich, und es gibt duplizierten Code. Aber ich verfüge ebenfalls über eine Suite von Unit-Tests, die alle diese umständlichen Code-Zeilen testen.

Dann massiere und verfeinere ich den Code, lagere Funktionen aus, ändere Namen und eliminiere Duplizierungen. Ich verkleinere die Methoden und stelle sie um. Manchmal breche ich ganze Klassen heraus, während ich gleichzeitig dafür Sorge, dass die Tests bestanden werden.

Schließlich folgen meine Funktionen den Regeln, die ich in diesem Kapitel beschrieben habe. Ich schreibe sie nicht in dieser Form, wenn ich anfangen. Ich glaube nicht, dass dies irgendjemand könnte.

3.13 Zusammenfassung

Jedes System wird mit einer domänenspezifischen Sprache konzipiert, die von dem Programmierer konzipiert wird, um das System zu beschreiben. Funktionen sind die Verben dieser Sprache, und Klassen sind die Substantive. Dies ist kein Rückschritt zu der scheußlichen alten Auffassung, die Substantive und die Verben in einem Anforderungsdokument lieferten die ersten Anhaltspunkte für die in einem System benötigten Klassen und Funktionen. Stattdessen kommt hier eine viel ältere Wahrheit zum Ausdruck. Die Kunst der Programmierung ist und war immer die Kunst des Sprachen-Designs.

Meister-Programmierer betrachten Systeme nicht als Programme, die geschrieben werden müssen, sondern als Geschichten, die erzählt werden wollen. Sie verwenden die Fähigkeiten der von ihnen gewählten Programmiersprache, um eine viel reichhaltigere und ausdrucksstärkere Sprache zu konstruieren, mit der diese Geschichte erzählt werden kann. Ein Teil dieser domänenspezifischen Sprache wird

von einer Hierarchie von Funktionen gebildet, die alle Aktionen beschreiben, die innerhalb des Systems ausgeführt werden. In einer kunstvollen Rekursion werden diese Aktionen so geschrieben, dass sie in derselben domänenspezifischen Sprache, die sie definieren, ihren eigenen kleinen Teil der Geschichte erzählen.

Dieses Kapitel behandelt die handwerklichen Fähigkeiten, um gute Funktionen zu schreiben. Wenn Sie die Regeln aus diesem Kapitel befolgen, werden Ihre Funktionen kurz sein, geeignete Namen haben und sauber strukturiert sein. Sie dürfen aber niemals vergessen, dass Ihr eigentliches Ziel darin besteht, die Geschichte des Systems zu erzählen, und dass Ihre Funktionen sauber zusammenpassen und in einer klaren und präzisen Sprache geschrieben sein müssen, um Ihre Erzählung zu unterstützen.

3.14 SetupTeardownIncluder

Listing 3.7: SetupTeardownIncluder.java

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
    private PageData pageData;
    private boolean isSuite;
    private WikiPage testPage;
    private StringBuffer newPageContent;
    private PageCrawler pageCrawler;

    public static String render(PageData pageData) throws Exception {
        return render(pageData, false);
    }

    public static String render(PageData pageData, boolean isSuite)
        throws Exception {
        return new SetupTeardownIncluder(pageData).render(isSuite);
    }

    private SetupTeardownIncluder(PageData pageData) {
        this.pageData = pageData;
        testPage = pageData.getWikiPage();
        pageCrawler = testPage.getPageCrawler();
        newPageContent = new StringBuffer();
    }

    private String render(boolean isSuite) throws Exception {
        this.isSuite = isSuite;
        if (isTestPage())
```

```
        includeSetupAndTeardownPages();
        return pageData.getHtml();
    }

    private boolean isTestPage() throws Exception {
        return pageData.hasAttribute("Test");
    }

    private void includeSetupAndTeardownPages() throws Exception {
        includeSetupPages();
        includePageContent();
        includeTeardownPages();
        updatePageContent();
    }

    private void includeSetupPages() throws Exception {
        if (isSuite)
            includeSuiteSetupPage();
        includeSetupPage();
    }

    private void includeSuiteSetupPage() throws Exception {
        include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
    }

    private void includeSetupPage() throws Exception {
        include("SetUp", "-setup");
    }

    private void includePageContent() throws Exception {
        newPageContent.append(pageData.getContent());
    }

    private void includeTeardownPages() throws Exception {
        includeTeardownPage();
        if (isSuite)
            includeSuiteTeardownPage();
    }

    private void includeTeardownPage() throws Exception {
        include("TearDown", "-teardown");
    }

    private void includeSuiteTeardownPage() throws Exception {
        include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
    }

    private void updatePageContent() throws Exception {
        pageData.setContent(newPageContent.toString());
    }
```

```
}

private void include(String pageName, String arg) throws Exception {
    WikiPage inheritedPage = findInheritedPage(pageName);
    if (inheritedPage != null) {
        String pagePathName = getPathNameForPage(inheritedPage);
        buildIncludeDirective(pagePathName, arg);
    }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
    return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}

private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
}

private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
        .append("\n!include ")
        .append(arg)
        .append(" .")
        .append(pagePathName)
        .append("\n");
}
}
```

Kommentare

»Kommentieren Sie schlechten Code nicht – schreiben Sie ihn um.«

– Brian W. Kernighan und P. J. Plaugher



Nichts kann so hilfreich sein wie ein wohlplatzierter Kommentar. Nichts kann ein Modul mehr zumüllen als leichtfertige dogmatische Kommentare. Nichts kann so viel Schaden anrichten wie ein alter überholter Kommentar, der Lügen und Fehlinformationen verbreitet.

Kommentare sind nicht wie Schindlers Liste. Sie sind nicht »das reine Gute«. Tatsächlich sind Kommentare bestenfalls ein erforderliches Übel. Wären unsere Programmiersprachen ausdrucksstark genug oder hätten wir genügend Talent, unsere Absichten in den vorhandenen Sprachen immer klar genug auszudrücken, wir würden wohl kaum Kommentare brauchen.

Der angemessene Einsatz von Kommentaren soll unsere Unfähigkeit ausgleichen, uns in unserem Code klar auszudrücken. Beachten Sie das Wort *Unfähigkeit*. Genau das meine ich. Kommentare sind immer ein Zeichen der Unfähigkeit. Wir brauchen sie, da es uns nicht immer gelingt herauszufinden, wie wir uns ohne sie ausdrücken können. Doch ein Kommentar ist kein Grund zum Feiern.

Wenn Sie also einen Kommentar schreiben müssen, sollten Sie vorher überlegen, ob Sie sich nicht doch noch in Ihrem Code besser ausdrücken könnten. Jedes Mal, wenn Sie sich in Ihrem Code ausdrücken können, können Sie stolz auf sich sein. Jedes Mal, wenn Sie einen Kommentar schreiben, sollten Sie sich mangelnder Ausdrucksfähigkeit im Code bewusst sein.

Warum schätze ich Kommentare so gering? Weil sie lügen. Nicht immer und nicht absichtlich, aber zu oft. Je älter ein Kommentar ist und je weiter er von dem Code entfernt ist, den er beschreibt, desto wahrscheinlicher ist er einfach falsch. Der Grund ist simpel. Realistisch betrachtet, können Programmierer Kommentare nicht warten.

Code ändert und entwickelt sich. Teile des Codes werden von hier nach da verschoben. Die Teile verzweigen, reproduzieren und vereinigen sich wieder und werden zu Schimären. Leider folgen die Kommentare ihnen nicht immer – sie *können ihnen nicht immer folgen*. Und allzu oft werden die Kommentare von dem Code getrennt, den sie beschreiben, und verwaisen als immer ungenauer werdende »Waschzettel«. Betrachten Sie beispielsweise das Schicksal des folgenden Kommentars und der Zeile, die er beschreiben sollte:

```
MockRequest request;  
private final String HTTP_DATE_REGEX =  
"[SMTWF][a-z]{2}\\s\\s[0-9]{2}\\s\\s[JFMASOND][a-z]{2}\\s"+  
"[0-9]{4}\\s\\s[0-9]{2}\\s\\s:[0-9]{2}\\s\\s:[0-9]{2}\\s\\sGMT";  
private Response response;  
private FitNesseContext context;  
private FileResponder responder;  
private Locale saveLocale;  
// Example: "Tue, 02 Apr 2003 22:18:49 GMT"
```

Wahrscheinlich wurden später andere Instanzvariablen zwischen der Konstanten HTTP_DATE_REGEX und ihrem erklärenden Kommentar eingefügt.

Natürlich könnte man fordern, dass Programmierer diszipliniert genug sein sollten, Kommentare immer zu aktualisieren, zu korrigieren usw. Ich stimme zu; sie sollten. Aber mir wäre es lieber, sie würden diese Energie aufwenden, um den Code so klar und ausdrucksstark zu formulieren, dass er überhaupt keine Kommentare benötigt.

Ungenaue Kommentare sind viel schlimmer als gar keine Kommentare. Sie führen in die Irre. Sie wecken Erwartungen, die niemals erfüllt werden. Sie schreiben alte Regeln fest, die nicht mehr befolgt werden müssen oder sogar nicht mehr befolgt werden sollten.

Die Wahrheit kann nur an einer Stelle gefunden werden: im Code. Nur der Code kann wirklich sagen, was er tut. Er ist die einzige Quelle für wirklich genaue Informationen. Deshalb sollten wir, auch wenn Kommentare manchmal erforderlich sind, uns anstrengen, um sie zu minimieren.

4.1 Kommentare sind kein Ersatz für schlechten Code

Einer der häufigeren Gründe, Kommentare zu schreiben, ist schlechter Code. Wir schreiben ein Modul und wir wissen, dass es verwirrend und schlecht strukturiert

ist. Wir wissen, dass es ein Chaos ist. Deshalb sagen wir uns: »Oh, das muss ich aber kommentieren!« Nein! Sie müssen Ihren Code säubern!

Klarer und ausdrucksstarker Code mit wenigen Kommentaren ist viel besser als unordentlicher und komplexer Code mit zahlreichen Kommentaren. Anstatt Ihre Zeit mit dem Schreiben von Kommentaren zu verbringen, die Ihr Chaos erklären, sollten Sie sie darauf verwenden, das Chaos zu beseitigen.

4.2 Erklären Sie im und durch den Code

Manchmal ist Code sicher kein gutes Medium für Erklärungen. Leider deuten viele Programmierer dies so um, dass Code selten, falls überhaupt, ein gutes Medium für Erklärungen ist. Dies ist grundfalsch. Was würden Sie lieber lesen? Dies:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Oder dies?

```
if (employee.isEligibleForFullBenefits())
```

In Deutsch etwa:

```
// Prüfen, ob der Mitarbeiter alle Benefits bekommen soll
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Oder dies?

```
if (employee.sollAlleBenefitsBekommen())
```

Meistens muss man nur wenige Sekunden nachdenken, um den Zweck durch den Code auszudrücken. In vielen Fällen reicht es einfach, eine Funktion zu erstellen, die dasselbe aussagt wie der Kommentar, den Sie schreiben wollen.

4.3 Gute Kommentare

Einige Kommentare sind notwendig oder vorteilhaft. Es folgen einige, die meiner Meinung die Bits wert sind, die sie verbrauchen. Denken Sie jedoch daran, dass der einzig wirklich gute Kommentar der ist, den Sie nicht schreiben müssen.

Juristische Kommentare

Manchmal zwingen uns die Codierstandards unseres Unternehmens, bestimmte Kommentare aus juristischen Gründen zu schreiben. Beispielsweise sind Copy-

right- und Autoren-Vermerke am Anfang aller Quelldateien oft notwendige und vernünftige Angaben.

Hier ist beispielsweise der standardmäßige Kommentar-Header, den wir am Anfang jeder Quelldatei in FitNesse einfügen. Glücklicherweise verbirgt unsere IDE diesen Kommentar, so dass er nicht den Bildschirm verunstaltet.

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.  
// Released under the terms of the GNU General Public License version 2 or later.
```

Derartige Kommentare sollten keine langen Verträge oder Ähnliches enthalten. Wenn möglich, sollten Sie auf eine Standardlizenz oder ein anderes externes Dokument verweisen, anstatt alle Bedingungen in den Kommentar einzufügen.

Informierende Kommentare

Manchmal ist es nützlich, grundlegende Informationen in einem Kommentar mitzuteilen. So erklärt etwa der folgende Kommentar den Rückgabewert einer abstrakten Methode:

```
// Gibt eine Instanz des getesteten Responders zurück  
protected abstract Responder responderInstance();
```

Ein derartiger Kommentar kann manchmal nützlich sein; aber es ist besser, die Information möglichst mit dem Namen der Funktion zu vermitteln. So könnte etwa der Kommentar in diesem Fall redundant gemacht werden, indem man die Funktion umbenennt: `responderBeingTested`.

Hier ist ein etwas besserer Fall:

```
// Vergleichsformat kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern.compile(  
    "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

In diesem Fall teilt uns der Kommentar mit, dass der reguläre Ausdruck zum Abgleich eines Ausdrucks mit Datum und Zeit verwendet werden soll, der mit der Funktion `SimpleDateFormat.format` und dem spezifizierten Format-String formatiert worden ist. Dennoch könnte es besser und klarer gewesen sein, wenn dieser Code in eine spezielle Klasse verschoben worden wäre, die für die Umwandlung der Formate von Datums- und Zeitangaben zuständig gewesen wäre. Dann wäre der Kommentar wahrscheinlich überflüssig gewesen.

Erklärung der Absicht

Manchmal liefert ein Kommentar nicht nur nützliche Informationen über die Implementierung, sondern auch über die Gründe hinter einer Entscheidung. Im folgenden Fall wird eine interessante Entscheidung durch einen Kommentar dokumentiert. Um zwei Objekte zu vergleichen, wollte der Autor sie so sortieren,

dass Objekte seiner Klasse in der Reihenfolge vor den Objekten aller anderen Klassen stehen.

```
public int compareTo(Object o)
{
    if(o instanceof WikiPagePath)
    {
        WikiPagePath p = (WikiPagePath) o;
        String compressedName = StringUtil.join(names, "");
        String compressedArgumentName = StringUtil.join(p.names, "");
        return compressedName.compareTo(compressedArgumentName);
    }
    return 1; // Wir sind größer, weil wir den richtigen Typ haben.
}
```

Hier ist ein noch besseres Beispiel. Vielleicht stimmen Sie nicht mit der Problemlösung des Programmierers überein, aber wenigstens wissen Sie, was er versuchte.

```
public void testConcurrentAddWidgets() throws Exception {
    WidgetBuilder widgetBuilder =
        new WidgetBuilder(new Class[]{BoldWidget.class});
    String text = ""'"bold text'"";
    ParentWidget parent =
        new BoldWidget(new MockWidgetRoot(), ""'"bold text'"");
    AtomicBoolean failFlag = new AtomicBoolean();
    failFlag.set(false);

    // Dies ist unser bester Versuch, eine Race-Condition
    // (Gleichzeitigkeitsbedingung) zu erzeugen,
    // indem eine große Anzahl von Threads erstellt wird.
    for (int i = 0; i < 25000; i++) {
        WidgetBuilderThread widgetBuilderThread =
            new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
        Thread thread = new Thread(widgetBuilderThread);
        thread.start();
    }
    assertEquals(false, failFlag.get());
}
```

Klarstellungen

Manchmal ist es einfach hilfreich, die Bedeutung eines obskuren Arguments oder Rückgabewerts in etwas Lesbares zu übersetzen. Im Allgemeinen ist es besser, eine Methode zu finden, die das Argument oder den Rückgabewert von sich aus klar macht; aber wenn sie zu einer Standard-Library gehört oder in Code steht, den Sie nicht ändern können, dann kann ein hilfreicher klärender Kommentar nützlicher sein.

Natürlich besteht ein beträchtliches Risiko, dass ein klärender Kommentar falsch ist. So zeigt ein Blick auf das vorhergehende Beispiel, wie schwierig seine Korrektur

```
public void testCompareTo() throws Exception
{
    WikiPagePath a = PathParser.parse("PageA");
    WikiPagePath ab = PathParser.parse("PageA.PageB");
    WikiPagePath b = PathParser.parse("PageB");
    WikiPagePath aa = PathParser.parse("PageA.PageA");
    WikiPagePath bb = PathParser.parse("PageB.PageB");
    WikiPagePath ba = PathParser.parse("PageB.PageA");

    assertTrue(a.compareTo(a) == 0);    // a == a
    assertTrue(a.compareTo(b) != 0);    // a != b
    assertTrue(ab.compareTo(ab) == 0);  // ab == ab
    assertTrue(a.compareTo(b) == -1);   // a < b
    assertTrue(aa.compareTo(ab) == -1); // aa < ab
    assertTrue(ba.compareTo(bb) == -1); // ba < bb
    assertTrue(b.compareTo(a) == 1);    // b > a
    assertTrue(ab.compareTo(aa) == 1);  // ab > aa
    assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

heit zu verifizieren ist. Dies erklärt, warum die Klarstellung erforderlich ist und warum sie riskant ist. Bevor Sie derartige Kommentare schreiben, sollten Sie deshalb sorgfältig prüfen, ob es keine bessere Alternative gibt, und dann noch sorgfältiger darauf achten, dass sie korrekt sind.

Warnungen vor Konsequenzen

Manchmal ist es nützlicher, andere Programmierer vor bestimmten Konsequenzen zu warnen. Beispielsweise erklärt der folgende Kommentar, warum ein spezieller Testfall abgeschaltet wurde:

```
// Nur ausführen, wenn Sie Zeit totschiessen wollen.
public void _testWithReallyBigFile()
{
    writelinesToFile(100000000);
    response.setBody(testFile);
    response.readyToSend(this);
    String responseString = output.toString();
    assertSubString("Content-Length: 1000000000", responseString);
    assertTrue(bytesSent > 1000000000);
}
```

Heutzutage schalten wir den Testfall natürlich mit einem `@Ignore`-Attribut und einem entsprechenden aussagekräftigen String ab. `@Ignore("Die Ausführung dauert zu lange.")`. Doch bevor es JUnit 4 gab, war es eine verbreitete Konvention, ein Unterstrichungszeichen vor den Methodennamen zu setzen. Der Kommentar ist zwar etwas flapsig, drückt aber den Punkt ziemlich gut aus.

Hier ist ein anderes, treffenderes Beispiel:

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
    // SimpleDateFormat ist nicht thread-sicher;
    // deshalb müssen wir jede Instanz unabhängig erstellen.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}
```

Sie könnten einwenden, dass es bessere Methoden zur Lösung dieses Problems gibt. Ich würde Ihnen vielleicht sogar zustimmen. Aber der hier gezeigte Kommentar ist vernünftig. Er warnt übereifrige Programmierer davor, aus Gründen der Effizienz einen statischen Initialisierer zu verwenden.

TODO-Kommentare

Manchmal ist es vernünftig, »To do«-Vermerke in Form von //TODO-Kommentaren in den Code einzufügen. Im folgenden Fall erklärt der TODO-Kommentar, warum die Funktion eine veraltete Implementierung hat und wie die Funktion in Zukunft aussehen sollte.

```
// TODO-MdM dies wird nicht benötigt
// Wir erwarten, dass sie mit unserem Checkout-Modell überflüssig wird.
protected VersionInfo makeVersion() throws Exception
{
    return null;
}
```

TODOs sind Aufgaben, die nach Meinung des Programmierers erledigt werden sollten, aber im Moment, aus welchem Grund auch immer, nicht gelöst werden können. Es könnte sich um eine Erinnerung handeln, eine veraltete Funktion zu ändern, oder eine Bitte an einen anderen Entwickler, sich des Problems anzunehmen. Es könnte eine Bitte sein, einen besseren Namen zu erfinden, oder eine Erinnerung, eine Änderung vorzunehmen, die von einem geplanten Ereignis abhängig ist. Doch eines ist ein TODO keinesfalls: Es ist *keine Entschuldigung* dafür, schlechten Code im System zu lassen.

Heutzutage stellen die meisten guten IDEs spezielle Funktionen zur Verfügung, um alle TODO-Kommentare zu finden. Deshalb ist es unwahrscheinlich, dass sie verloren gehen. Dennoch sollten Sie Ihren Code nicht mit TODOs verunstalten. Deshalb sollten Sie Ihren Code regelmäßig durchsuchen und nicht mehr aktuelle TODOs löschen.

Verstärkung

Mit einem Kommentar kann auch die Bedeutung eines Teil des Codes unterstrichen werden, der andernfalls als unbedeutend angesehen werden könnte.

```
String listItemContent = match.group(3).trim();  
// Der trim-Befehl ist wirklich wichtig. Er entfernt die führenden  
// Leerzeichen, die dazu führen könnten, dass das Element als  
// eine weitere Liste interpretiert wird.  
new ListItemWidget(this, listItemContent, this.level + 1);  
return buildList(text.substring(match.end()));
```

Javadocs in öffentlichen APIs

Kaum etwas anderes ist so hilfreich und befriedigend wie ein wohlgeschriebenes öffentliches API. Die Javadocs für die Standard-Java-Library sind ein Beispiel dafür. Es wäre bestenfalls schwierig, Java-Programme ohne sie zu schreiben.

Wenn Sie ein öffentliches API schreiben, sollten Sie sicher gute Javadocs dafür schreiben. Aber Sie sollten dabei an die anderen Empfehlungen in diesem Kapitel denken. Javadocs können genauso irreführend, nicht-lokal und unehrlich sein wie andere Kommentare.

4.4 Schlechte Kommentare

Die meisten Kommentare gehören zu dieser Kategorie. Normalerweise handelt es sich um Krücken oder Entschuldigungen für schlechten Code oder Rechtfertigungen für unzureichende Entscheidungen, die kaum über ein Selbstgespräch des Programmierers hinausgehen.

Geraune

Einen Kommentar nur deshalb einzufügen, weil Sie das Gefühl haben, Sie sollten dies tun, oder weil der Prozess es erfordert, ist ein Hack. Wenn Sie sich dafür entscheiden, einen Kommentar zu schreiben, sollten Sie die erforderliche Zeit aufwenden, um den Ihren Fähigkeiten entsprechend bestmöglichen Kommentar zu schreiben.

Hier ist beispielsweise ein Fall aus FitNesse, bei dem ein Kommentar möglicherweise nützlicher gewesen wäre. Aber der Autor war in Eile oder einfach nur nachlässig. Sein »Geraune« hinterließ ein Geheimnis:

```
public void loadProperties()  
{  
    try  
    {  
        String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;  
        FileInputStream propertiesStream = new FileInputStream(propertiesPath);  
        loadedProperties.load(propertiesStream);  
    }  
    catch(IOException e)  
    {  
        // No properties files means all defaults are loaded  
    }  
}
```

(Deutsche Übersetzung des Kommentars: »Keine Properties-Dateien bedeutet, dass alle Defaults oder Standardwerte geladen sind oder werden«; nicht eindeutig!) Was bedeutet der Kommentar in dem `catch`-Block? Man muss annehmen, dass es für den Autor etwas bedeutete, doch was, geht kaum klar daraus hervor. Was können wir aus dem Code schließen? Wenn eine `IOException` ausgelöst wird, gab es offensichtlich keine Properties-Datei; und in diesem Fall werden alle Standardwerte geladen. Aber wer lädt die Standardwerte? Waren sie bereits vor dem Aufruf von `loadProperties.load` geladen? Oder fängt `loadProperties.load` die Ausnahme ab, lädt die Standardwerte und übergeht dann die Ausnahme, so dass wir uns nicht darum kümmern müssen? Oder hat `loadProperties.load` alle Standardwerte geladen, bevor sie versuchte, die Datei zu laden? Versuchte der Autor nur, sich über die Tatsache hinwegzumogeln, dass er den `catch`-Block leer ließ? Oder – und dies ist die beängstigende Möglichkeit – wollte der Autor sich ermahnen, später an diese Stelle zurückzukommen und den Code zu schreiben, der die Standardwerte lädt?

Unsere einzige Option besteht darin, den Code in den Teilen des Systems zu studieren, um herauszufinden, was passiert. Jeder Kommentar, der Sie zwingt, in anderen Modulen nach seiner Bedeutung zu suchen, ist ein gescheiterter Kommunikationsversuch und die Bits nicht wert, die er konsumiert.

Redundante Kommentare

Listing 4.1 zeigt eine einfache Funktion mit einem Header-Kommentar, der vollkommen redundant ist. Den Kommentar zu lesen, dauert wahrscheinlich länger, als den Code selbst zu lesen.

```
Listing 4.1:    waitForClose
// Utility method that returns when this.closed is true. Throws an
// exception if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
    if(!closed)
    {
        wait(timeoutMillis);
        if(!closed)
            throw new Exception("MockResponseSender could not be closed");
    }
}
```

(Übersetzung des Kommentars: »Utility-Methode, die zurückkehrt, wenn `this.closed` wahr ist. Löst eine Ausnahme aus, wenn der Timeout erreicht ist.«) Welchen Zweck erfüllt dieser Kommentar? Es ist sicher nicht informativer als der Code. Weder rechtfertigt er den Code noch nennt er seinen Zweck oder Existenzgrund. Er ist nicht leichter zu lesen als der Code. Tatsächlich ist er weniger präzise als der Code und verführt den Leser, diesen Mangel an Präzision zu akzeptieren, anstatt sich zu

bemühen, den Code zu verstehen. Der Kommentar ähnelt einem Gebrauchtwagenhändler, der Sie mit schönen Worten davon abhalten will, unter die Haube zu schauen.

Betrachten Sie jetzt die zahlreichen nutzlosen und redundanten Javadocs in Listing 4.2 aus Tomcat. Diese Kommentare bewirken nur, dass der Code unübersichtlicher und unklarer wird. Sie erfüllen gar keinen dokumentarischen Zweck. Um die Sache schlimmer zu machen: Ich zeige nur die ersten Kommentare. Dieses Modul enthält zahlreiche weitere.

Listing 4.2: ContainerBase.java (Tomcat)

```
public abstract class ContainerBase
    implements Container, Lifecycle, Pipeline,
    MBeanRegistration, Serializable {

    /**
     * The processor delay for this component.
     */
    protected int backgroundProcessorDelay = -1;

    /**
     * The lifecycle event support for this component.
     */
    protected LifecycleSupport lifecycle =
        new LifecycleSupport(this);

    /**
     * The container event listeners for this Container.
     */
    protected ArrayList listeners = new ArrayList();

    /**
     * The Loader implementation with which this Container is
     * associated.
     */
    protected Loader loader = null;

    /**
     * The Logger implementation with which this Container is
     * associated.
     */
    protected Log logger = null;

    /**
     * Associated logger name.
     */
    protected String logName = null;

    /**
     * The Manager implementation with which this Container is
```

```
* associated.
*/
protected Manager manager = null;

/**
 * The cluster with which this Container is associated.
 */
protected Cluster cluster = null;

/**
 * The human-readable name of this Container.
 */
protected String name = null;

/**
 * The parent Container to which this Container is a child.
 */
protected Container parent = null;

/**
 * The parent class loader to be configured when we install a
 * Loader.
 */
protected ClassLoader parentClassLoader = null;

/**
 * The Pipeline object with which this Container is
 * associated.
 */
protected Pipeline pipeline = new StandardPipeline(this);

/**
 * The Realm with which this Container is associated.
 */
protected Realm realm = null;

/**
 * The resources DirContext object with which this Container
 * is associated.
 */
protected DirContext resources = null;
```

Irreführende Kommentare

Selbst bei den besten Absichten fügt ein Programmierer manchmal Aussagen in seine Kommentare ein, die nicht präzise genug sind, um korrekt zu sein. Betrachten Sie noch einmal den redundanten, doch auch subtil irreführenden Kommentar aus Listing 4.1.

Haben Sie bemerkt, wo der Kommentar Sie in die Irre führt? Die Methode kehrt nicht zurück, *wenn* `this.closed` den Wert `true` annimmt. Sie kehrt zurück, *falls* `this.closed` den Wert `true` hat; andernfalls wartet sie auf einen blinden Timeout und löst dann eine Ausnahme aus, *falls* `this.closed` immer noch nicht den Wert `true` hat.

Diese subtile Fehlinformation, die in einem Kommentar eingebettet ist, der schwerer zu lesen ist als der Body des Codes selbst, könnte einen anderen Programmierer veranlassen, die Funktion unbekümmert aufzurufen und zu erwarten, dass sie zurückkehrt, sobald `this.closed` den Wert `true` annimmt. Dieser arme Programmierer würde dann seine Zeit mit Debugging-Sitzungen vergeuden müssen, um herauszufinden, warum sein Code so langsam ausgeführt wird.

Vorgeschriebene Kommentare

Es ist einfach albern, per Regel vorzuschreiben, dass jede Funktion eine Javadoc oder jede Variable einen Kommentar haben müsse. Derartige Kommentare machen den Code nur unübersichtlicher, verbreiten Lügen und führen zu einer allgemeinen Verwirrung und Unordnung.

Beispielsweise führt die Forderung, jede Funktion müsse Javadocs haben, zu Abscheulichkeiten wie etwa in Listing 4.3. Dieser Wust von Kommentar liefert keine zusätzlichen Informationen, macht den Code nur unklarer und birgt das Potenzial für Lügen und Irreführungen.

Listing 4.3

```
/**
 *
 * @param title The title of the CD
 * @param author The author of the CD
 * @param tracks The number of tracks on the CD
 * @param durationInMinutes The duration of the CD in minutes
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
    CD cd = new CD();
    cd.title = title;
    cd.author = author;
    cd.tracks = tracks;
    cd.duration = duration;
    cdList.add(cd);
}
```

Tagebuch-Kommentare

Manche Entwickler fügen jedes Mal, wenn sie ein Modul editieren, an seinem Anfang einen Kommentar ein. Diese Kommentare bilden im Laufe der Zeit eine Art von Tagebuch oder Protokoll aller jemals vorgenommenen Änderungen. Ich habe

Module gesehen, die Dutzende von Seiten dieser fortlaufenden Tagebucheinträge enthielten.

```
~Changes (from 11-Oct-2001)
* -----
*11-Oct-2001 : Re-organised the class and moved it to new package
               com.jrefinery.date (DG);
*05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
               class (DG);
*12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
               class is gone (DG); Changed getPreviousDayOfWeek(),
               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
               bugs (DG);
*05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
*29-May-2002 : Moved the month constants into a separate interface
               (MonthConstants) (DG);
*27-Aug-2002 : Fixed bug in addMonths() method, thanks to N????levka Petr (DG);
*03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
*13-Mar-2003 : Implemented Serializable (DG);
*29-May-2003 : Fixed bug in addMonths method (DG);
*04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
*05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
```

Vor langer Zeit gab es einen guten Grund für die Erstellung und Pflege solcher Protokolleinträge am Anfang jedes Moduls. Es gab keine Sourcecode-Control-Systeme, die dies für uns erledigten. Heute sind diese langen Journale jedoch eine weitere Form von Stördaten, die das Modul unübersichtlicher machen. Sie sollten vollkommen entfernt werden.

Geschwätz

Manchmal stoßen Sie auf Kommentare, die reines Geschwätz sind. Sie wiederholen das Offensichtliche und liefern keine neuen Informationen.

```
/**
 *Default constructor.
 */
protected AnnualDateRule() {
}
```

Tatsächlich? Oder wie wäre es damit:

```
/** Der Tag des Monats. */
private int dayOfMonth;
```

Und dann der Inbegriff der Redundanz (dt.: »Den Tag des Monats zurückgeben.«):

```
/**
 * Returns the day of the month.
 */
```

```
* @return the day of the month.  
*/  
public int getDayOfMonth() {  
    return dayOfMonth;  
}
```

Diese Kommentare sind so geschwätzig, dass wir lernen, sie zu ignorieren. Wenn wir den Code lesen, springen unsere Augen einfach über sie hinweg. Irgendwann fangen die Kommentare dann an zu lügen, wenn sich der umliegende Code ändert.

Der erste Kommentar in Listing 4.4 scheint angemessen zu sein. (Der gegenwärtige Trend bei der Entwicklung der IDEs, auch die Rechtschreibung der Kommentare zu prüfen, ist Balsam für die Seelen von Entwicklern, die sehr viel Code lesen.) Er erklärt, warum der `catch`-Block ignoriert wird. Aber der zweite Kommentar ist reines Geschwätz. Anscheinend war der Programmierer einfach so frustriert, `try/catch`-Blöcke in dieser Funktion zu schreiben, dass er Dampf ablassen musste.

Listing 4.4: startSending

```
private void startSending()  
{  
    try  
    {  
        doSending();  
    }  
    catch(SocketException e)  
    {  
        // normal. someone stopped the request.  
    }  
    catch(Exception e)  
    {  
        try  
        {  
            response.add(Responder.makeExceptionString(e));  
            response.closeAll();  
        }  
        catch(Exception e1)  
        {  
            //Give me a break!  
        }  
    }  
}
```

Statt seine Mühe in einen wertlosen Kommentar zu stecken, hätte der Programmierer erkennen müssen, dass seine Frustration durch eine Verbesserung der Struktur seines Codes hätte behoben werden können. Er hätte seine Energie darauf lenken sollen, den letzten `try/catch`-Block in eine separate Funktion zu extrahieren (siehe Listing 4.5).

Listing 4.5: startSending (nach Refactoring)

```
private void startSending()
{
    try
    {
        doSending();
    }
    catch(SocketException e)
    {
        // normal. someone stopped the request.
    }
    catch(Exception e)
    {
        addExceptionAndCloseResponse(e);
    }
}

private void addExceptionAndCloseResponse(Exception e)
{
    try
    {
        response.add(ErrorResponder.makeExceptionString(e));
        response.closeAll();
    }
    catch(Exception e1)
    {
    }
}
```

Widerstehen Sie der Versuchung, geschwätzige Kommentare einzufügen, mit der Entschlossenheit, Ihren Code zu säubern. Sie werden dadurch ein besserer und glücklicherer Programmierer.

Beängstigendes Geschwätz

Javadocs können ebenfalls geschwätzig sein. Welchen Zweck erfüllen die folgenden Javadocs (aus einer bekannten Open-Source-Library)? Antwort: keinen! Sie sind nur redundante geschwätzige Kommentare, die aus einem fehlgeleiteten Wunsch heraus geschrieben wurden, den Code zu dokumentieren.

```
/** The name. */
private String name;

/** The version. */
private String version;

/** The licenceName. */
private String licenceName;
```

```
/** The version. */  
private String info;
```

Lesen Sie diese Kommentare noch einmal etwas sorgfältiger. Erkennen Sie den Cut-Paste-Fehler (Ausschneiden-Einfügen-Fehler)? Wenn Autoren nicht aufpassen, wenn sie Kommentare schreiben (oder einfügen), warum sollten sie dann von den Lesern erwarten dürfen, von den Kommentaren zu profitieren?

Verwenden Sie keinen Kommentar, wenn Sie eine Funktion oder eine Variable verwenden können

Betrachten Sie den folgenden Code-Ausschnitt:

```
// does the module from the global list <mod> depend on the  
// subsystem we are part of?  
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Man könnte diesen Code auch ohne den Kommentar wie folgt schreiben:

```
ArrayList moduleDependees = smodule.getDependSubsystems();  
String ourSubSystem = subSysMod.getSubSystem();  
if (moduleDependees.contains(ourSubSystem))
```

Möglicherweise hat der Autor des ursprünglichen Codes den Kommentar zuerst geschrieben (was unwahrscheinlich ist) und dann den Code ergänzt, um den Kommentar zu erfüllen. Doch dann hätte der Autor ein Refactoring des Codes durchführen sollen, wie ich es getan habe, um den Kommentar zu entfernen.

Positionsbezeichner

Manchmal markieren Programmierer spezielle Positionen in einer Quelldatei. Beispielsweise fand ich kürzlich folgende Zeile in einem Programm:

```
// Actions //////////////////////////////////////
```

Gelegentlich ist es sinnvoll, bestimmte Funktionen unter einem derartigen Banner zusammenzufassen. Aber im Allgemeinen sind solche Zeilen Ballast, der eliminiert werden sollte – besonders die überladen wirkende Folge von Schrägstrichen am Ende.

Sehen Sie es einmal so: Banner sind nur dann auffällig, wenn sie sparsam verwendet werden. Sie sind nur nützlich, wenn sie einen beträchtlichen Vorteil bieten. Wenn Sie zu viele Banner benutzen, wirken sie wie Hintergrundgeschwätz und werden ignoriert.

Kommentare hinter schließenden Klammern

Manchmal setzen Programmierer besondere Kommentare hinter schließende Klammern (siehe Listing 4.6). Auch wenn dies bei langen Funktionen mit tief verschachtelten Strukturen sinnvoll sein mag, bedeutet dies bei den von uns bevorzugten kleinen und eingekapselten Funktionen nur zusätzlichen Ballast. Wenn Sie also versucht sind, Ihre schließenden Klammern zu kommentieren, sollten Sie stattdessen versuchen, Ihre Funktionen zu verkürzen.

Listing 4.6: wc.java

```
public class wc {
    public static void main(String[] args) {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String line;
        int lineCount = 0;
        int charCount = 0;
        int wordCount = 0;
        try {
            while ((line = in.readLine()) != null) {
                lineCount++;
                charCount += line.length();
                String words[] = line.split("\\W");
                wordCount += words.length;
            } //while
            System.out.println("wordCount = " + wordCount);
            System.out.println("lineCount = " + lineCount);
            System.out.println("charCount = " + charCount);
        } //try
        catch (IOException e) {
            System.err.println("Error:" + e.getMessage());
        } //catch
    } //main
}
```

Zuschreibungen und Nebenbemerkungen

```
/* Added by Rick */
```

Sourcecode-Control-Systeme können sich sehr gut merken, wer, wann, was hinzugefügt hat. Es ist nicht notwendig, den Code mit kleinen Nebenbemerkungen zu verschmutzen. Vielleicht meinen Sie, solche Kommentare wären nützlich, um andere zu informieren, die über den Code reden. Aber in Wirklichkeit bleiben die Kommentare einfach jahrelang stehen und werden immer ungenauer und unwichtiger.

Auch hier ist das Sourcecode-Control-System ein besserer Platz für diese Art von Informationen.

Auskommentierter Code

Nur wenige Techniken sind so widerlich wie auskommentierter Code. Lassen Sie es einfach!

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Andere, die diesen auskommentierten Code sehen, werden nicht den Mut aufbringen, ihn zu löschen. Sie werden glauben, dass es einen Grund gibt, warum der Code dort steht, und dass er zu wichtig ist, um gelöscht zu werden. Deshalb sammelt sich auskommentierter Code an wie der Satz auf dem Boden einer Flasche mit schlechtem Wein.

Betrachten Sie den folgenden Code von Apache Commons:

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
    writeEnd();
    this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
else {
    this.pngBytes = null;
}
return this.pngBytes;
```

Warum sind diese beiden Codezeilen auskommentiert? Sind sie wichtig? Wurden sie als Erinnerungen an erforderliche Änderungen stehen gelassen? Oder sind sie einfach Abfall, den jemand vor Jahren auskommentiert hat, ohne sich dann darum zu kümmern, den Code zu bereinigen?

Vor langer Zeit, in den 60er-Jahren, war das Auskommentieren von Code manchmal nützlich. Aber seit sehr langer Zeit verfügen wir schon über gute Sourcecode-Control-Systeme. Diese Systeme merken sich den Code für uns. Wir müssen ihn nicht mehr auskommentieren. Löschen Sie einfach den Code. Wir werden ihn nicht verlieren. Versprochen.

HTML-Kommentare

HTML in Sourcecode-Kommentaren ist abscheulich. Lesen Sie nur den folgenden Beispielcode. HTML erschwert das Lesen des Kommentars an einer Stelle, wo er leicht zu lesen sein sollte – im Editor oder in der IDE. Wenn Kommentare durch ein

Werkzeug (wie etwa Javadoc) für eine Webseite extrahiert werden sollen, dann sollte es Aufgabe dieses Werkzeugs sein, die Kommentare mit geeigneten HTML-Tags auszuzeichnen, nicht die des Programmierers.

```
/**
 *Task to run fit tests.
 *This task runs fitness tests and publishes the results.
 *<p/>
 *<pre>
 *Usage:
 *<taskdef name="execute-fitness-tests"
 *classname="fitnesse.ant.ExecuteFitnessTestsTask"
 *classpathref="classpath" />
 *OR
 *<taskdef classpathref="classpath"
 *resource="tasks.properties" />
 *<p/>
 *<execute-fitness-tests
 *suitepage="FitNesse.SuiteAcceptanceTests"
 *fitnessesport="8082"
 *resultsdir="${results.dir}"
 *resultshmlpage="fit-results.html"
 *classpathref="classpath" />
 */
```

Nicht-lokale Informationen

Wenn Sie einen Kommentar schreiben, müssen Sie dafür sorgen, dass er in der Nähe des beschriebenen Codes steht. Das bedeutet zum Beispiel: keine systemweit gültigen Informationen im Kontext eines lokalen Kommentars! Betrachten Sie beispielsweise den folgenden Javadoc-Kommentar. Abgesehen von der Tatsache, dass er schrecklich redundant ist, enthält er auch Informationen über den Standard-Port. Dennoch hat die Funktion absolut keine Kontrolle über diesen Standardwert. Der Kommentar beschreibt nicht die Funktion, sondern einen anderen, weit entfernten Teil des Systems. Natürlich gibt es keine Garantie, dass dieser Kommentar geändert wird, wenn der Code mit dem Standardwert geändert wird.

```
/**
 *Port on which fitness would run. Defaults to <b>8082</b>.
 *
 *@param fitnessPort
 */
public void setFitnessPort(int fitnessPort)
{
    this.fitnessPort = fitnessPort;
}
```

Zu viele Informationen

Fügen Sie keine interessanten historischen Diskussionen oder irrelevante Beschreibungen von Details in Ihre Kommentare ein. Der folgende Kommentar stammt aus einem Modul, mit dem eine Funktion getestet werden kann, die base64 codieren und decodieren kann. Abgesehen von der RFC-Nummer sind alle anderen tief-schürfenden Informationen für einen Leser des Kommentars irrelevant.

```
/*  
  RFC 2045 – Multipurpose Internet Mail Extensions (MIME)  
  Part One: Format of Internet Message Bodies  
  section 6.8. Base64 Content-Transfer-Encoding  
  The encoding process represents 24-bit groups of input bits as output  
  strings of 4 encoded characters. Proceeding from left to right, a  
  24-bit input group is formed by concatenating 3 8-bit input groups.  
  These 24 bits are then treated as 4 concatenated 6-bit groups, each  
  of which is translated into a single digit in the base64 alphabet.  
  When encoding a bit stream via the base64 encoding, the bit stream  
  must be presumed to be ordered with the most-significant-bit first.  
  That is, the first bit in the stream will be the high-order bit in  
  the first 8-bit byte, and the eighth bit will be the low-order bit in  
  the first 8-bit byte, and so on.  
*/
```

Unklarer Zusammenhang

Der Zusammenhang zwischen einem Kommentar und dem von ihm beschriebenen Code sollte offensichtlich sein. Wenn Sie sich die Mühe machen, einen Kommentar zu schreiben, sollte der Leser wenigstens den Kommentar und den Code anschauen und verstehen können, worum es in dem Kommentar geht.

Betrachten Sie beispielsweise den folgenden Kommentar aus Apache Commons:

```
/*  
 * start with an array that is big enough to hold all the pixels  
 * (plus filter bytes), and an extra 200 bytes for header info  
 */  
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Was ist ein Filter-Byte? Hat es mit der +1 zu tun? Oder mit dem *3? Beidem? Ist ein Pixel ein Byte? Warum 200? Der Zweck eines Kommentars besteht darin, Code zu erklären, der sich nicht selbst erklärt. Es ist eine Schande, dass ein Kommentar selbst erklärt werden muss.

Funktions-Header

Kurze Funktionen müssen kaum beschrieben werden. Ein wohlgewählter Name für eine kleine Funktion, die eine Aufgabe erledigt, ist normalerweise besser als ein Kommentar-Header.

Javadocs in nicht-öffentlichem Code

So nützlich Javadocs für öffentliche APIs sind, so schädlich sind sie bei Code, der nicht für den öffentlichen Gebrauch bestimmt ist. Javadoc-Seiten für die Klassen und Funktionen innerhalb eines Systems zu generieren, ist im Allgemeinen nicht nützlich; und die zusätzliche Formalität der Javadoc-Kommentare erzeugt kaum mehr als noch mehr Ballast und Ablenkung.

Beispiel

Ich schrieb das Modul in Listing 4.7 für die erste *XP Immersion*. Es sollte ein Beispiel für einen schlechten Codier- und Kommentarstil zeigen. Kent Beck führte dann ein Refactoring des folgenden Codes vor mehreren Dutzend begeisterten Studenten in eine viel angenehmere Form durch. Später passte ich das Beispiel für mein Buch *Agile Software Development, Principles, Patterns, and Practices* sowie den ersten meiner *Craftsman*-Artikel in der Zeitschrift *Software Development* an.

Eines fasziniert mich bei diesem Modul ganz besonders: Es gab eine Zeit, in der viele von uns dieses Modul als »wohldokumentiert« bezeichnet hätten. Heute betrachten wir es als ein kleines Chaos. Schauen Sie, wie viele verschiedene Kommentar-Probleme Sie entdecken können.

Listing 4.7: GeneratePrimes.java

```
/**
 *This class Generates prime numbers up to a user specified
 *maximum. The algorithm used is the Sieve of Eratosthenes.
 *<p>
 *Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
 *d. c. 194, Alexandria. The first man to calculate the
 *circumference of the Earth. Also known for working on
 *calendars with leap years and ran the library at Alexandria.
 *<p>
 *The algorithm is quite simple. Given an array of integers
 *starting at 2. Cross out all multiples of 2. Find the next
 *uncrossed integer, and cross out all of its multiples.
 *Repeat until you have passed the square root of the maximum
 *value.
 *
 *@author Alphonse
 *@version 13 Feb 2002 atp
 */
import java.util.*;

public class GeneratePrimes
{
    /**
     *@param maxValue is the generation limit.
     */
    public static int[] generatePrimes(int maxValue)
    {
```

```
if (maxValue >= 2) // the only valid case
{
    // declarations
    int s = maxValue + 1; // size of array
    boolean[] f = new boolean[s];
    int i;

    // initialize array to true.
    for (i = 0; i < s; i++)
        f[i] = true;

    // get rid of known non-primes
    f[0] = f[1] = false;

    // sieve
    int j;
    for (i = 2; i < Math.sqrt(s) + 1; i++)
    {
        if (f[i]) // if i is uncrossed, cross its multiples.
        {
            for (j = 2 * i; j < s; j += i)
                f[j] = false; // multiple is not prime
        }
    }

    // how many primes are there?
    int count = 0;
    for (i = 0; i < s; i++)
    {
        if (f[i])
            count++; // bump count.
    }

    int[] primes = new int[count];
    // move the primes into the result
    for (i = 0, j = 0; i < s; i++)
    {
        if (f[i]) // if prime
            primes[j++] = i;
    }

    return primes; // return the primes
}
else // maxValue < 2
    return new int[0]; // return null array if bad input.
}
```

Listing 4.8 enthält eine Version desselben Moduls nach dem durchgeführten Refactoring. Beachten Sie, dass es erheblich weniger Kommentare enthält. Es gibt nur zwei Kommentare in dem gesamten Modul. Beide Kommentare erklären etwas.

Listing 4.8: PrimeGenerator.java (nach Refactoring)

```
/**
 *This class Generates prime numbers up to a user specified
 *maximum. The algorithm used is the Sieve of Eratosthenes.
 *Given an array of integers starting at 2:
 *Find the first uncrossed integer, and cross out all its
 *multiples. Repeat until there are no more multiples
 *in the array.
 */

public class PrimeGenerator
{
    private static boolean[] crossedOut;
    private static int[] result;

    public static int[] generatePrimes(int maxValue)
    {
        if (maxValue < 2)
            return new int[0];
        else
        {
            uncrossIntegersUpTo(maxValue);
            crossOutMultiples();
            putUncrossedIntegersIntoResult();
            return result;
        }
    }

    private static void uncrossIntegersUpTo(int maxValue)
    {
        crossedOut = new boolean[maxValue + 1];
        for (int i = 2; i < crossedOut.length; i++)
            crossedOut[i] = false;
    }

    private static void crossOutMultiples()
    {
        int limit = determineIterationLimit();
        for (int i = 2; i <= limit; i++)
            if (notCrossed(i))
                crossOutMultiplesOf(i);
    }

    private static int determineIterationLimit()
    {
        // Every multiple in the array has a prime factor that
        // is less than or equal to the root of the array size,
        // so we don't have to cross out multiples of numbers
        // larger than that root.
        double iterationLimit = Math.sqrt(crossedOut.length);
        return (int) iterationLimit;
    }
}
```

```
private static void crossOutMultiplesOf(int i)
{
    for (int multiple = 2*i;
        multiple < crossedOut.length;
        multiple += i)
        crossedOut[multiple] = true;
}

private static boolean notCrossed(int i)
{
    return crossedOut[i] == false;
}

private static void putUncrossedIntegersIntoResult()
{
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            result[j++] = i;
}

private static int numberOfUncrossedIntegers()
{
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
        if (notCrossed(i))
            count++;

    return count;
}
}
```

Natürlich könnte man auch einwenden, dass der erste Kommentar redundant sei, weil er im Wesentlichen dasselbe sagt wie die `generatePrimes`-Funktion selbst. Dennoch meine ich, dass der Kommentar dem Leser das Lesen des Algorithmus erleichtert; deshalb neige ich dazu, den Kommentar stehen zu lassen.

Der zweite Kommentar ist fast mit Sicherheit erforderlich. Er erklärt, warum die Quadratwurzel als Schleifengrenze verwendet wird. Ich konnte weder einen einfachen Variablennamen noch eine andere Codestruktur finden, der bzw. die diesen Punkt klar macht. Andererseits wirkt der Gebrauch der Quadratwurzel etwas extravagant. Spare ich wirklich so viel Zeit, wenn ich die Iteration durch die Quadratwurzel begrenze? Könnte die Berechnung der Quadratwurzel nicht mehr Zeit erfordern, als ich einspare?

Dies ist bedenkenswert. Die Quadratwurzel als Iterationsgrenze zu benutzen, befriedigt den alten C- und Assembler-Hacker in mir, aber ich bin nicht überzeugt, ob es die Zeit und den Aufwand lohnt, dies auch anderen so zu erklären, dass sie es verstehen.

Formatierung



Wenn Entwickler unter die Haube schauen, wollen wir, dass sie von der Ordentlichkeit, der Konsistenz und der Aufmerksamkeit im Detail beeindruckt sind, die sie wahrnehmen. Sie sollen die gute Struktur bestaunen. Ihre Augen sollen sich weiten, wenn sie die Module durchblättern. Sie sollen erkennen, dass hier Profis am Werk waren. Wenn sie stattdessen eine verknäuelte Masse von Code sehen, der aussieht, als wäre er von einer Schar betrunkenen Seeleute geschrieben worden, würden sie wahrscheinlich daraus schließen, dass dieselbe Missachtung der Details auch in allen anderen Aspekten des Projekts vorherrscht.

Sie sollten dafür sorgen, dass Ihr Code sauber formatiert ist. Sie sollten einen Satz einfacher Regeln für die Formatierung Ihres Codes auswählen und dann konsistent anwenden. Wenn Sie in einem Team arbeiten, dann sollte sich das Team auf einen einzigen Satz von Formatierungsregeln festlegen, die von allen Mitgliedern konsequent angewendet werden. Es hilft, wenn Sie über ein automatisiertes Tool verfügen, das diese Formatierungsregeln für Sie anwenden kann.

5.1 Der Zweck der Formatierung

Zunächst einmal möchte ich klarstellen: Code-Formatierung ist *wichtig*. Sie ist zu wichtig, um sie zu ignorieren, und sie ist zu wichtig, um sie dogmatisch zu behan-

deln. Code-Formatierung hat mit Kommunikation zu tun, und Kommunikation ist die vordringlichste Aufgabe eines professionellen Entwicklers.

Vielleicht dachten Sie, dass »den Code zum Laufen bringen« die vordringlichste Aufgabe eines professionellen Entwicklers wäre. Ich hoffe jedoch, dass dieses Buch Ihnen inzwischen diese Vorstellung geraubt hat. Die Funktionalität, die Sie heute erstellen, hat gute Chancen, das nächste Release Ihrer Software zu ändern, aber die Lesbarkeit Ihres Codes wird eine nachhaltige Auswirkung auf alle Änderungen haben, die jemals gemacht werden. Der Codierstil und die Lesbarkeit geben Maßstäbe vor, die die Wartbarkeit und Erweiterbarkeit beeinflussen, lange nachdem der ursprüngliche Code bis jenseits aller Erkennbarkeit geändert worden ist. Ihr Stil und Ihre Disziplin überleben Ihren Code.

Was also sind die Formatierungsprobleme, die wir lösen müssen, um bestmöglich zu kommunizieren?

5.2 Vertikale Formatierung

Beginnen wir mit der vertikalen Größe. Wie groß sollte eine Quelldatei sein? In Java ist die Dateigröße eng mit der Klassengröße verbunden. Wir kommen zur Klassengröße, wenn wir Klassen behandeln. Im Moment soll die Dateigröße unser Thema sein.

Wie groß sind die meisten Java-Quelldateien? Es zeigt sich, dass es riesige Unterschiede in der Größe und einige bemerkenswerte Unterschiede im Stil gibt. Abbildung 5.1 zeigt einige dieser Unterschiede.

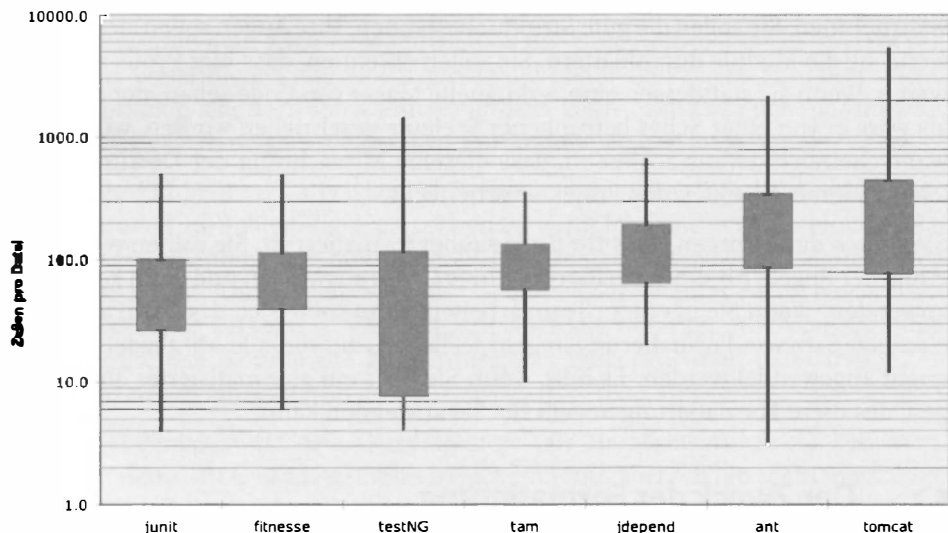


Abb. 5.1: Verteilung der Dateigrößen in einer logarithmischen Skala (Kastenhöhe = Sigma)

Es werden sieben verschiedene Projekte dargestellt. JUnit, FitNesse, TestNG, Time and Money, JDepend, Ant und Tomcat. Die Linien durch die Kästen zeigen die minimalen und maximalen Dateigrößen in jedem Projekt. Der Kasten zeigt etwa ein Drittel (eine Standardabweichung) der Dateien. Die Mitte eines Kastens entspricht dem Mittelwert. Der Kasten zeigt $\text{Sigma}/2$ über und unter dem Mittelwert. (Ja, ich weiß, dass die Dateigrößen nicht normalverteilt sind und dass deshalb die Standardabweichung mathematisch nicht genau ist. Aber hier geht es nicht um Präzision. Wir versuchen nur, ein Gefühl für die Größen zu bekommen.)

Die durchschnittliche Dateigröße in dem FitNesse-Projekt beträgt also etwa 65 Zeilen, und über ein Drittel der Dateien ist zwischen 40 und 100+ Zeilen lang. Die größte Datei in FitNesse ist über 400 Zeilen, die kleinste 6 Zeilen lang. Beachten Sie, dass dies eine logarithmische Skala ist; deshalb bedeuten die kleinen Unterschiede in der vertikalen Position sehr große Unterschiede in der absoluten Größe.

JUnit, FitNesse und Time and Money bestehen aus relativ kleinen Dateien. Keine ist über 500 Zeilen lang und die meisten Dateien sind kürzer als 200 Zeilen. Dagegen enthalten Tomcat und Ant einige Dateien, die mehrere Tausend Zeilen lang sind; etwa die Hälfte ihrer Dateien sind über 200 Zeilen lang.

Was bedeutet das für uns? Es scheint möglich zu sein, umfangreiche Systeme (FitNesse umfasst annähernd 50.000 Zeilen) aus Dateien zu erstellen, die typischerweise 200 Zeilen lang sind, wobei die Obergrenze bei 500 Zeilen liegt. Obwohl dies keine unumstößliche Regel sein sollte, sollten diese Werte als sehr erstrebenswert gelten. Kleine Dateien sind normalerweise leichter zu verstehen als große.

Die Zeitungs-Metapher

Stellen Sie sich einen gut geschriebenen Zeitungsartikel vor. Sie lesen ihn vertikal. Am Anfang erwarten Sie eine Überschrift, die Ihnen sagt, wovon die Geschichte handelt und es Ihnen ermöglicht zu entscheiden, ob Sie den Artikel lesen wollen. Der erste Absatz gibt Ihnen eine Zusammenfassung der gesamten Geschichte. Er verbirgt alle Details und stellt die wesentlichen allgemeinen Konzepte vor. Wenn Sie nach unten weiterlesen, erfahren Sie zunehmend feinere Details, bis Sie alle Daten, Namen, Zitate, Behauptungen und andere Kleinigkeiten kennen.

Eine Quelldatei sollte wie ein Zeitungsartikel lesbar sein. Der Name sollte einfach, aber aussagekräftig sein. Der Name allein sollte uns sagen können, ob wir im richtigen Modul sind oder nicht. Die oberen Teile der Quelldatei sollten die allgemeineren Konzepte und Algorithmen enthalten. Die Detailtiefe sollte nach unten hin zunehmen, bis wir am Ende der Quelldatei die am niedrigsten angesiedelten Funktionen und Details finden.

Eine Zeitung besteht aus vielen Artikeln; die meisten sind sehr klein. Einige sind etwas länger. Sehr wenige enthalten so viel Text, wie auf eine Seite passt. Dadurch wird die Zeitung *benutzbar*. Enthielte die Zeitung nur eine lange Geschichte mit einer ungeordneten Anhäufung von Fakten, Daten und Namen, würden wir sie einfach nicht lesen.

Vertikale Offenheit zwischen Konzepten

Fast der gesamte Code wird von links nach rechts und von oben nach unten gelesen. Jede Zeile repräsentiert einen Ausdruck oder eine Klausel, und jede Gruppe von Zeilen repräsentiert einen vollständigen Gedanken. Diese Gedanken sollten voneinander durch Leerzeilen getrennt werden.

Betrachten Sie beispielsweise Listing 5.1. Es enthält Leerzeilen, die die Package-Deklaration, die Import-Anweisung(en) und die Funktionen trennen. Diese extrem einfache Regel hat eine nachhaltige Auswirkung auf das visuelle Layout des Codes. Jede Leerzeile ist ein visueller Hinweis, der ein neues und separates Konzept identifiziert. Wenn Sie das Listing überfliegen, werden Ihre Augen jeweils zur ersten Zeile nach einer Leerzeile gezogen.

Listing 5.1: BoldWidget.java

```
package fitness.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "''.+?'';";
    private static final Pattern pattern = Pattern.compile("''.(.*?)''",
        Pattern.MULTILINE + Pattern.DOTALL
    );

    public BoldWidget(ParentWidget parent, String text) throws Exception {
        super(parent);
        Matcher match = pattern.matcher(text);
        match.find();
        addChildWidgets(match.group(1));
    }

    public String render() throws Exception {
        StringBuffer html = new StringBuffer("<b>");
        html.append(childHtml()).append("</b>");
        return html.toString();
    }
}
```

Wenn wir diese Leerzeilen auslassen (siehe Listing 5.2), wird die Lesbarkeit des Codes deutlich verschlechtert.

Listing 5.2: BoldWidget.java

```
package fitness.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
    public static final String REGEXP = "''.+?'';";
    private static final Pattern pattern = Pattern.compile("''.(.*?)''",
        Pattern.MULTILINE + Pattern.DOTALL);
}
```

```

public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
}
}

```

Dieser Effekt wird noch deutlicher, wenn Sie den Fokus Ihrer Augen verändern. Im ersten Beispiel fallen die verschiedenen Zeilengruppen sofort auf, während das zweite Beispiel wie ein großer Brei wirkt. Der Unterschied zwischen diese beiden Listings besteht nur in einer gewissen vertikalen Offenheit.

Vertikale Dichte

Wenn die Offenheit Konzepte trennt, dann bedeutet die vertikale Dichte eine enge Zusammengehörigkeit. Deshalb sollten Codezeilen, die eng zusammengehören, vertikal dicht beieinanderstehen. Beachten Sie, wie die nutzlosen Kommentare in Listing 5.3 die enge Beziehung der beiden Instanzvariablen stören.

Listing 5.3:

```

public class ReporterConfig {

    /**
     * The class name of the reporter listener
     */
    private String m_className;

    /**
     * The properties of the reporter listener
     */
    private List<Property> m_properties = new ArrayList<Property>();

    public void addProperty(Property property) {
        m_properties.add(property);
    }
}

```

Listing 5.4 ist viel leichter zu lesen. Es ist gerade ein »Auge voll«, wenigstens für mich. Ich kann es anschauen und sehe, dass es sich um eine Klasse mit zwei Variablen und einer Methode handelt, ohne meinen Kopf oder meine Augen nennenswert zu bewegen. Bei dem vorherigen Listing musste ich meine Augen und meinen Kopf viel stärker bewegen, um den Code genauso gut zu verstehen.

Listing 5.4:

```
public class ReporterConfig {  
    private String m_className;  
    private List<Property> m_properties = new ArrayList<Property>();  
  
    public void addProperty(Property property) {  
        m_properties.add(property);  
    }  
}
```

Vertikaler Abstand

Haben Sie jemals wie eine Katze ihren Schwanz eine Klasse gejagt und sind von einer Funktion zur nächsten gesprungen, wie wild durch die Quelldatei gescrollt, versucht herauszufinden, wie die Funktionen zusammengehören und arbeiten, nur um dann in einem Rattennest der Verwirrung zu landen? Haben Sie sich jemals eine Vererbungskette emporgehängt, um die Definition einer Variablen oder Funktion zu finden? Dies ist frustrierend, weil Sie versuchen zu verstehen, was das System tut, Sie aber Ihre Zeit und mentale Energie bei dem Versuch vergeuden, herauszufinden und sich zu merken, wo die Teile stehen.

Konzepte, die eng verwandt sind, sollten vertikal eng beisammenstehen [Gro]. Natürlich funktioniert diese Regel nicht bei Konzepten, die in separate Dateien gehören. Doch andererseits sollten eng verwandte Konzepte nicht auf verschiedene Dateien verteilt werden, es sei denn, Sie haben einen übergeordneten Grund dafür. Tatsächlich ist dies einer der Gründe dafür, dass `protected` Variablen vermieden werden sollten.

Bei Konzepten, die so eng verwandt sind, dass sie in dieselbe Quelldatei gehören, sollte ihre vertikale Trennung ein Maß dafür sein, wie wichtig das eine Konzept ist, um das jeweils andere zu verstehen. Wir wollen unsere Leser nicht zwingen, in unseren Quelldateien und Klassen hin und her zu springen.

Variablendeklarationen. Variablen sollten so eng bei ihrem Verwendungsort wie möglich deklariert werden. Weil unsere Funktionen sehr kurz sind, sollten lokale Variablen am Anfang einer Funktion stehen, wie etwa in dieser längeren Funktion aus JUnit4.3.1.

```
private static void readPreferences() {  
    InputStream is= null;  
    try {  
        is= new FileInputStream(getPreferencesFile());  
        setPreferences(new Properties(getPreferences()));  
        getPreferences().load(is);  
    } catch (IOException e) {  
        try {  
            if (is != null)  
                is.close();  
        } catch (IOException e1) {  
        }  
    }  
}
```

Kontrollvariablen für Schleifen sollten normalerweise innerhalb der Schleifenanweisung deklariert werden, wie etwa bei dieser hübschen kleinen Funktion aus derselben Quelle.

```
public int countTestCases() {
    int count= 0;
    for (Test each : tests)
        count += each.countTestCases();
    return count;
}
```

In seltenen Fällen kann eine Variable in einer längeren Funktion auch am Anfang eines Blocks oder unmittelbar vor einer Schleife deklariert werden. Eine solche Variable finden Sie in diesem Code-Ausschnitt aus der Mitte einer sehr langen Funktion in TestNG.

```
***
for (XmlTest test : m_suite.getTests()) {
    TestRunner tr = m_runnerFactory.newTestRunner(this, test);
    tr.addListener(m_textReporter);
    m_testRunners.add(tr);

    invoker = tr.getInvoker();
    for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
    }

    for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
        afterSuiteMethods.put(m.getMethod(), m);
    }
}
***
```

Instanzvariablen sollten dagegen am Anfang der Klasse deklariert werden. Dies sollte nicht den vertikalen Abstand dieser Variablen vergrößern, weil sie in einer gut konzipierten Klasse von vielen, oder sogar von allen der Methoden der Klasse benutzt werden.

Es hat zahlreiche Diskussionen über den geeigneten Platz von Instanzvariablen gegeben. In C++ haben wir üblicherweise die so genannte *Scissors-Regel* (*Scheren-Regel*) angewendet, nach der alle Instanzvariablen am Ende platziert werden. In Java werden sie jedoch per Konvention alle am Anfang der Klasse aufgeführt. Ich sehe keinen Grund, einer anderen Konvention zu folgen. Das Wichtige ist, dass die Instanzvariablen an einer wohlbekannten Stelle deklariert werden. Jeder sollte wissen, wo er die Deklarationen finden kann.

Betrachten Sie beispielsweise den seltsamen Fall der `TestSuite`-Klasse in JUnit 4.3.1. Ich habe diese Klasse stark ausgedünnt, um den Punkt deutlich zu machen.

Etwa in der Mitte des Listings werden zwei Instanzvariablen deklariert. Es wäre schwer, sie an einer geeigneteren Stelle zu verstecken. Wer den folgenden Code liest, muss schon per Zufall auf die Deklarationen stoßen (wie es mir ging).

```
public class TestSuite implements Test {
    static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
        ***
    }

    public static Constructor<? extends TestCase>
    getTestConstructor(Class<? extends TestCase> theClass)
    throws NoSuchMethodException {
        ***
    }

    public static Test warning(final String message) {
        ***
    }

    private static String exceptionToString(Throwable t) {
        ***
    }

    private String fName;

    private Vector<Test> fTests= new Vector<Test>(10);

    public TestSuite() {
    }

    public TestSuite(final Class<? extends TestCase> theClass) {
        ***
    }

    public TestSuite(Class<? extends TestCase> theClass, String name) {
        ***
    }
    ***
}
```

Abhängige Funktionen. Wenn eine Funktion eine andere aufruft, sollten sie vertikal eng beisammenstehen, und die aufrufende Funktion sollte über der aufgerufenen Funktion stehen, falls möglich. Dadurch erhält das Programm einen natürlichen Fluss. Wenn die Konventionen zuverlässig eingehalten werden, kann der Leser darauf vertrauen, dass die Funktionsdefinitionen kurz nach ihrer Verwendung folgen werden. Betrachten Sie beispielsweise den Code-Ausschnitt von FitNesse in Listing 5.5. Die obere Funktion ruft die Funktion unter ihr auf, diese ihrerseits die unter ihr

stehende Funktion usw. So kann der Leser leicht die aufgerufenen Funktionen finden; und die Lesbarkeit des ganzen Moduls wird erheblich verbessert.

Listing 5.5: WikiPageResponder.java

```
public class WikiPageResponder implements SecureResponder {
    protected WikiPage page;
    protected PageData pageData;
    protected String pageTitle;
    protected Request request;
    protected PageCrawler crawler;

    public Response makeResponse(FitNesseContext context, Request request)
        throws Exception {
        String pageName = getPageNameOrDefault(request, "FrontPage");
        loadPage(pageName, context);
        if (page == null)
            return notFoundResponse(context, request);
        else
            return makePageResponse(context);
    }

    private String getPageNameOrDefault(Request request, String defaultPageName)
    {
        String pageName = request.getResource();
        if (StringUtil.isBlank(pageName))
            pageName = defaultPageName;

        return pageName;
    }

    protected void loadPage(String resource, FitNesseContext context)
        throws Exception {
        WikiPagePath path = PathParser.parse(resource);
        crawler = context.root.getPageCrawler();
        crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
        page = crawler.getPage(context.root, path);
        if (page != null)
            pageData = page.getData();
    }

    private Response notFoundResponse(FitNesseContext context, Request request)
        throws Exception {
        return new NotFoundResponder().makeResponse(context, request);
    }

    private SimpleResponse makePageResponse(FitNesseContext context)
        throws Exception {
        pageTitle = PathParser.render(crawler.getFullPath(page));
        String html = makeHtml(context);
    }
}
```



```
SimpleResponse response = new SimpleResponse();
response.setMaxAge(0);
response.setContent(html);
return response;
}
```

Eine Nebenbemerkung: Dieser Code-Ausschnitt zeigt ein hübsches Beispiel dafür, wie Konstanten auf der passenden Stufe verwaltet werden [G35]. Die Konstante "FrontPage" hätte in der `getPageNameOrDefault`-Funktion vergraben werden können, aber dadurch wäre eine wohlbekannte und erwartete Konstante unangemessen in einer niedrig angesiedelten Funktion vergraben worden. Es war besser, diese Konstante von der Stelle aus, an der sie bekannt sein sollte, nach unten an die Stelle weiterzureichen, an der sie tatsächlich benutzt wird.

Konzeptionelle Affinität. Bestimmte Code-Stücke *wollen* in der Nähe anderer Code-Stück stehen. Sie haben eine bestimmte konzeptionelle Affinität (Verwandtschaft). Je stärker diese Affinität ist, desto geringer sollte der vertikale Abstand zwischen ihnen sein.

Wie wir gesehen haben, könnte diese Affinität auf einer direkten Abhängigkeit basieren (Beispiele: Eine Funktion ruft eine andere auf; eine Funktion verwendet eine Variable). Aber es gibt andere mögliche Ursachen für die Affinität. Sie könnte entstehen, weil mehrere Funktionen eine ähnliche Operation ausführen. Betrachten Sie den folgenden Code-Ausschnitt aus dem Code von JUnit 4.3.1:

```
public class Assert {
    static public void assertTrue(String message, boolean condition) {
        if (!condition)
            fail(message);
    }

    static public void assertTrue(boolean condition) {
        assertTrue(null, condition);
    }

    static public void assertFalse(String message, boolean condition) {
        assertTrue(message, !condition);
    }

    static public void assertFalse(boolean condition) {
        assertFalse(null, condition);
    }
}
```

Diese Funktionen haben eine starke konzeptionelle Affinität, weil sie ein gemeinsames Namensschema haben und Varianten derselben grundlegenden Aufgabe

ausführen. Die Tatsache, dass sie sich gegenseitig aufrufen, ist sekundär. Selbst wenn sie es nicht täten, sollten sie eng beieinander stehen.

Vertikale Anordnung

Im Allgemeinen sollen Funktionsaufruf-Abhängigkeiten nach unten zeigen. Das heißt, eine Funktion, die aufgerufen wird, sollte sich unter einer Funktion befinden, von der sie aufgerufen wird. Dadurch entsteht ein hübscher Fluss durch das Source-code-Modul, der von der hohen Ebene am Anfang nach unten zu den niedrigeren Stufen fließt. (Dies ist das genaue Gegenteil von Sprachen wie Pascal, C und C++, die erzwingen, dass Funktionen definiert oder zumindest deklariert werden müssen, bevor sie aufgerufen werden.)

Wie bei Zeitungsartikeln erwarten wir, dass die wichtigsten Konzepte zuerst kommen und dass sie mit der geringstmöglichen Verunreinigung durch Details ausgedrückt werden. Wir erwarten, dass die Details der niedrigen Ebenen zum Schluss kommen. So können wir die Quelldateien überfliegen und aus den ersten Funktionen das Wesentliche ablesen, ohne in die Details eintauchen zu müssen. Listing 5.5 ist nach diesem Prinzip geordnet. Vielleicht noch bessere Beispiele finden Sie in den Listings 15.5 und 3.7.

5.3 Horizontale Formatierung

Wie breit sollte eine Zeile sein? Um diese Frage zu beantworten, wollen wir einen Blick auf die Breite der Zeilen in typischen Programmen werfen. Auch hier untersuchen wir die sieben verschiedenen Projekte. Abbildung 5.2 zeigt die Verteilung der Zeilenbreiten für alle sieben Projekte. Die Regelmäßigkeit ist beeindruckend, besonders im Bereich von 45 Zeichen. Tatsächlich machen alle Größen von 20 bis 60 jeweils über ein Prozent der Gesamtzahl der Zeilen aus. Das sind 40 Prozent! Etwa 30 weitere Prozent sind weniger als 10 Zeichen breit. Vergessen Sie nicht, dass dies eine logarithmische Skala ist; deshalb ist das lineare Aussehen des Abfalls über 80 Zeichen wirklich sehr signifikant. Programmierer ziehen ganz klar kurze Zeilen vor.

Dies weist darauf hin, dass wir möglichst kurze Zeilen bevorzugen sollten. Die alte Hollerith-Grenze von 80 ist ein wenig willkürlich, und ich habe nichts dagegen, die Zeilenbreite auf 100 oder sogar 120 auszudehnen. Aber darüber hinaus ist wahrscheinlich nur sorglos.

Früher befolgte ich die Regel, dass man niemals nach rechts scrollen müssen sollte. Aber die heutigen Monitore sind dafür zu breit. Jüngere Programmierer können die Schriftart so weit verkleinern, dass sie 200 Zeichen in eine Zeile bekommen. Tun Sie dies nicht. Ich habe meine Grenze auf 120 gesetzt.

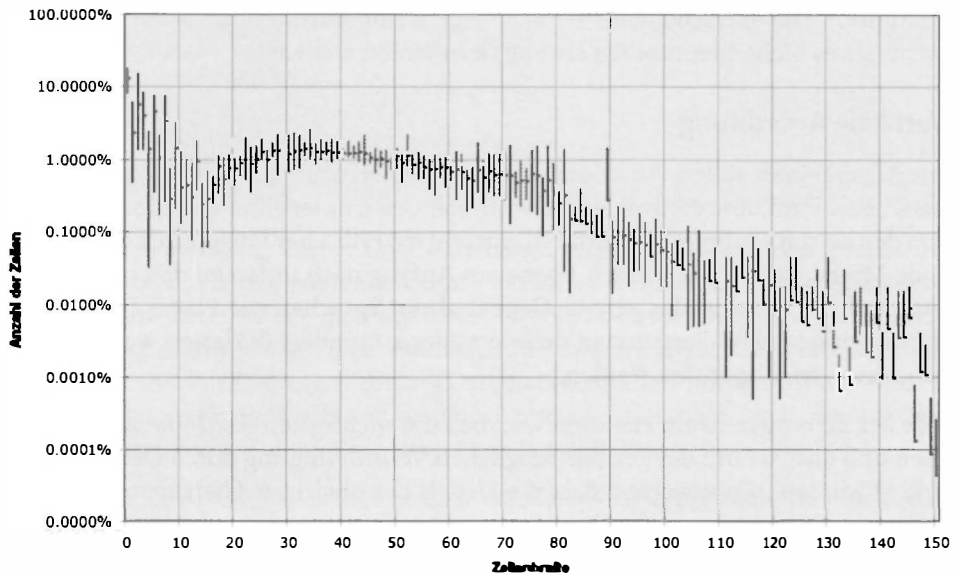


Abb. 5.2: Verteilung der Breite von Java-Zeilen

Horizontale Offenheit und Dichte

Wir verwenden horizontalen Whitespace, um Aufgaben zusammenzufassen, die enge Beziehungen haben, und Aufgaben zu trennen, die schwächere Beziehungen haben. Betrachten Sie die folgende Funktion:

```
private void measureLine(String line) {  
    lineCount++;  
    int lineSize = line.length();  
    totalChars += lineSize;  
    lineWidthHistogram.addLine(lineSize, lineCount);  
    recordWidestLine(lineSize);  
}
```

Ich habe die Zuweisungsoperatoren mit Whitespace umgeben, um sie zu betonen. Zuweisungsanweisungen bestehen aus zwei separaten Hauptelementen: die linke Seite und die rechte Seite. Die Leerzeichen machen diese Trennung deutlich.

Andererseits habe ich kein Leerzeichen zwischen die Funktionsnamen und die öffnende Klammer eingefügt, weil die Funktionen und ihre Argumente eng zusammengehören. Eine optische Trennung würde sie als nicht zusammengehörig erscheinen lassen. Ich trenne Argumente innerhalb der Klammern eines Funktionsaufrufs, um das Komma zu betonen und zu zeigen, dass die Argumente separat sind.

Mit Whitespace kann man auch den Vorrang von Operatoren hervorheben:

```

public class Quadratic {
    public static double root1(double a, double b, double c) {
        double determinant = determinant(a, b, c);
        return (-b + Math.sqrt(determinant)) / (2*a);
    }

    public static double root2(int a, int b, int c) {
        double determinant = determinant(a, b, c);
        return (-b - Math.sqrt(determinant)) / (2*a);
    }

    private static double determinant(double a, double b, double c) {
        return b*b - 4*a*c;
    }
}

```

Beachten Sie, wie gut sich die Gleichungen lesen lassen. Zwischen den Faktoren steht kein Whitespace, weil sie einen hohen Vorrang haben. Die Terme sind durch Whitespace getrennt, weil Addition und Subtraktion einen geringeren Vorrang haben.

Leider sind die meisten Formatierungs-Tools blind für den Vorrang von Operatoren und wenden durchgehend denselben Abstand an. Deshalb gehen subtil gesetzte Abstände wie in dem obigen Beispiel verloren, wenn Sie den Code mit einem solchen Tool neu formatieren.

Horizontale Ausrichtung

Als ich noch in Assembler programmierte, benutzte ich die horizontale Ausrichtung, um bestimmte Strukturen hervorzuheben. Als ich anfang, in C, C++ und schließlich in Java zu programmieren, versuchte ich weiterhin, alle Variablennamen in einem Satz von Deklarationen oder alle `values` in einem Satz von Zuweisungsanweisungen auszurichten. Mein Code könnte etwa wie folgt ausgesehen haben:

```

public class FitNesseExpediter implements ResponseSender
{
    private Socket      socket;
    private InputStream  input;
    private OutputStream output;
    private Request     request;
    private Response     response;
    private FitNesseContext context;
    protected long      requestParsingTimeLimit;
    private long         requestProgress;
    private long         requestParsingDeadline;
    private boolean      hasError;
}

```

```
public FitNesseExpediter(Socket      s,
                          FitNesseContext context) throws Exception
{
    this.context =      context;
    socket =           s;
    input =            s.getInputStream();
    output =           s.getOutputStream();
    requestParsingTimeLimit = 10000;
}
```

Ich habe jedoch festgestellt, dass diese Art von Ausrichtung nicht nützlich ist. Die Ausrichtung scheint die falschen Aufgaben zu betonen und lenkt mein Auge vom eigentlichen Zweck ab. Beispielsweise ist man bei der obigen Deklarationsliste versucht, die Liste der Variablennamen von oben nach unten zu lesen, ohne auf ihre Typen zu achten. Ähnlich ist man in der Liste der Zuweisungsanweisungen versucht, die Liste der `values` zu lesen, ohne auf den Zuweisungsoperator zu achten. Die Sache wird noch dadurch verschlimmert, dass automatische Formatierungstools diese Art von Ausrichtung normalerweise eliminieren.

Deshalb verwende ich diese Art von Ausrichtung nicht mehr. Heute ziehe ich nicht ausgerichtete Deklarationen und Zuweisungen vor (siehe unten), weil sie ein wichtiges Manko aufzeigen. Wenn ich lange Listen habe, die ausgerichtet werden müssen, *ist die Länge der Listen das Problem*, nicht das Fehlen der Ausrichtung. Die Länge der unten gezeigten Liste von Deklarationen in `FitNesseExpediter` weist darauf hin, dass diese Klasse aufgeteilt werden sollte.

```
public class FitNesseExpediter implements ResponseSender
{
    private Socket socket;
    private InputStream input;
    private OutputStream output;
    private Request request;
    private Response response;
    private FitNesseContext context;
    protected long requestParsingTimeLimit;
    private long requestProgress;
    private long requestParsingDeadline;
    private boolean hasError;

    public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
    {
        this.context = context;
        socket = s;
        input = s.getInputStream();
        output = s.getOutputStream();
        requestParsingTimeLimit = 10000;
    }
}
```

Einrückung

Eine Quelldatei ist eine Hierarchie, ähnlich einer Gliederung. Es gibt Informationen, die zu der Datei insgesamt, zu den einzelnen Klassen innerhalb der Datei, zu den Methoden innerhalb der Klassen, zu den Blöcken innerhalb der Methoden und rekursiv zu den Blöcken innerhalb von Blöcken gehören. Jede Ebene dieser Hierarchie ist ein Geltungsbereich, in dem Namen deklariert werden können und in dem Deklarationen und ausführbare Anweisungen interpretiert werden.

Um diese Hierarchie der Geltungsbereiche sichtbar zu machen, rücken wir die Zeilen des Sourcecodes entsprechend ihrer Position in der Hierarchie ein. Anweisungen auf der Datei-Ebene, wie etwa die meisten Klassendeklarationen, werden überhaupt nicht eingerückt. Methoden innerhalb einer Klasse werden eine Stufe nach rechts innerhalb ihrer Klasse eingerückt. Implementierungen dieser Methoden werden eine Stufe weiter rechts innerhalb der Methoden-Definition eingerückt. Block-Implementierungen werden eine Stufe weiter nach rechts innerhalb des einschließenden Blocks eingerückt usw.

Programmierer verlassen sich auf dieses Einrückungsschema. Sie orientieren sich am linken Rand der Zeilen, um zu sehen, zu welchem Geltungsbereich sie gehören. Auf diese Weise können sie schnell Geltungsbereiche, wie etwa Implementierungen von `if`- oder `while`-Anweisungen, überspringen, die in ihrer gegenwärtigen Situation nicht relevant sind. Sie scannen die linke Seite nach neuen Methoden-Deklarationen, neuen Variablen und sogar neuen Klassen. Ohne Einrückung wären Programme für Menschen praktisch unlesbar.

Betrachten Sie die folgenden Programme, die syntaktisch und semantisch identisch sind:

```
public class FitNesseServer implements SocketServer { private FitNesseContext context;
public FitNesseServer(FitNesseContext context) { this.context = context; } public void
serve(Socket s) { serve(s, 10000); } public void serve(Socket s, long requestTimeout)
{ try { FitNesseExpediter sender = new FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }
```

```
public class FitNesseServer implements SocketServer {
    private FitNesseContext context;
    public FitNesseServer(FitNesseContext context) {
        this.context = context;
    }

    public void serve(Socket s) {
        serve(s, 10000);
    }
}
```

```
public void serve(Socket s, long requestTimeout) {
    try {
        FitNesseExpediter sender = new FitNesseExpediter(s, context);
        sender.setRequestParsingTimeLimit(requestTimeout);
        sender.start();
    }
    catch (Exception e) {
        e.printStackTrace();
    }
}
```

Ihr Auge kann schnell die Struktur der eingerückten Datei erkennen. Sie finden fast sofort die Variablen, Konstruktoren, Accessoren und Methoden. Es dauert nur ein paar Sekunden, um zu erkennen, dass Sie es mit einem einfachen Front-End für ein Socket mit einem Timeout zu tun haben. Die nicht eingerückte Version ist jedoch ohne intensives Studium praktisch undurchdringlich.

Einrückungsregeln brechen. Manchmal ist man versucht, die Einrückungsregel für kurze `if`-Anweisungen, kurze `while`-Schleifen oder kurze Funktionen zu brechen. Wenn ich dieser Versuchung nachgegeben habe, bin ich fast immer zurückgegangen und habe die Einrückung wieder eingefügt. Deshalb vermeide ich es, Geltungsbereiche etwa wie folgt auf eine Zeile zu reduzieren:

```
public class CommentWidget extends TextWidget
{
    public static final String REGEXP = "^#[^\\r\\n]*(?:(?:\\r\\n)|\\n|\\r)?";

    public CommentWidget(ParentWidget parent, String text){super(parent, text);}
    public String render() throws Exception {return ""; }
}
```

Ich ziehe es vor, die Geltungsbereiche wie folgt zu erweitern und einzurücken:

```
public class CommentWidget extends TextWidget {
    public static final String REGEXP = "^#[^\\r\\n]*(?:(?:\\r\\n)|\\n|\\r)?";

    public CommentWidget(ParentWidget parent, String text) {
        super(parent, text);
    }

    public String render() throws Exception {
        return "";
    }
}
```

Dummy-Bereiche

Manchmal ist der Body einer `while`- oder `for`-Anweisung ein Dummy (siehe unten). Ich mag diese Art von Strukturen nicht und versuche, sie zu vermeiden.

Wenn ich sie nicht vermeiden kann, Sorge ich dafür, dass der Dummy-Body korrekt eingerückt und durch Klammern eingeschlossen wird. Ich weiß nicht, wie oft ich mich von einem Semikolon am Ende einer `while`-Schleife in derselben Zeile habe täuschen lassen. Solange Sie dieses Semikolon nicht *sichtbar* machen, indem Sie es in einer separaten Zeile einrücken, ist es einfach schwer zu sehen.

```
while (dis.read(buf, 0, readBufferSize) != -1)
;
```

5.4 Team-Regeln

Jeder Programmierer hat seine eigenen Formatierungsregeln; doch wenn er im Team arbeitet, hat das Team Vorrang.

Ein Team von Entwicklern sollte sich auf einen einzigen gemeinsamen Formatierungsstil festlegen. Jedes Mitglied dieses Teams sollte diesen Stil benutzen. Die Software soll in einem konsistenten Stil geschrieben werden. Sie soll nicht so aussehen, als wäre sie von einer Horde Individuen geschrieben worden, die sich nicht einigen konnten.

Als ich 2002 mit der Arbeit an dem FitNesse-Projekt begann, setzte ich mich mit dem Team zusammen, um einen Codierstil auszuarbeiten. Dies dauerte etwa zehn Minuten. Wir legten fest, wo wir unsere Klammern setzen wollten, wie groß die Einrückung sein sollte, wie die Klassen, Variablen und Methoden usw. benannt sein sollten. Dann legten wir diese Regeln in dem Code-Formatierer unserer IDE fest und haben uns seitdem daran gehalten. Es waren nicht die Regeln, die ich bevorzuge; es waren die Regeln, auf die sich das Team geeinigt hatte. Als Mitglied dieses Teams befolgte ich sie, wenn ich Code für das FitNesse-Projekt schrieb.

Vergessen Sie nicht: Ein gutes Software-System besteht aus einem Satz von gut lesbaren Dokumenten. Sie müssen einen konsistenten und geschmeidigen Stil haben. Der Leser muss darauf vertrauen können, dass die Formatierungskonstrukte, die er in einer Quelldatei gesehen hat, in anderen Dateien dasselbe bedeuten. Wir wollen keinesfalls die Komplexität des Sourcecodes mit einem Durcheinander verschiedener einzelner Stile vergrößern.

5.5 Uncle Bobs Formatierungsregeln

Die Regeln, die ich persönlich verwende, sind sehr einfach. Sie werden durch den Code in Listing 5.6 illustriert. Betrachten Sie das Folgende als Beispiel dafür, wie der Code selbst am besten Codierstandards dokumentiert.

Listing 5.6: `CodeAnalyzer.java`

```
public class CodeAnalyzer implements JavaFileAnalysis {
    private int lineCount;
```



```
private int maxLineWidth;  
private int widestLineNumber;  
private LineWidthHistogram lineWidthHistogram;  
private int totalChars;  
  
public CodeAnalyzer() {  
    lineWidthHistogram = new LineWidthHistogram();  
}  
  
public static List<File> findJavaFiles(File parentDirectory) {  
    List<File> files = new ArrayList<File>();  
    findJavaFiles(parentDirectory, files);  
    return files;  
}  
  
private static void findJavaFiles(File parentDirectory, List<File> files) {  
    for (File file : parentDirectory.listFiles()) {  
        if (file.getName().endsWith(".java"))  
            files.add(file);  
        else if (file.isDirectory())  
            findJavaFiles(file, files);  
    }  
}  
  
public void analyzeFile(File javaFile) throws Exception {  
    BufferedReader br = new BufferedReader(new FileReader(javaFile));  
    String line;  
    while ((line = br.readLine()) != null)  
        measureLine(line);  
}  
  
private void measureLine(String line) {  
    lineCount++;  
    int lineSize = line.length();  
    totalChars += lineSize;  
    lineWidthHistogram.addLine(lineSize, lineCount);  
    recordWidestLine(lineSize);  
}  
  
private void recordWidestLine(int lineSize) {  
    if (lineSize > maxLineWidth) {  
        maxLineWidth = lineSize;  
        widestLineNumber = lineCount;  
    }  
}  
  
public int getLineCount() {  
    return lineCount;  
}
```

```
public int getMaxLineWidth() {
    return maxLineWidth;
}

public int getWidestLineNumber() {
    return widestLineNumber;
}

public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
}

public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
}

public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
        cumulativeLineCount += lineCountForWidth(width);
        if (cumulativeLineCount > lineCount/2)
            return width;
    }
    throw new Error("Cannot get here");
}

private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesForWidth(width).size();
}

private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
}
}
```


Objekte und Datenstrukturen



Es gibt einen Grund, warum wir unsere Variablen privat halten. Wir wollen nicht, dass jemand von ihnen abhängig ist. Wir wollen uns die Freiheit bewahren, ihren Typ oder ihre Implementierung nach Belieben zu ändern. Warum fügen dann so viele Programmierer automatisch Getter und Setter zu ihren Objekten hinzu und enthüllen damit ihre privaten Variablen so, als wären sie öffentlich?

6.1 Datenabstraktion

Betrachten Sie den Unterschied zwischen Listing 6.1 und Listing 6.2. Beide repräsentieren die Daten eines Punktes in einem Kartesischen Koordinatensystem. Und dennoch enthüllt das eine Listing die Implementierung, während das andere sie vollkommen verbirgt.

Listing 6.1: Konkreter Punkt

```
public class Point {  
    public double x;  
    public double y;  
}
```

Listing 6.2: Abstrakter Punkt

```
public interface Point {  
    double getX();  
    double getY();  
    void setCartesian(double x, double y);  
    double getR();  
    double getTheta();  
    void setPolar(double r, double theta);  
}
```

Das Schöne an Listing 6.2 ist, dass Sie daraus nicht erkennen können, ob die Implementierung in Rechteck- oder in Polar-Koordinaten erfolgt. Es könnte auch keines von beiden sein! Und dennoch repräsentiert das Interface zweifellos eine Datenstruktur.

Aber es repräsentiert mehr als nur eine Datenstruktur. Die Methoden erzwingen gewisse Zugriffsregeln. Sie können die einzelnen Koordinaten unabhängig lesen, aber Sie müssen die Koordinaten zusammen als atomare Operation setzen.

Dagegen ist Listing 6.1 sehr klar in Rechteck-Koordinaten implementiert. Hier sind wir gezwungen, diese Koordinaten unabhängig zu manipulieren. Dadurch wird die Implementierung enthüllt. Tatsächlich würde die Implementierung selbst dann enthüllt, wenn die Variablen privat wären und wir nur Getter und Setter für die einzelnen Variablen verwenden würden.

Die Implementierung zu verbergen, bedeutet nicht, die Variablen einfach hinter einer Schicht von Funktionen zu verstecken. Die Implementierung zu verbergen, hat nichts mit Abstraktionen zu tun! Eine Klasse setzt und liest ihre Variablen nicht einfach durch Getter und Setter, sondern sie enthüllt abstrakte Interfaces, mit denen der Benutzer die *Essenz* der Daten manipulieren kann, ohne deren Implementierung kennen zu müssen.

Betrachten Sie Listing 6.3 und Listing 6.4. Das erste verwendet konkrete Termini, um den Füllstand des Tanks eines Fahrzeugs zu kommunizieren, während das zweite zu diesem Zweck mit der Abstraktion des Prozentsatzes arbeitet. In dem konkreten Fall können Sie ziemlich sicher sein, dass es sich einfach um Accessoren der Variablen handelt. In dem abstrakten Fall haben Sie überhaupt keinen Anhaltspunkt für die Form der Daten.

Listing 6.3: Konkretes Fahrzeug

```
public interface Vehicle {  
    double getFuelTankCapacityInGallons();  
    double getGallonsOfGasoline();  
}
```

Listing 6.4: Abstraktes Fahrzeug

```
public interface Vehicle {  
    double getPercentFuelRemaining();  
}
```

In beiden genannten Beispielen ist jeweils die zweite Variante vorzuziehen. Wir wollen die Details unserer Daten nicht enthüllen, sondern wir wollen unsere Daten in abstrakten Termini ausdrücken. Dies wird nicht einfach durch die Verwendung von Interfaces und/oder Gettern und Settern erreicht. Sie müssen gründlich nachdenken, was der beste Weg ist, die Daten in einem Objekt zu repräsentieren. Unbekümmert Getter und Setter hinzuzufügen, ist die schlimmste Option.

6.2 Daten/Objekt-Anti-Symmetrie

Diese beiden Beispiele zeigen den Unterschied zwischen Objekten und Datenstrukturen. Objekte verbergen ihre Daten hinter Abstraktionen und enthüllen Funktionen, die mit diesen Daten arbeiten. Datenstrukturen enthüllen ihre Daten und haben keine Funktionen. Lesen Sie dies bitte noch einmal. Beachten Sie die komplementäre Natur der beiden Definitionen. Sie sind praktisch Gegenteile. Dieser Unterschied mag trivial erscheinen, aber er hat weitreichende Konsequenzen.

Betrachten Sie das prozedurale Shape-Beispiel (Geometrische-Form-Beispiel) in Listing 6.5. Die Geometry-Klasse arbeitet mit drei Shape-Klassen (Form-Klassen). Die Shape-Klassen sind einfache Datenstrukturen ohne Verhalten. Alle Verhaltensweisen befinden sich in der Geometry-Klasse.

Listing 6.5: Prozedurales Shape-Beispiel

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Rectangle {
    public Point topLeft;
    public double height;
    public double width;
}

public class Circle {
    public Point center;
    public double radius;
}

public class Geometry {
    public final double PI = 3.141592653589793;

    public double area(Object shape) throws NoSuchShapeException
    {
        if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
        }
    }
}
```

```
else if (shape instanceof Rectangle) {
    Rectangle r = (Rectangle)shape;
    return r.height * r.width;
}
else if (shape instanceof Circle) {
    Circle c = (Circle)shape;
    return PI * c.radius * c.radius;
}
throw new NoSuchShapeException();
}
}
```

Objektorientierte Programmierer würden wahrscheinlich die Nase rümpfen und sich beschweren, dass dies prozedural wäre – und sie hätten recht. Aber vielleicht ist das Naserümpfen nicht ganz berechtigt. Was würde passieren, wenn *Geometry* um eine *perimeter()*-Funktion (Umfangsberechnung) erweitert würde? Die *Shape*-Klassen wären davon nicht betroffen! Alle anderen Klassen, die von den *Shapes* abhängen, wären ebenfalls nicht betroffen! Wenn ich andererseits ein neues *Shape* hinzufügen wollte, müsste ich alle Funktionen in *Geometry* ändern, um das neue *Shape* zu verarbeiten. Bitte lesen Sie auch diese Aussage noch einmal. Die beiden Bedingungen sind diametral entgegengesetzt.

Betrachten Sie jetzt die objektorientierte Lösung in Listing 6.6. Hier ist die *area()*-Methode polymorph. Es ist keine *Geometry*-Klasse erforderlich. Wenn ich jetzt ein neues *Shape* hinzufügen wollte, wäre keine der vorhandenen *Funktionen* betroffen, aber wenn ich eine neue Funktion hinzufügen wollte, müssten alle *Shapes* geändert werden! (Es gibt Methoden, dieses Problem zu umgehen, die erfahrenen objektorientierten Programmierern wohlvertraut sind, zum Beispiel *Visitor* oder *Dual-Dispatch*. Aber diese Techniken sind selbst mit Kosten verbunden und kehren im Allgemeinen zu der Struktur des prozeduralen Programms zurück.)

Listing 6.6: Polymorphe Shapes

```
public class Square implements Shape {
    private Point topLeft;
    private double side;

    public double area() {
        return side*side;
    }
}

public class Rectangle implements Shape {
    private Point topLeft;
    private double height;
    private double width;

    public double area() {
        return height * width;
    }
}
```

```

    }
}

public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

    public double area() {
        return PI * radius * radius;
    }
}

```

Auch hier sehen wir die komplementäre Natur dieser beiden Definitionen; sie sind praktisch Gegenteile! Dies enthüllt die fundamentale Dichotomie zwischen Objekten und Datenstrukturen:

Prozeduraler Code (Code, der Datenstrukturen verwendet) macht es leicht, neue Funktionen hinzuzufügen, ohne die vorhandenen Datenstrukturen zu ändern. Dagegen macht es OO-Code leicht, neue Klassen hinzuzufügen, ohne vorhandene Funktionen zu ändern.

Die komplementäre Aussage ist ebenfalls wahr:

Prozeduraler Code macht es schwer, neue Datenstrukturen hinzuzufügen, weil alle Funktionen geändert werden müssen. OO-Code macht es schwer, neue Funktionen hinzuzufügen, weil alle Klassen geändert werden müssen.

Also sind die Dinge, die für OO schwer sind, für Prozeduren leicht; und die Dinge, die für Prozeduren schwer sind, sind leicht für OO!

In jedem komplexen System treten Situationen auf, in denen wir neue Datentypen und keine neuen Funktionen hinzufügen wollen. In diesen Fällen sind Objekte und OO am besten geeignet. Es gibt jedoch auch Situationen, in denen wir neue Funktionen und keine Datentypen hinzufügen wollen. In diesem Fall sind prozeduraler Code und Datenstrukturen besser geeignet.

Gestandene Programmierer wissen, dass die Vorstellung, alles wäre ein Objekt, *ein Mythos ist*. Manchmal wollen Sie *wirklich* mit einfachen Datenstrukturen und mit Prozeduren arbeiten, die diese Strukturen manipulieren.

6.3 Das Law of Demeter

Es gibt eine bekannte Heuristik, die als *Law of Demeter* (LoD; Gesetz von Demeter; http://en.wikipedia.org/wiki/Law_of_Demeter; http://de.wikipedia.org/wiki/Gesetz_von_Demeter) bezeichnet wird, das besagt, dass ein Modul nichts über das Innere der Objekte wissen sollte, die es manipuliert. Wie im letzten Abschnitt gezeigt wurde, verbergen Objekte ihre Daten und enthüllen Operationen.

Das bedeutet, dass ein Objekt seine interne Struktur nicht durch Accessoren enthüllen sollte; denn dadurch wird seine interne Struktur enthüllt, nicht verborgen.

Genauer formuliert, sagt das Law of Demeter, dass eine Methode *f* einer Klasse *C* nur Methoden der folgenden Komponenten aufrufen sollte:

- *C*,
- ein Objekt, das von *f* erstellt wird,
- ein Objekt, das als Argument an *f* übergeben wird,
- ein Objekt, das in einer Instanzvariablen von *C* enthalten ist.

Die Methode sollte *keine* Methoden von Objekten aufrufen, die von einer der erlaubten Funktionen zurückgegeben werden. Anders ausgedrückt: Sprich nur zu Freunden, nicht zu Fremden.

Der folgende Code (irgendwo aus dem Apache-Framework) scheint (unter anderem) gegen das Law of Demeter zu verstoßen, weil er die `getScratchDir()`-Funktion des Rückgabewertes von `getOptions()` und dann `getAbsolutePath()` des Rückgabewerts von `getScratchDir()` aufruft.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Zugkatastrophe

Diese Art von Code wird oft als *Train Wreck* (Zugkatastrophe) bezeichnet, weil er wie eine Reihe zusammengekoppelter Eisenbahnwagen aussieht. Derartige Aufrufketten gelten im Allgemeinen als nachlässiger Stil und sollten vermieden werden [G36]. Es ist normalerweise üblich, sie wie folgt zu zerlegen:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Verstoßen diese beiden Code-Ausschnitte gegen das Law of Demeter? Sicher weiß das enthaltende Modul, dass das `ctxt`-Objekt Optionen enthält, die ein Scratch-Directory (Arbeitsverzeichnis) enthalten, das über einen absoluten Pfad verfügt. Das ist sehr viel Wissen für eine Funktion. Die aufrufende Funktion weiß, wie sie zu vielen verschiedenen Objekten navigieren kann.

Ob dies ein Verstoß gegen das Law of Demeter ist, hängt davon ab, ob `ctxt`, `Options` und `ScratchDir` Objekte oder Datenstrukturen sind. Sind sie Objekte, dann sollte ihre interne Struktur nicht enthüllt, sondern verborgen werden; dann wäre die Kenntnis ihrer inneren Struktur ein klarer Verstoß gegen das Law of Demeter. Sind `ctxt`, `Options` und `ScratchDir` dagegen einfach nur Datenstrukturen ohne Verhalten, dann enthüllen sie natürlicherweise ihre interne Struktur; in diesem Fall wäre das Law of Demeter nicht anwendbar.

Die Verwendung von Accessor-Funktionen verwirrt. Wäre der Code wie folgt geschrieben worden, wäre die Frage nach einem Verstoß gegen das Law of Demeter wahrscheinlich nicht aufgekommen:

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Das Problem wäre sehr viel weniger verwirrend, hätten Datenstrukturen einfach öffentliche Variablen und keine Funktionen und Objekte private Variablen und öffentliche Funktionen. Doch es gibt Frameworks und Standards (zum Beispiel »Beans«), die verlangen, dass selbst einfache Datenstrukturen Accessoren und Mutatoren haben sollen.

Hybride

Diese Verwirrung führt manchmal zu unglücklichen hybriden Strukturen, die halb Objekt und halb Datenstruktur sind. Sie haben Funktionen, die wichtige Aufgaben erfüllen, und sie haben auch entweder öffentliche Variablen oder öffentliche Accessoren und Mutatoren, die die privaten Variablen praktisch enthüllen und andere externe Funktionen dazu verleiten, diese Variablen so zu verwenden, wie ein prozedurales Programm eine Datenstruktur nutzen würde. Dies wird manchmal als *Feature Envy* ([Refactoring]; »Funktionsneid«) bezeichnet.

Solche hybriden Strukturen erschweren sowohl das Hinzufügen neuer Funktionen als auch das Hinzufügen neuer Datenstrukturen. Sie repräsentieren das Schlimmste aus beiden Welten. Sie sollten vermeiden, solche Strukturen zu erstellen. Sie sind ein Anzeichen für unsauberes Design, dessen Autoren sich nicht sicher sind – oder schlimmer: nicht wissen –, ob sie Schutz vor Funktionen oder Typen brauchen.

Struktur verbergen

Was wäre, wenn `ctxt`, `options` und `scratchDir` Objekte mit echten Verhaltensweisen wären? Weil Objekte ihre interne Struktur (hoffentlich) verbergen, sollten wir dann nicht in der Lage sein, durch die Objekte zu navigieren. Wie würden wir dann den absoluten Pfad des Scratch-Verzeichnisses bekommen?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

oder

```
ctx.getScratchDirectoryOption().getAbsolutePath()
```

Die erste Option könnte zu einer Explosion von Methoden in dem `ctxt`-Objekt führen. Die zweite Option nimmt an, dass `getScratchDirectoryOption()` eine Datenstruktur und kein Objekt zurückgibt. Keine der beiden Optionen schaut optimal aus.

Wenn `ctxt` ein Objekt ist, sollten wir es anweisen, *etwas zu tun*; wir sollten es nicht nach internen Daten befragen. Warum brauchen wir denn den absoluten Pfad des Scratch-Verzeichnisses? Was wollen wir damit tun? Betrachten Sie den folgenden Code aus demselben Modul (viele Zeilen weiter unten):

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Die Mischung der verschiedenen Detailebenen [G34][G6] ist etwas besorgniserregend. Dots, Slashes, Dateierweiterungen und `File`-Objekte sollten nicht so sorglos miteinander und mit dem einschließenden Code vermengt werden. Lassen wir dies jedoch beiseite, sehen wir, dass der absolute Pfad des Scratch-Verzeichnisses abgerufen wurde, um eine Scratch-Datei (Arbeitsdatei) eines bestimmten Namens zu erstellen.

Und wenn wir nun das `ctxt`-Objekt anwiesen, dies zu tun?

```
BufferedOutputStream bos = ctxt.createScratchFileStream(className);
```

Das sieht wie eine vernünftige Aufgabe für ein Objekt aus! Damit kann `ctxt` seine interne Struktur verbergen; und die gegenwärtige Funktion muss nicht gegen das Law of Demeter verstoßen, indem sie zu Objekten navigiert, von denen sie gar nichts wissen sollte.

6.4 Datentransfer-Objekte

Die idealtypische Form einer Datenstruktur ist eine Klasse mit öffentlichen Variablen und keinen Funktionen. Sie wird manchmal als *Datentransfer-Objekt* (DTO; engl. *Data Transfer Object*) bezeichnet. DTOs sind sehr nützliche Strukturen, besonders für die Kommunikation mit Datenbanken oder beim Parsen von Nachrichten von Sockets usw. Sie bilden oft die erste Stufe einer Reihe von Übersetzungsstufen, die rohe Daten aus einer Datenbank in Objekte im Anwendungscode übersetzen.

Etwas vertrauter ist die »Bean«-Form (siehe Listing 6.7). Beans haben private Variablen, die von Gettern und Settern manipuliert werden. Die Quasi-Einkapselung bei Beans scheint einigen OO-Puristen ein besseres Gefühl zu vermitteln, aber normalerweise bietet sie keine anderen Vorteile.

Listing 6.7: `address.java`

```
public class Address {
    private String street;
    private String streetExtra;
    private String city;
    private String state;
    private String zip;
```

```
public Address(String street, String streetExtra,
               String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
}

public String getStreet() {
    return street;
}

public String getStreetExtra() {
    return streetExtra;
}

public String getCity() {
    return city;
}

public String getState() {
    return state;
}

public String getZip() {
    return zip;
}
}
```

Active Record

Active Records sind eine spezielle Form von DTOs. Sie sind Datenstrukturen mit öffentlichen (oder mit Bean-Methoden zugänglichen) Variablen. Sie verfügen aber üblicherweise auch über Navigationsmethoden wie `save` oder `find`. Typischerweise sind diese Active Records direkte Übersetzungen von Datenbanktabellen oder anderen Datenquellen.

Leider stellen wir oft fest, dass Entwickler versuchen, diese Datenstrukturen wie Objekte zu behandeln, indem sie Methoden mit Geschäftsregeln in sie einfügen. Dies ist gefährlich, weil dadurch ein Hybrid aus einer Datenstruktur und einem Objekt entsteht.

Die Lösung besteht natürlich darin, das Active Record als Datenstruktur zu behandeln und die Geschäftsregeln in separate Objekte zu packen, die ihre internen Daten verbergen (die wahrscheinlich nur Instanzen des Active Records sind).

6.5 Zusammenfassung

Objekte enthüllen Verhaltensweisen und verbergen Daten. Dadurch können leicht neue Arten von Objekten hinzugefügt werden, ohne vorhandene Verhaltensweisen zu ändern. Dadurch wird es aber auch schwerer, neue Verhaltensweisen zu vorhandenen Objekten hinzuzufügen. Datenstrukturen enthüllen Daten und haben kein wesentliches Verhalten. Dadurch können leicht neue Verhaltensweisen zu vorhandenen Datenstrukturen hinzugefügt werden. Dadurch wird es aber auch schwerer, neue Datenstrukturen zu vorhandenen Funktionen hinzuzufügen.

In jedem System wünschen wir manchmal die Flexibilität, neue Datentypen hinzuzufügen, und bevorzugen deshalb in diesem Teil des Systems Objekte. Manchmal wünschen wir jedoch auch die Flexibilität, neue Verhaltensweisen hinzuzufügen, und bevorzugen deshalb in diesem Teil des Systems Datentypen und Prozeduren. Gute Software-Entwickler betrachten diese Probleme unvoreingenommen und wählen den Ansatz, der für die anstehende Aufgabe am besten geeignet ist.

Fehler-Handling

von Michael Feathers



Vielleicht scheint es etwas merkwürdig zu sein, in einem Buch über sauberen Code ein Kapitel über das Fehler-Handling zu bringen. Fehler-Handling gehört zu den ungeliebten, aber notwendigen Aufgaben, die wir beim Programmieren erledigen müssen. Der Input kann das falsche Format haben, Geräte können ausfallen. Kurz gesagt: Dinge können schiefgehen, und wenn sie dies tun, müssen wir als Programmierer dafür sorgen, dass unser Code tut, was er tun muss.

Die Verbindung mit sauberem Code sollte jedoch klar sein. Viele Code-Basen werden vollkommen vom Fehler-Handling dominiert. Wenn ich sage »dominiert«, will ich nicht sagen, dass sie sich nur mit dem Fehler-Handling befassen, sondern dass es fast unmöglich ist, zu erkennen, was der Code tut, weil seine Intention durch die überall verstreuten Anweisungen für das Fehler-Handling verdeckt wird. Fehler-Handling ist wichtig, aber *wenn es die Logik verschleiert, ist es falsch*.

In diesem Kapitel gebe ich einen Überblick über mehrere Techniken und Überlegungen, mit deren Hilfe Sie Code schreiben können, der sowohl sauber als auch robust ist – Code, der Fehler elegant und stilvoll handhabt.

7.1 Ausnahmen statt Rückgabe-Codes

Ganz früher kannten viele Programmiersprachen keine Ausnahmen. In diesen Sprachen waren die Techniken für die Behandlung und Meldung von Fehlern

beschränkt. Man setzte entweder ein Fehler-Flag oder gab einen Fehler-Code zurück, den der Aufrufer prüfen konnte. Der Code in Listing 7.1 illustriert diese Ansätze.

Listing 7.1: DeviceController.java

```
public class DeviceController {
    ...
    public void sendShutDown() {
        DeviceHandle handle = getHandle(DEV1);
        // Check the state of the device
        if (handle != DeviceHandle.INVALID) {
            // Save the device status to the record field
            retrieveDeviceRecord(handle);
            // If not suspended, shut down
            if (record.getStatus() != DEVICE_SUSPENDED) {
                pauseDevice(handle);
                clearDeviceWorkQueue(handle);
                closeDevice(handle);
            } else {
                logger.log("Device suspended. Unable to shut down");
            }
        } else {
            logger.log("Invalid handle for: " + DEV1.toString());
        }
    }
    ...
}
```

Diese Ansätze haben ein Problem: Sie überhäufen den Aufrufer mit logikfremdem Code. Der Aufrufer muss den Fehlerstatus unmittelbar nach dem Aufruf prüfen. Leider kann man dies leicht vergessen. Aus diesem Grund ist es besser, eine Ausnahme auszulösen, wenn ein Fehler auftritt. Der aufrufende Code ist sauberer. Die Logik wird nicht durch das Fehler-Handling verschleiert.

Listing 7.2 zeigt den Code, wenn in Methoden, die Fehler entdecken können, Ausnahmen ausgelöst werden.

Listing 7.2: DeviceController.java (mit Ausnahmen)

```
public class DeviceController {
    ...

    public void sendShutDown() {
        try {
            tryToShutDown();
        } catch (DeviceShutDownError e) {
            logger.log(e);
        }
    }
}
```

```

private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
}

private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
}
...
}

```

Beachten Sie, wie viel sauberer dieser Code ist. Dies ist nicht nur eine Frage der Ästhetik. Der Code ist besser, weil zwei miteinander vermengte Concerns (Belange), nämlich der Algorithmus für das Geräte-Shutdown und das Fehler-Handling, jetzt getrennt sind. Sie können jeden Concern unabhängig vom anderen studieren und verstehen.

7.2 Try-Catch-Finally-Anweisungen zuerst schreiben

Einer der interessantesten Aspekte von Ausnahmen ist die Tatsache, dass sie *einen Geltungsbereich* innerhalb Ihres Programms *definieren*. Wenn Sie Code in dem `try`-Abschnitt einer `try-catch-finally`-Anweisung ausführen, bringen Sie zum Ausdruck, dass das Programm an jedem Punkt abgebrochen und dann mit dem `catch`-Abschnitt fortgesetzt werden kann.

In gewisser Weise ähneln `try`-Blöcke Transaktionen. Ihr `catch` muss Ihr Programm in einem konsistenten Zustand zurücklassen, etwas, was in dem `try` passiert. Aus diesem Grund ist es sinnvoll, mit einer `try-catch-finally`-Anweisung zu beginnen, wenn Sie Code schreiben, der Ausnahmen auslösen könnte. Dies hilft Ihnen zu definieren, was der Benutzer dieses Codes erwarten sollte, egal was in dem Code schiefgeht, der in dem `try` ausgeführt wird.

Ein Beispiel: Wir müssen Code schreiben, der auf eine Datei zugreift und einige serialisierte Objekte liest.

Wir beginnen mit einem Unit-Test, der zeigt, dass eine Ausnahme ausgelöst wird, wenn die Datei nicht existiert:

```

@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {

```



```
sectionStore.retrieveSection("invalid - file");  
}
```

Der Test veranlasst uns, den folgenden Stub zu erstellen:

```
public List<RecordedGrip> retrieveSection(String sectionName) {  
    // Dummy-Rückgabe, bis wir eine echte Implementierung haben  
    return new ArrayList<RecordedGrip>();  
}
```

Unser Test scheitert, weil er keine Ausnahme auslöst. Als Nächstes ändern wir unsere Implementierung so, dass sie versucht, auf eine ungültige Datei zuzugreifen. Diese Operation löst eine Ausnahme aus:

```
public List<RecordedGrip> retrieveSection(String sectionName) {  
    try {  
        FileInputStream stream = new FileInputStream(sectionName)  
    } catch (Exception e) {  
        throw new StorageException("retrieval error", e);  
    }  
    return new ArrayList<RecordedGrip>();  
}
```

Jetzt besteht unser Test, weil wir die Ausnahme abgefangen haben. An diesem Punkt können wir das Refactoring ansetzen. Wir können den Typ der Ausnahme eingrenzen, damit er genau dem Typ von Ausnahme entspricht, die tatsächlich von dem `FileInputStream`-Konstruktor ausgelöst wird: `FileNotFoundException`:

```
public List<RecordedGrip> retrieveSection(String sectionName) {  
    try {  
        FileInputStream stream = new FileInputStream(sectionName);  
        stream.close();  
    } catch (FileNotFoundException e) {  
        throw new StorageException("retrieval error", e);  
    }  
    return new ArrayList<RecordedGrip>();  
}
```

Da wir jetzt den Geltungsbereich mit einer try-catch-Struktur definiert haben, können wir per TDD den Rest der erforderlichen Logik aufbauen. Diese Logik wird zwischen der Erstellung von `FileInputStream` und dem `close` eingefügt und kann so tun, als könne nichts schiefgehen.

Versuchen Sie, Tests zu schreiben, die Ausnahmen erzwingen. Fügen Sie dann Verhaltensweisen zu Ihrem Handler hinzu, um Ihre Testbedingungen zu erfüllen. Diese Vorgehensweise bewirkt, dass Sie zuerst den Transaktionsgeltungsbereich des try-Blocks erstellen, und hilft Ihnen, den Transaktionscharakter dieses Geltungsbereiches zu erhalten.

7.3 Unchecked Exceptions

Die Debatte ist vorbei. Jahrlang haben Java-Programmierer über die Vor- und Nachteile von Checked Exceptions diskutiert. Als Checked Exceptions in der ersten Version von Java eingeführt wurden, schien dies eine großartige Idee zu sein. Die Signatur jeder Methode würde alle Ausnahmen auflisten, die sie an ihren Aufrufer übergeben könnte. Darüber hinaus gehörten diese Ausnahmen zum Typ der Methode. Ihr Code ließ sich buchstäblich nicht kompilieren, wenn die Signatur nicht mit dem übereinstimmte, was Ihr Code tun konnte.

Damals dachten wir, dass Checked Exceptions eine großartige Idee wären; und ja, sie können *einige* Vorteile bieten. Doch heute wissen wir, dass sie nicht erforderlich sind, um robuste Software zu produzieren. C# hat keine Checked Exceptions, und trotz beherzter Versuche hat C++ auch keine. Dasselbe gilt für Python oder Ruby. Dennoch kann man in allen diesen Sprachen robuste Software schreiben. Weil dies der Fall ist, müssen wir – ernsthaft – überlegen, ob Checked Exceptions ihren Preis wert sind.

Welcher Preis? Der Preis von Checked Exceptions ist ein Verstoßen gegen das Open-Closed-Prinzip [Martin]. Wenn Sie eine Checked Exception von einer Methode in Ihrem Code auslösen und der `catch` drei Ebenen höher erfolgt, *müssen Sie diese Ausnahme in der Signatur aller Methoden deklarieren, die zwischen Ihnen und dem `catch` liegen*. Dies bedeutet, dass eine Änderung auf einer niedrigen Ebene der Software Signaturänderungen auf vielen höheren Ebenen erforderlich machen kann. Die geänderten Module müssen neu erstellt und verteilt werden, selbst wenn an ihren Funktionen nichts geändert wurde.

Betrachten Sie die Aufruf-Hierarchie eines großen Systems. Funktionen an der Spitze rufen untergeordnete Funktionen auf, die ihrerseits ihnen untergeordnete Funktionen aufrufen usw. Nehmen wir jetzt an, dass eine der Funktionen auf der untersten Ebene so modifiziert werde, dass sie eine Ausnahme auslösen muss. Wenn es sich um eine Checked Exception handelt, muss die Funktionssignatur um eine `throws`-Klausel ergänzt werden. Doch dies bedeutet, dass jede Funktion, die unsere modifizierte Funktion aufruft, ebenfalls modifiziert werden muss, um die neue Ausnahme entweder abzufangen oder die entsprechende `throws`-Klausel an ihre Signatur anzuhängen. Bis hin zur Spitze der Hierarchie. Netto ist eine Kaskade von Änderungen erforderlich, die von den unteren Ebenen der Software bis zu den höchsten reicht! Die Einkapselung ist defekt, weil alle Funktionen auf dem Pfad einer `throw`-Klausel die Details dieser Low-Level-Ausnahme kennen müssen. Bedenkt man, dass der Zweck von Ausnahmen darin besteht, Fehler separat behandeln zu können, ist es eine Schande, dass Checked Exceptions die Einkapselung in dieser Weise durchbrechen.

Checked Exceptions können manchmal nützlich sein, wenn Sie eine kritische Library schreiben: Sie müssen sie abfangen. Aber bei der allgemeinen Anwendungsentwicklung überwiegen die Kosten der Abhängigkeit die Vorteile.

7.4 Ausnahmen mit Kontext auslösen

Jede ausgelöste Ausnahme sollte genügend Kontext liefern, um die Quelle und den Ort eines Fehlers zu bestimmen. In Java können Sie für jede Ausnahme ein Stack-Trace abrufen; doch ein Stack-Trace kann Ihnen nicht den Zweck der gescheiterten Operation mitteilen.

Erstellen Sie informative Fehlermeldungen und übergeben Sie diese zusammen mit Ihren Ausnahmen. Erwähnen Sie die gescheiterte Operation sowie den Typ des Scheiterns. Wenn Sie sich bei Ihrer Anwendung einloggen, übergeben Sie genügend Informationen, damit Sie den Fehler in Ihrem `catch` protokollieren können.

7.5 Definieren Sie Exception-Klassen mit Blick auf die Anforderungen des Aufrufers

Es gibt viele Möglichkeiten, Fehler zu klassifizieren. Wir können Sie nach ihren Quellen unterscheiden: Kamen sie aus der einen Komponente oder einer anderen? Oder nach ihrem Typ: Sind es Gerätefehler, Netzwerkfehler oder Programmierfehler? Doch wenn wir Exception-Klassen in einer Anwendung definieren, sollten wir uns vor allem darum kümmern, *wie sie abgefangen werden*.

Betrachten wir ein Beispiel für eine suboptimale Ausnahmeklassifikation. Hier ist eine `try-catch-finally`-Anweisung für den Aufruf einer Drittanbieter-Library. Sie deckt alle Ausnahmen ab, die von den Aufrufen ausgelöst werden können:

```
ACMEPort port = new ACMEPort(12);

try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}
```

In dieser Anweisung wird viel Code dupliziert, und wir sollten nicht überrascht sein. Die meisten Ausnahmen werden, unabhängig von der tatsächlichen Ursache, relativ standardmäßig behandelt. Wir müssen einen Fehler registrieren und dafür sorgen, dass wir weiterarbeiten können.

Weil wir wissen, dass die zu leistende Arbeit unabhängig von der Ausnahme im Wesentlichen dieselbe ist, können wir in diesem Fall unseren Code erheblich vereinfachen, indem wir das aufgerufene API einhüllen und dafür sorgen, dass es einen gemeinsamen Ausnahmetyp zurückgibt:

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

Unsere LocalPort-Klasse ist einfach nur ein Wrapper (eine Umhüllung), der die von der ACMEPort-Klasse ausgelösten Ausnahmen abfängt und übersetzt:

```
public class LocalPort {
    private ACMEPort innerPort;

    public LocalPort(int portNumber) {
        innerPort = new ACMEPort(portNumber);
    }

    public void open() {
        try {
            innerPort.open();
        } catch (DeviceResponseException e) {
            throw new PortDeviceFailure(e);
        } catch (ATM1212UnlockedException e) {
            throw new PortDeviceFailure(e);
        } catch (GMXError e) {
            throw new PortDeviceFailure(e);
        }
    }
    ...
}
```

Wrapper wie der hier für ACMEPort definierte können sehr nützlich sein. Tatsächlich zählt das Einhüllen von Drittanbieter-APIs zu den Best Practices. Wenn Sie ein Drittanbieter-API einhüllen, minimieren Sie Ihre Abhängigkeiten von dem API: Sie können ohne größere Schwierigkeiten in der Zukunft auf eine andere Library umsteigen. Das Einhüllen erleichtert es auch, Drittanbieter-Aufrufe zu simulieren, wenn Sie eigenen Code testen.

Ein letzter Vorteil des Einhüllens liegt darin, dass Sie nicht an die API-Design-Entscheidungen eines speziellen Anbieters gebunden sind. Sie können ein API defi-

nieren, das für Sie nützlich ist. In dem vorangegangenen Beispiel haben wir einen einzigen Ausnahmetyp für `port-Device-Fehler` definiert und festgestellt, dass wir viel saubereren Code schreiben konnten.

Oft reicht eine einzige Exception-Klasse für einen speziellen Bereich des Codes aus. Die Informationen, die mit der Ausnahme gesendet werden, können die Fehler unterscheiden. Verwenden Sie andere Klassen nur, wenn Sie gewisse Ausnahmen abfangen und andere passieren lassen wollen.

7.6 Den normalen Ablauf definieren

Wenn Sie die Ratschläge aus den vorhergehenden Abschnitten befolgen, erreichen Sie eine gute Trennung zwischen Ihrer Geschäftslogik und Ihrem Fehler-Handling. Der Hauptteil Ihres Codes sieht einem schnörkellosen Algorithmus ähnlicher. Doch damit verdrängen Sie die Fehlererkennung an den Rand Ihres Programms. Sie hüllen externe APIs ein, damit Sie eigene Ausnahmen auslösen können, und Sie definieren einen Handler außerhalb Ihres normalen Codes, damit Sie abgebrochene Berechnungen handhaben können. Meistens ist dies ein passender Ansatz, aber manchmal wollen Sie das Programm nicht abbrechen.

Ein Beispiel. Der folgende Code addiert die Ausgaben in einer Abrechnungsanwendung recht umständlich:

```
try {
    MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
    m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
    m_total += getMealPerDiem();
}
```

Hat der Mitarbeiter Ausgaben für Mahlzeiten eingereicht, gehen diese in die Ausgabensumme ein, andernfalls wird ein *Tagessatz* (per diem amount) für diesen Tag angerechnet. Die Ausnahme verschleiert die Logik. Wäre es nicht besser, wenn wir den Sonderfall nicht behandeln müssten? Denn dann würde unser Code viel einfacher aussehen, nämlich so:

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Glücklicherweise können wir den Code entsprechend vereinfachen. Wir können `ExpenseReportDAO` so ändern, dass immer ein `MealExpense`-Objekt zurückgegeben wird. Wurden keine Ausgaben für Mahlzeiten abgerechnet, wird ein `MealExpense`-Objekt zurückgegeben, das den Tagessatz als Summe zurückgibt:

```
public class PerDiemMealExpenses implements MealExpenses {
    public int getTotal() {
```

```
    // den standardmäßigen Tagessatz zurückgeben  
    }  
}
```

Dies wird als *Special Case Pattern* [Fowler] (Sonderfall-Pattern) bezeichnet. Sie erstellen eine Klasse oder konfigurieren ein Objekt so, dass es einen Sonderfall für Sie handhabt. Dann muss sich der Client-Code nicht mit dem Ausnahmeverhalten befassen. Die Verhaltensweise ist in dem Sonderfall-Objekt eingekapselt.

7.7 Keine Null zurückgeben

Jede Beschreibung des Fehler-Handlings muss auch Dinge erwähnen, die Fehler geradezu einladen. Das erste auf meiner Liste ist die Rückgabe von `null`. Ich weiß nicht, wie viele Anwendungen ich gesehen habe, in denen fast jede zweite Zeile eine Prüfung auf `null` enthielt. Ein Beispiel:

```
public void registerItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.getItemRegistry();  
        if (registry != null) {  
            Item existing = registry.getItem(item.getID());  
            if (existing.getBillingPeriod().hasRetailOwner()) {  
                existing.register(item);  
            }  
        }  
    }  
}
```

Wenn Sie mit einer Code-Basis arbeiten, die derartigen Code enthält, sieht dies für Sie möglicherweise nicht so schlecht aus, wie es tatsächlich ist! Wenn wir `null` zurückgeben, schaffen wir im Wesentlichen Arbeit für uns selbst und wälzen Probleme auf unsere Aufrufer ab. Man muss nur die `null`-Prüfung vergessen, und schon läuft eine Anwendung aus dem Ruder.

Haben Sie bemerkt, dass in der zweiten Zeile der ersten verschachtelten `if`-Anweisung eine `null`-Prüfung fehlt? Was würde zur Laufzeit passieren, wenn `persistentStore` den Wert `null` hätte? Wir würden eine `NullPointerException` zur Laufzeit auslösen. Entweder wird diese `NullPointerException` auf oberer Ebene abgefangen oder aber auch nicht. Beides wäre *schlecht*. Was genau sollten Sie bei einer `NullPointerException` tun, die in den Tiefen Ihrer Anwendung ausgelöst wird?

Es ist leicht zu sagen, dass der obige Code das Problem einer fehlenden `null`-Prüfung hat. Doch tatsächlich liegt sein Problem darin, dass er *zu viele* Prüfungen dieser Art enthält. Wenn Sie versucht sind, `null` von einer Methode zurückzugeben, sollten Sie alternativ erwägen, eine Ausnahme auszulösen oder stattdessen ein *Special Case*-Objekt zurückzugeben. Wenn Sie eine Methode aus einem Drittanbieter-API

aufzurufen, die `null` zurückgibt, sollten Sie überlegen, wie Sie diese Methode in eine Methode einhüllen können, die entweder eine Ausnahme auslöst oder ein *Special Case*-Objekt zurückgibt.

In vielen Fällen bieten *Special Case*-Objekte eine leichte Abhilfe. Angenommen, Sie hätten Code wie diesen:

```
List<Employee> employees = getEmployees();
if (employees != null) {
    for(Employee e : employees) {
        totalPay += e.getPay();
    }
}
```

Im Moment kann `getEmployees` den Wert `null` zurückgeben, aber ist dies erforderlich? Wenn wir `getEmployees` so ändern, dass die Funktion eine leere Liste zurückgibt, können wir den Code aufräumen:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
    totalPay += e.getPay();
}
```

Glücklicherweise verfügt Java über `Collections.emptyList()`, die eine vordefinierte *immutable* (unveränderbare) Liste zurückgibt, die wir für diesen Zweck verwenden können:

```
public List<Employee> getEmployees() {
    if( .. keine Mitarbeiter vorhanden sind .. )
        return Collections.emptyList();
}
```

Wenn Sie so codieren, werden Sie die Gefahr von `NullPointerExceptions` minimieren, und Ihr Code wird sauberer sein.

7.8 Keine Null übergeben

Den Wert `null` von Methoden zurückzugeben, ist schlecht, aber `null` an Methoden zu übergeben, schlechter. Wenn Sie nicht gerade mit einem API arbeiten, das von Ihnen die Übergabe von `null` erwartet, sollten Sie, falls möglich, nie den Wert `null` in Ihrem Code übergeben.

Ein Beispiel soll zeigen, warum. Hier ist eine einfache Methode, die eine Metrik für zwei Punkte berechnet:

```
public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
```

```

        return (p2.x - p1.x) * 1.5;
    }
    ...
}

```

Was passiert, wenn jemand null als Argument übergibt?

```
calculator.xProjection(null, new Point(12, 13));
```

Natürlich bekommen wir eine `NullPointerException`.

Wie können wir sie beheben? Wir könnten einen neuen Ausnahmetyp erstellen und auslösen:

```

public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        if (p1 == null || p2 == null) {
            throw IllegalArgumentException(
                "Invalid argument for MetricsCalculator.xProjection");
        }
        return (p2.x - p1.x) * 1.5;
    }
}

```

Ist dies besser? Vielleicht ein wenig besser als eine `null-Pointer-Ausnahme`. Vergessen Sie aber nicht, dass wir einen Handler für `IllegalArgumentException` definieren müssen. Was sollte der Handler tun? Gibt es eine brauchbare Vorgehensweise?

Es gibt eine weitere Alternative. Wir könnten einen Satz von Zusicherungen verwenden:

```

public class MetricsCalculator
{
    public double xProjection(Point p1, Point p2) {
        assert p1 != null : "p1 should not be null";
        assert p2 != null : "p2 should not be null";
        return (p2.x - p1.x) * 1.5;
    }
}

```

Dies ist zwar eine gute Dokumentation, löst aber das Problem nicht. Wenn jemand null übergibt, erhalten wir immer noch einen Laufzeitfehler.

In den meisten Programmiersprachen gibt es keine gute Methode, mit einem Wert null umzugehen, der aus Versehen von einem Aufrufer übergeben wird. Deswegen besteht die rationale Lösung darin, die Übergabe von null standardmäßig zu verbieten. Denn dann können Sie mit dem Wissen codieren, dass der Wert null in

einer Argumentenliste ein Problem anzeigt, und machen viel weniger Fehler aus Unachtsamkeit.

7.9 Zusammenfassung

Sauberer Code ist lesbar, aber er muss auch robust sein. Dies sind keine konkurrierenden Ziele. Wir können robusten sauberen Code schreiben, wenn wir das Fehler-Handling als separaten Concern betrachten, der unabhängig von unserer Hauptlogik behandelt werden muss. In dem Maße, wie uns dies gelingt, können wir isoliert darüber nachdenken und die Wartbarkeit unseres Codes verbessern.

Grenzen

von James Grenning



Wir kontrollieren selten die gesamte Software in unseren Systemen. Manchmal kaufen wir Drittanbieter-Packages oder verwenden Open-Source-Software. Manchmal hängen wir davon ab, dass andere Teams in unserem Unternehmen Komponenten oder Subsysteme für uns erstellen. Irgendwie müssen wir diesen fremden Code sauber mit unserem eigenen integrieren. In diesem Kapitel geht es um Verfahren und Techniken, um Grenzen unserer Software sauber zu halten.

8.1 Mit Drittanbieter-Code arbeiten

Es gibt eine natürliche Spannung zwischen dem Lieferanten eines Interfaces und dem Benutzer eines Interfaces. Lieferanten von Drittanbieter-Packages und -Frameworks streben nach einer breiten Anwendbarkeit, damit sie in vielen Umgebungen arbeiten und eine größere Zielgruppe ansprechen können. Dagegen wünschen sich die Benutzer ein Interface, das auf ihre speziellen Anforderungen zugeschnitten ist.

Diese Spannung kann an den Grenzen unserer Systeme zu Problemen führen.

Betrachten wir die Methoden `java.util.Map` als Beispiel:

```
clear() void - Map
containsKey(Object key) boolean - Map
containsValue(Object value) boolean - Map
entrySet() Set - Map
equals(Object o) boolean - Map
get(Object key) Object - Map
getClass() Class<? extends Object> - Object
hashCode() int - Map
isEmpty() boolean - Map
keySet() Set - Map
notify() void - Object
notifyAll() void - Object
put(Object key, Object value) Object - Map
putAll(Map t) void - Map
remove(Object key) Object - Map
size() int - Map
toString() String - Object
values() Collection - Map
wait() void - Object
wait(long timeout) void - Object
wait(long timeout, int nanos) void - Object
```

Diese Liste zeigt, dass **Maps** über ein umfangreiches Interface mit zahlreichen Funktionen verfügen. Diese Leistungsstärke und Flexibilität sind sicher nützlich, können aber auch eine Belastung sein. Angenommen, unsere Anwendung erstellte eine **Map** und reichte sie herum. Wir wollen nicht, dass einer der Empfänger unserer **Map** Einträge in der **Map** löscht. Aber direkt am Anfang der Liste steht die `clear()`-Methode. Jeder Benutzer der **Map** kann ihren Inhalt löschen. Vielleicht sieht unser Design auch vor, dass nur spezielle Typen von Objekten in der **Map** gespeichert werden sollen; aber **Maps** schränken die Typen der Objekte, die in ihnen gespeichert werden, nicht zuverlässig ein. Jeder entschlossene Benutzer kann in jeder **Map** Elemente beliebigen Typs speichern.

Wenn unsere Anwendung eine **Map** von Sensoren benötigt, werden die Sensoren möglicherweise wie folgt definiert:

```
Map sensors = new HashMap();
```

In einem anderen Teil des Codes wird dann möglicherweise wie folgt auf einen Sensor zugegriffen:

```
Sensor s = (Sensor)sensors.get(sensorId);
```

Dieses Konstrukt finden Sie an vielen Stellen des Codes. Der Client dieses Codes ist dafür verantwortlich, ein **Object** aus der **Map** abzurufen und per **Cast** in den rich-

tigen Typ umzuwandeln. Dies funktioniert, ist aber kein sauberer Code. Außerdem erzählt der folgende Code seine Geschichte nicht so gut, wie er könnte. Die Lesbarkeit dieses Codes kann durch Generics erheblich verbessert werden; etwa so:

```
Map<Sensor> sensors = new HashMap<Sensor>();  
* * *  
Sensor s = sensors.get(sensorId );
```

Doch damit ist das Problem nicht gelöst, dass `Map<Sensor>` mehr Funktionalität anbietet, als wir brauchen oder wollen.

Wenn wir eine Instanz von `Map<Sensor>` freigiebig im System herumreichen, müssen zahlreiche Stellen korrigiert werden, sollte sich das Interface von `Map` jemals ändern. Vielleicht halten Sie eine solche Änderung für unwahrscheinlich; aber genau das ist passiert, als die Generics in Java 5 hinzugefügt wurden. Tatsächlich haben wir Systeme gesehen, die Generics nur deshalb nicht nutzen können, weil der schiere Änderungsaufwand für einen freizügigen Einsatz von Maps einfach zu groß wäre.

Ein saubererer Einsatz von `Map` könnte wie folgt aussehen. Dem Benutzer von `Sensors` ist der Einsatz von Generics egal. Diese Entscheidung ist (wie es sich gehört) zu einem Implementierungsdetail geworden.

```
public class Sensors {  
    private Map sensors = new HashMap();  
    public Sensor getById(String id) {  
        return (Sensor) sensors.get(id);  
    }  
  
    // ...  
}
```

Das Interface an der Grenze (`Map`) ist verborgen. Es kann mit sehr geringen Auswirkungen auf den Rest der Anwendung verändert werden. Der Einsatz von Generics ist kein großes Thema mehr, weil das Casting und die Typverwaltung innerhalb der `Sensors`-Klasse gehandhabt werden.

Außerdem ist dieses Interface auf die Anforderungen der Anwendung zugeschnitten. Es schränkt die zugänglichen Funktionen erheblich ein. Der resultierende Code ist leichter zu verstehen und schwerer zu missbrauchen. Die `Sensors`-Klasse kann das Design und die Geschäftsregeln durchsetzen.

Wir schlagen nicht vor, jede Anwendung von `Map` auf diese Weise einzukapseln, sondern raten Ihnen nur, Maps (oder andere Interfaces an der Grenze) nicht in Ihrem System herumzureichen. Wenn Sie ein Boundary-Interface wie `Map` verwenden, sollte Sie es auf die Klasse oder die enge Familie von Klassen beschränken, in der es benutzt wird. Sie sollten es beim Einsatz von öffentlichen APIs möglichst nicht zurückgeben und auch nicht als Argument übergeben.

8.2 Grenzen erforschen und kennen lernen

Drittanbieter-Code hilft uns, mehr Funktionalität in weniger Zeit zu realisieren. Wo beginnen wir, wenn wir ein Drittanbieter-Package nutzen wollen? Es ist nicht unsere Aufgabe, den Drittanbieter-Code zu testen. Aber es kann in unserem besten Interesse liegen, Tests für den Drittanbieter-Code zu schreiben, den wir benutzen.

Angenommen, Sie wüssten nicht, wie Sie unsere Drittanbieter-Library nutzen können. Sie könnten einige Tage die Dokumentation lesen, um herauszufinden, wie Sie es einsetzen wollen. Dann können Sie Ihren Code schreiben, der auf den Drittanbieter-Code zugreift, um zu prüfen, ob er tut, was Sie glauben. Sie wären nicht überrascht, durch lange Debugging-Sitzungen aufgehalten zu werden, um herauszufinden, ob auftretende Bugs durch Ihren oder den fremden Code verursacht werden.

Drittanbieter-Code nutzen zu lernen, ist schwer. Drittanbieter-Code zu integrieren, ist ebenfalls schwer. Beides gleichzeitig zu tun, ist doppelt so schwer. Was wäre, wenn wir einen anderen Ansatz wählen würden? Anstatt herumzuprobieren und den neuen Code in unserem Produktionscode zu testen, könnten wir einige Tests schreiben, um den Drittanbieter-Code besser kennen zu lernen. Jim Newkirk bezeichnet solche Tests als *Lern-Tests* (engl. *Learning Tests*; [BeckTDD], S. 136–137).

Bei Lern-Tests rufen wir das Drittanbieter-API so auf, wie wir es wahrscheinlich in unserer Anwendung benutzen werden. Wir führen im Wesentlichen kontrollierte Experimente durch, um zu prüfen, wie gut wir dieses API verstehen. Die Tests konzentrieren sich auf das, was wir uns von dem API versprechen.

8.3 log4j kennen lernen

Angenommen, wir wollten das log4j-Package von Apache anstelle unseres eigenen benutzerspezifischen Loggers verwenden. Wir haben das Package heruntergeladen und öffnen die einführende Dokumentationsseite. Ohne zu viel zu lesen, schreiben wir unseren ersten Testfall und erwarten, dass er »hello« auf der Konsole ausgibt.

```
@Test
public void testLogCreate() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.info("hello");
}
```

Wenn wir ihn ausführen, erzeugt der Logger einen Fehler, der uns mitteilt, dass wir etwas benötigen, das als **Appender** bezeichnet wird. Wenn wir etwas mehr lesen, stellen wir fest, dass es einen **ConsoleAppender** gibt. Deshalb erstellen wir einen **ConsoleAppender** und prüfen, ob wir die Geheimnisse des Loggings auf der Konsole gelüftet haben.

```

@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    ConsoleAppender appender = new ConsoleAppender();
    logger.addAppender(appender);
    logger.info("hello");
}

```

Diesmal stellen wir fest, dass der `Appender` keinen `Output-Stream` hat. Seltsam – es scheint logisch zu sein, dass er keinen hat. Mit ein wenig Hilfe von Google probieren wir das Folgende:

```

@Test
public void testLogAddAppender() {
    Logger logger = Logger.getLogger("MyLogger");
    logger.removeAllAppenders();
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("hello");
}

```

Das funktioniert; eine Protokollmeldung, die ein »hello« enthält, wird auf der Konsole ausgegeben! Es scheint seltsam zu sein, dass wir den `ConsoleAppender` anweisen müssen, auf die Konsole zu schreiben.

Interessanterweise wird »hello« immer noch ausgegeben, wenn wir das Argument `ConsoleAppender.SYSTEM_OUT` entfernen. Doch wenn wir das `PatternLayout` herausnehmen, erscheint wieder die Beschwerde über das Fehlen eines `Output-Streams`. Dieses Verhalten ist sehr seltsam.

Wenn wir uns die Dokumentation etwas sorgfältiger anschauen, stellen wir fest, dass der Default-Konstruktor `ConsoleAppender` »unconfigured« (nicht konfiguriert) ist, was nicht allzu offensichtlich oder nützlich zu sein scheint. Dies fühlt sich wie ein Bug oder zumindest eine Inkonsistenz in `log4j` an.

Ein wenig mehr Googelei, Lesen und Testen bringt uns schließlich zu Listing 8.1. Wir haben viel darüber gelernt, wie `log4j` funktioniert, und wir haben dieses Wissen in einem Satz einfacher Unit-Tests codiert.

Listing 8.1: `LogTest.java`

```

public class LogTest {
    private Logger logger;

    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        logger.getRootLogger().removeAllAppenders();
    }
}

```

```
}

@Test
public void basicLogger() {
    BasicConfigurator.configure();
    logger.info("basicLogger");
}

@Test
public void addAppenderWithStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
    logger.info("addAppenderWithStream");
}

@Test
public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n")));
    logger.info("addAppenderWithoutStream");
}
}
```

Jetzt wissen wir, wie wir einen einfachen Konsolen-Logger initialisieren müssen, und können dieses Wissen in unserer eigenen Logger-Klasse einkapseln, damit der Rest unserer Anwendung von dem `log4j`-Boundary-Interface isoliert ist.

8.4 Lern-Tests sind besser als kostenlos

Lern-Tests kosten nichts. Wir mussten das API trotzdem lernen; und diese Tests zu schreiben, war eine leichte und isolierte Methode, dieses Wissen zu erwerben. Die Lern-Tests waren präzise Experimente, die uns halfen, unser Verständnis zu vertiefen.

Lern-Tests sind nicht nur kostenlos, sondern haben ein positives Return-on-Investment. Wenn ein neues Release des Drittanbieter-Packages veröffentlicht wird, führen wir die Lern-Tests aus, um zu prüfen, ob es Verhaltensunterschiede gibt.

Lern-Tests verifizieren, ob die Drittanbieter-Packages so funktionieren, wie wir es erwarten. Nach der Integration gibt es keine Garantien dafür, dass der Drittanbieter-Code mit unseren Anforderungen kompatibel bleibt. Die ursprünglichen Autoren stehen unter Druck, ihren Code an neue, eigene Anforderungen anzupassen. Sie werden Bugs beheben und neue Fähigkeiten hinzufügen. Jedes Release birgt neue Risiken. Wenn die Änderungen des Drittanbieter-Packages inkompatibel mit unseren Tests sind, werden wir dies sofort feststellen.

Ob Sie das Wissen aus den Lern-Tests brauchen oder nicht, eine saubere Grenze sollte von einem Satz von nach außen gerichteten Tests unterstützt werden, die das Interface auf dieselbe Weise prüfen, wie wir den Produktionscode testen.

Ohne diese *Boundary-Tests*, die die Migration erleichtern, könnten wir versucht sein, die alte Version länger zu benutzen, als wir sollten.

8.5 Code verwenden, der noch nicht existiert

Es gibt eine andere Art von Grenzen: Sie trennt das Bekannte von dem Unbekannten. Es gibt oft Stellen im Code, wo unser Wissen vor einem Abgrund zu stehen scheint. Manchmal ist das, was auf der anderen Seite der Grenze ist, unergründlich (zumindest im Moment). Manchmal schauen wir absichtlich nicht weiter als bis zu der Grenze.

Vor einigen Jahren war ich Mitglied eines Teams, das Software für ein Radio-Kommunikationssystem entwickelte. Es gab ein Subsystem, den »Transmitter«, über das wir wenig wussten; und die Entwickler, die für das Subsystem verantwortlich waren, hatten ihr Interface noch nicht definiert. Da wir uns nicht blockieren lassen wollten, begannen wir unsere Arbeit an einem Teil des Systems, der von dem unbekannten Teil des Codes weit entfernt war.

Wir hatten eine ziemlich klare Vorstellung davon, wo unsere Welt endete und die neue Welt begann. Bei unserer Arbeit stießen wir manchmal an diese Grenze. Obwohl Nebel und Wolken der Unwissenheit unsere Sicht auf die andere Seite der Grenze behinderten, machte uns unsere Arbeit klar, wie das Boundary-Interface unsere Meinung nach aussehen *sollte*. Wir wollten dem Transmitter etwa Folgendes sagen können:

Stelle den Transmitter auf die angegebene Frequenz ein und gib eine analoge Repräsentation der Daten aus, die aus diesem Stream kommen.

Wir hatten keine Vorstellung davon, wie das bewerkstelligt werden könnte, weil das API noch nicht konzipiert worden war. Deshalb schoben wir die Ausarbeitung der Details auf später auf.

Damit wir nicht weiter blockiert waren, definierten wir unser eigenes Interface. Wir wählten einen griffigen Namen wie Transmitter. Wir gaben ihm eine Methode namens `transmit`, die eine Frequenz und einen Daten-Stream als Argumente übernahm. Dies war das Interface, das wir uns *wünschten*.

Ein solches Wunsch-Interface hat einen Vorteil: Es unterliegt der eigenen Kontrolle. Dies trägt dazu bei, die Lesbarkeit des Client-Codes und seinen Fokus auf seinen Zweck zu erhalten.

Abbildung 8.2 zeigt, dass wir die `CommunicationsController`-Klassen von dem Transmitter-API (das nicht unserer Kontrolle unterlag und noch nicht definiert war) isolierten. Indem wir unser eigenes anwendungsspezifisches Interface verwendeten, konnten wir unseren `CommunicationsController`-Code sauber und ausdrucksstark halten. Nachdem das Transmitter-API definiert war, schrieben wir den `TransmitterAdapter`, um die Lücke zu überbrücken. Der *Adapter* [GOF] kapselt

die Interaktion mit dem API ein und ist die einzige Stelle, die bei einer Weiterentwicklung des APIs geändert werden muss.

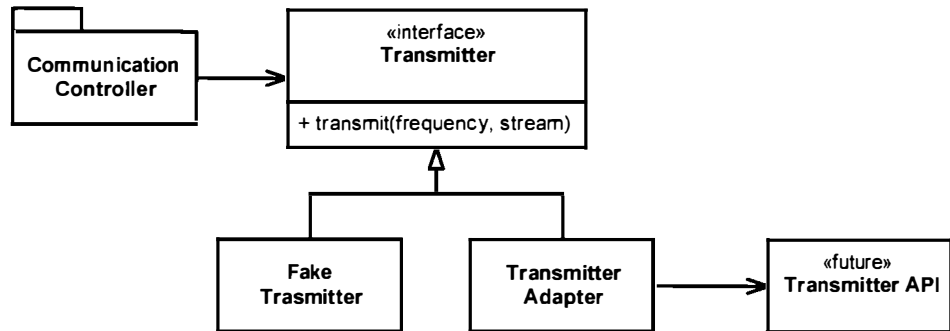


Abb. 8.1: Den Transmitter vorhersagen

Dieses Design stellt uns in dem Code auch einen sehr bequemen Seam (Saum; mehr über Seams siehe [WELC]) für das Testen zur Verfügung. Mit einem geeigneten **FakeTransmitter** können wir die **CommunicationsController**-Klassen testen. Wenn dann das **TransmitterAPI** zur Verfügung steht, können wir auch **Boundary-Tests** erstellen, um zu prüfen, ob wir das API korrekt verwenden.

8.6 Saubere Grenzen

An Grenzen passieren interessante Dinge. Änderung zählt zu diesen Dingen. Gute Software-Designs ermöglichen Änderungen ohne riesige Investitionen und Überarbeitungen. Wenn wir Code verwenden, der nicht unserer Kontrolle unterliegt, müssen wir besonders darauf achten, unsere Investition zu schützen und dafür zu sorgen, dass künftige Änderungen nicht zu teuer sind.

Code an den Grenzen braucht eine klare Trennung und Tests, die Erwartungen definieren. Wir sollten vermeiden, dass unser Code zu viel über die Details der Drittanbieter-Software weiß. Es ist besser, von etwas abhängig zu sein, das Sie kontrollieren können, als von etwas, das Sie nicht kontrollieren können, sonst werden Sie von ihm kontrolliert.

Wir halten Drittanbieter-Boundaries in Schach, indem wir sie nur an sehr wenigen Stellen im Code referenzieren. Wir könnten sie, ähnlich wie bei **Map** weiter vorne, einhüllen oder wir könnten unser perfektes Interface mit einem **Adapter** in das zur Verfügung gestellte Interface umwandeln. Auf jeden Fall spricht unser Code besser zu uns. Er fördert intern eine konsistente Anwendung über die Grenze hinweg; und er enthält weniger Wartungspunkte, wenn sich der Drittanbieter-Code ändert.

Unit-Tests



Unser Beruf hat in den letzten zehn Jahren eine wesentliche Entwicklung durchlaufen. 1997 hatte noch niemand von der Test Driven Development (TDD) gehört. Für die allermeisten Programmierer waren Unit-Tests kurze Abschnitte von Wegwerf-Code, den wir schrieben, um zu prüfen, ob unsere Programme »funktionierten«. Wir gaben uns große Mühe, unsere Klassen und Methoden zu schreiben, und dann flickten wir Ad-hoc-Code zusammen, um sie zu testen. Üblicherweise schrieben wir dabei eine Art von Treiber-Programm, mit dem wir manuell mit dem Programm interagieren konnten, das wir geschrieben hatten.

Ich erinnere mich an ein C++-Programm für ein eingebettetes Echtzeit-System, das ich Mitte der 90er-Jahre entwickelt hatte. Das Programm war ein einfacher Timer mit der folgende Signatur:

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

Die Idee war einfach: Die `execute`-Methode von `Command` sollte nach der spezifizierten Anzahl von Millisekunden in einem neuen Thread ausgeführt werden. Das Problem war, wie ich es testen konnte.

Ich schusterte ein einfaches Treiber-Programm zusammen, das die Tastatur überwachte. Jedes Mal, wenn ein Zeichen eingetippt wurde, wurde ein Schedule für einen Befehl erstellt, der dasselbe Zeichen fünf Sekunden später noch einmal tippen sollte. Dann tippte ich eine rhythmische Melodie auf der Tastatur ein und wartete darauf, dass dieses Lied fünf Sekunden später noch einmal abgespielt werden würde.

»Hänschen ... klein ... ging ... allein ... in ... die ... weite ... Welt ... hinein«.

Tatsächlich sang ich das Lied beim Eintippen mit, während ich die ».«-Tasten anschlug; und dann sang ich es noch einmal, als die Punkte auf dem Bildschirm erschienen.

Das war mein Test! Als ich sah, dass er funktionierte, und ich ihn meinen Kollegen demonstriert hatte, warf ich den Testcode weg.

Wie gesagt, unser Beruf hat eine lange Entwicklung hinter sich. Heute würde ich einen Test schreiben, mit dem ich prüfen könnte, ob jeder Haken und jede Öse dieses Codes so funktioniert, wie ich es erwarte. Ich würde meinen Code von dem Betriebssystem isolieren, anstatt einfach die eingebaute Timing-Funktion aufzurufen. Ich würde diese Timing-Funktion simulieren, um absolute Kontrolle über die Zeit zu haben. Ich würde Befehle planen, die boolesche Flags setzen würden, und dann würde ich die Zeit vorstellen, diese Flags beobachten und dafür sorgen, dass sie von `false` auf `true` gesetzt würden, sobald sich die Zeit auf den richtigen Wert setzte.

Nachdem ich dafür gesorgt hätte, dass alle Tests dieser Suite bestanden werden, würde ich dafür sorgen, dass diese Tests bequem von späteren Wartungsprogrammierern dieses Codes genutzt werden könnten. Ich würde dafür sorgen, dass die Tests und der Code zusammen in dasselbe Source-Package eingepackt werden würden.

Ja, wir haben einen langen Weg hinter uns; aber wir sind noch längst nicht am Ziel. Die Agile- und die TDD-Bewegung haben viele Programmierer ermutigt, automatisierte Unit-Tests zu schreiben. Heute werden diese Techniken von immer mehr Entwicklern übernommen. Aber in der raschen Anstrengung, das Testen fest in ihren täglichen Arbeitsablauf zu integrieren, haben viele Programmierer einige subtilere und wichtige Punkte übersehen, die zum Schreiben guter Tests unverzichtbar sind.

9.1 Die drei Gesetze der TDD

Heute weiß jeder, dass die TDD von uns verlangt, erst die Unit-Tests zu schreiben, bevor wir Produktionscode schreiben. Aber diese Regel ist nur die Spitze des Eisbergs. Betrachten Sie die folgenden drei Gesetze [Martino7]:

Erstes Gesetz – Sie dürfen Produktionscode erst schreiben, wenn Sie einen scheiternden Unit-Test geschrieben haben.

Zweites Gesetz – Der Unit-Test darf nicht mehr Code enthalten, als für das Scheitern und ein korrektes Kompilieren des Tests erforderlich ist.

Drittes Gesetz – Sie dürfen nur so viel Produktionscode schreiben, wie für das Bestehen des gegenwärtig scheiternden Tests ausreicht.

Diese drei Gesetze zwingen Sie in einen Zyklus, der vielleicht 30 Sekunden dauert. Die Tests und der Produktionscode werden *zusammen* geschrieben, wobei die Tests dem Produktionscode nur wenige Sekunden vorausseilen.

Wenn wir auf diese Weise arbeiten, schreiben wir Dutzende von Tests pro Tag, Hunderte von Tests pro Monat und Tausende von Tests pro Jahr. Wenn wir auf diese Weise arbeiten, decken diese Tests praktisch unseren gesamten Produktionscode ab. Der schiere Umfang dieser Tests kann mit dem Umfang des Produktionscodes selbst konkurrieren und Sie vor ein entmutigendes Verwaltungsproblem stellen.

9.2 Tests sauber halten

Vor einigen Jahren wurde ich gebeten, ein Team anzuleiten, das sich ausdrücklich dafür entschieden hatte, seinen Testcode *nicht* mit denselben Qualitätsstandards wie für den Produktionscode zu warten. Jeder hatte die Lizenz, Regeln in den Unit-Tests zu brechen. »Quick and dirty« war angesagt. Die Variablen mussten keine guten Namen haben, die Testfunktionen mussten nicht kurz und beschreibend sein. Der Testcode musste nicht wohlkonzipiert und gut durchdacht partitioniert sein. Solange der Testcode funktionierte und solange er den Produktionscode abdeckte, war er gut genug.

Einige Leser mögen mit dieser Entscheidung sympathisieren. Vielleicht haben sie, natürlich in einer weit zurückliegenden Vergangenheit, ähnliche Tests geschrieben wie ich für diese Timer-Klasse. Es ist ein riesiger Schritt vom Schreiben derartiger Wegwerf-Tests zum Schreiben einer Suite automatisierter Unit-Tests. Deshalb könnten Sie, wie das Team, das ich anleiten sollte, für sich beschließen, dass schmutzige Tests immer noch besser wären als gar keine Tests.

Doch dieses Team erkannte nicht, dass schmutzige Tests zu haben gleichbedeutend, wenn nicht sogar schlimmer ist, als keine Tests zu haben. Das Problem liegt darin, dass die Tests geändert werden müssen, wenn der Produktionscode weiterentwickelt wird. Je schmutziger die Tests sind, desto schwieriger sind die Änderungen. Je verschlungener der Testcode ist, desto wahrscheinlicher müssen Sie mehr Zeit damit verbringen, neue Tests in die Suite einzufügen, als Sie für das Schreiben des neuen Produktionscodes benötigen. Wenn Sie den Produktionscode modifizieren, fangen alte Tests an zu scheitern; und das Chaos in dem Testcode macht es schwer, die Tests so zu ändern, dass sie wieder bestanden werden. Deshalb werden diese Tests als eine immer größer werdende Belastung empfunden.

Von Release zu Release nahmen die Kosten für die Wartung der Test-Suite meines Teams zu. Im Laufe der Zeit entwickelte sie sich zu dem hauptsächlichen Stein des

Anstoßes der Entwickler. Wenn die Manager fragten, warum die Schätzungen zu groß geworden waren, schoben die Entwickler die Schuld auf die Tests. Schließlich waren sie gezwungen, die Test-Suite komplett aufzugeben.

Doch ohne eine Test-Suite hatten sie die Fähigkeit verloren, zu prüfen, ob Änderungen an ihrer Code-Basis so funktionierten, wie sie es erwarteten. Ohne eine Test-Suite konnten sie nicht garantieren, dass Änderungen in einem Teil ihres Systems nicht zu Defekten in anderen Teilen ihres Systems führten. Deshalb begann ihre Defektrate zu steigen. Als die Anzahl der nicht beabsichtigten Defekte zunahm, wuchs die Angst vor weiteren Änderungen. Sie hörten auf, ihren Produktionscode zu säubern, weil sie fürchteten, die Änderungen würden mehr Schaden als Nutzen bringen. Ihr Produktionscode begann zu verrotten. Zum Schluss standen sie da ohne Tests, mit verschlungenem Produktionscode voller Bugs, frustrierten Kunden und dem Gefühl, dass ihre Testanstrengungen nicht den erhofften Erfolg gebracht hätten.

In einer Hinsicht hatten sie recht. Ihre Testanstrengungen *hatten* nicht den erhofften Erfolg gebracht. Aber es war ihre Entscheidung, das Chaos in den Tests zuzulassen, die den Keim für dieses Scheitern pflanzte. Hätten sie ihre Tests sauber gehalten, dann hätten ihre Testanstrengungen sie nicht im Stich gelassen. Ich kann dies mit einiger Bestimmtheit sagen, weil ich in vielen Teams mitgearbeitet habe und viele Teams angeleitet habe, die mit *sauberen* Unit-Tests erfolgreich Projekte abgewickelt haben.

Die Moral der Geschichte ist einfach: *Testcode ist genauso wichtig wie Produktionscode*. Tests sind keine Bürger zweiter Klasse. Testcode erfordert Nachdenken, Design und Pflege. Er muss genauso sauber wie der Produktionscode gehalten werden.

Tests ermöglichen die -heiten und -keiten

Wenn Sie Ihre Tests nicht sauber halten, werden sie Ihnen entgleiten. Und ohne die Tests verlieren Sie das Einzige, das ihnen die Flexibilität Ihres Produktionscodes garantiert. Ja, Sie haben richtig gelesen: Es sind die *Unit-Tests*, die dafür sorgen, dass unser Code flexibel, wartbar und wiederverwendbar bleibt. Der Grund dafür ist einfach: Mit Tests haben Sie keine Angst, den Code zu ändern! Ohne Tests ist jede Änderung ein möglicher Bug. Egal, wie flexibel Ihre Architektur sein mag oder wie sauber Sie Ihr Design partitioniert haben, ohne Tests werden Sie zögern, Code zu ändern, weil Sie Angst haben, damit unentdeckte Bugs einzuführen.

Aber *mit* Tests ist diese Angst praktisch verschwunden. Je vollständiger Ihre Testabdeckung ist, desto geringer ist Ihre Angst. Sie können Änderungen praktisch straflos durchführen, auch wenn der Code eine weniger umwerfende Architektur und ein verworrenes und undurchschaubares Design hat. Tatsächlich können Sie diese Architektur und dieses Design furchtlos *verbessern*!

Deshalb ist die Arbeit mit einer automatisierten Suite von Unit-Tests, die den Produktionscode abdecken, der Schlüssel dazu, Ihr Design und Ihre Architektur so sau-

ber wie möglich zu halten. Tests ermöglichen all die -keiten, weil Tests *Änderung* ermöglichen.

Wenn also Ihre Tests schmutzig sein sollten, dann ist Ihre Fähigkeit, Ihren Code zu ändern, ernsthaft eingeschränkt, und Sie beginnen die Fähigkeit zu verlieren, die Struktur dieses Codes zu verbessern. Je schmutziger Ihre Tests sind, desto schmutziger wird Ihr Code. Schließlich entgleiten Ihnen die Tests, und Ihr Code verrottet.

9.3 Saubere Tests

Was zeichnet einen sauberen Test aus? Drei Dinge: Lesbarkeit, Lesbarkeit und Lesbarkeit. Lesbarkeit ist bei Unit-Tests vielleicht noch wichtiger als bei Produktionscode. Was macht Tests lesbar? Dasselbe, was allen Code lesbar macht: Klarheit, Einfachheit und Ausdrucksdichte. In einem Test wollen Sie so viel wie möglich mit so wenig Ausdrücken wie möglich sagen.

Betrachten Sie den Code aus FitNesse in Listing 9.1. Diese drei Tests sind schwierig zu verstehen und können bestimmt verbessert werden. Zunächst wird in den wiederholten Aufrufen von `addPage` und `assertSubString` schrecklich viel Code dupliziert [G5]. Doch wichtiger ist: Der folgende Code ist einfach mit Details überladen, die die Ausdruckskraft des Tests schwächen.

Listing 9.1: SerializedPageResponderTest.java

```
public void testGetPageHieratchyAsXml() throws Exception
{
    crawler.addPage(root, PathParser.parse("PageOne"));
    crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
    crawler.addPage(root, PathParser.parse("PageTwo"));

    request.setResource("root");
    request.addInput("type", "pages");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("<name>PageOne</name>", xml);
    assertSubString("<name>PageTwo</name>", xml);
    assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
    WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
```

```
crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
crawler.addPage(root, PathParser.parse("PageTwo"));

PageData data = pageOne.getData();
WikiPageProperties properties = data.getProperties();
WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
symLinks.set("SymPage", "PageTwo");
pageOne.commit(data);

request.setResource("root");
request.addInput("type", "pages");
Responder responder = new SerializedPageResponder();
SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
        new FitNesseContext(root), request);
String xml = response.getContent();

assertEquals("text/xml", response.getContentType());
assertSubString("<name>PageOne</name>", xml);
assertSubString("<name>PageTwo</name>", xml);
assertSubString("<name>ChildOne</name>", xml);
assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception
{
    crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

    request.setResource("TestPageOne");
    request.addInput("type", "data");
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
        (SimpleResponse) responder.makeResponse(
            new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
}
```

Betrachten Sie beispielsweise die `PathParser`-Aufrufe. Sie transformieren Strings in `PagePath`-Instanzen, die von den Crawlern verwendet werden. Diese Transformation ist für den gegenwärtigen Test vollständig irrelevant und verschleiert nur den Zweck. Die Details, die die Erstellung des `responder`-Objekts umgeben und die Sammlung und das Casting der `response`-Objekte sind ebenfalls nur Rauschen. Dann gibt es die ungeschickte Methode, wie der Anfrage-URL aus einer `resource` und einem Argument zusammengesetzt wird. (Ich habe geholfen, diesen Code zu schreiben; deswegen nehme ich bei meiner Kritik kein Blatt vor den Mund.)

Schließlich wurde dieser Code nicht konzipiert, um gelesen zu werden. Der arme Leser wird mit einer Unmenge von Details überhäuft, die er verstehen muss, bevor er die Bedeutung der Tests wirklich begreifen kann.

Betrachten Sie jetzt die verbesserten Tests in Listing 9.2. Diese Tests leisten genau dasselbe, wurden aber durch Refactoring in eine viel saubere und aussagekräftigere Form gebracht.

Listing 9.2: SerializedPageResponderTest.java (nach Refactoring)

```
public void testGetPageHierarchyAsXml() throws Exception {
    makePages("PageOne", "PageOne.ChildOne", "PageTwo");

    submitRequest("root", "type:pages");

    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
    WikiPage page = makePage("PageOne");
    makePages("PageOne.ChildOne", "PageTwo");

    addLinkTo(page, "PageTwo", "SymPage");
    submitRequest("root", "type:pages");
    assertResponseIsXML();
    assertResponseContains(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
    assertResponseDoesNotContain("SymPage");
}

public void testGetDataAsXml() throws Exception {
    makePageWithContent("TestPageOne", "test page");

    submitRequest("TestPageOne", "type:data");

    assertResponseIsXML();
    assertResponseContains("test page", "<Test>");
}
```

Das *Build-Operate-Check*-Pattern (<http://fitnesse.org/FitNesse.Acceptance-TestPatterns>) ist in der Struktur dieser Tests klar erkennbar. Jeder Test wird erkennbar in drei Teile zerlegt. Der erste Teil erstellt die Testdaten, der zweite Teil manipuliert diese Testdaten, und der dritte Teil prüft, ob die Operation die erwarteten Ergebnisse erzeugt hat.

Beachten Sie, dass der überwiegende Teil der ärgerlichen Details eliminiert worden ist. Die Tests kommen direkt zur Sache und verwenden nur die Datentypen und Funktionen, die sie wirklich benötigen. Jeder Leser dieser Tests sollte sehr schnell herausfinden können, was sie tun, ohne durch Details in die Irre geführt oder überwältigt zu werden.

Domänenspezifische Testsprache

Die Tests in Listing 9.2 demonstrieren die Technik, eine domänenspezifische Sprache für Ihre Tests zu erstellen. Anstatt die APIs zu verwenden, mit denen Programmierer das System manipulieren, erstellen wir einen Satz von Funktionen und Utilities, die diese APIs nutzen und uns das Schreiben und Lesen der Tests erleichtern. Diese Funktionen und Utilities bilden ein spezielles API, das von den Tests benutzt wird. Sie bilden eine *Testsprache*, mit denen Programmierer sich selbst helfen, ihre Tests zu schreiben, und auch denen helfen, die diese Tests später lesen müssen.

Dieses Testing-API wird nicht vorher konzipiert, sondern entwickelt sich im Zuge des laufenden Refactoring von Testcode, der selbst durch verschleiernde Details zu unsauber geworden ist. So wie ich bei Listing 9.1 zu Listing 9.2, bringen disziplinierte Entwickler ihren Testcode durch Refactoring in immer präzisere und ausdrucksstärkere Formen.

Ein Doppelstandard

In einer Hinsicht machte das Team, das ich am Anfang dieses Kapitels erwähne, etwas richtig. Der Code in dem Testing-API folgt tatsächlich einem anderen Satz von technischen Standards als der Produktionscode. Er muss immer noch einfach, präzise und ausdrucksstark sein, aber er muss nicht so effizient wie der Produktionscode sein. Schließlich wird er in einer Test-Umgebung und nicht in einer Produktionsumgebung ausgeführt; und diese beiden Umgebungen haben sehr verschiedene Anforderungen.

Betrachten Sie den Test in Listing 9.3. Ich habe diesen Test als Teil eines Systems zur Umgebungskontrolle geschrieben, für das ich einen Prototyp entwickelte. Ohne in die Details zu gehen, kann ich Ihnen sagen, dass dieser Test prüft, ob der Niedrigtemperatur-Alarm, der Heizer und der Ventilator alle eingeschaltet sind, wenn die Temperatur »way too cold« (»viel zu kalt«) ist.

Listing 9.3: EnvironmentControllerTest.java

```
@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
}
```

```

assertFalse(hw.coolerState());
assertFalse(hw.hiTempAlarm());
assertTrue(hw.loTempAlarm());
}

```

Es gibt hier natürlich zahlreiche Details. Was hat es beispielsweise mit der `tic`-Funktion auf sich? Tatsächlich sollten Sie sich darüber keine Gedanken machen, wenn Sie diesen Test lesen. Wichtig ist, dass Sie überlegen, ob der Endzustand des Systems konsistent mit der Aussage ist, dass die Temperatur »way too cold« sei.

Beachten Sie, wie Ihre Augen beim Lesen des Tests zwischen dem Namen des zu prüfenden Zustands und dem *Wert* dieses Zustands hin und her springen müssen. Sie sehen `heaterState`, und dann gleiten Ihre Augen nach links zu `assertTrue`. Sie sehen `coolerState`, und Ihre Augen müssen nach links, um `assertFalse` aufzunehmen. Dies ist mühsam und unzuverlässig. Der Test wird dadurch schwer lesbar.

Ich habe die Lesbarkeit dieses Tests erheblich verbessert, indem ich ihn umgeschrieben habe (siehe Listing 9.4).

Listing 9.4: EnvironmentControllerTest.java (nach Refactoring)

```

@Test
public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
}

```

Natürlich habe ich das Detail der `tic`-Funktion verborgen, indem ich eine `wayTooCold`-Funktion erstellt habe. Doch das Wichtige ist der seltsame String in `assertEquals`. Großbuchstabe bedeutet »an«, Kleinbuchstabe bedeutet »aus«, und die Buchstaben haben immer die folgende Reihenfolge: {heater, blower, cooler, hi-temp-alarm, lo-temp-alarm}.

Doch obwohl dies eng an einen Verstoß gegen die Regel über das mentale Mapping (siehe den Abschnitt *Mentale Mappings vermeiden* in Kapitel 2) herankommt, scheint es in diesem Fall angemessen zu sein. Denn sobald Sie die Bedeutung kennen, gleiten Ihre Augen schnell über den String, und Sie können die Ergebnisse schnell interpretieren. Den Test zu lesen, kann fast ein Vergnügen sein. Werfen Sie nur einen Blick auf Listing 9.5 und sehen Sie, wie leicht diese Tests zu verstehen sind.

Listing 9.5: EnvironmentControllerTest.java (bessere Übersicht)

```

@Test
public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
}

@Test
public void turnOnHeaterAndBlowerIfTooCold() throws Exception {

```

```
        tooCold();
        assertEquals("HBchl", hw.getState());
    }

    @Test
    public void turnOnHiTempAlarmAtThreshold() throws Exception {
        wayTooHot();
        assertEquals("hBCHl", hw.getState());
    }

    @Test
    public void turnOnLoTempAlarmAtThreshold() throws Exception {
        wayTooCold();
        assertEquals("HBchl", hw.getState());
    }
}
```

Die `getState`-Funktion wird in Listing 9.6 gezeigt. Beachten Sie, dass dieser Code nicht sehr effizient ist. Um ihn effizient zu machen, hätte ich wahrscheinlich einen `StringBuffer` benutzen sollen.

Listing 9.6: MockControlHardware.java

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
}
```

`StringBuffers` sind etwas hässlich. Selbst bei Produktionscode vermeide ich sie, wenn die Kosten gering sind. Man könnte auch einwenden, dass die Kosten des Codes in Listing 9.6 sehr klein wären. Doch diese Anwendung ist sicher ein eingebettetes Echtzeit-System, und wahrscheinlich sind die Computer- und Speicher-Ressourcen sehr eingeschränkt. Doch in der *Test*-Umgebung bestehen wahrscheinlich gar keine Einschränkungen.

Das liegt in der Natur des Doppelstandards. Es gibt Dinge, die Sie in einer Produktionsumgebung niemals tun dürfen, die aber in einer *Test*-Umgebung absolut in Ordnung sind. Normalerweise geht es dabei um Probleme der Speichernutzung oder der CPU-Effizienz, aber *niemals* um Probleme der Sauberkeit.

9.4 Ein assert pro Test

Es gibt eine Denkschule (siehe <http://www.artima.com/weblogs/viewpost.jsp?thread=35578>, ein Blog-Beitrag von Dave Astels), die fordert, jede Testfunk-

tion in einem JUnit-Test solle eine und nur eine `assert`-Anweisung enthalten. Diese Regel mag drakonisch erscheinen, aber der Vorteil zeigt sich in Listing 9.5. Diese Tests kommen zu einer einzigen Schlussfolgerung, die schnell und leicht zu verstehen ist.

Aber was ist mit Listing 9.2? Es scheint unvernünftig zu sein, die Zusicherung, dass der Output XML ist, leicht mit der Zusicherung kombinieren zu können, dass er bestimmte Substrings enthält. Doch wir können den Test in zwei separate Tests zerlegen, die jeweils eine eigene spezielle Zusicherung enthalten (siehe Listing 9.7).

Listing 9.7: SerializedPageResponderTest.java (nurein assert)

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldBeXML();
}

public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldContain(
        "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
}
```

Beachten Sie, dass ich die Namen der Funktionen geändert habe, um die übliche `given-when-then`-Konvention zu befolgen [RSpec]. Dadurch werden die Tests noch leichter lesbar. Leider führt die Zerlegung des Tests zu einer erheblichen Code-Duplizierung.

Wir können die Duplizierung mit dem *Template Method*-Pattern [GOF] eliminieren und die `given/when`-Teile in die Basisklasse und die `then`-Teile in verschiedene abgeleitete Klassen einfügen. Oder wir könnten eine vollkommen separate Testklasse erstellen und die `given`- und `when`-Teile in die `@Before`-Funktion und die `then`-Teile in jede `@Test`-Funktion einfügen. Aber dieser Aufwand scheint mir für ein solches kleines Problem zu groß zu sein. Letztlich ziehe ich die mehrfachen `assert`-Anweisungen in Listing 9.2 vor.

Ich betrachte die Regel, nur eine einzige `assert`-Anweisung zu verwenden, als eine brauchbare Richtlinie. (»Halte dich an den Code!«) Normalerweise versuche ich, eine domänenspezifische Testsprache zu erstellen, die diese Regel unterstützt (siehe Listing 9.5). Aber ich habe keine Angst davor, mehr als eine `assert`-Anweisung in einen Test einzufügen. Ich glaube, das Beste, was wir sagen können, ist, dass die Anzahl der `assert`-Anweisungen in einem Test minimiert werden sollte.

Ein Konzept pro Test

Vielleicht wäre es besser, die Regel aufzustellen, dass in jeder Testfunktion nur ein einziges Konzept getestet werden sollte. Wir wollen keine langen Testfunktionen haben, die einen beliebigen Aspekt nach dem anderen testen. Listing 9.8 ist ein Beispiel für einen solchen Test. Dieser Test sollte in drei unabhängige Tests zerlegt werden, weil er drei unabhängige Dinge testet. Alle Tests in einer Funktion zusammenzufassen, zwingt den Leser herauszufinden, warum ein bestimmter Abschnitt vorhanden ist und was dieser Abschnitt testet.

Listing 9.8: Verschiedene Tests für die addMonths()-Methode

```
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

Die drei Testfunktionen sollten wahrscheinlich wie folgt formuliert werden:

- *Gegeben* sei der letzte Tag eines Monats mit 31 Tagen (wie Mai):
 1. *Wenn* Sie einen Monat addieren und der letzte Tag dieses Monats der 30ste ist (wie Juni), *dann* sollte das Datum der 30ste dieses Monats, nicht der 31ste sein.
 2. *Wenn* Sie zwei Monate zu diesem Datum addieren, so dass der letzte Monat 31 Tage hat, *dann* sollte das Datum der 31ste sein.
- *Gegeben* sei der letzte Tag eines Monats mit 30 Tagen (wie Juni):
 1. *Wenn* Sie einen Monat addieren, so dass der letzte Tag dieses Monats der 31ste ist, *dann* sollte das Datum der 30ste, nicht der 31ste sein.

So formuliert, können Sie eine allgemeine Regel erkennen, die in den verschiedenen Tests verborgen ist. Wenn Sie den Monat inkrementieren, kann das Datum nicht größer werden als der letzte Tag dieses Monats. Dies bedeutet, dass die Inkre-

mentierung des Monats am 28. Februar den 28. März ergeben sollte. *Dieser* Test fehlt und wäre eine nützliche Ergänzung.

Deshalb wird das Problem nicht durch die Mehrzahl der `assert`-Anweisungen in den Abschnitten von Listing 9.8 verursacht, sondern durch die Tatsache, dass mehr als ein Konzept getestet wird. Deshalb wäre es wahrscheinlich am besten, die Anzahl der `assert`-Anweisungen pro Konzept zu minimieren und nur ein Konzept pro Testfunktion zu testen.

9.5 F.I.R.S.T.

Saubere Tests folgen fünf anderen Regeln, die die Abkürzung »F.I.R.S.T.« bilden (aus den Schulungsunterlagen von Object Mentor):

Fast (schnell) – Tests sollten schnell sein. Sie sollten schnell laufen. Wenn Tests langsam laufen, wollen Sie sie seltener ausführen. Wenn Sie sie nicht regelmäßig ausführen, finden Sie Probleme nicht früh genug, um sie leicht beheben zu können. Sie fühlen sich beim Aufräumen des Codes nicht frei genug. Schließlich fängt der Code an zu verfallen.

Independent (unabhängig) – Tests sollten nicht voneinander abhängen. Ein Test sollte keine Bedingungen für den nächsten Test setzen. Sie sollten jeden Test unabhängig ausführen können. Sie sollten die Tests in beliebiger Reihenfolge ausführen können. Wenn Tests voneinander abhängen, dann löst der erste, der scheitert, eine Kaskade weiterer nicht bestandener Tests stromabwärts aus, was die Diagnose erschwert und stromabwärts liegende Defekte verbirgt.

Repeatable (wiederholbar) – Tests sollten in jeder Umgebung wiederholbar sein. Sie sollten die Tests in der Produktionsumgebung, in der QA-Umgebung und auf Ihrem Laptop im Zug ohne Netzwerk ausführen können. Wenn Ihre Tests nicht in jeder Umgebung wiederholbar sind, haben Sie immer eine Entschuldigung dafür, warum sie scheitern. Dann kann es auch passieren, dass Sie die Tests nicht ausführen können, wenn die Umgebung nicht zur Verfügung steht.

Self-Validating (selbst-validierend) – Die Tests sollten einen booleschen Output haben. Entweder sie werden bestanden oder sie scheitern. Sie sollten keine Protokolldatei studieren müssen, um zu erfahren, ob die Tests bestanden werden. Sie sollten nicht manuell zwei verschiedene Textdateien vergleichen müssen, um zu erfahren, ob die Tests bestanden werden. Wenn die Tests sich nicht selbst validieren, kann das Scheitern subjektiv werden, und die Ausführung der Tests kann eine lange manuelle Auswertung erfordern.

Timely (zeitgerecht) – Die Tests müssen rechtzeitig geschrieben werden. Unit-Tests sollten *kurz vor* dem Produktionscode geschrieben werden, der dafür sorgt, dass sie bestanden werden. Wenn Sie Tests nach dem Produktionscode schreiben, finden Sie den Produktionscode möglicherweise schwer zu testen. Vielleicht kommen Sie

zu der Auffassung, ein Teil des Produktionscodes sei zu schwer zu testen. Sie dürfen keinen Produktionscode entwickeln, der nicht testbar ist.

9.6 Zusammenfassung

Wir haben kaum die Oberfläche dieses Themas angekratzt. Tatsächlich glaube ich, dass man ein ganzes Buch über *Saubere Tests* schreiben könnte. Tests sind für die Gesundheit eines Projekts genauso wichtig wie der Produktionscode. Vielleicht sind sie sogar noch wichtiger, weil Tests die Flexibilität, Wartbarkeit und Wiederverwendbarkeit des Produktionscodes verbessern und sichern. Deshalb sollten Sie Ihre Tests immer sauber halten. Bemühen Sie sich, sie ausdrucksstark zu formulieren und auf den Punkt zu bringen. Erfinden Sie Test-APIs, die als domänenspezifische Sprache agieren und Sie beim Schreiben der Tests unterstützen.

Wenn Sie die Tests verkommen lassen, wird auch Ihr Code verkommen. Halten Sie Ihre Tests sauber!

Klassen

mit Jeff Langr



Bis jetzt haben wir uns in diesem Buch darauf konzentriert, gute Codezeilen und Codeblöcke zu schreiben. Wir haben uns mit der sauberen Zusammensetzung von Funktionen und ihren Beziehungen befasst. Doch ausdrucksstarke Anweisungen, die sauber zu Funktionen zusammengesetzt werden, garantieren noch keinen sauberen Code. Wir müssen uns auch um die höheren Ebenen der Organisation des Codes kümmern: saubere Klassen.

10.1 Klassenaufbau

Folgt man den »offiziellen« Java-Konventionen, sollte eine Klasse mit einer Liste von Variablen beginnen. An erster Stelle sollten, falls vorhanden, `public static` Konstanten stehen. Dann die `private static` Variablen, gefolgt von den `private` Instanzvariablen. Gute Gründe für `public` Variablen sind selten.

Die `public` Funktionen sollten nach der Liste der Variablen stehen. Wir ziehen es vor, `private` Utilities, die von einer `public` Funktion aufgerufen werden, unmittelbar hinter die `public` Funktion selbst zu setzen. Damit erfüllen wir die Step-down-Regel. Es hilft uns und anderen, das Programm wie einen Zeitungsartikel lesen zu können.

Einkapselung

Unsere Variablen und Utility-Funktionen sollten möglichst `private` sein, aber wir sehen dies nicht dogmatisch. Manchmal müssen wir eine Variable oder Utility-Funktion als `protected` deklarieren, damit ein Test auf sie zugreifen kann. Für uns haben Tests Priorität. Wenn ein Test im selben Package eine Funktion aufrufen oder auf eine Variable zugreifen muss, deklarieren wir sie als `protected` oder mit einem Package-Geltungsbereich. Doch zunächst suchen wir nach Möglichkeiten, die Privatheit zu erhalten. Die Einkapselung zu lockern ist immer ein letzter Ausweg.

10.2 Klassen sollten klein sein!

Die erste Regel für Klassen fordert, dass sie klein sein sollten. Die zweite Regel für Klassen fordert, dass sie nicht noch kleiner sein sollten. Nein, wir werden hier nicht den genauen Wortlaut aus dem Kapitel *Funktionen* wiederholen. Aber wie bei Funktionen hat die Forderung, kleine Klassen zu schreiben, Priorität. Und wie bei Funktionen schließt daran immer sofort die Frage an: »Wie klein?«

Bei Funktionen haben wir die Größe gemessen, indem wir die Anzahl der Codezeilen gezählt haben. Bei Klassen verwenden wir ein anderes Maß. Wir zählen die *Verantwortlichkeiten* oder *Responsibilities* [RDD] (Zuständigkeiten, Belange).

Listing 10.1 zeigt eine Klasse, `SuperDashboard`, die über 70 `public` Methoden veröffentlicht. Die meisten Entwickler würden der Meinung zustimmen, dass die Klasse etwas zu groß geraten ist. Einige Entwickler bezeichnen eine Klasse wie `SuperDashboard` als »Gott-Klasse«.

Listing 10.1: Zu viele Verantwortlichkeiten

```
public class SuperDashboard extends JFrame implements MetadataUser {
    public String getCustomizerLanguagePath()
    public void setSystemConfigPath(String systemConfigPath)
    public String getSystemConfigDocument()
    public void setSystemConfigDocument(String systemConfigDocument)
    public boolean getGuruState()
    public boolean getNoviceState()
    public boolean getOpenSourceState()
    public void showObject(MetaObject object)
    public void showProgress(String s)
    public boolean isMetadataDirty()
    public void setIsMetadataDirty(boolean isMetadataDirty)
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public void setMouseSelectState(boolean isMouseSelected)
    public boolean isMouseSelected()
    public LanguageManager getLanguageManager()
    public Project getProject()
    public Project getFirstProject()
}
```

```

public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDatabaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
    MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPage(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
public void processProjectChangedState(boolean hasProjectChanged)
public void processObjectNameChanged(MetaObject object)
public void runProject()
public void setAllowDragging(boolean allowDragging)
public boolean allowDragging()
public boolean isCustomizing()

```

```
public void setTitle(String title)
public IdeMenuBar getIdMenuBar()
public void showHelper(MetaObject metaObject, String propertyName)
// ... viele weitere non-public Methoden ...
}
```

Aber was wäre, wenn SuperDashboard nur die Methoden in Listing 10.2 enthielte?

Listing 10.2: Klein genug?

```
public class SuperDashboard extends JFrame implements MetaDataUser {
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Fünf Methoden sind nicht zu viel, oder?

In diesem Fall doch, weil SuperDashboard trotz der geringen Anzahl von Methoden zu viele *Verantwortlichkeiten* hat.

Der Name einer Klasse sollte ihre Verantwortlichkeiten beschreiben. Tatsächlich ist die Wahl des Namens wahrscheinlich die erste Methode, um die Größe einer Klasse abzugrenzen. Wenn wir keinen prägnanten Namen für eine Klasse finden, ist sie wahrscheinlich zu groß. Je mehrdeutiger der Klassenname ist, desto eher hat sie zu viele Verantwortlichkeiten. Beispielsweise geben uns Klassennamen, die Sammelausdrücke wie *Processor*, *Manager* oder *Super* enthalten, oft einen Hinweis auf eine unglückliche Bündelung von Verantwortlichkeiten.

Wir sollten ebenfalls eine kurze Beschreibung der Klasse in etwa 25 Wörtern schreiben können, ohne die Wörter »wenn«, »und«, »oder« oder »aber« zu verwenden. Wie würden wir SuperDashboard beschreiben? »Die SuperDashboard-Klasse ermöglicht den Zugriff auf die Komponente, die zuletzt den Fokus hatte, und sie ermöglicht es uns auch, die Versions- und Build-Nummern zu verfolgen.« Das erste »und« ist ein Hinweis darauf, dass SuperDashboard zu viele Verantwortlichkeiten hat.

Das Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip (SRP; Eine-Verantwortlichkeit-Prinzip; ausführliche Informationen darüber finden Sie in [PPP]) fordert, dass eine Klasse oder ein Modul einen und nur einen *Grund zur Änderung* haben sollte. Dieses Prinzip liefert uns sowohl eine Definition der Verantwortlichkeit als auch eine Richtlinie für die Klassengröße. Klassen sollten eine Verantwortlichkeit haben – einen Grund zur Änderung.

Die auf den ersten Blick kleine SuperDashboard-Klasse in Listing 10.2 enthält zwei Gründe zur Änderung. Erstens überwacht sie die Versionsinformationen, die anscheinend bei jeder Auslieferung der Software aktualisiert werden müssen. Zweitens verwaltet sie Java-Swing-Komponenten. (Sie ist von der Klasse JFrame abgeleitet, der Swing-Repräsentation eines GUI-Fensters auf der obersten Ebene.) Zweifellos sollte die Versionsnummer aktualisiert werden, wenn der Swing-Code geändert wird; aber das Umgekehrte ist nicht unbedingt wahr: Wir können die Versionsinformationen auch wegen Änderungen an anderem Code im System aktualisieren.

Der Versuch, Verantwortlichkeiten (Gründe zur Änderung) zu identifizieren, hilft uns oft, bessere Abstraktionen in unserem Code zu erkennen und zu erstellen. Wir können leicht alle drei SuperDashboard-Methoden, die mit Versionsinformationen zu tun haben, in eine separate Klasse namens Version extrahieren (siehe Listing 10.3). Die Version-Klasse ist ein Konstrukt, das potenziell in vielen anderen Anwendungen wiederverwendet werden kann!

Listing 10.3: Klasse mit einer einzigen Verantwortlichkeit

```
public class Version {  
    public int getMajorVersionNumber()  
    public int getMinorVersionNumber()  
    public int getBuildNumber()  
}
```

Das SRP zählt zu den wichtigeren Konzepten des OO-Designs. Außerdem ist es sowohl vom Verständnis als auch von der Befolgung her eines der einfacheren Konzepte. Doch seltsamerweise ist das SRP oft das am meisten missbrauchte Klassendesign-Prinzip. Wir stoßen regelmäßig auf Klassen, die viel zu viele Aufgaben erfüllen. Warum?

Software zum Laufen zu bringen und saubere Software zu schreiben, sind zwei ganz verschiedene Aktivitäten. Da unsere Bewusstseinskapazität grundsätzlich beschränkt ist, konzentrieren wir uns meist eher darauf, den Code zum Laufen bringen, und weniger auf seinen Aufbau und seine Sauberkeit. Das ist absolut in Ordnung. Die Concerns (Belange) zu trennen, ist bei unserer *Programmiertätigkeit* genauso wichtig wie in unseren Programmen.

Das Problem liegt darin, dass zu viele denken, sie wären fertig, wenn das Programm funktioniert. Sie *versäumen* es dann, sich um die anderen Concerns der Organisation und Sauberkeit zu kümmern, und wenden sich dem nächsten Problem zu, statt zurückzugehen und ihre übervollen Klassen in Einheiten zu zerlegen, die nur noch eine einzige Verantwortlichkeit haben.

Zugleich fürchten viele Entwickler, dass eine große Zahl kleiner, auf einen Zweck beschränkter Klassen das Verstehen des Großen und Ganzen erschwere. Sie sind besorgt, dass sie von Klasse zu Klasse navigieren müssen, um herauszufinden, wie eine größere Komponente des Ganzen funktioniert.

Doch ein System mit vielen kleinen Klassen hat nicht mehr bewegliche Teile als ein System mit einigen wenigen großen Klassen. Bei einem System mit einigen wenigen großen Klassen muss man genauso viel lernen. Deshalb lautet die Frage: Haben Sie lieber einen Werkzeugkasten mit vielen kleinen Fächern, in dem alle Werkzeuge sauber geordnet und beschriftet aufbewahrt werden? Oder ziehen Sie es vor, alles zusammen in einige wenige große Fächer zu werfen?

Jedes größere System enthält eine umfangreiche Logik und Komplexität. Das Hauptziel bei der Verwaltung einer solchen Komplexität besteht darin, sie so zu *organisieren*, dass ein Entwickler weiß, wo er nachschauen muss, um bestimmte Dinge zu finden, und jeweils nur die direkt betroffene Komplexität verstehen muss. Dagegen behindert uns ein System mit größeren Mehrzweckklassen immer, weil es uns zwingt, durch zahlreiche Dinge zu waten, die wir im Moment gar nicht wissen müssen.

Um diesen Punkt noch einmal zu betonen: Unsere Systeme sollten aus vielen kleinen Klassen, nicht aus wenigen großen aufgebaut sein. Jede kleine Klasse kapselt eine einzige Verantwortlichkeit ein, hat einen einzigen Grund zur Änderung und arbeitet mit wenigen anderen zusammen, um das gewünschte Systemverhalten zu realisieren.

Kohäsion

Klassen sollten eine kleine Anzahl von Instanzvariablen haben. Jede Methode einer Klasse sollte eine oder mehrere dieser Variablen manipulieren. Im Allgemeinen hängen eine Methode und eine Klasse um so kohäsiver zusammen, je mehr Variablen die Methode manipuliert. Eine Klasse, in der jede Variable von jeder Methode verwendet wird, ist maximal kohäsiv.

Im Allgemeinen ist es weder ratsam noch möglich, solche maximal kohäsiven Klassen zu erstellen; andererseits soll die Kohäsion hoch sein. Eine hohe Kohäsion ist ein Kennzeichen dafür, dass die Methoden und Variablen der Klasse als logische Gesamtheit voneinander abhängen.

Betrachten Sie die Implementierung von `Stack` in Listing 10.4. Dies ist eine hoch kohäsive Klasse. Von den drei Methoden verwendet nur `size()` nicht beide Variablen.

Listing 10.4: `Stack.java`, eine kohäsive Klasse

```
public class Stack {  
    private int topOfStack = 0;  
    List<Integer> elements = new LinkedList<Integer>();  
  
    public int size() {  
        return topOfStack;  
    }  
}
```

```
public void push(int element) {
    topOfStack++;
    elements.add(element);
}

public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
        throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
}
}
```

Die Strategie, kleine Funktionen zu schreiben und kurze Parameterlisten zu verwenden, kann manchmal zu einer Proliferation (Vervielfältigung) von Instanzvariablen führen, die nur von einer Teilmenge der Methoden verwendet werden. Dies ist fast immer ein Zeichen dafür, dass wenigstens eine andere Klasse versucht, aus der größeren Klasse auszubrechen. Sie sollten dann versuchen, die Variablen und Methoden so auf zwei oder mehr Klassen zu verteilen, dass die neuen Klassen eine höhere Kohäsion aufweisen.

Kohäsion zu erhalten, führt zu vielen kleinen Klassen

Allein die Zerlegung großer Funktionen in kleinere Funktionen führt zu einer Proliferation von Klassen. Betrachten Sie eine große Funktion, in der viele Variablen deklariert werden. Angenommen, Sie wollten einen kleinen Teil dieser Funktion in eine separate Funktion auslagern. Doch der Code, den Sie extrahieren wollen, verwendet vier Variablen, die in der Funktion deklariert sind. Müssen Sie diese vier Variablen alle als Argumente an die neue Funktion übergeben?

Überhaupt nicht! Wenn wir diese vier Variablen zu Instanzvariablen der Klasse befördern, könnten wir den Code extrahieren, *ohne irgendeine* Variable zu übergeben. Es wäre *leicht*, die Funktion in kleinere Teile zu zerlegen.

Leider bedeutet dies auch, dass die Kohäsion unserer Klassen schwächer wird, weil sich immer mehr Instanzvariablen ansammeln, die nur existieren, damit sie von einigen Funktionen gemeinsam genutzt werden können. Doch ist nicht gerade die Tatsache, dass einige Funktionen bestimmte Variablen gemeinsam nutzen wollen, ein Zeichen dafür, dass sie in eine separate Klasse gehören? Natürlich! Wenn Klassen ihre Kohäsion verlieren, sollten Sie sie aufteilen!

Deshalb gibt uns die Zerlegung einer großen Funktion in viele kleinere Funktionen oft die Gelegenheit, auch mehrere kleinere Klassen herauszulösen. Damit verbessern wir die Struktur und Transparenz unseres Programms.

Um zu demonstrieren, was ich meine, möchte ich ein bewährtes Beispiel aus dem wundervollen Buch *Literate Programming* von Donald Knuth [Knuth92] verwenden.

Listing 10.5 zeigt eine Übersetzung von Knuths `PrintPrimes`-Programm in Java. Fairerweise muss man sagen, dass dies nicht das Programm ist, das Knuth geschrieben hat, sondern die Version, die von seinem WEB-Tool generiert wurde. Ich verwende dieses Programm, weil es hervorragend als Ausgangspunkt dient, um zu zeigen, wie eine große Funktion in viele kleinere Funktionen und Klassen zerlegt werden kann.

Listing 10.5: `PrintPrimes.java`

```
package literatePrimes;

public class PrintPrimes {
    public static void main(String[] args) {
        final int M = 1000;
        final int RR = 50;
        final int CC = 4;
        final int WW = 10;
        final int ORDMAX = 30;
        int P[] = new int[M + 1];
        int PAGENUMBER;
        int PAGEOFFSET;
        int ROWOFFSET;
        int C;
        int J;
        int K;
        boolean JPRIME;
        int ORD;
        int SQUARE;
        int N;
        int MULT[] = new int[ORDMAX + 1];

        J = 1;
        K = 1;
        P[1] = 2;
        ORD = 2;
        SQUARE = 9;

        while (K < M) {
            do {
                J = J + 2;
                if (J == SQUARE) {
                    ORD = ORD + 1;
                    SQUARE = P[ORD] * P[ORD];
                    MULT[ORD - 1] = J;
                }
                N = 2;
                JPRIME = true;
                while (N < ORD && JPRIME) {
                    while (MULT[N] < J)
                        MULT[N] = MULT[N] + P[N] + P[N];
```

```

        if (MULT[N] == J)
            JPRIME = false;
        N = N + 1;
    }
} while (!JPRIME);
K = K + 1;
P[K] = J;
}
{
    PAGENUMBER = 1;
    PAGEOFFSET = 1;
    while (PAGEOFFSET <= M) {
        System.out.println("The First " + M +
            " Prime Numbers --- Page " + PAGENUMBER);
        System.out.println("");
        for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++) {
            for (C = 0; C < CC; C++)
                if (ROWOFFSET + C * RR <= M)
                    System.out.format("%10d", P[ROWOFFSET + C * RR]);
                    System.out.println("");
        }
        System.out.println("\f");
        PAGENUMBER = PAGENUMBER + 1;
        PAGEOFFSET = PAGEOFFSET + RR * CC;
    }
}
}
}

```

Dieses Programm, geschrieben als eine einzige Funktion, ist chaotisch. Es verfügt über eine tief verschachtelte Struktur, zahlreiche seltsame Variablen und eine stark gekoppelte Struktur. Zumindest sollte die eine große Funktion in einige kleinere Funktionen zerlegt werden.

Die Listings 10.6 bis 10.8 zeigen das Ergebnis einer Zerlegung des Codes aus Listing 10.5 in kleinere Klassen und Funktionen sowie der Wahl aussagefähigerer Namen für diese Klassen, Funktionen und Variablen.

Listing 10.6: PrimePrinter.java (nach Refactoring)

```

package literatePrimes;

public class PrimePrinter {
    public static void main(String[] args) {
        final int NUMBER_OF_PRIMES = 1000;
        int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

        final int ROWS_PER_PAGE = 50;
        final int COLUMNS_PER_PAGE = 4;
        RowColumnPagePrinter tablePrinter =

```



```
        new RowColumnPagePrinter(ROWS_PER_PAGE,
                                   COLUMNS_PER_PAGE,
                                   "The First " + NUMBER_OF_PRIMES +
                                   " Prime Numbers");
    tablePrinter.print(primes);
}
}
```

Listing 10.7: RowColumnPagePrinter.java

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
    private int rowsPerPage;
    private int columnsPerPage;
    private int numbersPerPage;
    private String pageHeader;
    private PrintStream printStream;

    public RowColumnPagePrinter(int rowsPerPage,
                                 int columnsPerPage,
                                 String pageHeader) {
        this.rowsPerPage = rowsPerPage;
        this.columnsPerPage = columnsPerPage;
        this.pageHeader = pageHeader;
        numbersPerPage = rowsPerPage * columnsPerPage;
        printStream = System.out;
    }

    public void print(int data[]) {
        int pageNumber = 1;
        for (int firstIndexOnPage = 0;
             firstIndexOnPage < data.length;
             firstIndexOnPage += numbersPerPage) {
            int lastIndexOnPage =
                Math.min(firstIndexOnPage + numbersPerPage - 1,
                        data.length - 1);
            printPageHeader(pageHeader, pageNumber);
            printPage(firstIndexOnPage, lastIndexOnPage, data);
            printStream.println("\f");
            pageNumber++;
        }
    }

    private void printPage(int firstIndexOnPage,
                           int lastIndexOnPage,
                           int[] data) {
        int firstIndexOfLastRowOnPage =
            firstIndexOnPage + rowsPerPage - 1;
```

```

    for (int firstIndexInRow = firstIndexOnPage;
        firstIndexInRow <= firstIndexOfLastRowOnPage;
        firstIndexInRow++) {
        printRow(firstIndexInRow, lastIndexOnPage, data);
        printStream.println("");
    }
}

private void printRow(int firstIndexInRow,
                    int lastIndexOnPage,
                    int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
        int index = firstIndexInRow + column * rowsPerPage;
        if (index <= lastIndexOnPage)
            printStream.format("%10d", data[index]);
    }
}

private void printPageHeader(String pageHeader,
                            int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
}

public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
}
}

```

Listing 10.8: PrimeGenerator.java

```

package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
    private static int[] primes;
    private static ArrayList<Integer> multiplesOfPrimeFactors;

    protected static int[] generate(int n) {
        primes = new int[n];
        multiplesOfPrimeFactors = new ArrayList<Integer>();
        set2AsFirstPrime();
        checkOddNumbersForSubsequentPrimes();
        return primes;
    }

    private static void set2AsFirstPrime() {
        primes[0] = 2;
        multiplesOfPrimeFactors.add(2);
    }
}

```

```
private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
        primeIndex < primes.length;
        candidate += 2) {
        if (isPrime(candidate))
            primes[primeIndex++] = candidate;
    }
}

private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
        multiplesOfPrimeFactors.add(candidate);
        return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
}

private static boolean
isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
    int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
}

private static boolean
isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
        if (isMultipleOfNthPrimeFactor(candidate, n))
            return false;
    }
    return true;
}

private static boolean
isMultipleOfNthPrimeFactor(int candidate, int n) {
    return candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
}

private static int
smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
        multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
}
}
```

Vielleicht fällt Ihnen zunächst auf, dass das Programm länger geworden ist. War es vorher etwas über eine Seite lang, braucht es jetzt fast drei Seiten. Dafür gibt es mehrere Gründe. Erstens verwendet das refaktorierte Programm längere, aussagekräftigere Variablennamen. Zweitens verwendet das refaktorierte Programm Funktionen- und Klassendeklarationen als Möglichkeit, den Code zu kommentieren. Drittens verwenden wir Whitespace und Formatierungstechniken, um die Lesbarkeit des Programms zu erhalten.

Beachten Sie, wie das Programm in drei Hauptverantwortlichkeiten zerlegt worden ist. Das Hauptprogramm steht ganz allein in einer separaten `PrimePrinter`-Klasse. Es ist für die Verwaltung der Ausführungsumgebung zuständig. Es ändert sich, wenn sich die Methode seines Aufrufs ändert. Würde dieses Programm beispielsweise in einen SOAP-Service umgewandelt, wäre diese Klasse davon betroffen.

Die Klasse `RowColumnPagePrinter` weiß alles über die Formatierung einer Liste von Zahlen in Form von Seiten mit einer bestimmten Anzahl von Zeilen und Spalten. Müsste die Formatierung des Outputs geändert werden, wäre diese Klasse betroffen.

Die Klasse `PrimeGenerator` weiß, wie man eine Liste mit Primzahlen generiert. Beachten Sie, dass die Klasse nicht als Objekt instanziiert werden soll. Sie ist nur ein nützlicher Geltungsbereich, in dem ihre Variablen deklariert und verborgen werden können. Diese Klasse ändert sich, wenn der Algorithmus zur Berechnung der Primzahlen geändert wird.

Das Programm wurde nicht neu geschrieben! Wir begannen nicht von vorne und schrieben das Programm noch einmal. Wenn Sie sich die beiden Programme anschauen, werden Sie feststellen, dass sie ihre Aufgaben mit denselben Algorithmen und Techniken erfüllen.

Die Änderungen wurden gemacht, indem wir eine Test-Suite schrieben, die die *genaue* Verhaltensweise des ersten Programms verifizierte. Dann wurden zahlreiche Änderungen nacheinander durchgeführt. Nach jeder Änderung wurde das Programm ausgeführt, um zu prüfen, dass sich sein Verhalten nicht geändert hatte. In kleinen aufeinanderfolgenden Schritten wurde das erste Programm bereinigt und in das zweite transformiert.

10.3 Änderungen einplanen

Die meisten Systeme werden laufend geändert. Jede Änderung setzt uns dem Risiko aus, dass der Rest des Systems nicht mehr wie beabsichtigt funktioniert. In einem sauberen System sind die Klassen so strukturiert, dass dieses Risiko bei Änderungen reduziert wird.

Mit der `Sql`-Klasse in Listing 10.9 werden aus entsprechenden Metadaten korrekt formulierte SQL-Strings generiert. Das Programm befindet sich noch in der Ent-

wicklung. Deshalb unterstützt es bestimmte SQL-Funktionen wie etwa `update`-Anweisungen noch nicht. Wenn wir diese Funktionalität in die `Sql`-Klasse einbauen wollen, müssen wir diese Klasse »öffnen«. Das Öffnen einer Klasse ist immer mit einem Risiko verbunden. Jede Änderung der Klasse kann potenziell anderen Code in ihr beschädigen. Sie muss komplett neu getestet werden.

Listing 10.9: Eine Klasse, die zwecks Änderung geöffnet werden muss

```
public class Sql {  
    public Sql(String table, Column[] columns)  
    public String create()  
    public String insert(Object[] fields)  
    public String selectAll()  
    public String findByKey(String keyColumn, String keyValue)  
    public String select(Column column, String pattern)  
    public String select(Criteria criteria)  
    public String preparedInsert()  
    private String columnList(Column[] columns)  
    private String valuesList(Object[] fields, final Column[] columns)  
    private String selectWithCriteria(String criteria)  
    private String placeholderList(Column[] columns)  
}
```

Die `Sql`-Klasse muss geändert werden, wenn wir einen neuen Anweisungstyp hinzufügen. Sie muss ebenfalls geändert werden, wenn wir die Details eines einzigen Anweisungstyps ändern – etwa wenn wir die `select`-Funktion ändern müssen, damit sie Subselects unterstützt. Diese beiden Änderungsgründe bedeuten, dass die `Sql`-Klasse gegen das SRP verstößt.

Wir können diese SRP-Verletzung rein organisatorisch erkennen. Die Methodenauzählung von `Sql` zeigt, dass es `private` Methoden, wie etwa `selectWithCriteria`, gibt, die sich anscheinend nur auf `select`-Anweisungen beziehen.

Das Verhalten von `private` Methoden, die nur einer kleinen Untermenge einer Klasse dienen, kann uns nützliche Hinweise auf Verbesserungsmöglichkeiten liefern. Doch der Hauptanstoß für Aktionen sollte von den Systemänderungen selbst kommen. Wenn die `Sql`-Klasse als logisch vollständig eingestuft wird, müssen wir uns keine Gedanken über die Trennung der Verantwortlichkeiten machen. Wenn wir die `update`-Funktionalität in absehbarer Zukunft nicht brauchen, sollten wir `Sql` in Ruhe lassen. Doch sobald wir vor der Aufgabe stehen, eine Klasse zu öffnen, sollten wir überlegen, wie wir unser Design verbessern können.

Was wäre, wenn wir eine Lösung wie die in Listing 10.10 ins Auge fassen würden? Jede `Sql`-Methode mit einer öffentlichen Schnittstelle aus dem vorhergehenden Listing 10.9 wurde durch Refactoring in eine separate abgeleitete Klasse der `Sql`-Klasse umgewandelt. Beachten Sie, dass `private` Methoden, wie etwa `valuesList`, direkt

dort hin gewandert sind, wo sie gebraucht werden. Das gemeinsame private Verhalten wurde in zwei Utility-Klassen, `Where` und `ColumnList`, isoliert.

Listing 10.10: Ein Satz geschlossener Klassen

```
abstract public class Sql {
    public Sql(String table, Column[] columns)
    abstract public String generate();
}

public class CreateSql extends Sql {
    public CreateSql(String table, Column[] columns)
    @Override public String generate()
}

public class SelectSql extends Sql {
    public SelectSql(String table, Column[] columns)
    @Override public String generate()
}

public class InsertSql extends Sql {
    public InsertSql(String table, Column[] columns, Object[] fields)
    @Override public String generate()
    private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
    public SelectWithCriteriaSql(
        String table, Column[] columns, Criteria criteria)
    @Override public String generate()
}

public class SelectWithMatchSql extends Sql {
    public SelectWithMatchSql(
        String table, Column[] columns, Column column, String pattern)
    @Override public String generate()
}

public class FindByKeySql extends Sql {
    public FindByKeySql(
        String table, Column[] columns, String keyColumn, String keyValue)
    @Override public String generate()
}

public class PreparedInsertSql extends Sql {
    public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate() {
    private String placeholderList(Column[] columns)
}
```

```
public class Where {  
    public Where(String criteria)  
    public String generate()  
}  
  
public class ColumnList {  
    public ColumnList(Column[] columns)  
    public String generate()  
}
```

Der Code in den einzelnen Klassen ist jetzt außergewöhnlich einfach. Der Zeitaufwand, um eine Klasse zu verstehen, ist praktisch auf null gesunken. Das Risiko, dass eine Funktion eine andere beschädigen könnte, ist auch fast null. Und es ist leichter geworden, alle Komponenten dieser Lösung zu testen, da die Klassen jetzt alle voneinander isoliert sind.

Und was gleichermaßen wichtig ist: Wenn die `update`-Anweisungen hinzugefügt werden sollen, muss keine vorhandene Klasse geändert werden! Die Logik für die Erstellung von `update`-Anweisungen wird einfach in eine neue Unterklasse von `Sql` namens `UpdateSql` eingefügt. Anderer Code in dem System wird durch diese Änderung nicht beschädigt.

Unsere umstrukturierte `Sql`-Logik repräsentiert die beste aller Welten. Sie unterstützt das SRP. Sie unterstützt ebenfalls ein anderes Schlüsselprinzip des OO-Klassendesigns, nämlich das *Open-Closed-Prinzip* oder OCP [PPP]: Klassen sollten offen für Erweiterungen, aber geschlossen für Änderungen sein. Unsere umstrukturierte `Sql`-Klasse ist offen für neue Funktionalitäten, die durch Unterklassen zur Verfügung gestellt werden. Doch diese Änderungen können erfolgen, während alle anderen Klassen geschlossen bleiben. Wir fügen einfach unsere `UpdateSql`-Klasse zu dem System hinzu.

Die Struktur eines Systems sollte so beschaffen sein, dass wir sie so wenig wie möglich ändern müssen, wenn wir neue Funktionen hinzufügen oder vorhandene ändern. Idealerweise sind neue Funktionen einfach Erweiterungen des Systems, die den vorhandenen Code nicht ändern.

Änderungen isolieren

Anforderungen ändern sich, deshalb ändert sich der Code. Wir haben bei der Einführung in die OO gelernt, dass es konkrete Klassen gibt, die Implementierungsdetails (Code) enthalten, und abstrakte Klassen, die nur Konzepte repräsentieren. Eine Client-Klasse, die von konkreten Details abhängt, ist gefährdet, wenn sich diese Details ändern. Wir können Interfaces und abstrakte Klassen einführen, die dazu beitragen, die Auswirkungen dieser Details zu isolieren.

Dependencies von konkreten Details verursachen zusätzliche Probleme beim Testen des Systems. Wenn wir eine `Portfolio`-Klasse erstellen und die von einem

externen TokyoStockExchange-API abhängt, um den Wert des Portfolios zu berechnen, werden unsere Testfälle von der Volatilität eines solchen Lookups (Datenabrufs) beeinflusst. Es ist schwer, einen Test zu schreiben, wenn wir alle fünf Minuten eine andere Antwort erhalten!

Anstatt Portfolio so zu konzipieren, dass die Klasse direkt von TokyoStockExchange abhängt, erstellen wir ein Interface, StockExchange, das eine einzige Methode deklariert:

```
public interface StockExchange {  
    Money currentPrice(String symbol);  
}
```

Wir konzipieren TokyoStockExchange so, dass diese Klasse dieses Interface implementiert. Wir sorgen auch dafür, dass der Konstruktor von Portfolio eine StockExchange-Referenz als Argument übernimmt:

```
public Portfolio {  
    private StockExchange exchange;  
    public Portfolio(StockExchange exchange) {  
        this.exchange = exchange;  
    }  
    // ...  
}
```

Jetzt kann unser Test eine testbare Implementierung des StockExchange-Interface erstellen, die die TokyoStockExchange-Klasse emuliert. Diese Testimplementierung hält den gegenwärtigen Wert aller Symbole fest, die wir beim Testen verwenden. Wenn unser Test den Kauf von fünf Microsoft-Aktien für unser Portfolio demonstriert, codieren wir die Testimplementierung so, dass sie immer 100 US-Dollar pro Microsoft-Aktie zurückgibt. Unsere Testimplementierung des StockExchange-Interface ist auf ein einfaches Tabellen-Lookup geschrumpft. Dann können wir einen Test schreiben, der 500 US-Dollar als Gesamtwert des Portfolios erwartet.

```
public class PortfolioTest {  
    private FixedStockExchangeStub exchange;  
    private Portfolio portfolio;  
  
    @Before  
    protected void setUp() throws Exception {  
        exchange = new FixedStockExchangeStub();  
        exchange.fix("MSFT", 100);  
        portfolio = new Portfolio(exchange);  
    }  
  
    @Test  
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {  
        portfolio.add(5, "MSFT");  
    }  
}
```



```
        Assert.assertEquals(500, portfolio.value());  
    }  
}
```

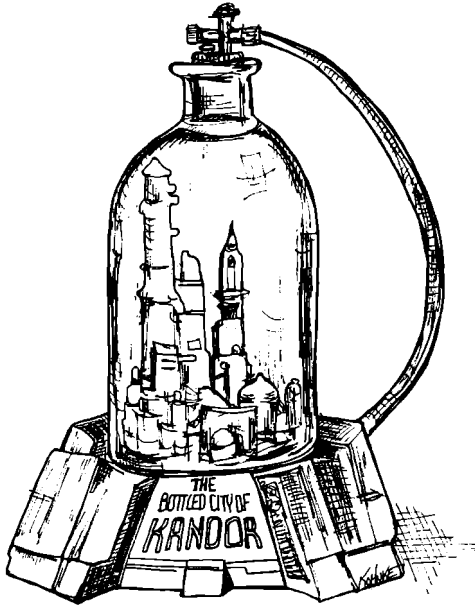
Wenn ein System so weit entkoppelt ist, dass es auf diese Weise getestet werden kann, ist es auch flexibler und kann leichter wiederverwendet werden. Die schwache Kopplung bedeutet, dass die Elemente unseres Systems besser voneinander und von Änderungen isoliert sind. Durch diese Isolierung ist es leichter, alle Elemente des Systems zu verstehen.

Indem wir so die Kopplung minimieren, sorgen wir dafür, dass unsere Klassen ein anderes Prinzip des Klassendesigns erfüllen, das sogenannte *Dependency-Inversion-Prinzip* (DIP; Prinzip der Abhängigkeitsumkehr; [PPP]). Im Wesentlichen fordert das DIP, dass unsere Klassen von Abstraktionen, nicht von konkreten Details abhängen sollten.

Unsere Portfolio-Klasse ist jetzt nicht mehr von den Implementierungsdetails der TokyoStockExchange-Klasse, sondern von dem StockExchange-Interface abhängig. Das StockExchange-Interface repräsentiert das abstrakte Konzept, den aktuellen Kurs eines Symbols (einer Aktie) abzurufen. Diese Abstraktion isoliert alle speziellen Details des Kursabrufs, einschließlich seines Herkunftsortes.

Systeme

von Dr. Kevin Dean



»Komplexität tötet. Sie saugt den Entwicklern das Leben aus und erschwert das Planen, Erstellen und Testen von Produkten.«

– Ray Ozzie, CTO, Microsoft Corporation

11.1 Wie baut man eine Stadt?

Könnten Sie alle Details selbst gestalten? Wahrscheinlich nicht. Selbst die Verwaltung einer vorhandenen Stadt ist zu viel für eine Person. Dennoch funktionieren Städte (meistens). Sie funktionieren, weil Städte über Teams von Entwicklern verfügen, die spezielle Aspekte der Stadt verwalten: die Wasserversorgung, die Stromversorgung, den Verkehr, die Einhaltung der Gesetze, Bauvorschriften usw. Einige dieser Entwickler sind für das *Gesamtbild* verantwortlich, andere kümmern sich um die Details.

Städte funktionieren auch deshalb, weil sie geeignete Abstraktionsebenen und Teilbereiche entwickelt haben, die es einzelnen Personen und den von ihnen verwalteten »Abteilungen (Kammern, Ämtern)« ermöglichen, effizient zu arbeiten, auch ohne das Gesamtbild zu verstehen.

Obwohl Software-Teams oft ähnlich aufgebaut sind, enthalten die Systeme, an denen sie arbeiten, häufig nicht dieselbe Trennung von Zuständigkeiten und Abstraktionsebenen. Sauberer Code hilft uns, dies auf den unteren Abstraktionsebenen zu erreichen. In diesem Kapitel wollen wir überlegen, wie wir auf den höheren Abstraktionsebenen, der Systemebene, sauber bleiben können.

11.2 Konstruktion und Anwendung eines Systems trennen

Zunächst müssen wir erkennen, dass *Konstruktion* und *Anwendung* ganz verschiedene Prozesse sind. Während ich dies schreibe, blicke ich durch mein Fenster in Chicago auf eine Baustelle, auf der ein neues Hotel errichtet wird. Heute ist es ein unverkleideter Betonkasten mit einem Baukran und einem Fahrstuhl, der an einer Außenwand befestigt ist. Die geschäftigen Ingenieure und Bauarbeiter tragen alle schwere Schutzhelme und Arbeitskleidung. In etwa einem Jahr wird das Hotel fertig sein. Der Kran und der Aufzug werden verschwunden sein. Das Gebäude wird sauber und die Wände werden mit Glasfenstern verkleidet und attraktiv angestrichen sein. Die Personen, die sich dann dort aufhalten werden, werden ganz anders aussehen.

Software-Systeme sollten den Startup-Prozess, wenn die Anwendungsobjekte konstruiert und die Abhängigkeiten »verdrahtet« werden, von der Laufzeit-Logik trennen, die nach dem Startup die Kontrolle übernimmt.

Der Startup-Prozess ist ein *Concern* (Belang, Verantwortlichkeit, Zuständigkeit), der bei jeder Anwendung geregelt sein muss. Es ist der erste *Concern*, den wir in diesem Kapitel untersuchen werden. Die *Trennung der Concerns* (Trennung der Verantwortlichkeiten) gehört zu den ältesten und wichtigsten Design-Techniken unserer Zukunft.

Leider behandeln die meisten Anwendungen diesen Concern nicht separat. Der Code für den Startup-Prozess wird ad hoc geschrieben und mit der Laufzeit-Logik vermischt. Hier ist ein typisches Beispiel:

```
public Service getService() {  
    if (service == null)  
        service = new MyServiceImpl(...); // In den meisten Fällen ausreichend?  
    return service;  
}
```

Dies ist das Idiom der *Lazy Initialization/Evaluation*, das mehrere Vorteile bietet: Der zusätzliche Aufwand für die Konstruktion eines Objekts fällt erst dann an, wenn

das Objekt tatsächlich verwendet wird. Folglich kann das Startup entsprechend schneller erfolgen. Außerdem sorgen wir dafür, dass niemals `null` zurückgegeben wird.

Doch jetzt gibt es eine fest encodierte Dependency (Abhängigkeit) von `MyServiceImpl` und allem, was deren Konstruktor benötigt (was ich ausgelassen habe). Ohne diese Dependencies können wir den Code nicht kompilieren, selbst wenn wir zur Laufzeit kein Objekt dieses Typs verwenden!

Das Testen kann ein Problem sein. Wenn `MyServiceImpl` ein schwergewichtiges Objekt ist, müssen wir dafür sorgen, dass dem `service`-Feld ein geeignetes *Test Double* [Mezzaroso7] oder *Mock Object* zugewiesen wird, bevor diese Methode während der Unit-Tests aufgerufen wird. Weil wir die Konstruktionslogik mit der normalen Laufzeit-Verarbeitung vermengt haben, sollten wir alle Ausführungspfade testen (zum Beispiel auch den `null`-Test und seinen Block). Die Methode hat also zwei Verantwortlichkeiten, das heißt mehr als eine Aufgabe, und verstößt damit im Kleinen gegen das *Single-Responsibility-Prinzip*.

Was vielleicht am schlimmsten ist: Wir wissen nicht, ob `MyServiceImpl` in allen Fällen das richtige Objekt ist. Der Kommentar soll diese Unklarheit zum Ausdruck bringen. Warum muss diese Klasse mit dieser Methode den globalen Kontext kennen? Können wir *jemals* wirklich wissen, welches Objekt an dieser Stelle richtig ist? Kann es überhaupt sein, dass ein Typ für alle möglichen Kontexte richtig ist?

Eine Anwendung der *Lazy Initialization* ist natürlich kein ernstes Problem. Doch Anwendungen enthalten normalerweise viele Instanzen kleiner derartiger Setup-Idiome. Deshalb ist die globale Setup-Strategie (falls es eine gibt) über die Anwendung *verstreut*. Sie ist gering modularisiert und enthält oft eine beträchtliche Duplizierung.

Wenn wir wirklich *sorgfältig* gut strukturierte und robuste Systeme erstellen wollen, sollten wir uns niemals dazu hinreißen lassen, die Modularität durch *bequeme* Idiome zu gefährden. Der Startup-Prozess der Objekt-Konstruktion und -Verknüpfung bildet keine Ausnahme. Wir sollten diesen Prozess von der normalen Laufzeit-Logik trennen und für die Anwendung einer globalen, konsistenten Strategie zur Auflösung der Haupt-Dependencies sorgen.

Trennung in main

Eine Möglichkeit, die Konstruktion von der Anwendung zu trennen, besteht darin, einfach alle Aspekte der Konstruktion nach `main` oder in Module zu verschieben, die von `main` aufgerufen werden, und das restliche System unter der Annahme zu konzipieren, dass alle Objekte konstruiert und korrekt verknüpft worden sind (siehe Abbildung 11.1).

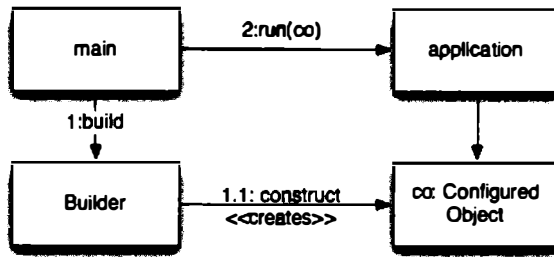


Abb. 11.1: Die Konstruktion nach main verlagern

Der Kontrollfluss ist leicht nachzuvollziehen. Die `main`-Funktion erstellt die für das System benötigten Objekte und übergibt sie dann an die Anwendung, die sie einfach benutzt. Beachten Sie, in welche Richtung die Dependency-Pfeile die Barriere zwischen `main` und der Anwendung überschreiten. Sie weisen alle in eine Richtung, die von `main` wegzeigt. Dies bedeutet, dass die Anwendung nichts über `main` oder den Konstruktionsprozess weiß. Sie nimmt einfach an, dass alles korrekt erstellt worden ist.

Factories

Natürlich ist manchmal die Anwendung selbst verantwortlich dafür, wann ein Objekt erstellt wird. Beispielsweise muss die Anwendung in einem Auftragsverwaltungssystem die `LineItem`-Instanzen (Positionen) eines Auftrags erstellen. In diesem Fall können wir der Anwendung mit dem *Abstract Factory*-Pattern [GOF] die Kontrolle über den Zeitpunkt der Erstellung von Auftragspositionen geben, aber die Details dieser Konstruktion von dem Anwendungscode trennen (siehe Abbildung 11.2).

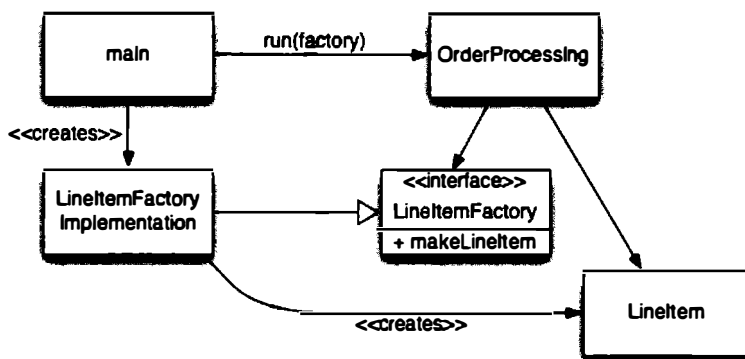


Abb. 11.2: Abtrennung der Konstruktion mit Factory

Beachten Sie, dass auch hier alle Dependencies von `main` weg auf die `OrderProcessing`-Anwendung zeigen. Dies bedeutet, dass die Anwendung von den Details

der Erstellung von `LineItem`-Objekten entkoppelt ist. Die entsprechende Fähigkeit ist in der `LineItemFactoryImplementation` eingekapselt, die sich auf der `main`-Seite der Trennlinie befindet. Dennoch hat die Anwendung die vollständige Kontrolle über den Zeitpunkt, zu dem `LineItem`-Instanzen erstellt werden, und kann sogar anwendungsspezifische Konstruktor-Argumente übergeben.

Dependency Injection

Eine leistungsstarke Technik, um die Konstruktion von der Anwendung zu trennen, ist die *Dependency Injection* (DI; Injektion der Abhängigkeit), die Anwendung der *Inversion of Control* (IoC; Umkehrung der Kontrolle) auf das Dependency-Management ([Fowler] u.a.). Die Inversion of Control verlagert untergeordnete Verantwortlichkeiten von einem Objekt auf andere Objekte, die dem jeweiligen Zweck dienen, und unterstützt damit das *Single-Responsibility-Prinzip*. Im Kontext des Dependency-Managements sollte ein Objekt nicht die Verantwortung dafür tragen, Dependencies selbst zu instanzieren, sondern es sollte diese Verantwortung an einen anderen »zuständigen« Mechanismus delegieren und damit die »Kontrolle umkehren«. Weil das Setup ein globaler Concern ist, sollte dieser zuständige Mechanismus normalerweise die »main«-Routine oder ein spezieller, diesem Zweck dienender *Container* sein.

JNDI-Lookups sind eine »partielle« Implementierung der DI, in der ein Objekt einen Verzeichnis-Server auffordert, einen »Service« bereitzustellen, der einem speziellen Namen entspricht.

```
MyService myService = (MyService) (jndiContext.lookup("NameOfMyService"));
```

Das aufrufende Objekt hat keine Kontrolle über die Art von Objekt, das tatsächlich zurückgegeben wird. (Natürlich muss es das entsprechende Interface implementieren.) Dennoch löst das aufrufende Objekt aktiv die Abhängigkeit auf.

Echte Dependency Injection geht einen Schritt weiter. Die Klasse unternimmt keine direkten Schritte, ihre Abhängigkeiten aufzulösen; sie ist vollständig passiv. Stattdessen stellt sie Setter-Methoden oder Konstruktor-Argumente (oder beides) zur Verfügung, mit denen die Dependencies injiziert werden. Während der Konstruktion instanziiert der DI-Container die erforderlichen Objekte (normalerweise auf Anfrage) und benutzt die bereitgestellten Konstruktor-Argumente oder Setter-Methoden, um die Dependencies zu verknüpfen. Welche abhängigen Objekte tatsächlich verwendet werden, wird durch eine Konfigurationsdatei oder per Programm in einem speziellen Konstruktionsmodul festgelegt.

Das Spring Framework ist der bekannteste DI-Container für Java. (Siehe [Spring]. Es gibt auch ein Spring.NET Framework.) Sie definieren in einer XML-Konfigurationsdatei, welche Objekte verknüpft werden sollen, und fordern dann spezielle Objekte namentlich in Ihrem Java-Code an. Etwas später finden Sie ein Beispiel.

Aber was ist mit den Vorzügen der *Lazy Initialization*?³ Dieses Idiom ist auch mit DI manchmal nützlich. Erstens: Die meisten DI-Container konstruieren ein Objekt nicht, bevor es nicht gebraucht wird. Zweitens: Viele dieser Container stellen Mechanismen zur Verfügung, um Factories aufzurufen oder Proxies zu konstruieren, die für eine *Lazy Evaluation* und ähnliche *Optimierungen* verwendet werden können. (Sie dürfen nicht vergessen, dass *Lazy Initialization/Evaluation* nur eine Optimierungstechnik ist, die vielleicht zu früh angewendet wird.)

11.3 Aufwärtsskalierung

Städte sind gewachsene Dörfer und diese wiederum gewachsene Ansiedlungen. Zuerst sind die Straßen schmal und praktisch nicht vorhanden; dann werden sie gepflastert; dann werden sie mit der Zeit verbreitert. Kleine Gebäude werden durch größere Gebäude ersetzt, leere Grundstücke bebaut. Einige dieser Gebäude werden später von Wolkenkratzern abgelöst.

Zunächst gibt es keine Versorgung mit Strom, Gas oder Wasser, kein Abwässersystem und kein Internet (oh Schreck!). Diese Dienste werden nach und nach auf- und ausgebaut, wenn die Bevölkerung wächst und die Bebauungsdichte zunimmt.

Dieses Wachstum verläuft oft nicht schmerzlos. Wie oft haben Sie Stoßstange an Stoßstange an einer Baustelle eines »Verbesserungsprojekts« im Stau gestanden und sich gefragt: »Warum hat man das nicht gleich richtig gebaut!«?

Aber anders wäre es gar nicht möglich gewesen. Wer kann die Kosten für den Ausbau einer sechsspurigen Schnellstraße mitten durch eine Kleinstadt rechtfertigen, die ein Wachstum erwartet? Wer *wünscht* sich eine solche Straße durch seine Stadt?

Es ist ein Mythos, dass wir Systeme »gleich beim ersten Mal richtig« erstellen können. Stattdessen sollten wir nur die heutigen *Stories* (Geschichten) implementieren, dann refaktorisieren und das System erweitern, um morgen neue Stories zu implementieren. Dies ist die Essenz der iterativen und inkrementellen Agilität. Test Driven Development (TDD; testgesteuerte Entwicklung), Refaktorisierung und der saubere Code, der dadurch erstellt wird, sorgen dafür, dass dies auf Code-Ebene funktioniert.

Aber was ist mit der Systemebene? Verlangt die Systemarchitektur keine Vorausplanung? Sicher kann sie nicht schrittweise vom Einfachen zum Komplexen wachsen, *oder etwa doch?*

Software-Systeme sind im Vergleich zu physikalischen Systemen einzigartig. Ihre Architekturen können schrittweise wachsen, wenn wir eine geeignete Trennung der Concerns beachten.

Dies wird durch die flüchtige Natur von Software-Systemen ermöglicht, wie Sie gleich sehen werden. Zunächst wollen wir ein Gegenbeispiel einer Architektur betrachten, bei der die Concerns nicht sauber getrennt sind.

Bei den ursprünglichen EJB1- und EJB2-Architekturen waren die Concerns nicht sauber getrennt. Dadurch wurden unnötige Barrieren für ein organisches Wachstum aufgebaut. Betrachten Sie etwa eine *Entity Bean* für eine persistente Bank-Klasse. Eine Entity Bean ist eine speicherresidente Repräsentation relationaler Daten, anders ausgedrückt: einer Tabellenzeile.

Zuerst musste man ein lokales (in process) oder fernes (separates JVM-) Interface definieren, das die Clients verwenden sollten. Listing 11.1 zeigt ein mögliches lokales Interface:

Listing 11.1: Ein lokales Interface von EJB2 für eine Bank-EJB

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
    String getStreetAddr1() throws EJBException;
    String getStreetAddr2() throws EJBException;
    String getCity() throws EJBException;
    String getState() throws EJBException;
    String getZipCode() throws EJBException;
    void setStreetAddr1(String street1) throws EJBException;
    void setStreetAddr2(String street2) throws EJBException;
    void setCity(String city) throws EJBException;
    void setState(String state) throws EJBException;
    void setZipCode(String zip) throws EJBException;
    Collection getAccounts() throws EJBException;
    void setAccounts(Collection accounts) throws EJBException;
    void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Ich habe mehrere Attribute der Adresse der Bank und eine Collection von Konten gezeigt, die der Bank gehören und deren Daten jeweils durch eine separate Account-EJB gehandhabt werden würden. Listing 11.2 zeigt die entsprechende Implementierungsklasse der Bank-Bean.

Listing 11.2: Die entsprechende EJB2-Entity-Bean-Implementierung

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
    // Geschäftslogik ...
    public abstract String getStreetAddr1();
    public abstract String getStreetAddr2();
    public abstract String getCity();
    public abstract String getState();
    public abstract String getZipCode();
    public abstract void setStreetAddr1(String street1);
```



```
public abstract void setStreetAddr2(String street2);
public abstract void setCity(String city);
public abstract void setState(String state);
public abstract void setZipCode(String zip);
public abstract Collection getAccounts();
public abstract void setAccounts(Collection accounts);
public void addAccount(AccountDTO accountDTO) {
    InitialContext context = new InitialContext();
    AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
    AccountLocal account = accountHome.create(accountDTO);
    Collection accounts = getAccounts();
    accounts.add(account);
}
// EJB-Container-Logik
public abstract void setId(Integer id);
public abstract Integer getId();
public Integer ejbCreate(Integer id) { ... }
public void ejbPostCreate(Integer id) { ... }
// Der Rest musste implementiert werden, war aber gewöhnlich leer:
public void setEntityContext(EntityContext ctx) {}
public void unsetEntityContext() {}
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbLoad() {}
public void ejbStore() {}
public void ejbRemove() {}
}
```

Ich habe das entsprechende LocalHome-Interface nicht gezeigt. Es handelt sich im Wesentlichen um eine Factory zum Erstellen von Objekten. Auch mögliche Abfragemethoden von Bank, die man hinzufügen könnte, sind nicht angegeben.

Schließlich musste man eine oder mehrere XML-Deployment-Deskriptoren schreiben, um die Details des Objekt-relationalen Mappings auf einen persistenten Speicher, das gewünschte Transaktionsverhalten, Sicherheitseinschränkungen usw. zu beschreiben.

Die Geschäftslogik ist eng an den EJB2-Anwendungs-»Container« gekoppelt. Sie müssen Unterklassen von Container-Typen bilden und viele Lifecycle-Methoden zur Verfügung stellen, die von dem Container benötigt werden.

Wegen dieser Kopplung an den schwergewichtigen Container sind isolierte Unit-Tests schwierig. Es ist erforderlich, den Container durch ein Mock-Objekt zu ersetzen, was schwierig ist, oder viel Zeit damit zu verschwenden, EJBs und Tests auf einem echten Server zu deployen. Wegen der engen Kopplung ist eine Wiederverwendung außerhalb der EJB2-Architektur praktisch unmöglich.

Schließlich wird sogar die objektorientierte Programmierung untergraben. Eine Bean kann nichts von einer anderen erben. Beachten Sie die Logik, um ein neues

Konto hinzuzufügen. Bei EJB2-Beans werden üblicherweise »Data Transfer Objects« (DTOs; Datentransferobjekte) definiert, die im Wesentlichen aus »structs« ohne Verhaltensweisen bestehen. Dies führt normalerweise zu redundanten Typen, die hauptsächlich dieselben Daten speichern; und es wird Standard-Code benötigt, um Daten von einem Objekt zu einem anderen zu kopieren.

Cross-Cutting Concerns

In einigen Bereichen kommt die EJB2-Architektur nahe an eine echte Trennung von Concerns heran. Beispielsweise wird das gewünschte Transaktions-, Sicherheits- und ein Teil des Persistenzverhaltens unabhängig vom Sourcecode in den Deployment-Deskriptoren deklariert.

Beachten Sie, dass *Concerns* wie Persistenz oft über die natürlichen Objektgrenzen einer Domäne hinweggreifen. Im Allgemeinen wollen Sie alle Ihre Objekte mit derselben Strategie dauerhaft speichern, etwa mit einem speziellen DBMS (Datenbank-Management-System) und nicht in flachen Dateien. Sie wollen dabei bestimmte Namenskonventionen für Tabellen und Spalten einhalten, eine konsistente Transaktionssemantik verwenden usw.

Theoretisch können Sie Ihre Persistenzstrategie mit gekapselten Modulen konzipieren. Doch in der Praxis müssen Sie im Wesentlichen identischen Code, mit dem Sie die Persistenzstrategie implementieren, auf viele Objekte verteilen. Für dieses Phänomen wurde die Bezeichnung *Cross-Cutting Concerns* geprägt (wörtlich »querschneidende Belange«; Probleme, die unabhängig vom Einzelobjekt in jedem Objekttyp gelöst werden müssen, zum Beispiel speichern oder drucken). Auch hier kann das Persistenz-Framework modular sein; und auch unsere Bereichslogik könnte isoliert eine modulare Struktur haben. Das Problem ist die feinkörnige *Überschneidung* der Domänen (Bereiche).

Tatsächlich nahm die Methode, wie die EJB-Architektur mit Persistenz, Sicherheit und Transaktionen das *Aspect-oriented Programming* (AOP, Aspektorientierte Programmierung) vorweg, einen Allzweckansatz, um die Modularität auch bei Cross-Cutting Concerns zu realisieren. Allgemeine Informationen über Aspekte und die AOP finden Sie in [AOSD]. [AspectJ] und [Colyer] liefern AspectJ-spezifische Informationen.

In der AOP spezifizieren modulare Konstrukte, die so genannten *Aspekte*, an welchen Punkten des Systems das Verhalten in einer konsistenten Weise implementiert werden soll, um einen speziellen Concern zu unterstützen. Diese Spezifikation erfolgt durch Einbettung bestimmter Deklarationen und Anweisungen in den Programmcode.

Betrachten wir Persistenz als Beispiel. Sie deklarieren, welche Objekte und Attribute (oder *Patterns* von Objekten und Attributen) gespeichert werden sollen, und delegieren dann die Persistenz-Tasks an Ihr Persistenz-Framework. Das AOP-Framework ändert das Verhalten des Target-Codes *nicht-invasiv*, was bedeutet, dass Sie den

Sourcecode des Targets manuell nicht ändern müssen. Schauen wir uns drei Aspekte oder aspektähnliche Mechanismen in Java an.

11.4 Java-Proxies

Java-Proxies eignen sich in einfachen Situationen, etwa um Methodenaufrufe in einzelne Objekte oder Klassen einzuhüllen. Doch die dynamischen Proxies, die in dem JDK zur Verfügung gestellt werden, funktionieren nur mit Interfaces. Für Klassen-Proxies müssen Sie ein Library zur Byte-Code-Manipulation verwenden, wie etwa CGLIB, ASM oder Javassist (siehe [CGLIB], [ASM] oder [Javassist]).

Listing 11.3 zeigt das Gerüst für ein JDK-Proxy, das eine Persistenz-Unterstützung für unsere Bank-Anwendung zur Verfügung stellt. Es werden nur die Methoden dargestellt, um die Liste der Konten abzurufen und zu speichern.

Listing 11.3: JDK-Proxy-Beispiel

```
// Bank.java (suppressing package names...)
import java.util.*;

// Die Abstraktion einer Bank
public interface Bank {
    Collection<Account> getAccounts();
    void setAccounts(Collection<Account> accounts);
}

// BankImpl.java
import java.util.*;

// Das "Plain Old Java Object" (POJO), das die Abstraktion implementiert
public class BankImpl implements Bank {
    private List<Account> accounts;

    public Collection<Account> getAccounts() {
        return accounts;
    }
    public void setAccounts(Collection<Account> accounts) {
        this.accounts = new ArrayList<Account>();
        for (Account account: accounts) {
            this.accounts.add(account);
        }
    }
}

// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;

// "InvocationHandler", der von dem Proxy-API benötigt wird
```

```

public class BankProxyHandler implements InvocationHandler {
    private Bank bank;

    public BankHandler (Bank bank) {
        this.bank = bank;
    }

    // Methode, die in InvocationHandler definiert ist
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        String methodName = method.getName();
        if (methodName.equals("getAccounts")) {
            bank.setAccounts(getAccountsFromDatabase());
            return bank.getAccounts();
        } else if (methodName.equals("setAccounts")) {
            bank.setAccounts((Collection<Account>) args[0]);
            setAccountsToDatabase(bank.getAccounts());
            return null;
        } else {
            ***
        }
    }

    // Zahlreiche Details hier ...
    protected Collection<Account> getAccountsFromDatabase() { ... }
    protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}

// An anderer Stelle ...

Bank bank = (Bank) Proxy.newProxyInstance(
    Bank.class.getClassLoader(),
    new Class[] { Bank.class },
    new BankProxyHandler(new BankImpl()));

```

Wir definieren ein Interface, `Bank`, das von dem Proxy *eingehüllt* (engl. *wrapped*) wird, und ein *Plain-Old Java Object* (POJO), `BankImpl`, das die Geschäftslogik implementiert. (Wir kommen in Kürze auf POJOs zurück.)

Das Proxy-API benötigt ein `InvocationHandler`-Objekt, das es aufruft, um die Aufrufe von `Bank`-Methoden zu implementieren, die an das Proxy gerichtet werden. Unser `BankProxyHandler` benutzt das Java Reflection API, um die generischen Methodenaufrufe den entsprechenden Methoden in `BankImpl` zuzuordnen, usw.

Der Code hier ist *ziemlich umfangreich* und selbst in diesem einfachen Fall relativ kompliziert. (Ausführlichere Beispiele für das Proxy-API und Anwendungsbeispiele finden Sie beispielsweise in [Goetz].) Der Einsatz einer der Libraries zur Byte-Manipulation ist ähnlich schwierig. Der Umfang und die Komplexität dieses Codes sind zwei Nachteile von Proxies. Dadurch wird es schwer, sauberen Code zu erstellen.

len! Außerdem stellen Proxies keinen Mechanismus zur Verfügung, um systemweit interessante »Ausführungspunkte« zu definieren, was für eine echte AOP-Lösung erforderlich ist. (AOP wird manchmal mit den Techniken verwechselt, mit denen es implementiert wird, wie etwa die Methoden-Interception und das »Wrapping« mittels Proxies. Der wirkliche Wert eines AOP-Systems basiert auf seiner Fähigkeit, systemische Verhaltensweise klar und knapp zu beschreiben.)

11.5 Reine Java-AOP-Frameworks

Glücklicherweise kann der meiste Proxy-Standardcode automatisch mit Werkzeugen erstellt und bearbeitet werden. Proxies werden intern in mehreren Java-Frameworks (beispielsweise Spring AOP und JBoss AOP) verwendet, um Aspekte in reinem Java zu implementieren. (Siehe [Spring] und [JBoss]. »Reines Java« bedeutet Java ohne Aspect[.]) In Spring schreiben Sie Ihre Geschäftslogik als *Plain-Old Java Objects*. POJOs sind ausschließlich auf Ihre Domäne fokussiert. Sie haben keine Dependencies von Enterprise Frameworks (oder anderen Domänen). Deshalb sind sie konzeptionell einfacher und leichter zu testen. Wegen der relativen Einfachheit können Sie auch die entsprechenden Benutzer-Stories leichter korrekt implementieren. Ihr Code wird wartungsfreundlicher und kann bei zukünftigen Stories besser erweitert werden.

Die erforderliche Anwendungsinfrastruktur, einschließlich der Cross-Cutting Concerns wie Persistenz, Transaktionen, Sicherheit, Caching, Ausfallsicherheit usw., wird mithilfe deklarativer Konfigurationsdateien oder APIs in die Programme eingebaut. In vielen Fällen spezifizieren Sie tatsächlich Spring- oder JBoss-Library-Aspekte, wobei das Framework die mechanische Anwendung der Java-Proxies oder Byte-Code-Libraries für den Benutzer transparent handhabt. Diese Deklarationen steuern den Dependency Injection Container (DI-Container), der die Hauptobjekte auf Anforderung instanziert und miteinander verknüpft.

Listing 11.4 zeigt ein typisches Fragment der Konfigurationsdatei `app.xml` von Spring V2.5 (adaptiert von <http://www.theserverside.com/articles/article.tss?l=IntrotoSpring25>):

Listing 11.4: Konfigurationsdatei von Spring 2.X

```
<beans>

  ...
  <bean id="appDataSource"
    class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close"
    p:driverClassName="com.mysql.jdbc.Driver"
    p:url="jdbc:mysql://localhost:3306/mydb"
    p:username="me"/>

  <bean id="bankDataAccessObject"
    class="com.example.banking.persistence.BankDataAccessObject"
```

```

p:dataSource-ref="appDataSource"/>

<bean id="bank"
class="com.example.banking.model.Bank"
p:dataAccessObject-ref="bankDataAccessObject"/>
* * *
</beans>

```

Jede »bean« ist einer Figur einer verschachtelten »Russischen Puppe« vergleichbar. Ein Domänenobjekt (Bank) wird von einem Proxy, einem Data Access Object (DAO; Datenzugriffs-Objekt), eingehüllt, das seinerseits von einer JDBC-Driver-Data-Source eingehüllt ist (siehe Abbildung 11.3).

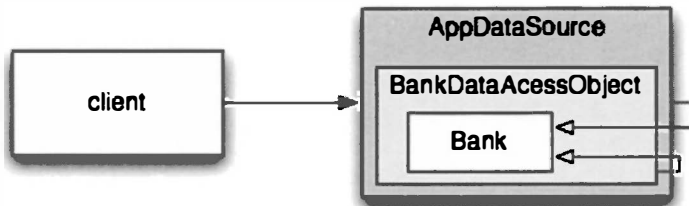


Abb. 11.3: Die »Russische Puppe« der Decorators

Der Client glaubt, er würde `getAccounts()` eines `Bank`-Objekts aufrufen; tatsächlich redet er aber mit der äußeren Schicht verschachtelter *Decorator*-Objekte [GOF], die das grundlegende Verhalten des `Bank`-POJOs erweitern. Wir könnten andere Decorators für Transaktionen, Caching und andere Funktionen hinzufügen.

In der Anwendung werden einige Zeilen benötigt, um den DI-Container nach den Top-Level-Objekten des Systems zu fragen, die in der XML-Datei spezifiziert sind.

```

XmlBeanFactory bf =
    new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");

```

Weil nur so wenige Zeilen von Spring-spezifischem Java-Code benötigt werden, ist die Anwendung *fast vollständig von Spring entkoppelt*. Damit sind alle Probleme der engen Systemkopplung wie bei EJB2 nicht existent.

Obwohl XML wortreich und schwer zu lesen ist, ist die »Policy«, die in diesen Konfigurationsdateien festgelegt wird, einfacher als die komplizierte Proxy- und Aspekt-Logik, die unsichtbar bleibt und automatisch erstellt wird. (Das Beispiel könnte noch durch Mechanismen vereinfacht werden, die dem Prinzip *Convention over Configuration*, Konvention über Konfiguration, folgen und Java-5-Annotationen verwenden, um den Umfang der ausdrücklich anzugebenden Verknüpfungslogik zu reduzieren.) Diese Art von Architektur ist so überzeugend, dass Frameworks wie Spring zu einer vollständigen Überarbeitung des EJB-Standards führten. EJB3 folgt

zu einem großen Teil dem Spring-Modell der deklarativen Unterstützung von Cross-Cutting Concerns mit XML-Konfigurationsdateien und/oder Java-5-Annotationen.

Listing 11.5 zeigt unser Bank-Objekt umgeschrieben in EJB3 (adaptiert von <http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api.html>).

Listing 11.5: Eine EJB3-Bank-EJB

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    @Embeddable // Ein "inlined" Objekt in der DB-Zeile von Bank
    public class Address {
        protected String streetAddr1;
        protected String streetAddr2;
        protected String city;
        protected String state;
        protected String zipCode;
    }

    @Embedded
    private Address address;

    @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
        mappedBy="bank")
    private Collection<Account> accounts = new ArrayList<Account>();

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public void addAccount(Account account) {
        account.setBank(this);
        accounts.add(account);
    }

    public Collection<Account> getAccounts() {
```

```
    return accounts;
}

public void setAccounts(Collection<Account> accounts) {
    this.accounts = accounts;
}
}
```

Dieser Code ist viel sauberer als der ursprüngliche EJB2-Code. Einige Entity-Details sind immer noch vorhanden, und zwar in den Annotationen. Doch weil sich keine dieser Informationen außerhalb von Annotationen befindet, ist der Code sauber und klar und deshalb leicht zu testen, zu warten usw.

Einige oder alle Persistenz-Informationen in den Annotationen können auf Wunsch in XML-Deployment-Deskriptoren ausgelagert werden, um wirklich reine POJOs zu erhalten. Wenn sich die Details des Persistenz-Mappings nur selten ändern, ziehen es viele Teams möglicherweise vor, die Annotationen zu behalten. Dies hat aber erheblich weniger schädliche Nachteile als das invasive Verhalten von EJB2.

11.6 AspectJ-Aspekte

Schließlich ist die AspectJ-Sprache (siehe [AspectJ] und [Colyer]) das funktionsreichste Werkzeug zur Trennung von Concerns. AspectJ ist eine Erweiterung von Java, die eine »erstklassige« Unterstützung zur Modularisierung von Aspekten bietet. Die reinen Java-Ansätze von Spring AOP und JBoss AOP reichen in 80 bis 90 Prozent der Fälle aus, in denen Aspekte am nützlichsten sind. Doch AspectJ stellt einen sehr reichhaltigen und leistungsstarken Werkzeugsatz für die Trennung von Concerns zur Verfügung. Der Nachteil von AspectJ liegt darin, dass Sie mehrere neue Tools, Sprachkonstrukte und Anwendungssidiome lernen müssen.

Die Probleme der Übernahme sind etwas kleiner geworden, seit vor kurzem die »Annotation-Form« von AspectJ eingeführt worden ist, bei der Aspekte mit Java-5-Annotationen in reinem Java-Code definiert werden können. Außerdem enthält das Spring Framework eine Reihe von Funktionen, die das Einfügen von annotations-basierten Aspekten für ein Team mit begrenzter AspectJ-Erfahrung erheblich erleichtern.

Ein komplette Beschreibung von AspectJ sprengt den Rahmen dieses Buches. Näheres finden Sie in [AspectJ], [Colyer] und [Spring].

11.7 Die Systemarchitektur testen

Die Wirksamkeit der Trennung von Concerns durch aspektorientierte Ansätze kann gar nicht überbetont werden. Wenn Sie Ihre Anwendungslogik mit POJOs schreiben können, die auf Code-Ebene von allen architektonischen Concerns entkoppelt

sind, können Sie Ihre Architektur wirklich testgesteuert entwickeln. Sie können sie nach Bedarf vom Einfachem zum Komplexen hin ausbauen und je nach Anforderung neue Technologien einbauen. Es ist kein *Big Design Up Front* (BDUF; »Großes Design vor der Arbeit«) erforderlich. Tatsächlich ist BDUF sogar schädlich, weil es die Anpassung an Änderungen behindert. Denn es gibt einen psychologischen Widerstand dagegen, frühere Aufwendungen zu verwerfen. Außerdem schränken frühere Entscheidungen über die Architektur das nachfolgende Denken über das Design ein. (BDUF darf nicht mit der guten Praxis verwechselt werden, vorher ein Design zu erstellen. BDUF bedeutet, *alles und jedes* zu designen, bevor die Implementierung überhaupt beginnt.)

Architekten von Gebäuden müssen ein BDUF leisten, weil es nicht machbar ist, radikale Änderungen einer großen physischen Struktur durchzuführen, wenn die Konstruktion über die ersten Anfänge hinausgekommen ist. Dennoch gibt es auch hier beträchtliche iterative Anstrengungen und Diskussionen über Details. Selbst nachdem der Bau längst begonnen hat. Obwohl Software über eine eigene *Physik* verfügt, ist es ökonomisch machbar, die Struktur radikal zu ändern, *falls* die Struktur der Software ihre Concerns wirksam trennt. (Der Terminus *software physics*, Softwarephysik, wurde zum ersten Mal von [Kolence] verwendet.)

Dies bedeutet, dass wir ein Software-Projekt mit einer »naiv einfachen«, aber hübschenkoppelten Architektur starten, schnell funktionierende Benutzer-Stories liefern und dann die Infrastruktur ausbauen können, wenn wir das System aufwärtsskalieren. Einige der größten Websites der Welt haben eine sehr hohe Verfügbarkeit und ein hohes Leistungsverhalten erzielt, indem sie gekonnt und flexibel Daten-Caching, Sicherheitsfunktionen, Virtualisierung usw. einsetzen können, weil sie auf minimalgekoppelten Designs basieren, die auf jeder Abstraktionsebene und in dem Geltungsbereich entsprechend *einfach* sind.

Natürlich bedeutet dies nicht, dass wir ohne Zielvorstellungen und Steuerungsmittel in ein Projekt einsteigen. Wir haben einige Erwartungen an den allgemeinen Geltungsbereich, die Ziele und den Plan für das Projekt sowie an die allgemeine Struktur des fertigen Systems. Doch wir müssen unsere Fähigkeit bewahren, das System an neue Umstände anpassen zu können.

Die frühe EJB-Architektur ist nur eines der vielen bekannten APIs, die eng gekoppelte Komponenten verwendeten und die Trennung von Concerns unnötig erschwerten. Selbst gut konzipierte APIs können zu viel des Guten anbieten, wenn es nicht wirklich benötigt wird. Ein gutes API sollte die meiste Zeit über hauptsächlich *unsichtbar* sein, damit sich das Team vor allem auf die Implementierung der Benutzer-Stories konzentriert. Ist dies nicht der Fall, behindern architektonische Einschränkungen die effiziente Lieferung des optimalen Werts an den Kunden.

Zusammenfassend können wir also sagen:

Eine optimale Systemarchitektur besteht aus modularisierten Concern-Domänen, die jeweils mit Plain Old Java Objects (oder anderen Objekten) imple-

mentiert werden. Die verschiedenen Domänen werden durch minimal invasive Aspekte oder aspektähnliche Tools integriert. Diese Architektur kann, wie der Code, testgesteuert entwickelt werden.

11.8 Die Entscheidungsfindung optimieren

Modularität und Trennung von Concerns ermöglichen eine dezentralisierte Verwaltung und Entscheidungsfindung. Bei hinreichend großen Systemen, egal ob Stadt oder Software-Projekt, kann keine einzelne Person alle Entscheidungen treffen.

Wir wissen alle, dass es am besten ist, die Verantwortung an die am besten qualifizierten Personen zu delegieren. Oft vergessen wir, dass es auch am besten ist, *Entscheidungen bis zum letzten möglichen Moment* aufzuschieben. Dies ist keine Faulheit oder Verantwortungslosigkeit, sondern ermöglicht es uns, Entscheidungen mit dem bestmöglichen erreichbaren Informationsstand zu treffen. Eine voreilige Entscheidung wird immer mit einem suboptimalen Wissensstand getroffen. Wir verfügen über sehr viel weniger Kundenfeedback, haben weniger über das Projekt nachgedacht und wissen weniger über unsere Implementierungsoptionen, wenn wir zu schnell entscheiden.

Die Agilität, die ein POJO-System mit modularisierten Concerns bietet, ermöglicht es uns, optimale, Just-in-Time-Entscheidungen zu treffen, die auf dem aktuellen Wissensstand basieren. Die Komplexität dieser Entscheidungen wird ebenfalls reduziert.

11.9 Standards weise anwenden, wenn sie nachweisbar einen Mehrwert bieten

Dem Bau von Gebäuden zuzuschauen, ist ein bewundernswerter Anblick, weil die Geschwindigkeit, mit der heute neue Gebäude (selbst mitten im Winter) errichtet werden, außerordentlich hoch ist und mit der heutigen Technologie ungewöhnliche Designs möglich sind. Die Bauindustrie ist eine reife Branche mit hochgradig optimierten Bauteilen, Methoden und Standards, die sich unter ständigem Druck über Jahrhunderte hinweg entwickelt haben.

Viele Teams verwendeten die EJB2-Architektur, weil sie ein Standard war, selbst wenn leichgewichtigere und geradlinigere Designs ausgereicht hätten. Ich habe erlebt, wie Teams von verschiedenen *stark beworbenen* Standards besessen wurden und die Implementierung von Werten für ihre Kunden aus den Augen verloren.

Standards machen es leichter, Ideen und Komponenten wiederzuverwenden, Entwickler mit einschlägiger Erfahrung zu rekrutieren, gute Ideen einzukapseln und Komponenten zu verknüpfen. Doch der Prozess der Erstellung von Standards kann manchmal zu lange dauern, als dass die Branche warten

könnte; und einige Standards verlieren den Kontakt mit den wirklichen Bedürfnissen der Zielgruppe.

11.10 Systeme brauchen domänenspezifische Sprachen

Die Bauindustrie hat, wie die meisten Branchen, eine eigene Fachsprache mit einem reichhaltigen Vokabular, Idiomen und Patterns entwickelt, die wesentliche Informationen klar und präzise vermittelt. In der Software-Branche war die Arbeit von [Alexander] besonders einflussreich. Bei der Software-Entwicklung wurde in jüngerer Zeit das Interesse an der Erstellung von *Domain-Specific Languages* (DSL; domänenspezifische Sprache) wiederbelebt. (Siehe zum Beispiel [DSL]. [JMock] ist ein gutes Beispiel für ein Java-API, das eine DSL erstellt.) DSLs sind separate, kleine Skriptsprachen oder APIs in Standardsprachen, in denen man Code so schreiben kann, dass er sich wie ein strukturierter Prosatext eines Domänenexperten liest.

Eine gute DSL minimiert die »Kommunikationslücke« zwischen einem Domänenkonzept und dem Code, der es implementiert, ähnlich wie agile Praktiken die Kommunikation in einem Team und mit den Stakeholdern des Projekts optimieren. Wenn Sie Bereichslogik in derselben Sprache implementieren, die ein Domänenexperte verwendet, ist die Gefahr geringer, dass Sie die Logik falsch implementieren.

Eine effiziente Anwendung von DSLs hebt das Abstraktionsniveau über Code-Idiome und Design-Patterns hinaus. Sie ermöglicht es dem Entwickler, den Zweck des Codes auf der geeigneten Abstraktionsebene auszudrücken.

Domain-Specific Languages ermöglichen es, alle Abstraktionsebenen und alle Domänen der Anwendung als POJOs auszudrücken, und zwar von der abstrakt formulierten Policy bis hin zu den konkreten Details.

11.11 Zusammenfassung

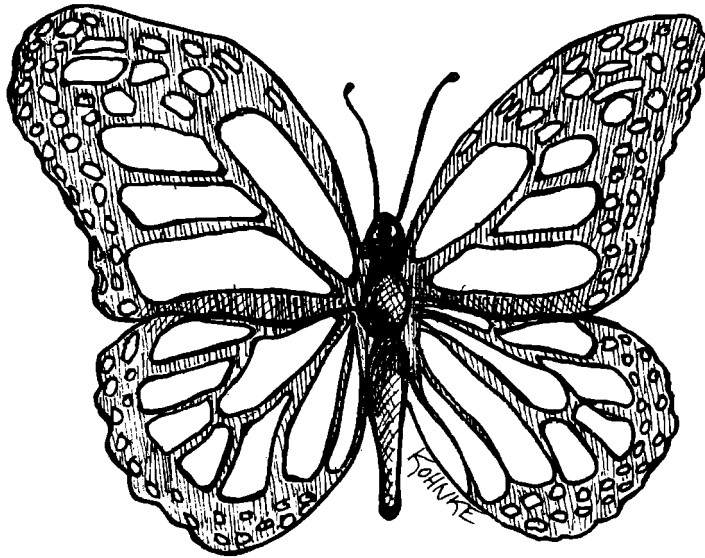
Auch Systeme müssen sauber sein. Eine invasive Architektur überwältigt die Bereichslogik und verschlechtert die Agilität. Wenn die Bereichslogik verdunkelt wird, leidet die Qualität, weil Bugs schwerer zu lokalisieren und Stories schwerer zu implementieren sind. Wenn die Agilität leidet, nimmt die Produktivität ab und die Vorteile der TDD gehen verloren.

Der Zweck sollte auf allen Abstraktionsebenen klar sein. Dies können Sie nur erreichen, wenn Sie POJOs schreiben und andere Implementierungs-Concerns nicht-invasiv mit aspektähnlichen Mechanismen realisieren.

Egal, ob Sie Systeme oder einzelne Module designen, dürfen Sie niemals vergessen: *Verwenden Sie die einfachste funktionsfähige Lösung!*

Emergenz

von Jeff Langr



Emergenz (lat. emergere: auftauchen, hervorkommen, sich zeigen) ist die spontane Herausbildung von Phänomenen oder Strukturen auf der Makroebene eines Systems auf der Grundlage des Zusammenspiels seiner Elemente. Dabei lassen sich die emergenten Eigenschaften des Systems nicht offensichtlich auf Eigenschaften der Elemente zurückführen, die diese isoliert aufweisen.

(Wikipedia: <http://de.wikipedia.org/wiki/Emergenz>)

12.1 Saubere Software durch emergentes Design

Was wäre, wenn es vier einfache Regeln gäbe, deren Befolgung Ihnen helfen könnte, brauchbare Designs zu erstellen? Was wäre, wenn Sie durch Befolgung dieser Regeln Einsichten in die Struktur und das Design Ihres Codes bekämen, die es Ihnen erleichtern würden, Prinzipien wie etwa SRP und DIP anzuwenden?

Was wäre, wenn diese vier Regeln die *Emergenz* eines guten Designs unterstützten?

Viele Entwickler sind der Auffassung, dass die vier Regeln aus *Simple Design* von Kent Beck [XPE] eine wesentliche Hilfe bei der Erstellung eines brauchbaren Software-Designs sind.

Laut Kent ist ein Design »einfach«, wenn es folgende Bedingungen (A.d.Ü.: engl. unglücklich *rules*, »Regeln«) erfüllt:

- Es besteht alle Tests.
- Es enthält keine Duplizierungen.
- Es verkörpert die Absicht der Programmierer.
- Es minimiert die Anzahl der Klassen und Methoden.

Die Regeln sind nach Wichtigkeit geordnet. Die wichtigste steht am Anfang.

12.2 Einfache Design-Regel 1: Alle Tests bestehen

Vor allem muss ein Design ein System erzeugen, das den gewollten Zweck erfüllt. Ein System kann auf dem Papier ein perfektes Design haben, aber wenn nicht auf einfache Weise geprüft werden kann, ob das System tatsächlich zweckerfüllend funktioniert, ist der gesamte Papieraufwand fraglich.

Ein System, das umfassend getestet ist und jederzeit alle Tests besteht, ist ein testbares System. Diese Aussage ist offensichtlich, aber wichtig. Systeme, die nicht testbar sind, können nicht verifiziert werden.

Man könnte fordern, dass ein System, das nicht verifiziert werden kann, niemals produktiv eingesetzt werden sollte.

Glücklicherweise drängt uns das Bemühen, unsere Systeme testbar zu machen, zu einem Design, bei dem unsere Klassen klein und einem einzigen Zweck dienen. Es ist eben einfacher, Klassen zu testen, die dem SRP folgen. Je mehr Tests wir schreiben, desto stärker bemühen wir uns, Code zu schreiben, der einfacher zu testen ist. Deshalb unterstützt unser Bemühen, komplett testbare Systeme zu erstellen, die Entwicklung eines besseren Designs.

Eine starke Kopplung erschwert das Schreiben von Tests. Hier ist der Gedankengang ähnlich: Je mehr Tests wir schreiben, desto mehr bemühen wir uns, die Kopplung zu minimieren, und wenden folglich desto häufiger Prinzipien wie DIP und Techniken wie Dependency Injection, Interfaces und Abstraktionen an.

Unsere Designs werden noch besser.

Bemerkenswerterweise führt das Befolgen einer einfachen und offensichtlichen Regel, die besagt, dass wir Tests schreiben und unsere Systeme laufend damit testen sollen, dazu, dass unser System quasi automatisch bestimmte Hauptziele des objektorientierten Designs, schwache Kopplung und starke Kohäsion, erfüllt.

Tests zu schreiben, führt zu besseren Designs.

12.3 Einfache Design-Regeln 2–4: Refactoring

Die Tests geben uns die Möglichkeit, unseren Code und unsere Klassen sauber zu halten. Wir erreichen dies, indem wir den Code schrittweise refaktorisieren. Wir fügen einige Codezeilen hinzu, halten dann inne und überdenken das neue Design. Haben wir es gerade verschlechtert? Wenn ja, säubern wir es und führen unsere Tests aus, um uns zu vergewissern, dass wir nichts beschädigt haben.

Die Tatsache, dass wir diese Tests haben, nimmt uns die Furcht, wir könnten den Code durch die Bereinigung beschädigen!

Bei der Refaktorisierung können wir unser gesamtes Wissen über gutes Software-Design einsetzen. Wir können die Kohäsion verstärken, die Kopplung verringern, Zuständigkeiten trennen, Systemaufgaben in Module auslagern, unsere Funktionen und Klassen verkleinern, bessere Namen verwenden usw. An dieser Stelle wenden wir auch die anderen drei Regeln des einfachen Designs an: Duplizierten Code eliminieren, die Ausdruckskraft des Codes verbessern und die Anzahl der Klassen und Methoden minimieren.

12.4 Keine Duplizierung

Die Duplizierung ist der Hauptfeind eines guten System-Designs. Sie bedeutet zusätzliche Arbeit, zusätzliche Risiken und zusätzliche unnötige Komplexität.

Duplizierung manifestiert sich in viele Formen. Natürlich sind identische Codezeilen eine Form der Duplizierung. Ähnlich aussehende Codezeilen können oft so umformuliert werden, dass sie sich noch ähnlicher werden und ein Refactoring leichter durchzuführen ist. Duplizierung kann auch in anderen Formen wie etwa der Duplizierung der Implementierung auftreten. Beispielsweise könnte eine Collection-Klasse zwei Methoden enthalten:

```
int size() {}  
boolean isEmpty() {}
```

Wir hätten beide Methoden separat implementieren können. Die `isEmpty`-Methode könnte einen booleschen Wert und `size` einen Zähler überwachen. Wir könnten diese Duplizierung eliminieren, indem wir `isEmpty` mit der Definition von `size` verbinden:

```
boolean isEmpty() {  
    return 0 == size();  
}
```

Die Erstellung eines sauberen Systems erfordert den Willen, Duplizierung zu eliminieren, selbst wenn nur wenige Codezeilen betroffen sind. Betrachten Sie beispielsweise den folgenden Code:

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

    RenderedOp newImage = ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
}

public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(
        image, degrees);
    image.dispose();
    System.gc();
    image = newImage;
}
```

Um dieses System sauber zu halten, sollten wir die geringe Duplizierung in den Methoden `scaleToOneDimension` und `rotate` eliminieren:

```
public void scaleToOneDimension(
    float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
        return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

    replaceImage(ImageUtilities.getScaledImage(
        image, scalingFactor, scalingFactor));
}

public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
}

private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
}
```

Wenn wir Gemeinsamkeiten auf dieser sehr winzigen Ebene extrahieren, beginnen wir, Verstöße gegen das SRP zu erkennen. Deshalb könnten wir eine neu extrahierte Methode in eine andere Klasse einfügen. Dadurch wird die Sichtbarkeit der Methode verbessert. Ein anderes Teammitglied könnte eine Gelegenheit erkennen, die neue

Methode weiter zu abstrahieren und in einem anderen Kontext wiederzuverwenden. Diese »Wiederverwendung im Kleinen« kann zu einer erheblichen Verringerung der Systemkomplexität führen. Wiederverwendung im Kleinen zu verstehen, ist ein wesentlicher Aspekt, um Wiederverwendung im Großen zu erzielen.

Das *Template Method*-Pattern [GOF] ist eine gebräuchliche Technik, um Duplizierung auf höheren Ebenen zu eliminieren. Ein Beispiel:

```
public class VacationPolicy {
    public void accrueUSDivisionVacation() {
        // Code zur Berechnung der Urlaubstage aufgrund der Arbeitsstunden
        // ...
        // Code zur Garantie des US-Mindesturlaubs
        // ...
        // Code für die Berechnung des Urlaubslohns
        // ...
    }

    public void accrueEUDivisionVacation() {
        // Code zur Berechnung der Urlaubstage aufgrund der Arbeitsstunden
        // ...
        // Code zur Garantie des EU-Mindesturlaubs
        // ...
        // Code für die Berechnung des Urlaubslohns
        // ...
    }
}
```

Der Code in `accrueUSDivisionVacation` und `accrueEUDivisionVacation` ist zu großen Teilen gleich; die Ausnahme ist die Berechnung des gesetzlichen Mindesturlaubs. Dieser Teil des Algorithmus hängt vom Mitarbeitertyp ab.

Mit dem *Template Method*-Pattern können wir die offensichtliche Duplizierung eliminieren:

```
abstract public class VacationPolicy {
    public void accrueVacation() {
        calculateBaseVacationHours();
        alterForLegalMinimums();
        applyToPayroll();
    }

    private void calculateBaseVacationHours() { /* ... */ };
    abstract protected void alterForLegalMinimums();
    private void applyToPayroll() { /* ... */ };
}

public class USVacationPolicy extends VacationPolicy {
    @Override protected void alterForLegalMinimums() {
        // US-spezifische Logik
    }
}
```



```
    }  
  }  
  
  public class EUVacationPolicy extends VacationPolicy {  
    @Override protected void alterForLegalMinimums() {  
      // EU-spezifische Logik  
    }  
  }  
}
```

Die Unterklassen füllen das »Loch« in dem `accrueVacation`-Algorithmus aus und liefern die einzigen Informationen, die nicht dupliziert wurden.

12.5 Ausdrucksstärke

Die meisten Entwickler habenschon mit verwickeltem Code gearbeitet. Viele haben auch selbst verwickelten Code produziert. Es ist leicht, Code zu schreiben, den man *selbst* versteht, weil man zu dem Zeitpunkt, an dem man ihn schreibt, tief in das betreffende Problem eingetaucht ist. Andere Entwickler, die den Code warten müssen, verstehen ihn nicht so gründlich.

Die Hauptkosten eines Software-Projekts werden durch die langfristige Wartung verursacht. Um potenzielle Defekte bei Änderungen zu vermeiden, müssen wir verstehen können, was ein System tut. Da Systeme immer komplexer werden, brauchen Entwickler immer mehr Zeit, um die Systeme zu verstehen, und die Gefahren von Missverständnissen werden immer größer. Deshalb sollte Code die Absichten seines Autors möglichst deutlich ausdrücken. Je klarer der Autor seinen Code schreiben kann, desto weniger Zeit müssen andere aufwenden, um seinen Zweck zu verstehen. Dadurch werden Defekte und Wartungskosten reduziert.

Sie können Ihre Zwecke durch geeignete Namen ausdrücken. Wenn wir den Namen einer Klasse oder Funktion lesen, wollen wir keine Überraschung erleben, wenn wir seine Verantwortlichkeiten entdecken.

Sie können Ihren Code auch ausdrucksstärker machen, indem Sie Ihre Funktionen und Klassen klein halten. Kleine Klassen und Funktionen sind normalerweise leichter zu benennen, zu schreiben und zu verstehen.

Sie können Ihren Code auch ausdrucksstärker machen, indem Sie sich an die Standardnomenklatur halten. So dreht es sich etwa bei Design-Patterns hauptsächlich um Kommunikation und Ausdruckskraft. Mit standardmäßigen Pattern-Namen, wie etwa *Command* oder *Visitor*, in den Namen der Klassen, die diese Patterns implementieren, können Sie anderen Entwicklern Ihr Design knapp und treffend mitteilen.

Gut geschriebene Unit-Tests sind ebenfalls ausdrucksstark. Tests dienen hauptsächlich auch als Beispiele für Dokumentationszwecke. Wer unsere Tests liest, sollte schnell verstehen können, worum es in einer Klasse geht.

Aber die wichtigste Methode, ausdrucksstarken Code zu schreiben, besteht darin, es *auszuprobieren*. Allzu oft bringen wir unseren Code zum Laufen und wenden uns dann dem nächsten Problem zu, ohne genügend zu fragen, ob der Code für den nächsten Leser aussagestark genug ist. Vergessen Sie nicht: Wahrscheinlich sind Sie der Nächste, der den Code lesen wird.

Seien Sie deshalb ein wenig stolz auf Ihr Können. Verbringen Sie ein wenig Zeit mit Ihren Funktionen und Klassen. Wählen Sie bessere Namen, zerlegen Sie große Funktionen in kleinere und kümmern Sie sich ganz allgemein um Ihre Produkte. Sorgfalt ist eine wertvolle Ressource.

12.6 Minimale Klassen und Methoden

Selbst Konzepte, die so grundlegend sind wie die Eliminierung der Duplizierung, die Ausdrucksstärke des Codes und das SRP, können übertrieben werden. Bei unserem Bemühen, unsere Klassen und Methoden zu verkleinern, erstellen wir möglicherweise zu viele winzige Klassen und Methoden. Deshalb schreibt diese Regel vor, dass wir auch die Anzahl unserer Funktionen und Klassen niedrig halten sollen.

Eine hohe Anzahl von Klassen und Methoden ist manchmal die Folge eines sinnlosen Dogmatismus. Betrachten Sie beispielsweise einen Codierstandard, der darauf besteht, dass man für jede Klasse ein Interface erstellen soll. Oder betrachten Sie Entwickler, die darauf bestehen, dass Felder und Verhaltensweisen immer in Daten-Klassen und Verhaltens-Klassen getrennt werden müssen. Auch Sie sollten sich solchen Dogmen widersetzen und einen pragmatischeren Ansatz verfolgen.

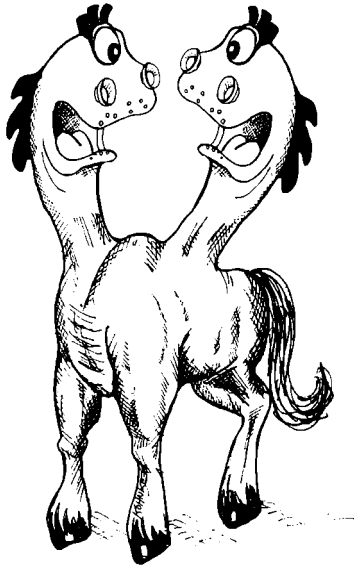
Unser Ziel besteht darin, das Gesamtsystem klein und zugleich die Anzahl unserer Funktionen und Klassen klein zu halten. Denken Sie jedoch, dass diese Regeln die niedrigste Priorität unter den Regeln des einfachen Designs haben. Deshalb ist es zwar wichtig, die Anzahl der Klassen und Funktionen klein zu halten, wichtiger ist jedoch, Tests zu schreiben, Duplizierung zu eliminieren und ausdrucksstarken Code zu schreiben.

12.7 Zusammenfassung

Gibt es einen Satz von einfachen Techniken, die die Erfahrung ersetzen können? Sicher nicht. Andererseits sind die Techniken (Regeln), die in diesem Kapitel und in diesem Buch beschrieben werden, die Quintessenz vieler Jahrzehnte Programmiererfahrung der Autoren. Die Prinzipien des einfachen Designs zu befolgen, hilft Entwicklern, bewährte Prinzipien und Patterns anzuwenden, für deren Aneignung sie sonst Jahre benötigen würden.

Nebenläufigkeit

von Brett L. Schuchert



»Objekte sind Abstraktionen der Verarbeitung. Threads sind Abstraktionen von Zeitabläufen.«

– James O. Coplien (in einer privaten Mitteilung)

Saubere nebenläufige Programme zu schreiben, ist schwer – sehr schwer. Es ist viel leichter, Code zu schreiben, der in einem einzigen Thread ausgeführt wird. Es ist ebenfalls leicht, Multithreaded-Code zu schreiben, der an der Oberfläche gut aussieht, aber in einer tieferen Ebene Defekte hat. Solcher Code funktioniert so lange, bis das System unter Stress gerät.

In diesem Kapitel beschreiben wir, warum die nebenläufige Programmierung erforderlich ist und welche Schwierigkeiten mit ihr verbunden sind. Dann geben wir mehrere Empfehlungen für den Umgang mit diesen Schwierigkeiten und das Schreiben von sauberem nebenläufigem Code. Wir schließen mit Problemen, die beim Testen von nebenläufigem Code auftreten können.

Saubere Nebenläufigkeit ist ein komplexes Thema, das ein eigenes Buch verdient. In diesem Buch können wir nur einen Überblick über dieses Thema geben. Es wird in diesem Kapitel eingeführt und in Anhang A, *Nebenläufigkeit II*, vertieft. Wenn Sie zunächst nur wissen wollen, worum es bei Nebenläufigkeit geht, reicht dieses Kapitel für Sie im Moment aus. Wollen Sie tiefer in das Thema einsteigen, sollten Sie auch den Anhang lesen.

13.1 Warum Nebenläufigkeit?

Nebenläufigkeit ist eine Entkopplungsstrategie. Sie hilft das, was getan wird, davon zu entkoppeln, wann es getan wird. Bei single-threaded Anwendungen sind das Was und das Wann so stark gekoppelt, dass der Status der gesamten Anwendung oft durch einen Blick auf den Stack-Backtrace bestimmt werden kann. Ein Programmierer, der ein solches System debuggt, kann einen Breakpoint oder eine Folge von Breakpoints setzen und *kennt* den Zustand des Systems, das er mithilfe der Breakpoints untersucht.

Das Was von dem Wann zu entkoppeln, kann sowohl den Durchsatz als auch die Struktur einer Anwendung erheblich verbessern. Die Struktur einer solchen entkoppelten Anwendung ähnelt eher einer Menge kleiner zusammenarbeitender Computer als einer großen Hauptschleife. Dadurch kann das System transparenter gemacht werden; und die Zuständigkeiten können besser verteilt werden.

Betrachten Sie beispielsweise das Standard-»Servlet«-Modell von Webanwendungen. Diese Systeme laufen unter dem Schirm eines Web- oder EJB-Containers, der *einen Teil* der Nebenläufigkeit für Sie verwaltet. Die Servlets werden asynchron ausgeführt, wenn Webanfragen eingehen. Der Servlet-Programmierer muss nicht alle eingehenden Anfragen verwalten. *Im Prinzip* lebt jede Servlet-Ausführung in ihrer eigenen kleinen Welt und ist von allen anderen Servlet-Ausführungen entkoppelt.

Natürlich wäre dieses Kapitel nicht erforderlich, wenn es so leicht wäre. Tatsächlich ist die Entkopplung, die der Web-Container leistet, bei Weitem nicht perfekt. Servlet-Programmierer müssen sehr sorgfältig arbeiten, damit ihre nebenläufigen Programme korrekt sind. Dennoch bietet das Servlet-Modell erhebliche strukturelle Vorteile.

Aber die Strukturverbesserung ist nicht das einzige Motiv für die Einführung der Nebenläufigkeit. Einige Systeme haben Auflagen für ihr Antwortverhalten und ihren Durchsatz, die häufig manuell codierte nebenläufige Lösungen erfordern.

Betrachten Sie beispielsweise einen single-threaded Informationsaggregator, der Informationen von vielen verschiedenen Websites sammelt und daraus einer tägliche Zusammenfassung erstellt. Weil das System single-threaded ist, werden alle Websites nacheinander abgefragt. Eine Website wird erst fertig bearbeitet, bevor die nächste in Angriff genommen wird. Der tägliche Lauf muss in weniger als 24 Stunden ausgeführt werden. Doch je mehr Websites hinzugefügt werden, desto mehr

Zeit wird benötigt, bis mehr als 24 Stunden benötigt werden, um alle Daten zu sammeln. Die Single-Thread-Lösung führt zu langen Wartezeiten an den Websockets, während die I/O abgewickelt wird. Wir könnten das Leistungsverhalten mit einem multithreaded Algorithmus verbessern, der mehr als eine Website gleichzeitig besucht.

Oder betrachten Sie ein System, das immer nur einen Benutzer auf einmal abfertigt und nur eine Sekunde Zeit pro Benutzer benötigt. Bei wenigen Benutzern reagiert dieses System recht flott; aber wenn die Anzahl der Benutzer zunimmt, werden die Antwortzeiten des Systems länger. Kein Benutzer möchte in einer Schlange hinter 150 anderen warten! Wir könnten die Antwortzeit dieses Systems verbessern, indem wir viele Benutzer nebenläufig (parallel, gleichzeitig) abfertigen.

Oder betrachten Sie ein System, das große Datenmengen verarbeitet, aber erst dann eine komplette Lösung liefert, nachdem es alle Daten verarbeitet hat. Vielleicht könnten Teilmengen der Daten auf verschiedenen Computern parallel verarbeitet werden.

Mythen und falsche Vorstellungen

Die Beispiele sollten gezeigt haben, dass es überzeugende Gründe dafür gibt, nebenläufig zu programmieren. Doch wie bereits erwähnt: Nebenläufigkeit ist *schwer*. Wenn Sie nicht sehr sorgfältig arbeiten, können Sie einige sehr hässliche Situationen produzieren. Betrachten Sie die folgenden verbreiteten Mythen und Fehlvorstellungen:

- *Nebenläufigkeit verbessert immer das Leistungsverhalten.* Nebenläufigkeit kann das Leistungsverhalten manchmal verbessern, aber nur wenn es zahlreiche Wartezeiten gibt, die von mehreren Threads oder Prozessoren genutzt werden können. Keine Situation ist trivial.
- *Das Design ändert sich nicht, wenn man nebenläufige Programme schreibt.* Tatsächlich kann sich das Design eines nebenläufigen Algorithmus stark von dem eines single-threaded Systems unterscheiden. Die Entkopplung des Was vom Wann hat normalerweise einen riesigen Einfluss auf die Struktur des Systems.
- *Nebenläufigkeitsprobleme zu verstehen, ist nicht wichtig, wenn man mit einem Container wie etwa einem Web- oder EJB-Container arbeitet.* Tatsächlich sollten Sie genau wissen, was Ihr Container macht, um Probleme mit nebenläufigen Updates und Deadlocks, die später in diesem Kapitel beschrieben werden, möglichst zu vermeiden.

Hier sind einige Tipps für das Schreiben nebenläufiger Software:

- *Nebenläufigkeit verlangt einen gewissen Verwaltungsaufwand*, der das Leistungsverhalten etwas verschlechtert und zusätzlichen Code erfordert.
- *Korrekte Nebenläufigkeit ist komplex*, selbst bei einfachen Problemen.

- *Nebenläufigkeit-Bugs sind normalerweise nicht reproduzierbar*; deshalb werden sie oft als einmalige Erscheinungen (kosmische Strahlung, Glitches usw.) abgeschrieben und nicht, wie es erforderlich wäre, als echte Defekte behandelt.
- *Nebenläufigkeit erfordert oft eine grundlegende Änderung der Design-Strategie.*

13.2 Herausforderungen

Was macht die nebenläufige Programmierung so schwierig? Betrachten Sie die folgende triviale Klasse:

```
public class X {  
    private int lastIdUsed;  
    public int getNextId() {  
        return ++lastIdUsed;  
    }  
}
```

Angenommen, wir erstellen eine Instanz von `X`, setzen das `lastIdUsed`-Feld auf 42 und nutzen dann die Instanz in zwei Threads, die beide die Methode `getNextId()` aufrufen. Dann gibt es drei mögliche Ergebnisse:

- Thread eins erhält den Wert 43, Thread zwei erhält den Wert 44, `lastIdUsed` ist 44.
- Thread eins erhält den Wert 44, Thread zwei erhält den Wert 43, `lastIdUsed` ist 44.
- Thread eins erhält den Wert 43, Thread zwei erhält den Wert 43, `lastIdUsed` ist 43.

Das überraschende dritte Ergebnis (siehe den Abschnitt *Tiefer graben* in Anhang A.2) tritt ein, wenn die beiden Threads in Konflikt geraten. Dies passiert, weil es viele mögliche Pfade gibt, auf denen die beiden Threads diese Zeile des Java-Codes durchlaufen können und einige dieser Pfade falsche Ergebnisse erzeugen. Wie viele verschiedene Pfade gibt es? Um diese Frage wirklich zu beantworten, müssen Sie verstehen, was der Just-in-Time-Compiler mit dem generierten Byte-Code anstellt und was das Memory-Model von Java als *atomar* betrachtet.

Eine schnelle Antwort: Wenn wir nur den generierten Byte-Code betrachten, gibt es 12.870 verschiedene mögliche Ausführungspfade, wie diese beiden Threads innerhalb der `getNextId`-Methode ausgeführt werden können. (Eine detailliertere Analyse finden Sie in Anhang A.2 im Abschnitt *Mögliche Ausführungspfade*.) Wenn der Typ von `lastIdUsed` von `int` in `long` geändert wird, wächst die Anzahl der möglichen Pfade auf 2.704.156. Natürlich erzeugen die meisten dieser Pfade gültige Ergebnisse. Das Problem liegt darin, dass *einige falsche Ergebnisse generieren*.

13.3 Prinzipien einer defensiven Nebenläufigkeitsprogrammierung

Die folgenden Abschnitte enthalten mehrere Prinzipien und Techniken, mit denen Sie Ihre Systeme gegen Probleme mit nebenläufigem Code schützen können.

Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip (SRP; Prinzip der einzigen Verantwortlichkeit; [PPP]) besagt, dass eine Methode/Klasse/Komponente nur einen einzigen Grund haben sollte, sich zu ändern. Das Design von nebenläufigen Programmen ist komplex genug. Änderungen durch nebenläufigen Code sollten nicht mit Änderungen durch den restlichen Code vermengt werden. Leider sind die Details einer Nebenläufigkeitsimplementierung allzu oft direkt in anderen Produktionscode eingebettet. Hier sind einige Punkte, die Sie beachten sollten:

- *Nebenläufiger Code hat einen eigenen Lebenszyklus mit Entwicklung, Änderung und Feinschliff.*
- *Nebenläufiger Code ist mit besonderen Problemen verbunden, die eine andere Form und oft auch einen höheren Schwierigkeitsgrad haben als nicht nebenläufiger Code.*
- Die Anzahl der Fehlermöglichkeiten bei schlecht geschriebenem nebenläufigen Code macht die Programmierung auch ohne die zusätzliche Last des anderen Anwendungscodes schwer genug.

Empfehlung: *Trennen Sie den nebenläufigen Code von dem anderen Code (siehe das Client/Server-Beispiel in Anhang A.1).*

Korollar: Beschränken Sie den Gültigkeitsbereich von Daten

Wie wir gesehen haben, können zwei Threads, die dasselbe Feld eines gemeinsam genutzten Objekts modifizieren, in Konflikt geraten und ein unerwartetes Verhalten auslösen. Eine Lösung besteht darin, einen *kritischen Abschnitt* in dem Code, der auf das gemeinsam genutzte Objekt zugreift, mit dem Schlüsselwort `synchronized` zu schützen.

Es ist wichtig, die Anzahl solcher kritischen Abschnitte einzuschränken. Je mehr Stellen gemeinsam genutzter Daten aktualisiert werden können, desto wahrscheinlicher werden folgende Ereignisse:

- Sie vergessen, eine oder mehrere dieser Stellen zu schützen, wodurch praktisch der gesamte Code defekt wird, der die gemeinsam genutzten Daten modifiziert.
- Der erforderliche Aufwand für den wirksamen Schutz aller Stellen wird praktisch verdoppelt (ein Verstoß gegen das DRYP, Don't-repeat-Yourself-Prinzip; [PRAG]).

- Es wird noch schwerer werden, die Quelle von Fehlern zu finden, die bereits schwer genug zu finden sind.

Empfehlung: *Nehmen Sie sich die Datenkapselung zu Herzen; schränken Sie den Zugriff auf alle gemeinsam genutzten Daten nachhaltig ein.*

Korollar: Arbeiten Sie mit Kopien der Daten

Eine gute Methode, die gemeinsame Nutzung von Daten zu vermeiden, besteht darin, sie ganz zu umgehen. In einigen Situationen können Sie Objekte kopieren und als read-only behandeln. In anderen Fällen könnte es möglich sein, Objekte zu kopieren, Ergebnisse aus mehreren Threads in diese Kopien zu sammeln und dann die Ergebnisse in einem einzigen Thread zusammenzuführen.

Falls sich eine leichte Möglichkeit anbietet, sollten Sie die gemeinsame Nutzung von Objekten vermeiden. Der Code wird wahrscheinlich sehr viel weniger Probleme verursachen. Vielleicht machen Sie sich Gedanken über die Kosten, um all die zusätzlichen Objekte zu erstellen. Es lohnt sich, durch Experimente herauszufinden, ob dies wirklich ein Problem ist. Doch wenn es möglich ist, mit Kopien von Objekten eine Synchronisierung zu vermeiden, heben die Einsparungen des Aufwands für intrinsische Locks wahrscheinlich die Kosten für die zusätzliche Erstellung der Objekte und die Garbage Collection wieder auf.

Korollar: Threads sollten voneinander so unabhängig wie möglich sein

Wenn Sie nebenläufigen Code schreiben, sollten Sie jeden Thread möglichst so schreiben, als existierte er in seiner eigenen Welt, ohne Daten mit einem anderen Thread zu teilen. Jeder Thread verarbeitet eine Client-Anfrage, wobei alle erforderlichen Daten aus einer nicht mit anderen Threads geteilten Quelle stammen und als lokale Variablen gespeichert werden. Dadurch verhalten sich diese Threads so, als wären sie die einzigen Threads auf der Welt und als gäbe es keine Synchronisationsanforderungen.

Beispielsweise erhalten alle von `HttpServlet` abgeleiteten Klassen alle Informationen als Parameter, die in den `doGet`- und `doPost`-Methoden übergeben werden. Dadurch verhält sich jedes Servlet so, als hätte es eine eigene Maschine. Solange der Code in dem Servlet nur lokale Variablen verwendet, besteht keine Gefahr, dass das Servlet Synchronisationsprobleme verursacht. Natürlich verwenden die meisten mit Servlets arbeitenden Anwendungen irgendwann gemeinsam genutzte Ressourcen wie etwa Datenbankverbindungen.

Empfehlung: *Versuchen Sie, Daten in unabhängige Teilmengen zu zerlegen, die von unabhängigen Threads, möglicherweise in verschiedenen Prozessoren, verarbeitet werden können.*

13.4 Lernen Sie Ihre Library kennen

Java 5 enthält gegenüber früheren Versionen viele Verbesserungen für die Entwicklung von nebenläufigem Code. Sie müssen mehrere Dinge beachten, wenn Sie nebenläufigen Code in Java 5 schreiben:

- Verwenden Sie die zur Verfügung gestellten thread-sicheren Collections.
- Verwenden Sie das Executor Framework zur Ausführung unzusammenhängender Tasks.
- Verwenden Sie nicht blockierende Lösungen, falls möglich.
- Mehrere Library-Klassen sind nicht thread-sicher.

Thread-sichere Collections

In der Anfangszeit von Java schrieb Doug Lea das bahnbrechende Buch *Concurrent Programming in Java* [Lea99]. Zusätzlich zu dem Buch entwickelte er mehrere thread-sichere Collections, die später in das `java.util.concurrent`-Package des JDK übernommen wurden. Die Collections in diesem Package können gefahrlos in multithreaded Situationen verwendet werden und zeigen ein gutes Leistungsverhalten. Tatsächlich arbeitet die `ConcurrentHashMap`-Implementierung in fast allen Situationen besser als die `HashMap`. Sie ermöglicht auch gleichzeitige nebenläufige Lese- und Schreibvorgänge und verfügt über Methoden, die gebräuchliche zusammengesetzte Operationen unterstützen, die sonst nicht thread-sicher sind. Wenn die Deployment-Umgebung Java 5 verwendet, sollten Sie mit `ConcurrentHashMap` beginnen.

Mehrere andere Klassen unterstützen ebenfalls nebenläufigen Code. Einige Beispiele:

ReentrantLock	Ein Lock, das in einer Methode gesetzt und in einer anderen aufgehoben werden kann.
Semaphore	Eine Implementierung der klassischen Semaphore, ein Lock mit einem Zähler.
CountDownLatch	Ein Lock, das eine Anzahl von Ereignissen abwartet, bevor es alle auf es wartenden Threads freigibt. Dadurch haben alle Threads eine faire Chance, etwa zur gleichen Zeit zu starten.

Empfehlung: Studieren Sie die Klassen, die Ihnen zur Verfügung stehen. Bei Java sollten Sie sich mit `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks` vertraut machen.

13.5 Lernen Sie Ihre Ausführungsmodelle kennen

Es gibt mehrere Methoden, Verhalten in einer nebenläufigen Anwendung aufzuteilen. Zunächst müssen Sie einige grundlegende Definitionen kennen:

Bound Resources (gebundene Ressourcen)	Ressourcen fester Größe oder Anzahl, die in einer nebenläufigen Umgebung verwendet werden. Beispiele sind Datenbankverbindungen und Lese/Schreib-Puffer fester Größe.
Mutual-Exclusion (gegenseitiger Ausschluss)	Nur ein Thread kann gleichzeitig auf gemeinsam genutzte Daten oder eine gemeinsam genutzte Ressource zugreifen.
Starvation (Verhungern)	Ein Thread oder eine Gruppe von Threads wird außerordentlich lange oder für immer daran gehindert, weiterzuarbeiten. Wenn beispielsweise schnell ablaufende Threads immer zuerst bedient werden, könnte dies dazu führen, dass lange laufende Threads verhungern, wenn die schnell laufenden Threads kein Ende nehmen.
Deadlock	Zwei oder mehr Threads warten gegenseitig auf das Ende des/der jeweils anderen. Jeder Thread hat eine Ressource, die von dem/den anderen benötigt wird, und keiner kann aufhören, bevor er nicht die fehlende Ressource bekommt.
Livelock	Threads versuchen, im Gleichschritt zu laufen, stehen sich dabei aber gegenseitig im Weg. Aufgrund von Resonanz versuchen sie immer weiter, Fortschritte zu machen, kommen aber für außerordentlich lange Zeit oder niemals voran.

Mit diesen Definitionen können wir jetzt die verschiedenen Ausführungsmodelle beschreiben, die in der nebenläufigen Programmierung verwendet werden.

Erzeuger-Verbraucher

Das *Erzeuger-Verbraucher-Problem* (<http://de.wikipedia.org/wiki/Erzeuger-Verbraucher-Problem>; engl. *Producer-Consumer*) zählt zu den klassischen Problemen der Prozess-Synchronisation. Einer oder mehrere Erzeuger-Threads erstellen ein Produkt und fügen es in einen Puffer oder eine Queue ein. Einer oder mehrere Verbraucher-Threads rufen das Produkt aus der Queue ab und ergänzen es. Die Queue zwischen den Erzeugern und Verbrauchern ist eine *gebundene Ressource*. Dies bedeutet, dass die Erzeuger auf einen freien Platz in der Queue warten müssen, bevor sie schreiben können, und die Verbraucher warten müssen, bis die Queue etwas Konsumierbares enthält. Die Koordination zwischen den Erzeugern und Verbrauchern mittels der Queue erfordert den Austausch von Signalen zwischen den Erzeugern und Verbrauchern. Die Erzeuger schreiben in die Queue und signalisieren, dass die Queue nicht mehr leer ist. Die Verbraucher lesen aus der Queue und signalisieren, dass die Queue nicht mehr voll ist. Beide müssen möglicherweise auf ein Signal warten, dass sie fortfahren können.

Leser-Schreiber

Das *Leser-Schreiber-Problem* (engl. *Readers-Writers*) zählt ebenfalls zu den klassischen Problemen der Prozess-Synchronisation. Wenn Sie über eine gemeinsam genutzte Ressource verfügen, die hauptsächlich als Informationsquelle für Leser dient, aber gelegentlich von Schreibern aktualisiert wird, ist der Durchsatz ein Problem. Wird der Durchsatz betont, kann es verhungern und eine Ansammlung überholter Informationen kann eintreten. Lässt man Aktualisierungen zu, kann der Durchsatz beeinträchtigt werden. Leser und Schreiber so zu koordinieren, dass die Leser nicht lesen, während der Schreiber aktualisiert, und umgekehrt, ist ein schwieriger Balanceakt. Schreiber neigen dazu, viele Leser für lange Zeitspannen zu blockieren, und verursachen so Probleme mit dem Durchsatz.

Die Herausforderung liegt darin, die Anforderungen der Leser und der Schreiber so auszugleichen, dass eine korrekte Operation gewährleistet, ein vernünftiger Durchsatz erzielt und Verhungern vermieden wird. Eine einfache Strategie besteht darin, Schreiber warten zu lassen, bis keine Leser da sind, bevor der Schreiber eine Aktualisierung durchführen darf. Doch wenn es Dauerleser gibt, verhungern die Schreiber. Gibt es dagegen Schreiber mit häufigen Aktualisierungen und erhalten sie Priorität, leidet der Durchsatz. Hier besteht das Problem also darin, die geeignete Balance zu finden und Probleme mit den nebenläufigen Updates zu vermeiden.

Philosophenproblem

Das *Philosophenproblem* (<http://de.wikipedia.org/wiki/Philosophenproblem>; engl. *Dining-Philosophers-Problem*) ist ein weiteres klassisches Problem der Prozess-Synchronisation. Stellen Sie sich vor, dass mehrere Philosophen an einem runden Tisch sitzen. Links neben jedem Philosophen liegt eine Gabel. In der Mitte des Tisches steht eine große Schüssel Spaghetti. Die Philosophen denken nach, bis sie hungrig sind. Wenn sie hungrig sind, nehmen sie die Gabeln links und rechts neben sich auf und essen. Ein Philosoph kann nur essen, wenn er zwei Gabeln hat. Hat einer der Philosophen zu seiner Rechten oder Linken bereits eine der benötigten Gabeln in der Hand, muss der hungrige Philosoph warten, bis der Nachbar mit dem Essen fertig ist und seine Gabeln wieder hinlegt. Nachdem ein Philosoph gegessen hat, legt er beide Gabeln wieder auf den Tisch und wartet, bis er wieder hungrig ist.

Wenn Sie Philosophen durch Threads und Gabeln durch Ressourcen ersetzen, beschreibt dieses Problem viele Unternehmensanwendungen, in denen Prozesse um Ressourcen konkurrieren. Bei einem nachlässigen Design können derartig konkurrierende Systeme Deadlocks, Livelocks, Durchsatzabfälle und Effizienzverluste erleiden.

Die meisten Nebenläufigkeitsprobleme, denen Sie begegnen werden, werden wahrscheinlich eine Variante dieser drei Probleme sein. Studieren Sie die einschlägigen

Algorithmen und wenden Sie sie in Ihren eigenen Lösungen an, damit Sie auf diese Nebenläufigkeitsprobleme vorbereitet sind.

Empfehlung: *Studieren Sie die grundlegenden Algorithmen und ihre Anwendung in Lösungen.*

13.6 Achten Sie auf Abhängigkeiten zwischen synchronisierten Methoden

Abhängigkeiten zwischen synchronisierten Methoden in nebenläufigem Code verursachen subtile Bugs. Java enthält das Konstrukt `synchronized`, das eine einzelne Methode schützt. Doch da dieselbe gemeinsam genutzte Klasse mehr als eine `synchronized` Methode enthält, ist Ihr System möglicherweise falsch konzipiert (siehe den Abschnitt *Abhängigkeiten zwischen Methoden können nebenläufigen Code beschädigen* in Anhang A.4).

Empfehlung: *Vermeiden Sie es, mehr als eine Methode auf ein gemeinsam genutztes Objekt anzuwenden.*

Manchmal müssen Sie mehr als eine Methode auf ein gemeinsam genutztes Objekt anwenden. In diesem Fall gibt es drei Möglichkeiten, korrekten Code zu schreiben:

- Clientbasiertes Locking – Der Client sollte den Server sperren, bevor die erste Methode aufgerufen wird, und dafür sorgen, dass das Lock den Code einschließt, der die letzte Methode aufruft.
- Serverbasiertes Locking – Erstellen Sie im Server eine Methode, die den Server sperrt, alle Methoden aufruft und dann die Sperre aufhebt. Lassen Sie den Client die `new`-Methode aufrufen.
- Adapted Server – Erstellen Sie eine intermediäre Komponente, die die Sperre ausführt. Dies ist eine Variante des serverbasierten Lockings, wenn der ursprüngliche Server nicht geändert werden kann.

13.7 Halten Sie synchronisierte Abschnitte klein

Das Schlüsselwort `synchronized` setzt ein Lock (Sperre). Alle Code-Abschnitte, die durch dasselbe Lock geschützt werden, werden garantiert nur von einem Thread gleichzeitig ausgeführt. Locks sind teuer, weil sie Verzögerungen und zusätzlichen Verwaltungsaufwand verursachen. Deshalb sollten Sie in Ihrem Code `synchronized`-Anweisungen nur sparsam verwenden. Andererseits müssen kritische Abschnitte geschützt werden. (Ein kritischer Abschnitt ist ein Code-Abschnitt, der nur dann korrekt ausgeführt werden kann, wenn nicht gleichzeitig mehrere Threads auf ihn zugreifen.)

Deshalb sollte Ihr Code so wenig kritische Abschnitte wie möglich enthalten.

Einige naive Programmierer versuchen, dies zu erreichen, indem sie ihre kritischen Abschnitte sehr groß wählen. Doch wenn die Synchronisation über den minimalen kritischen Abschnitt hinaus vergrößert wird, nehmen die Konflikte zu und die Leistung verschlechtert sich (siehe den Abschnitt *Den Durchsatz verbessern* in Anhang A.5).

Empfehlung: *Halten Sie Ihre synchronisierten Abschnitte so klein wie möglich.*

13.8 Korrekten Shutdown-Code zu schreiben, ist schwer

Ein System zu schreiben, das aktiv bleiben und für immer laufen soll, unterscheidet sich von einem System, das eine Zeit lang arbeitet und dann kontrolliert abgeschaltet wird.

Ein kontrolliertes Abschalten ist oft schwer zu realisieren. Zu den üblichen Problemen zählen Deadlocks (siehe auch Anhang A.6), bei denen Threads auf Signale warten, die nie gesendet werden.

Stellen Sie sich beispielsweise ein System mit einem Parent-Thread vor, der mehrere Child-Threads hervorbringt und dann darauf wartet, dass sie alle beendet werden, bevor er seine Ressourcen freigibt und sich abschaltet. Was passiert, wenn einer der Child-Threads in einen Deadlock gerät? Der Parent-Thread wartet ewig, und das System wird niemals abgeschaltet.

Oder betrachten Sie ein ähnliches System, das *angewiesen* wurde, sich abzuschalten. Der Parent-Thread weist alle seine Child-Threads an, ihre Tasks abzubrechen und sich zu beenden. Aber was passiert, wenn zwei Children als Producer-Consumer-Paar zusammenarbeiten? Nehmen Sie an, der Producer empfangt das Signal von dem Parent und schalte sich schnell ab. Der Consumer könnte auf eine Nachricht von dem Producer gewartet haben und in einem Zustand blockiert sein, in dem er das Shutdown-Signal nicht empfangen kann. Er könnte auf ewig weiter auf den Producer warten und verhindern, dass der Parent auch beendet werden kann.

Situationen wie diese sind durchaus nicht selten. Wenn Sie also nebenläufigen Code schreiben müssen, bei dem es auch um ein kontrolliertes Abschalten geht, müssen Sie damit rechnen, viel Zeit darin zu investieren, den Shutdown korrekt zu programmieren.

Empfehlung: *Sie sollten bereits früh über einen Shutdown nachdenken und ihn möglichst früh zum Laufen bringen. Wenn Sie warten, dauert es immer länger. Studieren Sie vorhandene Algorithmen, weil diese Aufgabe wahrscheinlich schwerer ist, als Sie denken.*

13.9 Threaded-Code testen

Die Korrektheit von Code zu beweisen, ist unpraktisch. Tests können keine Korrektheit garantieren. Doch gute Tests können das Risiko vermindern. Dies gilt vor allem

für Lösungen, die mit einem einzigen Thread arbeiten. Sobald zwei oder mehr Threads denselben Code verwenden und dieselben Daten gemeinsam nutzen, wird alles sehr viel komplexer.

Empfehlung: Schreiben Sie Tests, die Probleme aufdecken können, und führen Sie sie dann mit verschiedenen Programm- und Systemkonfigurationen und Lasten aus. Sollten Tests scheitern, suchen Sie nach der Ursache und beheben Sie sie. Ignorieren Sie ein Scheitern nicht nur deshalb, weil die Tests bei einer nachfolgenden Ausführung bestanden werden.

Es gibt zahlreiche Faktoren, die Sie berücksichtigen müssen. Hier sind einige stärker aufgegliederte Empfehlungen:

- Behandeln Sie gelegentliche Fehler als potenzielle Threading-Probleme.
- Bringen Sie zunächst Ihren Nonthreaded-Code zum Laufen.
- Machen Sie Ihren Threaded-Code pluggable.
- Machen Sie Ihren Threaded-Code anpassbar.
- Führen Sie den Code mit mehr Threads als Prozessoren aus.
- Führen Sie den Code auf verschiedenen Plattformen aus.
- Instrumentieren Sie Ihren Code, um Fehler zu provozieren.

Behandeln Sie gelegentlich auftretende Fehler als potenzielle Threading-Probleme

Threaded-Code bringt Komponenten zum Scheitern, die »einfach nicht scheitern können«. Die meisten Entwickler haben kein intuitives Verständnis dafür, wie Threading mit anderem Code interagiert (die Autoren eingeschlossen). Bugs in Threaded-Code können ihre Symptome einmal in tausend oder in einer Million Ausführungen zeigen. Versuche, Abläufe zu reproduzieren, können frustrierend sein. Dies verleitet Entwickler oft dazu, einen Fehler auf kosmische Strahlen, einen Hardware-Glitch oder ein anderes einmaliges Ereignis zu schieben. Am besten gehen Sie davon aus, dass derartige »einmalige Ereignisse« nicht existieren. Je länger Sie sie ignorieren, desto mehr Code wird auf einem potenziell fehlerhaften Ansatz aufgebaut.

Empfehlung: Behandeln Sie Systemfehler nicht als »einmalige Ereignisse«.

Bringen Sie erst den Nonthreaded-Code zum Laufen

Dies mag offensichtlich sein, aber es schadet nicht, diesen Punkt extra zu betonen. Sorgen Sie dafür, dass der Code außerhalb der Threads funktioniert. Im Allgemeinen bedeutet dies, dass Sie POJOs erstellen, die von Ihren Threads aufgerufen werden. Die POJOs wissen nichts von den Threads und können deshalb außerhalb der

mit Threads arbeitenden Umgebung getestet werden. Je größer der Anteil Ihres Systems ist, den Sie in solche POJOs einfügen können, desto besser.

Empfehlung: Versuchen Sie nicht, nicht Thread-bezogene Bugs und Thread-bezogene Bugs gleichzeitig zu beheben. Sorgen Sie da für, dass Ihr Code außerhalb von Threads funktioniert.

Machen Sie Ihren Threaded-Code pluggable

Schreiben Sie nebenläufigen Code so, dass er in mehreren Konfigurationen ausgeführt werden kann:

- Ein Thread, mehrere Threads, bei der Ausführung variierend
- Threaded-Code, der sowohl mit echten Daten als auch mit Test-Doubles interagieren kann
- Ausführung mit Test-Doubles, die variabel schnell oder langsam laufen
- Tests so konfigurieren, dass die Anzahl der Iterationen beliebig ist

Empfehlung: Machen Sie Ihren threadbasierten Code besonders pluggable, damit Sie ihn in verschiedenen Konfigurationen ausführen können.

Schreiben Sie anpassbaren Threaded-Code

Normalerweise wird die richtige Balance von Threads nur durch Versuch und Irrtum gefunden. Sie sollten sich gleich am Anfang überlegen, wie Sie das Leistungsverhalten Ihres Systems unter verschiedenen Konfigurationen messen können. Treffen Sie Vorkehrungen, damit Sie die Anzahl der Threads leicht anpassen können, auch während das System läuft. Erwägen Sie auch, Möglichkeiten zu schaffen, dass sich das System selbst je nach Durchsatz und Nutzung optimiert.

Den Code mit mehr Threads als Prozessoren ausführen

Dinge passieren, wenn das System zwischen Tasks wechselt. Um den Task-Wechsel anzuregen, sollten Sie den Code mit mehr Threads ausführen, als Prozessoren oder Kerne vorhanden sind. Je häufiger Ihre Tasks gewechselt werden, desto wahrscheinlich stoßen Sie auf Code, der einen kritischen Abschnitt verpasst oder einen Deadlock verursacht.

Den Code auf verschiedenen Plattformen ausführen

2007 entwickelten wir einen Kurs über die nebenläufige Programmierung. Die Kursentwicklung erfolgte hauptsächlich unter OS X. Das Seminar wurde mit Windows XP in einer VM präsentiert. Tests, die geschrieben wurden, um Fehlerbedingungen zu demonstrieren, scheiterten in einer XP-Umgebung nicht so häufig wie unter OS X.

In allen Fällen war der zu testende Code bekanntermaßen defekt. Dies bestätigt nur die Tatsache, dass verschiedene Betriebssysteme unterschiedliche Threading-Policies haben, die die Ausführung des Codes beeinflussen. Multithreaded-Code verhält sich in verschiedenen Umgebungen unterschiedlich. (Wussten Sie, dass das Threading-Modell in Java kein präemptives Threading garantiert? Moderne Betriebssysteme unterstützen präemptives Threading; deshalb erhalten Sie es »kostenlos«. Doch selbst dann wird es von der JVM nicht garantiert.) Sie sollten Ihre Tests in jeder potenziellen Deployment-Umgebung ausführen.

Empfehlung: Führen Sie nebenläufigen Code früh und oft auf allen Zielplattformen aus.

Code-Scheitern durch Instrumentierung provozieren

Defekte in nebenläufigem Code sind normalerweise verborgen und lassen sich durch einfache Tests oft nicht aufdecken. Tatsächlich bleiben sie bei der normalen Verarbeitung verborgen. Sie treten nur alle paar Stunden oder Tage oder sogar Wochen auf!

Der Grund, warum Threading-Bugs so selten auftreten können und so schwer zu reproduzieren sind, liegt darin, dass nur sehr wenige der vielen Tausend möglichen Ausführungspfade durch einen verletzbaren Abschnitt tatsächlich scheitern. Deshalb kann die Wahrscheinlichkeit für das Scheitern eines Ausführungspfades erstaunlich gering sein. Dies macht das Erkennen und das Debugging sehr schwierig.

Wie können Sie Ihre Chancen verbessern, solche seltenen Ereignisse einzufangen? Sie können Ihren Code instrumentieren und ihn zwingen, in verschiedenen Reihenfolgen zu laufen, indem Sie Aufrufe an Methoden wie `Object.wait()`, `Object.sleep()`, `Object.yield()` und `Object.priority()` hinzufügen.

Jede dieser Methoden kann die Reihenfolge der Ausführung beeinflussen und deshalb die Chancen für die Entdeckung eines Mangels verbessern. Es ist besser, wenn defekter Code so früh und so oft wie möglich scheitert.

Es gibt zwei Optionen für Code-Instrumentierung:

- Manuelle Codierung
- Automatisiert

Manuelle Codierung

Sie können Aufrufe von `wait()`, `sleep()`, `yield()` und `priority()` manuell in Ihren Code einfügen. Wenn Sie einen verzwickten Code-Bereich testen, könnte dies genau die richtige Maßnahme sein.

Hier ist ein Beispiel für diese Vorgehensweise:

```
public synchronized String nextUrlOrNull() {  
    if(hasNext()) {  
        String url = urlGenerator.next();
```

```
Thread.yield(); // für Testzwecke eingefügt
updateHasNext();
return url;
}
return null;
}
```

Der eingefügte Aufruf von `yield()` ändert die Ausführungspfade des Codes und führt möglicherweise dazu, dass er an Stellen scheitert, die er zuvor problemlos passiert hatte. Wenn der Code scheitert, liegt das nicht an dem zusätzlichen Aufruf von `yield()`, sondern er war bereits vorher defekt; und jetzt wurde dieser defekte Bereich aufgedeckt. (Dies ist nicht ganz richtig. Da die JVM kein präemptives Threading garantiert, könnte ein bestimmter Algorithmus immer unter einem OS funktionieren, das Threads nicht präemptiv verarbeitet. Das Umgekehrte ist ebenfalls möglich, allerdings aus anderen Gründen.)

Bei diesem Ansatz gibt es mehrere Probleme:

- Sie müssen die entsprechenden Stellen für diese Maßnahme manuell suchen.
- Woher wissen Sie, wo Sie den Aufruf einfügen müssen und welche Art von Aufruf Sie verwenden sollten?
- Wird solcher Code in eine Produktionsumgebung übernommen, verlangsamt er der Ausführung unnötigerweise.
- Es ist ein Schrotschuss-Ansatz. Vielleicht entdecken Sie Mängel, vielleicht auch nicht. Tatsächlich stehen die Chancen gegen Sie.

Wir brauchen eine Methode, um diese Maßnahmen beim Testen, aber nicht in der Produktion durchzuführen. Außerdem brauchen wir eine Möglichkeit, Konfigurationen zwischen verschiedenen Ausführungen schnell und leicht zu ändern, um unsere Chancen zu verbessern, Fehler zu finden.

Wenn wir unser System in POJOs, die nichts über Threading wissen, und Klassen zerlegen, die das Threading steuern, wird es offensichtlich leichter sein, die entsprechenden Stellen für die Instrumentierung des Codes zu lokalisieren. Darüber hinaus könnten wir viele verschiedene Test-Jigs erstellen, die die POJOs mit verschiedenen Kombinationen von `sleep`, `yield` usw. aufrufen.

Automatisiert

Sie könnten Tools wie etwa Aspect-Oriented Framework, CGLIB oder ASM einsetzen, um Ihren Code per Programm zu instrumentieren. Beispielsweise könnten Sie eine Klasse mit einer einzigen Methode verwenden:

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

Sie könnten dann Aufrufe dieser Klasse an verschiedenen Stellen in Ihrem Code einfügen:

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJigglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJigglePoint.jiggle();
        updateHasNext();
        ThreadJigglePoint.jiggle();
        return url;
    }
    return null;
}
```

Jetzt verwenden Sie einen einfachen Aspekt, der zufällig eine der Aktionen »do nothing«, »sleeping« oder »yielding« auswählt.

Oder nehmen Sie an, die `ThreadJigglePoint`-Klasse habe zwei Implementierungen. Die erste implementiert `jiggle`, um nichts zu tun, und wird in der Produktion benutzt. Die zweite generiert eine Zufallszahl, um zwischen »sleeping«, »yielding« oder einfach »falling through« auszuwählen. Wenn Sie nun Ihre Tests tausend Mal mit Random-Jiggling ausführen, finden Sie möglicherweise einige Mängel. Werden die Tests bestanden, können Sie wenigstens sagen, dass Sie sorgfältig gearbeitet haben. Auch wenn dieser Ansatz etwas simplifizierend ist, könnte er eine vernünftige Option darstellen, wenn ausgefeiltere Tools nicht zur Verfügung stehen.

Es gibt ein Tool namens *ConTest* (<http://www.alphaworks.ibm.com/tech/contest>) von IBM, das, allerdings mit erheblich mehr Finesse, Ähnliches leistet.

Der Punkt ist, dass der Code so »durchgerüttelt« wird, dass die Threads zu verschiedenen Zeiten in unterschiedlicher Reihenfolge ausgeführt werden. Die Kombination aus brauchbaren Tests und Jiggling kann Ihre Chancen, Fehler zu finden, erheblich verbessern.

Empfehlung: *Verwenden Sie Jiggling-Strategien, um Fehler auszumerzen.*

13.10 Zusammenfassung

Es ist schwer, korrekten nebenläufigen Code zu schreiben. Einfach nachvollziehbarer Code kann zum Albtraum werden, wenn mehrere Threads und gemeinsam genutzte Daten ins Spiel kommen. Wenn Sie nebenläufigen Code schreiben müssen, sollte Ihr Code unbedingt sauber sein; andernfalls müssen Sie mit subtilen und kaum reproduzierbaren Fehlern rechnen.

Zuallererst sollten Sie das Single-Responsibility-Prinzip beachten. Zerlegen Sie Ihr System in POJOs, die den Thread-bezogenen Code von dem Thread-freien Code trennen. Achten Sie darauf, dass Sie beim Testen von Thread-bezogenen Code nur

diesen und nichts sonst testen. Anders ausgedrückt: Ihr Thread-bezogener Code sollte klein und fokussiert sein.

Sie müssen die möglichen Quellen für Nebenläufigkeitsprobleme kennen: mehrere Threads, die gemeinsam genutzte Daten verarbeiten oder einen gemeinsamen Ressourcen-Pool nutzen. Grenzfälle, wie etwa ein sauberes Herunterfahren oder das Beenden einer Schleifen-Iteration, können besonders verzwickelt sein.

Studieren Sie Ihre Library und die grundlegenden Algorithmen. Sie müssen verstehen, wie die Library und die Algorithmen die Lösung spezieller Probleme unterstützen.

Lernen Sie, wie Sie die Code-Bereiche finden, die gesperrt werden müssen, und sperren Sie sie. Sperren Sie keine Code-Bereiche, die nicht gesperrt werden müssen. Vermeiden Sie es, einen gesperrten Abschnitt von einem anderen aus aufzurufen. Dazu müssen Sie gründlich verstehen, was und was nicht gemeinsam genutzt wird. Halten Sie die Menge der gemeinsam genutzten Objekte und die Geltungsbereiche der Nutzungen so klein wie möglich. Passen Sie die Designs der Objekte mit gemeinsam genutzten Daten an die Clients an, anstatt diese zu zwingen, den Nutzungsstatus dieser Daten zu verwalten.

Probleme lassen sich nicht vermeiden. Probleme, die nicht frühzeitig auftreten, werden oft als einmalige Ereignisse abgeschrieben. Diese so genannten One-offs treten nur unter Last oder zu scheinbar zufälligen Zeitpunkten auf. Deshalb müssen Sie Ihren Thread-bezogenen Code unter vielen Konfigurationen wiederholt und laufend auf vielen Plattformen ausführen können. Testbarkeit, die sich natürlicherweise aus der Befolgung der *Drei Gesetze der TDD* ergibt, schließt eine Plug-Fähigkeit (»Konfigurierbarkeit«) in einem Umfang ein, der die für die Ausführung des Codes unter vielen Konfigurationen erforderliche Unterstützung bietet.

Sie können Ihre Chancen, fehlerhaften Code zu finden, erheblich verbessern, wenn Sie sich die Zeit nehmen, Ihren Code zu instrumentieren. Sie können dies manuell tun oder eine automatisierte Technologie einsetzen. Investieren Sie frühzeitig in diese Technologie. Sie sollten threadbasierten Code so lange wie möglich ausführen, bevor Sie ihn produktiv einsetzen.

Wenn Sie einen sauberen Ansatz befolgen, steigen Ihre Chancen auf einen Erfolg erheblich.

Schrittweise Verfeinerung

Fallstudie eines Parsers für Befehlszeilenargumente



Dieses Kapitel enthält eine Fallstudie der schrittweisen Verfeinerung. Sie lernen ein Modul kennen, das gut angelegt ist, aber dann nicht skaliert. Denn sehen Sie, wie ein Refactoring durchgeführt und das Modul bereinigt wird.

Die meisten Entwickler müssen dann und wann Befehlszeilenargumente parsen. Ohne ein geeignetes Utility durchlaufen sie dann einfach das Array von Strings, das an die `main`-Funktion übergeben wird. Es gibt mehrere brauchbare Utilities aus verschiedenen Quellen, aber keines leistet genau das, was ich haben möchte. Deshalb beschloss ich natürlich, mein eigenes zu schreiben. Ich nenne es `Args`.

`Args` ist sehr leicht anzuwenden. Sie konstruieren einfach ein `Args`-Objekt mit den Input-Argumenten und einem Format-String. Dann fragen Sie die `Args`-Instanz nach den Werten der Argumente ab. Betrachten Sie das folgende einfache Beispiel:

Listing 14.1: Einfaches Beispiel von `Args`

```
public static void main(String[] args) {  
    try {  
        Args arg = new Args("\1,p#,d*", args);
```

```
        boolean logging = arg.getBoolean('l');
        int port = arg.getInt('p');
        String directory = arg.getString('d');
        executeApplication(logging, port, directory);
    } catch (ArgsException e) {
        System.out.printf("Argument error: %s\n", e.errorMessage());
    }
}
```

Sie sehen, wie einfach dies ist. Wir erstellen einfach mit zwei Parametern eine Instanz der `Args`-Klasse. Der erste Parameter ist der Format- oder *Schema*-String: `"l,p#,d*"`. Er definiert drei Befehlszeilenargumente: Das erste, `-l`, ist ein boolesches Argument; das zweite, `-p`, ist ein Integer-Argument; das dritte, `-d`, ist ein String-Argument. Der zweite Parameter des `Args`-Konstruktors ist einfach das Array mit den Befehlszeilenargumenten, die an `main` übergeben werden.

Wenn der Konstruktor zurückkehrt, ohne eine `ArgsException` auszulösen, wurde die eingehende Befehlszeile geparkt, und die `Args`-Instanz steht für Abfragen bereit. Mit Methoden wie `getBoolean`, `getInteger` und `getString` können wir die Werte der Argumente mit ihren Namen abfragen.

Wenn es ein Problem mit dem Format-String und/oder den Befehlszeilenargumenten gibt, wird eine `ArgsException` ausgelöst. Mit der `errorMessage`-Methode der Ausnahme können wir eine genauere Beschreibung des Fehlers abrufen.

14.1 Args-Implementierung

Listing 14.2 zeigt die Implementierung der `Args`-Klasse. Bitte lesen Sie sie sehr sorgfältig. Ich habe hart an dem Stil und der Struktur gearbeitet und hoffe, dass sie sich als Vorlage eignet.

Listing 14.2: `Args.java`

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
    private Map<Character, ArgumentMarshaler> marshalers;
    private Set<Character> argsFound;
    private ListIterator<String> currentArgument;

    public Args(String schema, String[] args) throws ArgsException {
        marshalers = new HashMap<Character, ArgumentMarshaler>();
        argsFound = new HashSet<Character>();

        parseSchema(schema);
    }
}
```

```
    parseArgumentStrings(Arrays.asList(args));
}

private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
        if (element.length() > 0)
            parseSchemaElement(element.trim());
}

private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("-"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]"))
        marshalers.put(elementId, new StringArrayArgumentMarshaler());
    else
        throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
}

private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId))
        throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
}

private void parseArgumentStrings(List<String> argsList) throws ArgsException
{
    for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
    {
        String argString = currentArgument.next();
        if (argString.startsWith("-")) {
            parseArgumentCharacters(argString.substring(1));
        } else {
            currentArgument.previous();
            break;
        }
    }
}

private void parseArgumentCharacters(String argChars) throws ArgsException {
    for (int i = 0; i < argChars.length(); i++)
        parseArgumentCharacter(argChars.charAt(i));
}
```



```
}

private void parseArgumentCharacter(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null) {
        throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
    } else {
        argsFound.add(argChar);
        try {
            m.set(currentArgument);
        } catch (ArgsException e) {
            e.setErrorArgumentId(argChar);
            throw e;
        }
    }
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public int nextArgument() {
    return currentArgument.nextIndex();
}

public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
}

public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg));
}

public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
}

public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
}

public String[] getStringArray(char arg) {
    return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
}
}
```

Beachten Sie, dass Sie den Code von oben bis unten lesen können, ohne viel herumzuspringen oder vorausszuschauen. Der einzige Abschnitt, bei dem Sie vorausschauen müssen, ist die Definition von `ArgumentMarshaler`, den ich absichtlich

ausgelassen habe. Nachdem Sie den Code sorgfältig gelesen haben, sollten Sie verstehen, was das `ArgumentMarshaler`-Interface ist und was seine abgeleiteten Klassen tun. Einige dieser Klassen werden jetzt in den Listings 14.3 bis 14.6 gezeigt.

Listing 14.3: `ArgumentMarshaler.java`

```
public interface ArgumentMarshaler {  
    void set(Iterator<String> currentArgument) throws ArgsException;  
}
```

Listing 14.4: `BooleanArgumentMarshaler.java`

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {  
    private boolean booleanValue = false;  
  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        booleanValue = true;  
    }  
  
    public static boolean getValue(ArgumentMarshaler am) {  
        if (am != null && am instanceof BooleanArgumentMarshaler)  
            return ((BooleanArgumentMarshaler) am).booleanValue;  
        else  
            return false;  
    }  
}
```

Listing 14.5: `StringArgumentMarshaler.java`

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;  
  
public class StringArgumentMarshaler implements ArgumentMarshaler {  
    private String stringValue = "";  
  
    public void set(Iterator<String> currentArgument) throws ArgsException {  
        try {  
            stringValue = currentArgument.next();  
        } catch (NoSuchElementException e) {  
            throw new ArgsException(MISSING_STRING);  
        }  
    }  
  
    public static String getValue(ArgumentMarshaler am) {  
        if (am != null && am instanceof StringArgumentMarshaler)  
            return ((StringArgumentMarshaler) am).stringValue;  
        else  
            return "";  
    }  
}
```

Listing 14.6: IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            throw new ArgsException(MISSING_INTEGER);
        } catch (NumberFormatException e) {
            throw new ArgsException(INVALID_INTEGER, parameter);
        }
    }

    public static int getValue(ArgumentMarshaler am) {
        if (am != null && am instanceof IntegerArgumentMarshaler)
            return ((IntegerArgumentMarshaler) am).intValue;
        else
            return 0;
    }
}
```

Die anderen von `ArgumentMarshaler` abgeleiteten Klassen wiederholen einfach dieses Pattern für `doubles`- und `String`-Arrays und würden dieses Kapitel nur überfrachten. Ich überlasse sie Ihnen zur Übung.

Möglicherweise bereiten Ihnen einige andere Informationen Schwierigkeiten: die Definition der Fehlercode-Konstanten. Sie sind in der `ArgsException`-Klasse enthalten (Listing 14.7).

Listing 14.7: ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = null;
    private ErrorCode errorCode = OK;

    public ArgsException() {}

    public ArgsException(String message) {super(message);}

    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }
}
```

```
public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
}

public ArgsException(ErrorCode errorCode,
                    char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId() {
    return errorArgumentId;
}

public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter() {
    return errorParameter;
}

public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
}

public ErrorCodes getErrorCode() {
    return errorCode;
}

public void setErrorCode(ErrorCodes errorCode) {
    this.errorCode = errorCode;
}

public String errorMessage() {
    switch (errorCode) {
        case OK:
            return "TILT: Should not get here.";
        case UNEXPECTED_ARGUMENT:
            return String.format("Argument -%c unexpected.", errorArgumentId);
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                                errorArgumentId, errorParameter);
    }
}
```

```
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                                errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                                errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                                errorArgumentId);
        case INVALID_ARGUMENT_NAME:
            return String.format("'%' is not a valid argument name.",
                                errorArgumentId);
        case INVALID_ARGUMENT_FORMAT:
            return String.format("'%' is not a valid argument format.",
                                errorParameter);
    }
    return "";
}

public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE
}
```

Es ist bemerkenswert, wie viel Code erforderlich ist, um die Details dieses einfachen Konzepts zu realisieren. Einer der Gründe dafür liegt darin, dass wir eine besonders wortreiche Sprache verwenden. Java erfordert als statische typisierte Sprache zahlreiche Wörter, um das Typensystem zu befriedigen. In Sprachen wie Ruby, Python oder Smalltalk wäre dieses Programm viel kleiner. (Vor Kurzem schrieb ich dieses Programm in Ruby um. Es war nur 1/7 so groß und etwas besser strukturiert.)

Bitte lesen Sie den Code noch einmal. Achten Sie besonders auf die Benennung der Dinge, die Größe der Funktionen und die Formatierung des Codes. Wenn Sie ein erfahrener Programmierer sind, haben Sie möglicherweise hier und dort etwas am Stil oder der Struktur auszusetzen. Insgesamt hoffe ich jedoch, dass Sie zu dem Schluss kommen, dass dieses Programm gut geschrieben ist und eine saubere Struktur hat.

Beispielsweise sollte es offensichtlich sein, wie man einen neuen Argumenttyp, wie etwa ein Datum oder eine komplexe Zahl, hinzufügen kann. Es sollte auch klar sein, dass dafür nur ein trivialer Aufwand erforderlich ist. Kurz gesagt: Sie sollten einfach nur eine neue abgeleitete Klasse von `ArgumentMarshaler`, eine neue `getXXX`-Funktion und eine neue `case`-Anweisung in der `parseSchemaElement`-Funktion benötigen. Wahrscheinlich sollte es auch einen neuen `ArgsException.ErrorCode` und eine neue Fehlermeldung geben.

Wie habe ich dies gemacht?

Ich möchte Sie gleich am Anfang beruhigen. Ich habe dieses Programm in seiner gegenwärtigen Form nicht einfach aus dem Stand geschrieben. Und was noch wichtiger ist: Ich erwarte nicht, dass Sie saubere und elegante Programme in einem Durchgang schreiben können. Wenn wir überhaupt etwas in den letzten Jahrzehnten gelernt haben, dann eines: Programmierung ist eher ein Handwerk als eine Wissenschaft. Um sauberen Code zu schreiben, müssen Sie zunächst schmutzigen Code schreiben und *ihn dann bereinigen*.

Dies sollte Sie nicht überraschen. Wir haben diese Wahrheit schon in der Grundschule gelernt, als unsere Lehrer (normalerweise vergebens) versuchten, uns beizubringen, erst ins Unreine zu schreiben. Idealerweise sollten wir erst ein rohes Konzept schreiben und dieses in mehreren Schritten immer weiter bis zur endgültigen Version verfeinern. Saubere Ergebnisse, so die Botschaft, wären die Folge einer schrittweisen Verfeinerung.

Die meisten Programmieranfänger (wie die meisten Grundschüler) befolgen diesen Rat nicht besonders eifrig. Sie glauben, das Hauptziel bestehe darin, das Programm zum Laufen zu bringen. Sobald es »funktioniert«, wenden sie sich der nächsten Aufgabe zu und lassen das »funktionierende« Programm in dem Zustand zurück, in dem sie es schließlich »zum Laufen« brachten. Die meisten erfahrenen Programmierer wissen, dass dies professioneller Selbstmord ist.

14.2 Args: der Rohentwurf

Listing 14.8 zeigt eine frühere Version der Args-Klasse. Sie »funktioniert«. Und sie ist chaotisch.

Listing 14.8: Args.java (erster Entwurf)

```
import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs = new HashMap<Character, String>();
    private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;
```

```
private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
}

private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
        return true;
    parseSchema();
    try {
        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}

private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
        if (element.length() > 0) {
            String trimmedElement = element.trim();
            parseSchemaElement(trimmedElement);
        }
    }
    return true;
}

private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
        parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
        parseIntegerSchemaElement(elementId);
    } else {
        throw new ParseException(
            String.format("Argument: %c has invalid format: %s.",
                elementId, elementTail), 0);
    }
}

private void validateSchemaElementId(char elementId) throws ParseException {
```

```
        if (!Character.isLetter(elementId)) {
            throw new ParseException(
                "Bad character:" + elementId + "in Args format: " + schema, 0);
        }
    }

    private void parseBooleanSchemaElement(char elementId) {
        booleanArgs.put(elementId, false);
    }

    private void parseIntegerSchemaElement(char elementId) {
        intArgs.put(elementId, 0);
    }

    private void parseStringSchemaElement(char elementId) {
        stringArgs.put(elementId, "");
    }

    private boolean isStringSchemaElement(String elementTail) {
        return elementTail.equals("*");
    }

    private boolean isBooleanSchemaElement(String elementTail) {
        return elementTail.length() == 0;
    }

    private boolean isIntegerSchemaElement(String elementTail) {
        return elementTail.equals("#");
    }

    private boolean parseArguments() throws ArgsException {
        for (currentArgument = 0; currentArgument < args.length; currentArgument++)
        {
            String arg = args[currentArgument];
            parseArgument(arg);
        }
        return true;
    }

    private void parseArgument(String arg) throws ArgsException {
        if (arg.startsWith("-"))
            parseElements(arg);
    }

    private void parseElements(String arg) throws ArgsException {
        for (int i = 1; i < arg.length(); i++)
            parseElement(arg.charAt(i));
    }
}
```



```
private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    if (isBooleanArg(argChar))
        setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
        setStringArg(argChar);
    else if (isIntArg(argChar))
        setIntArg(argChar);
    else
        return false;

    return true;
}

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}

private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
    }
```

```
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private boolean isStringArg(char argChar) {
    return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}

private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                                errorArgumentId);
    }
    return "";
}
```

```
private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
}

private boolean falseIfNull(Boolean b) {
    return b != null && b;
}

private int zeroIfNull(Integer i) {
    return i == null ? 0 : i;
}

private String blankIfNull(String s) {
    return s == null ? "" : s;
}

public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
}

public int getInt(char arg) {
    return zeroIfNull(intArgs.get(arg));
}

public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public boolean isValid() {
    return valid;
}

private class ArgsException extends Exception {
}
}
```

Ich hoffe, dass Sie zunächst wie folgt auf diese Masse von Code reagiert haben: »Auf jeden Fall bin ich froh, dass er den Code nicht in dieser Form gelassen hat!« Wenn

Sie dieses Gefühl haben, sollten Sie daran denken, wie andere Entwickler auf Code reagieren, den Sie in der Form eines Rohentwurfs hinterlassen haben.

Wahrscheinlich ist »Rohentwurf« noch die freundlichste Bezeichnung für diesen Code. Es handelt sich offensichtlich um ein unfertiges Produkt. Die reine Anzahl der Instanzvariablen ist abschreckend. Die seltsamen Strings wie "TILT", die HashSets und TreeSets und die try-catch-catch-Blöcke ergeben zusammen einen Misthaufen.

Ich wollte keinen Misthaufen schreiben. Tatsächlich versuchte ich, alle Dinge möglichst vernünftig zu ordnen. Sie können dies wahrscheinlich an meiner Wahl der Funktions- und Variablenamen und der Tatsache ablesen, dass das eine grobe Struktur hat. Doch ganz offensichtlich ist mir das Problem entglitten.

Das Chaos entstand allmählich. Frühere Versionen waren längst nicht so hässlich. So zeigt etwa Listing 14.9 eine frühere Version, in der nur boolesche Argumente funktionierten.

Listing 14.9: Args.java (nur boolesche Argumente)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private int numberOfArguments = 0;

    public Args(String schema, String[] args) {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    public boolean isValid() {
        return valid;
    }

    private boolean parse() {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        parseArguments();
        return unexpectedArguments.size() == 0;
    }
}
```

```
private boolean parseSchema() {
    for (String element : schema.split(",")) {
        parseSchemaElement(element);
    }
    return true;
}

private void parseSchemaElement(String element) {
    if (element.length() == 1) {
        parseBooleanSchemaElement(element);
    }
}

private void parseBooleanSchemaElement(String element) {
    char c = element.charAt(0);
    if (Character.isLetter(c)) {
        booleanArgs.put(c, false);
    }
}

private boolean parseArguments() {
    for (String arg : args)
        parseArgument(arg);
    return true;
}

private void parseArgument(String arg) {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
    if (isBoolean(argChar)) {
        numberOfArguments++;
        setBooleanArg(argChar, true);
    } else
        unexpectedArguments.add(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
}
```

```
}

public int cardinality() {
    return numberOfArguments;
}

public String usage() {
    if (schema.length() > 0)
        return "["+schema+"]";
    else
        return "";
}

public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
    } else
        return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
}

public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
}
}
```

Obwohl es an diesem Code viel zu bemängeln gibt, ist er wirklich nicht so schlecht. Er ist kompakt und leicht zu verstehen. Doch in dem Code sind die Keime des späteren Misthaufens leicht zu erkennen. Es ist ziemlich klar, wie daraus das spätere Chaos entstand.

Beachten Sie, dass das spätere Chaos nur zwei zusätzliche Argumenttypen enthält: `String` und `integer`. Das Hinzufügen von nur zwei weiteren Argumenttypen hatte einen erheblichen negativen Einfluss auf den Code. Ich machte aus einem einigermaßen wartbaren Programm ein Gebilde, bei dem ich mit zahlreichen Bugs und Wanzen rechnen musste.

Ich fügte die beiden Argumenttypen schrittweise hinzu. Zuerst fügte ich das `String`-Argument hinzu und erhielt das folgende Programm:

Listing 14.10: Args.java (boolesche Argumente und String-Argumente)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, Boolean> booleanArgs =
        new HashMap<Character, Boolean>();
    private Map<Character, String> stringArgs =
        new HashMap<Character, String>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgument = '\0';

    enum ErrorCode {
        OK, MISSING_STRING
    }

    private ErrorCode errorCode = ErrorCode.OK;

    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
        parseSchema();
        parseArguments();
        return valid;
    }

    private boolean parseSchema() throws ParseException {
        for (String element : schema.split(",")) {
            if (element.length() > 0) {
                String trimmedElement = element.trim();
                parseSchemaElement(trimmedElement);
            }
        }
        return true;
    }

    private void parseSchemaElement(String element) throws ParseException {
        char elementId = element.charAt(0);
        String elementTail = element.substring(1);
```

```
validateSchemaElementId(elementId);
if (isBooleanSchemaElement(elementTail))
    parseBooleanSchemaElement(elementId);
else if (isStringSchemaElement(elementTail))
    parseStringSchemaElement(elementId);
}

private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
}

private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
}

private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("");
}

private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}

private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
}

private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

private void parseArgument(String arg) {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
    if (setArgument(argChar))
```



```
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        valid = false;
    }
}

private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
        setBooleanArg(argChar, true);
    else if (isString(argChar))
        setStringArg(argChar, "");
    else
        set = false;

    return set;
}

private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {
        stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgument = argChar;
        errorCode = ErrorCode.MISSING_STRING;
    }
}

private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}
```

```
}

public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
        return unexpectedArgumentMessage();
    } else
        switch (errorCode) {
            case MISSING_STRING:
                return String.format("Could not find string parameter for -%c.",
                                      errorArgument);

            case OK:
                throw new Exception("TILT: Should not get here.");
        }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
}

public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
}

private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
}

public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
}

private String blankIfNull(String s) {
    return s == null ? "" : s;
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public boolean isValid() {
    return valid;
}
}
```

Sie können erkennen, dass der Code meiner Kontrolle entglitt. Er war noch nicht schrecklich, aber das Chaos wurde sicherlich größer. Er bildet zwar einen Haufen, aber noch keinen Misthaufen. Ich musste noch den `integer`-Argumenttyp hinzufügen, um wirklich einen Misthaufen zu erhalten.

Deshalb hörte ich auf

Ich musste mindestens noch zwei weitere Argumenttypen hinzufügen und konnte erkennen, dass sie alles noch viel schlimmer machen würden. Wenn ich stur drauflos gearbeitet hätte, hätte ich den Code wahrscheinlich zum Laufen bringen können; aber ich würde ein Chaos hinterlassen, das für etwaige Wartungsarbeiten zu groß gewesen wäre. Wenn die Struktur dieses Codes überhaupt wartbar sein sollte, war jetzt der Zeitpunkt gekommen, sie zu korrigieren.

Deshalb hörte ich auf, Funktionen hinzuzufügen, und begann, ein Refactoring durchzuführen. Da ich gerade die `String`- und `integer`-Argumente hinzugefügt hatte, wusste ich, dass für jeden Argumenttyp neuer Code an drei hauptsächlichen Stellen eingefügt werden musste. Erstens wurde für jeden Argumenttyp eine Methode zum Parsen seines Schema-Elements benötigt, um die `HashMap` für diesen Typ auszuwählen. Zweitens musste jeder Argumenttyp in den Befehlszeilen-Strings geparkt und in seinen wahren Typ umgewandelt werden. Drittens benötigte jeder Argumenttyp eine `getXXX`-Methode, um ihn als echten Typ an einen Aufrufer zurückzugeben.

Viele verschiedene Typen, alle mit ähnlichen Methoden – das hört sich für mich wie eine Klasse an. Und so wurde das `ArgumentMarshaler`-Konzept geboren.

Über inkrementelle Entwicklung

Eine der besten Methoden, ein Programm zu ruinieren, besteht darin, im Namen der Verbesserung massive Änderungen an seiner Struktur vorzunehmen. Einige Programme erholen sich nie von solchen »Verbesserungen«. Das Problem liegt darin, dass es sehr schwer ist, das Programm dazu zu bringen, wie vor der »Verbesserung« zu arbeiten.

Um dies zu vermeiden, arbeite ich nach den Prinzipien der Test Driven Development (TDD). Eine der zentralen Doktrinen dieses Ansatzes fordert, dass das System jederzeit lauffähig sein muss. Anders ausgedrückt: Bei TDD darf ich das System nicht so ändern, dass es nicht mehr läuft. Nach jeder Änderung muss es wie zuvor arbeiten.

Um dies zu erreichen, benötige ich eine Suite automatisierter Tests, die ich jederzeit nach Gusto ausführen kann und die verifizieren, dass sich das Verhalten des Systems nicht geändert hat. Für die `Args`-Klasse hatte ich eine Suite von Unit- und Acceptance-Tests erstellt, während ich den Misthaufen zusammenbaute. Die Unit-Tests waren in Java geschrieben und wurden von JUnit verwaltet. Die Acceptance-Tests bestanden aus Wiki-Seiten in FitNesse. Ich konnte diese Tests jederzeit ausführen. Wurden sie bestanden, war ich sicher, dass das System meinen Spezifikationen entsprechend funktionierte.

Deshalb machte ich mich daran, eine große Anzahl winziger Änderungen vorzunehmen. Jede Änderung brachte die Struktur des Systems dem **ArgumentMarshaler**-Konzept näher. Dennoch blieb das System nach jeder Änderung lauffähig. Zuerst fügte ich am Ende des Misthaufens das Gerüst des **ArgumentMarshaler** ein (Listing 14.11).

Listing 14.11: ArgumentMarshaler am Ende von Args.java eingefügt

```
private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() {return booleanValue;}
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
}
}
```

Damit wurde der Code sicher nicht beschädigt. Dann nahm ich die einfachste Änderung vor, die so wenig wie möglich kaputtmachen konnte. Ich änderte die **HashMap** für die booleschen Argumente so, dass sie einen **ArgumentMarshaler** übernahm.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
```

Dadurch wurde einige Anweisungen ungültig, die ich schnell korrigierte.

```
***
private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());
}

**
private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).setBoolean(value);
}

***
public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg).getBoolean());
}
```

Beachten Sie, wie diese Änderungen genau die Stellen betreffen, die ich weiter vorne erwähnt habe: `parse`, `set` und `get` für den Argumenttyp. Doch selbst diese kleine Änderung brachte leider einige Tests zum Scheitern. Schauen Sie sich `getBoolean` sorgfältig an: Wenn Sie die Funktion mit `'y'`, aufrufen, aber es kein `y`-Argument gibt, dann gibt `booleanArgs.get('y')` den Wert `null` zurück und die Funktion löst eine `NullPointerException` aus. Die `falseIfNull`-Funktion war als Schutz gegen diese Situation verwendet worden, aber meine Änderung hatte diese Funktion irrelevant gemacht.

Eine schrittweise Entwicklung verlangte, dass ich dieses Problem schnell beheben musste, bevor ich weitere Änderungen vornahm. Tatsächlich war die Korrektur nicht zu schwierig. Ich musste nur die Prüfung auf `null` verschieben. Ich musste nicht mehr die boolesche Größe `null` prüfen, sondern den `ArgumentMarshaller`.

Zuerst entfernte ich den Aufruf von `falseIfNull` aus der `getBoolean`-Funktion. Er war jetzt nutzlos, deshalb eliminierte ich auch die Funktion selbst. Die Tests scheiterten immer noch auf dieselbe Weise; deshalb war ich ziemlich sicher, dass ich keine neuen Fehler eingeführt hatte.

```
public boolean getBoolean(char arg) {
    return booleanArgs.get(arg).getBoolean();
}
```

Als Nächstes zerlegte ich die Funktion in zwei Zeilen und fügte den `ArgumentMarshaller` in eine separate Variable namens `argumentMarshaller` ein. Mir gefiel der lange Variablenname nicht; er war höchst redundant und machte die Funktion unübersichtlich. Deshalb kürzte ich ihn in `am` [N5].

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaller am = booleanArgs.get(arg);
    return am.getBoolean();
}
```

Dann fügte ich die Prüfung auf `null` ein.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaller am = booleanArgs.get(arg);
    return am != null && am.getBoolean();
}
```

14.3 String-Argumente

Das Einfügen der `String`-Argumente ähnelte dem der `boolean`-Argumente. Ich musste die `HashMap` ändern und die Funktionen `parse`, `set` und `get` zum Laufen bringen. Der folgende Code sollte keine Überraschungen enthalten, außer vielleicht, dass ich die gesamte Marshalling-Implementierung in die Basisklasse `ArgumentMarshaller` einfüge, anstatt sie auf die abgeleiteten Klassen zu verteilen.

```
private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();

...
private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, new StringArgumentMarshaler());
}

...
private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.get(argChar).setString(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

...
public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : am.getString();
}

...
private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() {
        return booleanValue;
    }

    public void setString(String s) {
        stringValue = s;
    }

    public String getString() {
        return stringValue == null ? "" : stringValue;
    }
}
```

Auch diese Änderungen wurden einzeln nacheinander durchgeführt. Nach jeder Änderung sorgte ich jeweils dafür, dass alle Tests bestanden wurden, bevor ich mit der nächsten Änderung fortfuhr.

Inzwischen sollten Sie meine Absicht durchschaut haben: Zunächst wollte ich alle gegenwärtigen Marshalling-Verhaltensweisen in die Basisklasse `ArgumentMarshaler` einfügen. Dann wollte ich diese Verhaltensweisen in die abgeleiteten Klassen verlagern. So konnte ich den Code lauffähig erhalten, während ich die Struktur dieses Programms allmählich änderte.

Der nächste Schritt bestand offensichtlich darin, die Funktionalität für die `int`-Argumente in den `ArgumentMarshaler` zu verlagern. Auch hier gab es keine Überraschungen.

```
private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();

...
private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, new IntegerArgumentMarshaler());
}

...
private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (NumberFormatException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
    }
}

...
public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
}

...
private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }
}
```

```
public boolean getBoolean() {
    return booleanValue;
}

public void setString(String s) {
    stringValue = s;
}

public String getString() {
    return stringValue == null ? "" : stringValue;
}

public void setInteger(int i) {
    integerValue = i;
}

public int getInteger() {
    return integerValue;
}
}
```

Nachdem ich das gesamte Marshalling in den `ArgumentMarshaler` verschoben hatte, begann ich damit, die Funktionalität in die abgeleiteten Klassen zu verlagern. Der erste Schritt bestand darin, die `setBoolean`-Funktion in den `BooleanArgumentMarshaler` zu verschieben und dafür zu sorgen, dass sie korrekt aufgerufen wurde. Deshalb erstellte ich eine abstrakte `set`-Methode.

```
private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
        booleanValue = value;
    }

    public boolean getBoolean() {
        return booleanValue;
    }

    public void setString(String s) {
        stringValue = s;
    }

    public String getString() {
        return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
```



```
        integerValue = i;
    }

    public int getInteger() {
        return integerValue;
    }

    public abstract void set(String s);
}
```

Dann implementierte ich die `set`-Methode in `BooleanArgumentMarshaller`.

```
private class BooleanArgumentMarshaller extends ArgumentMarshaller {
    public void set(String s) {
        booleanValue = true;
    }
}
```

Und schließlich ersetzte ich den Aufruf von `setBoolean` durch einen Aufruf von `set`.

```
private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
}
```

Die Tests wurden immer noch alle bestanden. Weil `set` durch diese Änderung in dem `BooleanArgumentMarshaller` implementiert wurde, entfernte ich die `setBoolean`-Methode aus der Basisklasse `ArgumentMarshaller`.

Beachten Sie, dass die abstrakte `set`-Funktion ein `String`-Argument übernimmt, das aber von der Implementierung in dem `BooleanArgumentMarshaller` einfach ignoriert wird. Ich fügte dieses Argument ein, weil ich wusste, dass der `StringArgumentMarshaller` und der `IntegerArgumentMarshaller` es *benutzen würden*.

Als Nächstes wollte ich die `get`-Methode in den `BooleanArgumentMarshaller` verschieben. Derartige `get`-Funktionen zu verschieben ist immer hässlich, weil der Rückgabewert vom Typ `Object` sein muss und in diesem Fall ein `Cast` in `Boolean` erforderlich ist.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaller am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
}
```

Nur um diesen Code kompilierbar zu machen, fügte ich die `get`-Funktion zu `ArgumentMarshaller` hinzu.

```
private abstract class ArgumentMarshaller {
    ...

    public Object get() {
```

```

        return null;
    }
}

```

Dieser Code wurde kompiliert, scheiterte dann aber bei den Tests. Um die Tests wieder zu bestehen, musste ich einfach `get` zu einer abstrakten Methode machen und in `BooleanArgumentMarshaler` implementieren.

```

private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;

    ***

    public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}

```

Wieder wurden die Tests bestanden. Beide Funktionen, `get` und `set`, werden also von dem `BooleanArgumentMarshaler` aus ausgeführt! Dies erlaubte es mir, die alte `getBoolean`-Funktion aus `ArgumentMarshaler` zu entfernen, die geschützte `booleanValue`-Variable in die abgeleitete Klasse `BooleanArgumentMarshaler` zu verschieben und als `private` zu deklarieren.

Dann führte ich für `Strings` die analogen Änderungen aus. Ich verschob `set` und `get`, löschte die nicht mehr benötigten Funktionen und verschob die Variablen.

```

private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
        stringArgs.get(argChar).set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

***
public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : (String) am.get();
}

***

```

```
private abstract class ArgumentMarshaler {
    private int integerValue;

    public void setInteger(int i) {
        integerValue = i;
    }

    public int getInteger() {
        return integerValue;
    }

    public abstract void set(String s);

    public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
        stringValue = s;
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {

    public void set(String s) {

    }

    public Object get() {
        return null;
    }
}
```

Schließlich wiederholte ich den Prozess für `integers`. Dies war nur etwas komplizierter, weil `integers` geparkt werden müssen und die `parse`-Operation eine Ausnahme auslösen kann. Aber das Ergebnis ist besser, weil das ganze Konzept der `NumberFormatException` in dem `IntegerArgumentMarshaler` verborgen ist.

```
private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

...
private void setBooleanArg(char argChar) {
    try {
        booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
}

...
public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
}

...
private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
}

...
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        }
    }
}
```

```
    } catch (NumberFormatException e) {  
        throw new ArgsException();  
    }  
}  
  
public Object get() {  
    return intValue;  
}  
}
```

Natürlich wurden die Tests weiterhin bestanden. Als Nächstes eliminierte ich die drei verschiedenen Maps am Anfang des Algorithmus. Dadurch wurde das gesamte System viel generischer. Doch ich konnte die Maps nicht einfach löschen, weil ich dadurch das System kaputtgemacht hätte. Stattdessen fügte ich eine neue Map für den `ArgumentMarshaler` hinzu und änderte dann nacheinander die Methoden so, dass sie die neue Map anstelle der drei ursprünglichen Maps benutzten.

```
public class Args {  
    ***  
    private Map<Character, ArgumentMarshaler> booleanArgs =  
        new HashMap<Character, ArgumentMarshaler>();  
    private Map<Character, ArgumentMarshaler> stringArgs =  
        new HashMap<Character, ArgumentMarshaler>();  
    private Map<Character, ArgumentMarshaler> intArgs =  
        new HashMap<Character, ArgumentMarshaler>();  
    private Map<Character, ArgumentMarshaler> marshalers =  
        new HashMap<Character, ArgumentMarshaler>();  
    ***  
    private void parseBooleanSchemaElement(char elementId) {  
        ArgumentMarshaler m = new BooleanArgumentMarshaler();  
        booleanArgs.put(elementId, m);  
        marshalers.put(elementId, m);  
    }  
  
    private void parseIntegerSchemaElement(char elementId) {  
        ArgumentMarshaler m = new IntegerArgumentMarshaler();  
        intArgs.put(elementId, m);  
        marshalers.put(elementId, m);  
    }  
  
    private void parseStringSchemaElement(char elementId) {  
        ArgumentMarshaler m = new StringArgumentMarshaler();  
        stringArgs.put(elementId, m);  
        marshalers.put(elementId, m);  
    }  
}
```

Natürlich wurden die Tests alle immer noch bestanden. Als Nächstes änderte ich `isBooleanArg` aus

```
private boolean isBooleanArg(char argChar) {  
    return booleanArgs.containsKey(argChar);  
}
```

in

```
private boolean isBooleanArg(char argChar) {  
    ArgumentMarshaler m = marshalers.get(argChar);  
    return m instanceof BooleanArgumentMarshaler;  
}
```

Die Tests wurden immer noch bestanden. Deshalb änderte ich `isIntArg` und `isStringArg` analog.

```
private boolean isIntArg(char argChar) {  
    ArgumentMarshaler m = marshalers.get(argChar);  
    return m instanceof IntegerArgumentMarshaler;  
}  
  
private boolean isStringArg(char argChar) {  
    ArgumentMarshaler m = marshalers.get(argChar);  
    return m instanceof StringArgumentMarshaler;  
}
```

Die Tests wurden immer noch bestanden. Deshalb eliminierte ich alle doppelten Aufrufe von `marshalers.get`.

```
private boolean setArgument(char argChar) throws ArgsException {  
    ArgumentMarshaler m = marshalers.get(argChar);  
    if (isBooleanArg(m))  
        setBooleanArg(argChar);  
    else if (isStringArg(m))  
        setStringArg(argChar);  
    else if (isIntArg(m))  
        setIntArg(argChar);  
    else  
        return false;  
  
    return true;  
}  
  
private boolean isIntArg(ArgumentMarshaler m) {  
    return m instanceof IntegerArgumentMarshaler;  
}  
  
private boolean isStringArg(ArgumentMarshaler m) {  
    return m instanceof StringArgumentMarshaler;  
}
```

```
}  
  
private boolean isBooleanArg(ArgumentMarshaler m) {  
    return m instanceof BooleanArgumentMarshaler;  
}
```

Jetzt gab es auch keinen guten Grund mehr für die drei `isxxxArg`-Methoden. Deshalb fügte ich sie inline ein:

```
private boolean setArgument(char argChar) throws ArgsException {  
    ArgumentMarshaler m = marshalers.get(argChar);  
    if (m instanceof BooleanArgumentMarshaler)  
        setBooleanArg(argChar);  
    else if (m instanceof StringArgumentMarshaler)  
        setStringArg(argChar);  
    else if (m instanceof IntegerArgumentMarshaler)  
        setIntArg(argChar);  
    else  
        return false;  
  
    return true;  
}
```

Als Nächstes begann ich, die `marshalers`-Map in den `set`-Funktionen zu nutzen und damit die Nutzung der anderen drei Maps abzuklemmen. Ich begann mit den `booleans`.

```
private boolean setArgument(char argChar) throws ArgsException {  
    ArgumentMarshaler m = marshalers.get(argChar);  
    if (m instanceof BooleanArgumentMarshaler)  
        setBooleanArg(m);  
    else if (m instanceof StringArgumentMarshaler)  
        setStringArg(argChar);  
    else if (m instanceof IntegerArgumentMarshaler)  
        setIntArg(argChar);  
    else  
        return false;  
  
    return true;  
}  
  
***  
private void setBooleanArg(ArgumentMarshaler m) {  
    try {  
        m.set("true"); // war: booleanArgs.get(argChar).set("true");  
    } catch (ArgsException e) {  
    }  
}
```

Die Tests wurden immer noch bestanden. Deshalb wiederholte ich diese Änderungen für Strings und Integers. Dies ermöglichte es mir, einigen hässlichen Code zur Ausnahmebehandlung in die `setArgument`-Funktion zu integrieren.

```
private boolean setArgument(char argChar) throws ArgException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgException();
    } catch (ArgException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgException {
    currentArgument++;
    try {
        m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgException();
    }
}
```


Ich stand kurz vor dem Punkt, an dem ich die drei alten Maps entfernen konnte. Zuerst musste ich die `getBoolean`-Funktion von

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
}
```

in

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}
```

ändern. Diese letzte Änderung kommt vielleicht etwas überraschend. Warum beschloss ich plötzlich, die `ClassCastException` abzufangen? Der Grund ist, dass ich einen Satz von Unit-Tests und einen separaten Satz von Acceptance-Tests in FitNesse geschrieben hatte. Es zeigte sich, dass der FitNesse-Test dafür sorgte, dass man ein `false` erhält, wenn `getBoolean` mit einem nicht-`boolean` Argument aufgerufen wird. Die Unit-Tests tun dies nicht. Bis zu diesem Punkt hatte ich nur die Unit-Tests ausgeführt. (Um weitere derartige Überraschungen auszuschließen, fügte ich einen neuen Unit-Test hinzu, der alle FitNesse-Tests aufrief.)

Diese letzte Änderung ermöglichte es mir, eine weitere Verwendung der `boolean` Map zu entfernen:

```
private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
}
```

Und jetzt können wir die `boolean` Map löschen.

```
public class Args {
    ...
    private Map<Character, ArgumentMarshaler> booleanArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> stringArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
        new HashMap<Character, ArgumentMarshaler>();
}
```

```
private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
```

```
***
```

Als Nächstes verschob ich die `String`- und `Integer`-Argumente auf dieselbe Weise und bereinigte die `booleans` ein wenig.

```
private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
}
```

```
private void parseIntegerSchemaElement(char elementId) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
}
```

```
private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
}
```

```
***
```

```
public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}
```

```
public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}
```

```
***
```

```
public class Args {
```

```
***
```

```
    private Map<Character, ArgumentMarshaler> stringArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> intArgs =
        new HashMap<Character, ArgumentMarshaler>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
```

```
***
```

Als Nächstes fügte ich die drei `parse`-Methoden inline ein, weil sie nicht mehr viel tun:

```
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
}
```

Okay, werfen wir jetzt wieder einen Blick auf das gesamte Bild. Listing 14.12 zeigt die gegenwärtige Form der Args-Klasse.

Listing 14.12: Args.java (nach dem ersten Refactoring)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
    private String schema;
    private String[] args;
    private boolean valid = true;
    private Set<Character> unexpectedArguments = new TreeSet<Character>();
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private int currentArgument;
    private char errorArgumentId = '\\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    private enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT
    }

    public Args(String schema, String[] args) throws ParseException {
        this.schema = schema;
        this.args = args;
        valid = parse();
    }

    private boolean parse() throws ParseException {
        if (schema.length() == 0 && args.length == 0)
            return true;
    }
}
```

```
    parseSchema();
    try {
        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}

private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
        if (element.length() > 0) {
            String trimmedElement = element.trim();
            parseSchemaElement(trimmedElement);
        }
    }
    return true;
}

private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
}

private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
        throw new ParseException(
            "Bad character: " + elementId + "in Args format: " + schema, 0);
    }
}

private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("");
}

private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
}
```

```
private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
        String arg = args[currentArgument];
        parseArgument(arg);
    }
    return true;
}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
```

```
private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
        parameter = args[currentArgument];
        m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
        m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

private void setBooleanArg(ArgumentMarshaler m) {
    try {
        m.set("true");
    } catch (ArgsException e) {
    }
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
```

```
        return unexpectedArgumentMessage();
    case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                               errorArgumentId);
    case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                               errorArgumentId, errorParameter);
    case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                               errorArgumentId);
    }
    return "";
}

private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
        message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
}

public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}

public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}

public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}
```

```
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}

public boolean isValid() {
    return valid;
}

private class ArgsException extends Exception {
}

private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
        booleanValue = true;
    }

    public Object get() {
        return booleanValue;
    }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
        stringValue = s;
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }
}
```



```
    }  
  }  
  
  public Object get() {  
    return intValue;  
  }  
}  
}
```

Nach all der vielen Arbeit ist dies ein wenig enttäuschend. Die Struktur ist ein bisschen besser, aber am Anfang stehen immer noch diese vielen Variablen; und es gibt immer noch eine schreckliche Fallunterscheidung in `setArgument`; und alle diese `set`-Funktionen sind wirklich hässlich, ganz zu schweigen von der Fehlerverarbeitung. Wir haben noch viel Arbeit vor uns.

Ich möchte die Fallunterscheidung in `setArgument` loswerden [G23]. Am liebsten würde ich in `setArgument` einen einzigen Aufruf von `ArgumentMarshaler.set` verwenden. Dies bedeutet, dass ich `setIntArg`, `setStringArg` und `setBooleanArg` in die entsprechenden abgeleiteten Klassen von `ArgumentMarshaler` verschieben muss. Aber es gibt ein Problem.

Ein genauer Blick auf `setIntArg` zeigt, dass die Funktion zwei Instanzvariablen verwendet: `args` und `currentArg`. Wenn ich `setIntArg` nach unten in `BooleanArgumentMarshaler` verschiebe, muss ich sowohl `args` als auch `currentArgs` als Funktionsargumente übergeben. Das ist schmutzig [F1]. Ich übergebe lieber ein Argument als zwei. Glücklicherweise gibt es eine einfache Lösung. Wir können das `args`-Array in eine Liste umwandeln und einen Iterator an die `set`-Funktionen übergeben. Für den folgenden Code benötigte ich zehn Schritte. Nach jedem Schritte wurden alle Tests bestanden. Aber ich zeige Ihnen nur das Ergebnis. Sie sollten die meisten kleinen Schritte selbst erschließen können.

```
public class Args {  
  private String schema;  
  private String[] args;  
  private boolean valid = true;  
  private Set<Character> unexpectedArguments = new TreeSet<Character>();  
  private Map<Character, ArgumentMarshaler> marshalers =  
    new HashMap<Character, ArgumentMarshaler>();  
  private Set<Character> argsFound = new HashSet<Character>();  
  private Iterator<String> currentArgument;  
  private char errorArgumentId = '\0';  
  private String errorParameter = "TILT";  
  private ErrorCode errorCode = ErrorCode.OK;  
  private List<String> argsList;  
  
  private enum ErrorCode {  
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT  
  }  
}
```

```

public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    argList = Arrays.asList(args);
    valid = parse();
}

private boolean parse() throws ParseException {
    if (schema.length() == 0 && argList.size() == 0)
        return true;
    parseSchema();
    try {
        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}

---
private boolean parseArguments() throws ArgsException {
    for (currentArgument = argList.iterator(); currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
    }

    return true;
}

---
private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
        parameter = currentArgument.next();
        m.set(parameter);
    } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
    } catch (ArgsException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw e;
    }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
        m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
    }
}

```

Dies waren einfache Änderungen, nach denen jeweils alle Tests bestanden wurden. Jetzt können wir beginnen, die `set`-Funktionen in die entsprechenden abgeleiteten Klassen zu verschieben. Zuerst muss ich `setArgument` wie folgt ändern:

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);
        else
            return false;
    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}
```

Diese Änderung ist wichtig, weil wir die `if-else`-Kette komplett beseitigen wollen. Deshalb müssen wir auch die Fehlerbedingung herausnehmen.

Jetzt können wir mit der Verschiebung der `set`-Funktionen beginnen. Die `setBooleanArg`-Funktion ist trivial; deshalb wollen wir mit ihr anfangen. Unser Ziel besteht darin, die `setBooleanArg`-Funktion so zu ändern, dass sie einfach an den `BooleanArgumentMarshaler` weitergibt.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            setBooleanArg(m, currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);

    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
}
```

```

    return true;
}
--
private void setBooleanArg(ArgumentMarshaler m,
                          Iterator<String> currentArgument)
    throws ArgException {
    try {
        m.set("true");
    } catch (ArgException e) {
    }
}

```

Haben wir die Ausnahmebehandlung nicht gerade erst eingefügt? Dinge einzufügen, damit Sie sie wieder herausnehmen können, kommt beim Refactoring häufig vor. Die Kleinheit der Schritte und die Forderung, dass die Tests immer bestanden werden müssen, hat zur Folge, dass Sie Dinge sehr viel herumschieben. Das Refactoring ähnelt der Lösung von Rubicks Würfel. Es sind zahlreiche kleine Schritte erforderlich, um ein großes Ziel zu erreichen. Jeder Schritt ermöglicht den nächsten.

Warum übergeben wir diesen Iterator, wenn `setBooleanArg` ihn mit Sicherheit nicht benötigt? Weil `setIntArg` und `setStringArg` ihn brauchen! Und weil ich alle drei Funktionen über eine abstrakte Methode in `ArgumentMarshaler` ansteuern möchte, muss ich den Iterator auch an `setBooleanArg` übergeben.

Jetzt ist `setBooleanArg` nutzlos. Gäbe es eine `set`-Funktion in `ArgumentMarshaler`, könnten wir sie direkt aufrufen. Deshalb sollten wir diese Funktion jetzt erstellen! Der erste Schritt besteht darin, die neue abstrakte Methode in `ArgumentMarshaler` einzufügen.

```

private abstract class ArgumentMarshaler {
    public abstract void set(Iterator<String> currentArgument)
        throws ArgException;
    public abstract void set(String s) throws ArgException;
    public abstract Object get();
}

```

Natürlich funktionieren jetzt alle abgeleiteten Klassen nicht mehr. Deshalb müssen wir die neue Methode in jeder implementieren.

```

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(Iterator<String> currentArgument) throws ArgException {
        booleanValue = true;
    }

    public void set(String s) {
        booleanValue = true;
    }
}

```

```
}

    public Object get() {
        return booleanValue;
    }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgException {
    }

    public void set(String s) {
        stringValue = s;
    }

    public Object get() {
        return stringValue;
    }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgException {
    }

    public void set(String s) throws ArgException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

Und jetzt können wir `setBooleanArg` eliminieren!

```
private boolean setArgument(char argChar) throws ArgException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            m.set(currentArgument);
    }
```

```

        else if (m instanceof StringArgumentMarshaler)
            setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
            setIntArg(m);

    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

```

Die Tests werden alle bestanden, und die `set`-Funktion wird an `BooleanArgumentMarshaler` weitergereicht!

Jetzt können wir dieselben Änderungen für `Strings` und `Integers` durchführen.

```

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        if (m instanceof BooleanArgumentMarshaler)
            m.set(currentArgument);
        else if (m instanceof StringArgumentMarshaler)
            m.set(currentArgument);
        else if (m instanceof IntegerArgumentMarshaler)
            m.set(currentArgument);

    } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
    }
    return true;
}

---

private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_STRING;
            throw new ArgsException();
        }
    }
}

```

```
public void set(String s) {
}

public Object get() {
    return stringValue;
}
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            set(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (ArgsException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw e;
        }
    }

    public void set(String s) throws ArgsException {
        try {
            intValue = Integer.parseInt(s);
        } catch (NumberFormatException e) {
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}
```

Und jetzt die Krönung: Die Fallunterscheidung kann verschwinden! Touche!

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        m.set(currentArgument);
        return true;
    } catch (ArgsException e) {
        valid = false;
    }
}
```

```

        errorArgumentId = argChar;
        throw e;
    }
}

```

Jetzt können wir einige störende Funktionen in `IntegerArgumentMarshaler` entfernen und die Klasse ein wenig säubern.

```

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_INTEGER;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_INTEGER;
            throw new ArgsException();
        }
    }

    public Object get() {
        return intValue;
    }
}

```

Außerdem können wir `ArgumentMarshaler` in ein Interface umwandeln.

```

private interface ArgumentMarshaler {
    void set(Iterator<String> currentArgument) throws ArgsException;
    Object get();
}

```

Jetzt wollen wir prüfen, wie leicht wir einen neuen Argumenttyp zu unserer Struktur hinzufügen können. Die Aktionen sollten sehr wenige Änderungen erfordern, und diesen Änderungen sollten isoliert sein. Zuerst fügen wir einen neuen Testfall hinzu, um zu prüfen, ob das `double`-Argument korrekt funktioniert.

```

public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "42.3"});
    assertTrue(args.isValid());
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
}

```


Jetzt räumen wir den Schema-Parsing-Code auf und fügen die ##-Erkennung für den double-Argumenttyp hinzu.

```
private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
        throw new ParseException(String.format(
            "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}
```

Als Nächstes schreiben wir die DoubleArgumentMarshaler-Klasse.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ErrorCode.MISSING_DOUBLE;
            throw new ArgsException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ErrorCode.INVALID_DOUBLE;
            throw new ArgsException();
        }
    }

    public Object get() {
        return doubleValue;
    }
}
```

Dies zwingt uns, einen neuen ErrorCode hinzuzufügen.

```
private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Und wir brauchen eine `getDouble`-Funktion.

```
public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
        return 0.0;
    }
}
```

Und alle Tests werden bestanden! Dies war ziemlich schmerzlos. Jetzt wollen wir prüfen, ob die Fehlerverarbeitung korrekt funktioniert. Der nächste Testfall prüft, ob ein Fehler deklariert ist, wenn ein nicht parsbarer String in ein `##`-Argument übergeben wird.

```
public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x", "Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
        args.errorMessage());
}

---

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return unexpectedArgumentMessage();
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                errorArgumentId);
    }
    return "";
}
```

Und die Tests werden bestanden. Der nächste Test prüft, ob das Fehlen eines `double`-Arguments korrekt erkannt wird.

```
public void testMissingDouble() throws Exception {
    Args args = new Args("x##", new String[]{"-x"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
        args.errorMessage());
}
```

Dieser Test wird erwartungsgemäß bestanden. Ich habe ihn einfach der Vollständigkeit halber geschrieben.

Der Ausnahme-Code ist ziemlich hässlich und gehört eigentlich nicht in die `Args`-Klasse. Wir lösen auch eine `ParseException` aus, die nicht zu unserem Code gehört. Deshalb wollen wir alle Ausnahmen in einer einzigen `ArgsException`-Klasse zusammenfassen und in ein separates Modul einfügen.

```
public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    public ArgsException() {}

    public ArgsException(String message) {super(message);}

    public enum ErrorCode {
        OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
        MISSING_DOUBLE, INVALID_DOUBLE}
}

---
public class Args {
    ...
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
    private List<String> argsList;

    public Args(String schema, String[] args) throws ArgsException {
        this.schema = schema;
        argsList = Arrays.asList(args);
        valid = parse();
    }

    private boolean parse() throws ArgsException {
```

```

    if (schema.length() == 0 && argsList.size() == 0)
        return true;
    parseSchema();
    try {
        parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
}

private boolean parseSchema() throws ArgsException {
    ...
}

private void parseSchemaElement(String element) throws ArgsException {
    ...
    else
        throw new ArgsException(
            String.format("Argument: %c has invalid format: %s.",
                elementId, elementTail));
}

private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
        throw new ArgsException(
            "Bad character:" + elementId + "in Args format: " + schema);
    }
}

...

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        unexpectedArguments.add(argChar);
        errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
        valid = false;
    }
}

...

private class StringArgumentMarshaler implements ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
        try {
            stringValue = currentArgument.next();
        } catch (NoSuchElementException e) {

```

```
        errorCode = ArgumentException.ErrorCode.MISSING_STRING;
        throw new ArgumentException();
    }
}

public Object get() {
    return stringValue;
}
}

private class IntegerArgumentMarshaler implements ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgumentException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ArgumentException.ErrorCode.MISSING_INTEGER;
            throw new ArgumentException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ArgumentException.ErrorCode.INVALID_INTEGER;
            throw new ArgumentException();
        }
    }

    public Object get() {
        return intValue;
    }
}

private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgumentException {
        String parameter = null;
        try {
            parameter = currentArgument.next();
            doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
            errorCode = ArgumentException.ErrorCode.MISSING_DOUBLE;
            throw new ArgumentException();
        } catch (NumberFormatException e) {
            errorParameter = parameter;
            errorCode = ArgumentException.ErrorCode.INVALID_DOUBLE;
            throw new ArgumentException();
        }
    }
}
```

```

    }

    public Object get() {
        return doubleValue;
    }
}
}

```

Dies ist nett. Jetzt wird nur eine Ausnahme von `Args` ausgelöst: `ArgsException`. Indem wir `ArgsException` in ein separates Modul verlagern, können wir einen großen Teil des Codes zur Fehlerbehandlung aus dem `Args`-Modul herausnehmen und in diesem Modul unterbringen. Es stellt einen natürlichen und offensichtlichen Platz für all diesen Code zur Verfügung und hilft uns wirklich dabei, das `Args`-Modul aufzuräumen.

Jetzt haben wir den Code zur Ausnahme- und Fehlerbehandlung vollständig von dem `Args`-Modul getrennt (siehe Listings 14.13 bis 14.16). Dies wurde durch eine Folge von über 30 kleinen Schritten erreicht, wobei nach jedem Schritt dafür gesorgt wurde, dass alle Tests bestanden wurden.

Listing 14.13: `ArgsTest.java`

```

package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
    public void testCreateWithNoSchemaOrArguments() throws Exception {
        Args args = new Args("", new String[0]);
        assertEquals(0, args.cardinality());
    }

    public void testWithNoSchemaButWithOneArgument() throws Exception {
        try {
            new Args("", new String[]{"-x"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                e.getErrorCode());
            assertEquals('x', e.getErrorArgumentId());
        }
    }

    public void testWithNoSchemaButWithMultipleArguments() throws Exception {
        try {
            new Args("", new String[]{"-x", "-y"});
            fail();
        } catch (ArgsException e) {
            assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,

```

```
        e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }

}

public void testNonLetterSchema() throws Exception {
    try {
        new Args("*", new String[]{});
        fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
            e.getErrorCode());
        assertEquals('*', e.getErrorArgumentId());
    }
}

public void testInvalidArgumentFormat() throws Exception {
    try {
        new Args("f~", new String[]{});
        fail("Args constructor should have throws exception");
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
        assertEquals('f', e.getErrorArgumentId());
    }
}

public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args("x", new String[]{"-x"});
    assertEquals(1, args.cardinality());
    assertEquals(true, args.getBoolean('x'));
}

public void testSimpleStringPresent() throws Exception {
    Args args = new Args("x*", new String[]{"-x", "param"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals("param", args.getString('x'));
}

public void testMissingStringArgument() throws Exception {
    try {
        new Args("x*", new String[]{"-x"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }
}
```

```
public void testSpacesInFormat() throws Exception {
    Args args = new Args("x, y", new String[]{"-xy"});
    assertEquals(2, args.cardinality());
    assertTrue(args.has('x'));
    assertTrue(args.has('y'));
}

public void testSimpleIntPresent() throws Exception {
    Args args = new Args("x#", new String[]{"-x", "42"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42, args.getInt('x'));
}

public void testInvalidInteger() throws Exception {
    try {
        new Args("x#", new String[]{"-x", "Forty two"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
        assertEquals("Forty two", e.getErrorParameter());
    }
}

public void testMissingInteger() throws Exception {
    try {
        new Args("x#", new String[]{"-x"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }
}

public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x###", new String[]{"-x", "42.3"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
}

public void testInvalidDouble() throws Exception {
    try {
        new Args("x###", new String[]{"-x", "Forty two"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }
}
```



```
        assertEquals("Forty two", e.getErrorParameter());
    }
}

public void testMissingDouble() throws Exception {
    try {
        new Args("x#\"", new String[]{"-x"});
        fail();
    } catch (ArgsException e) {
        assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
        assertEquals('x', e.getErrorArgumentId());
    }
}
}
```

Listing 14.14: ArgsExceptionTest.java

```
public class ArgsExceptionTest extends TestCase {
    public void testUnexpectedMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                            'x', null);
        assertEquals("Argument -x unexpected.", e.errorMessage());
    }

    public void testMissingStringMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                            'x', null);
        assertEquals("Could not find string parameter for -x.", e.errorMessage());
    }

    public void testInvalidIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                            'x', "Forty two");
        assertEquals("Argument -x expects an integer but was 'Forty two'.",
                    e.errorMessage());
    }

    public void testMissingIntegerMessage() throws Exception {
        ArgsException e =
            new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
        assertEquals("Could not find integer parameter for -x.", e.errorMessage());
    }

    public void testInvalidDoubleMessage() throws Exception {
        ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                            'x', "Forty two");
        assertEquals("Argument -x expects a double but was 'Forty two'.",
                    e.errorMessage());
    }
}
```

```

}

public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                       'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
}
}

```

Listing 14.15: ArgsException.java

```

public class ArgsException extends Exception {
    private char errorArgumentId = '\0';
    private String errorParameter = "TILT";
    private ErrorCode errorCode = ErrorCode.OK;

    public ArgsException() {}

    public ArgsException(String message) {super(message);}

    public ArgsException(ErrorCode errorCode) {
        this.errorCode = errorCode;
    }

    public ArgsException(ErrorCode errorCode, String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
    }

    public ArgsException(ErrorCode errorCode, char errorArgumentId,
                        String errorParameter) {
        this.errorCode = errorCode;
        this.errorParameter = errorParameter;
        this.errorArgumentId = errorArgumentId;
    }

    public char getErrorArgumentId() {
        return errorArgumentId;
    }

    public void setErrorArgumentId(char errorArgumentId) {
        this.errorArgumentId = errorArgumentId;
    }

    public String getErrorParameter() {
        return errorParameter;
    }

    public void setErrorParameter(String errorParameter) {
        this.errorParameter = errorParameter;
    }
}

```

```
}

public ErrorCode getErrorCode() {
    return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
}

public String errorMessage() throws Exception {
    switch (errorCode) {
        case OK:
            throw new Exception("TILT: Should not get here.");
        case UNEXPECTED_ARGUMENT:
            return String.format("Argument -%c unexpected.", errorArgumentId);
        case MISSING_STRING:
            return String.format("Could not find string parameter for -%c.",
                errorArgumentId);
        case INVALID_INTEGER:
            return String.format("Argument -%c expects an integer but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_INTEGER:
            return String.format("Could not find integer parameter for -%c.",
                errorArgumentId);
        case INVALID_DOUBLE:
            return String.format("Argument -%c expects a double but was '%s'.",
                errorArgumentId, errorParameter);
        case MISSING_DOUBLE:
            return String.format("Could not find double parameter for -%c.",
                errorArgumentId);
    }
    return "";
}

public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE
}
```

Listing 14.16: Args.java

```
public class Args {
    private String schema;
    private Map<Character, ArgumentMarshaler> marshalers =
        new HashMap<Character, ArgumentMarshaler>();
    private Set<Character> argsFound = new HashSet<Character>();
    private Iterator<String> currentArgument;
    private List<String> argsList;
```

```
public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    parse();
}

private void parse() throws ArgsException {
    parseSchema();
    parseArguments();
}

private boolean parseSchema() throws ArgsException {
    for (String element : schema.split(",")) {
        if (element.length() > 0) {
            parseSchemaElement(element.trim());
        }
    }
    return true;
}

private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
        marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
        marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
        marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
        marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
        throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                                elementId, elementTail);
}

private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
        throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                                elementId, null);
    }
}

private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
        String arg = currentArgument.next();
        parseArgument(arg);
    }
}
```

```
}

private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
        parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
        parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
        argsFound.add(argChar);
    else {
        throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                                argChar, null);
    }
}

private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
        return false;
    try {
        m.set(currentArgument);
        return true;
    } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
    }
}

public int cardinality() {
    return argsFound.size();
}

public String usage() {
    if (schema.length() > 0)
        return "-[" + schema + "]";
    else
        return "";
}

public boolean getBoolean(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
        b = am != null && (Boolean) am.get();
    }
```

```

    } catch (ClassCastException e) {
        b = false;
    }
    return b;
}

public String getString(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
        return "";
    }
}

public int getInt(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
        return 0;
    }
}

public double getDouble(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
        return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
        return 0.0;
    }
}

public boolean has(char arg) {
    return argsFound.contains(arg);
}
}

```

Die Mehrzahl der Änderungen der `Args`-Klasse bestand aus Löschungen. Ein großer Teil des Codes wurde einfach aus `Args` herausgenommen und in `ArgsException` eingefügt. Hübscher. Wir haben auch alle `ArgumentMarshalers` in separate Dateien eingefügt. Hübscher!

Gutes Software-Design bedeutet einfach nur, den Code in geeignete Module zu zerlegen – also entsprechende Plätze für verschiedene Arten von Code zur Verfügung zu stellen. Diese Trennung der Zuständigkeiten vereinfacht den Code und macht ihn verständlicher und wartungsfreundlicher.

Besonders interessant ist die `errorMessage`-Methode von `ArgumentException`. Es war offensichtlich eine Verletzung des Single-Responsibility-Prinzips (SRP), die Formatierung von Fehlermeldungen in `Args` einzufügen. `Args` sollte sich um die Verarbeitung der Argumente, nicht um die Formatierung von Fehlermeldungen kümmern. Doch ist es wirklich sinnvoll, den Code für die Formatierung von Fehlermeldungen in `ArgumentException` unterzubringen?

Offen gesagt, es ist ein Kompromiss. Benutzer, denen die Fehlermeldungen in `ArgumentException` nicht gefallen, müssen ihre eigenen schreiben. Aber die Bequemlichkeit vorgefertigter Fehlermeldungen ist nicht zu verachten.

Inzwischen sollte Ihnen klar sein, dass wir der endgültigen Lösung vom Anfang dieses Kapitels sehr nahe gekommen sind. Ich überlasse Ihnen die letzten Umformungen als Übung.

14.4 Zusammenfassung

Es reicht nicht aus, dass Code funktioniert. Code, der funktioniert, enthält oft schwere Mängel. Programmierer, die sich mit Code zufriedengeben, der einfach nur funktioniert, verhalten sich unprofessionell. Wenn Sie fürchten, keine Zeit für die Verbesserung der Struktur und des Designs Ihres Codes zu haben, kann ich Ihnen nur sagen, dass ein Entwicklungsprojekt durch nichts so grundlegend und langfristig negativ beeinflusst wird wie durch schlechten Code. Schlechte Pläne können geändert werden; schlechte Anforderungen können umdefiniert werden; schlechte Zusammenarbeit in einem Team kann geheilt werden. Aber schlechter Code rottet und fault vor sich hin und wird zu einem unerbittlichen Gewicht, das das Team herunterzieht. Immer wieder habe ich erlebt, wie Teamarbeit nur noch schleichend vorankam, weil das Team in seiner Eile einen bösartigen Morast von Code erzeugt hatte, der für immer sein weiteres Schicksal bestimmte.

Natürlich kann schlechter Code bereinigt werden. Aber dies ist sehr teuer. Wenn Code vor sich hin rottet, verhaken sich die Module immer stärker miteinander, wodurch zahlreiche verborgene und verzwickte Abhängigkeiten entstehen. Alte Abhängigkeiten zu entdecken und zu beseitigen, ist ein langer und mühsamer Prozess. Andererseits ist es relativ einfach, Code sauber zu halten. Wenn Sie ein Modul am Morgen verunreinigt haben, können Sie es einfach am Nachmittag bereinigen. Oder noch besser: Haben Sie vor fünf Minuten ein kleines Chaos angerichtet, können Sie es sehr leicht jetzt im Moment beseitigen.

Deshalb besteht die Lösung darin, Ihren Code laufend so sauber und einfach wie möglich zu halten. Lassen Sie einfach nicht zu, dass er anfängt, zu verrotten!

JUnit im Detail



JUnit gehört zu den berühmtesten Java-Frameworks. Für ein Framework ist es einfach konzipiert, präzise definiert und elegant implementiert. Aber wie sieht der Code aus? In diesem Kapitel kritisieren wir ein Beispiel aus dem JUnit-Framework.

15.1 Das JUnit-Framework

JUnit hatte viele Autoren. Die erste Version wurde von Kent Beck und Eric Gamma zusammen auf einem Flug nach Atlanta entwickelt. Kent wollte Java lernen, und Eric wollte das Smalltalk Testing Framework von Kent kennen lernen. »Was könnte für einige Geeks, die auf engem Raum zusammenhocken, natürlicher sein, als ihre Laptops herauszunehmen und anfangen zu codieren?« ([Beck94], S. 43). Nach drei Stunden Arbeit in großer Höhe hatten sie die Grundlagen von JUnit geschrieben.

Das Modul, das wir uns anschauen, ist ein pfiffiges Stück Code, der hilft, Fehler beim Vergleich von Strings zu erkennen. Dieses Modul wird als `ComparisonCompactor` bezeichnet. Sind zwei unterschiedliche Strings, wie etwa `ABCDE` und `ABXDE`, gegeben, zeigt der Code den Unterschied mit einem anderen String an, etwa in der Form `<... B[X]D...>`.

Ich könnte ihn noch ausführlicher erklären, aber die Testfälle machen dies besser. Aus Listing 15.1 gehen die Anforderungen an dieses Modul bis ins Detail hervor. Wenn Sie schon dabei sind, sollten Sie die Struktur der Tests kritisch beurteilen. Könnten sie einfacher oder klarer sein?

Listing 15.1: ComparisonCompactorTest.java

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

    public void testMessage() {
        String failure= new ComparisonCompactor(0, "b", "c").compact("a");
        assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
    }

    public void testStartSame() {
        String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
        assertEquals("expected:<b[a]> but was:<b[c]>", failure);
    }

    public void testEndSame() {
        String failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
        assertEquals("expected:<[a]b> but was:<[c]b>", failure);
    }

    public void testSame() {
        String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
        assertEquals("expected:<ab> but was:<ab>", failure);
    }

    public void testNoContextStartAndEndSame() {
        String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
        assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
    }

    public void testStartAndEndContext() {
        String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
        assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
    }

    public void testStartAndEndContextWithEllipses() {
        String failure=
            new ComparisonCompactor(1, "abcde", "abfde").compact(null);
        assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
    }
}
```

```
public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
}

public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
}

public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
}

public void testComparisonErrorOverlappingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
    assertEquals("expected:<...[]...> but was:<...[b]...>", failure);
}

public void testComparisonErrorOverlappingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
}

public void testComparisonErrorOverlappingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[]...>", failure);
}

public void testComparisonErrorOverlappingMatches2Context() {
    String failure=
        new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
}

public void testComparisonErrorWithActualNull() {
    String failure= new ComparisonCompactor(0, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}

public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
}

public void testComparisonErrorWithExpectedNull() {
    String failure= new ComparisonCompactor(0, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
}
```

```
public void testComparisonErrorWithExpectedNullContext() {
    String failure= new ComparisonCompactor(2, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
}

public void testBug609972() {
    String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
    assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
}
}
```

Ich führte mit diesen Tests eine Code-Coverage-Analyse von `ComparisonCompactor` durch. Der Code ist zu 100 Prozent abgedeckt. Jede Zeile des Codes, jede `if`-Anweisung und jede `for`-Schleife wird von den Tests ausgeführt. Dies vermittelt mir ein großes Vertrauen, dass der Code funktioniert, und einen großen Respekt vor dem Können seiner Autoren.

Listing 15.2 enthält den Code von `ComparisonCompactor`. Sie sollten ihn etwas eingehender studieren. Meiner Meinung nach ist er übersichtlich partitioniert, ausdrucksstark genug und einfach strukturiert. Wenn Sie fertig sind, wollen wir uns gemeinsam über ihn hermachen.

Listing 15.2: `ComparisonCompactor.java` (Original)

```
package junit.framework;

public class ComparisonCompactor {

    private static final String ELLIPSIS = "...";
    private static final String DELTA_END = "]";
    private static final String DELTA_START = "[";

    private int fContextLength;
    private String fExpected;
    private String fActual;
    private int fPrefix;
    private int fSuffix;

    public ComparisonCompactor(int contextLength,
                               String expected,
                               String actual) {
        fContextLength = contextLength;
        fExpected = expected;
        fActual = actual;
    }

    public String compact(String message) {
        if (fExpected == null || fActual == null || areStringsEqual())
            return Assert.format(message, fExpected, fActual);
    }
}
```

```

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(fExpected);
    String actual = compactString(fActual);
    return Assert.format(message, expected, actual);
}

private String compactString(String source) {
    String result = DELTA_START +
        source.substring(fPrefix, source.length() -
            fSuffix + 1) + DELTA_END;
    if (fPrefix > 0)
        result = computeCommonPrefix() + result;
    if (fSuffix > 0)
        result = result + computeCommonSuffix();
    return result;
}

private void findCommonPrefix() {
    fPrefix = 0;
    int end = Math.min(fExpected.length(), fActual.length());
    for (; fPrefix < end; fPrefix++) {
        if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
            break;
    }
}

private void findCommonSuffix() {
    int expectedSuffix = fExpected.length() - 1;
    int actualSuffix = fActual.length() - 1;
    for (;
        actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
        actualSuffix--, expectedSuffix--) {
        if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
            break;
    }
    fSuffix = fExpected.length() - expectedSuffix;
}

private String computeCommonPrefix() {
    return (fPrefix > fContextLength ? ELLIPSIS : "") +
        fExpected.substring(Math.max(0, fPrefix - fContextLength),
            fPrefix);
}

private String computeCommonSuffix() {
    int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
        fExpected.length());
    return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
        (fExpected.length() - fSuffix + 1 < fExpected.length() -
            fContextLength ? ELLIPSIS : "");
}

```

```
}

private boolean areStringsEqual() {
    return fExpected.equals(fActual);
}
}
```

Möglicherweise haben Sie an diesem Modul einiges zu bemängeln. Es enthält einige lange Ausdrücke und einige seltsame +1s usw. Aber insgesamt ist dieses Modul ziemlich gut. Schließlich hätte es auch wie Listing 15.3 aussehen können.

Listing 15.3: ComparisonCompactor.java (nach Defactoring)

```
package junit.framework;

public class ComparisonCompactor {
    private int ctxt;
    private String s1;
    private String s2;
    private int pfx;
    private int sfx;

    public ComparisonCompactor(int ctxt, String s1, String s2) {
        this.ctxt = ctxt;
        this.s1 = s1;
        this.s2 = s2;
    }

    public String compact(String msg) {
        if (s1 == null || s2 == null || s1.equals(s2))
            return Assert.format(msg, s1, s2);

        pfx = 0;
        for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
            if (s1.charAt(pfx) != s2.charAt(pfx))
                break;
        }
        int sfx1 = s1.length() - 1;
        int sfx2 = s2.length() - 1;
        for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
            if (s1.charAt(sfx1) != s2.charAt(sfx2))
                break;
        }
        sfx = s1.length() - sfx1;
        String cmp1 = compactString(s1);
        String cmp2 = compactString(s2);
        return Assert.format(msg, cmp1, cmp2);
    }

    private String compactString(String s) {
        String result =
            "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    }
}
```

```

if (pfx > 0)
    result = (pfx > ctxt ? "... " : "") +
        s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
if (sfx > 0) {
    int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
    result = result + (s1.substring(s1.length() - sfx + 1, end) +
        (s1.length() - sfx + 1 < s1.length() - ctxt ? "... " : ""));
}
return result;
}
}

```

Obwohl die Autoren dieses Modul in sehr guter Verfassung hinterlassen haben, sagt uns die *Pfadfinder-Regel* (siehe Kapitel 1), dass wir es sauberer verlassen sollten, als wir es vorgefunden haben. Also: Wie können wir den ursprünglichen Code in Listing 15.2 verbessern?

Zunächst einmal stört mich das Präfix `f` der Member-Variablen [N6]. In den modernen Umgebungen ist diese Art von Codierung des Geltungsbereiches redundant. Deshalb wollen wir alle `fs` entfernen.

```

private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;

```

Als Nächstes finden wir am Anfang der `compact`-Funktion eine nicht eingekapselte Bedingung [G28].

```

public String compact(String message) {
    if (expected == null || actual == null || areStringsEqual())
        return Assert.format(message, expected, actual);

    findCommonPrefix();
    findCommonSuffix();
    String expected = compactString(this.expected);
    String actual = compactString(this.actual);
    return Assert.format(message, expected, actual);
}

```

Diese Bedingung sollte eingekapselt werden, um unsere Absicht zu verdeutlichen. Deshalb wollen wir eine Methode extrahieren, die den Zweck der Bedingung ausdrückt.

```

public String compact(String message) {
    if (shouldNotCompact())
        return Assert.format(message, expected, actual);
}

```

```
findCommonPrefix();  
findCommonSuffix();  
String expected = compactString(this.expected);  
String actual = compactString(this.actual);  
return Assert.format(message, expected, actual);  
}  
  
private boolean shouldNotCompact() {  
    return expected == null || actual == null || areStringsEqual();  
}
```

Ich finde die Notation `this.expected` und `this.actual` in der `compact`-Funktion störend. Sie wurde ergänzt, als wir den Namen von `fExpected` in `expected` geändert haben. Warum gibt es in dieser Funktion Variablen, die dieselben Namen wie die Member-Variablen haben? Repräsentieren sie nicht etwas anderes [N4]? Wir sollten die Namen eindeutig machen.

```
String compactExpected = compactString(expected);  
String compactActual = compactString(actual);
```

Negativ formulierte Bedingungen sind etwas schwerer zu verstehen als positiv formulierte [G29]. Deshalb wollen wir die `if`-Anweisung auf den Kopf stellen und die Bedeutung der Bedingung umkehren.

```
public String compact(String message) {  
    if (canBeCompacted()) {  
        findCommonPrefix();  
        findCommonSuffix();  
        String compactExpected = compactString(expected);  
        String compactActual = compactString(actual);  
        return Assert.format(message, compactExpected, compactActual);  
    } else {  
        return Assert.format(message, expected, actual);  
    }  
}  
  
private boolean canBeCompacted() {  
    return expected != null && actual != null && !areStringsEqual();  
}
```

Der Name der Funktion wird ihrer Aufgabe nicht gerecht [N7]. Sie komprimiert die Strings nämlich nur, wenn `canBeCompacted` `true` zurückgibt. Deshalb verbirgt der einfache Name `compact` dieser Funktion den Nebeneffekt der Fehlerprüfung. Beachten Sie auch, dass die Funktion eine formatierte Meldung, nicht nur die komprimierten Strings zurückgibt. Deshalb sollte der Name der Funktion besser `formatCompactedComparison` lauten. Dies erleichterte das Lesen erheblich, wenn man das Funktionsargument einbezieht:

```
public String formatCompactedComparison(String message) {
```

Die Komprimierung des erwarteten und des tatsächlichen Strings selbst erfolgt im Body der `if`-Anweisung. Wir sollten sie als separate Methode namens `compactExpectedAndActual` extrahieren. Doch die gesamte Formatierung sollte von der `formatCompactedComparison`-Funktion geleistet werden. Die `compact...`-Funktion sollte nur für die Komprimierung zuständig sein [G30]. Deshalb zerlegen wir die ursprüngliche Funktion wie folgt:

```
...
private String compactExpected;
private String compactActual;
...

public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
        compactExpectedAndActual();
        return Assert.format(message, compactExpected, compactActual);
    } else {
        return Assert.format(message, expected, actual);
    }
}

private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}
```

Beachten Sie, dass wir deswegen `compactExpected` und `compactActual` zu Member-Variablen machen mussten. Mir gefällt nicht, wie die letzten beiden Zeilen der neuen Funktion Variablen zurückgeben, während die ersten beiden dies nicht tun. Hier werden keine konsistenten Konventionen verwendet [G11]. Deshalb sollten wir `findCommonPrefix` und `findCommonSuffix` so ändern, dass sie die Präfix- und Suffix-Werte zurückgeben.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

private int findCommonPrefix() {
    int prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
```



```
for (; prefixIndex < end; prefixIndex++) {
    if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
}
return prefixIndex;
}

private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
    return expected.length() - expectedSuffix;
}
```

Wir sollten auch die Namen der Member-Variablen ändern, damit sie etwas genauer sind [N1]; schließlich sind beide Indizes.

Eine sorgfältige Untersuchung von `findCommonSuffix` deckt eine *verborgene zeitliche Kopplung* auf [G31]. Sie hängt von der Tatsache ab, dass `prefixIndex` von `findCommonPrefix` berechnet wird. Würden diese beiden Funktionen in der falschen Reihenfolge aufgerufen, stünde uns eine schwierige Debugging-Sitzung bevor. Deshalb wollen wir diese zeitliche Kopplung verdeutlichen und `prefixIndex` als Argument an `findCommonSuffix` übergeben.

```
private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix(prefixIndex);
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--) {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
    return expected.length() - expectedSuffix;
}
```

Ich bin damit nicht wirklich glücklich. Die Übergabe von `prefixIndex` als Argument wirkt ein wenig willkürlich [G32]. Sie stellt zwar die Reihenfolge sicher, erklärt aber nicht, warum diese Reihenfolge erforderlich ist. Ein anderer Programmierer

könnte unsere Änderung rückgängig machen, weil es keinen Hinweis darauf gibt, dass der Parameter tatsächlich erforderlich ist. Deshalb wollen wir einen anderen Ansatz wählen.

```
private void compactExpectedAndActual() {
    findCommonPrefixAndSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (;
        actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
        actualSuffix--, expectedSuffix--)
    {
        if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
            break;
    }
    suffixIndex = expected.length() - expectedSuffix;
}

private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++)
        if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
            break;
}
```

Wir stellen die ursprüngliche Form von `findCommonPrefix` und `findCommonSuffix` wieder her, ändern den Namen von `findCommonSuffix` in `findCommonPrefixAndSuffix` und rufen in dieser Funktion zuallererst `findCommonPrefix` auf. Dadurch wird die zeitliche Abhängigkeit der beiden Funktionen sehr viel dramatischer ausgedrückt als bei der vorherigen Lösung. Diese Lösung zeigt auch auf, wie hässlich `findCommonPrefixAndSuffix` ist. Dies wollen wir jetzt bereinigen.

```
private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int suffixLength = 1;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
        if (charFromEnd(expected, suffixLength) !=
            charFromEnd(actual, suffixLength))
            break;
    }
    suffixIndex = suffixLength;
}
```

```
private char charFromEnd(String s, int i) {
    return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength < prefixLength ||
        expected.length() - suffixLength < prefixLength;
}
```

Dies ist viel besser. Der Code zeigt jetzt, dass der `suffixIndex` eigentlich die Länge des Suffixes angibt und unpassend benannt ist. Dasselbe gilt für `prefixIndex`, obwohl in diesem Fall »index« und »length« synonym sind. Doch selbst dann ist es konsistenter, »length« zu verwenden. Das Problem ist, dass die `suffixIndex`-Variable nicht null-basiert, sondern eins-basiert ist und deshalb keine echte Länge angibt. Dies ist auch der Grund für die vielen `+1s` in `computeCommonSuffix` [G33]. Wir wollen dies korrigieren. Listing 15.4 zeigt das Ergebnis.

Listing 15.4: `ComparisonCompactor.java` (Zwischenlösung)

```
public class ComparisonCompactor {
    ...
    private int suffixLength;
    ...
    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength))
                break;
        }
    }

    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }

    private boolean suffixOverlapsPrefix(int suffixLength) {
        return actual.length() - suffixLength <= prefixLength ||
            expected.length() - suffixLength <= prefixLength;
    }

    ...
    private String compactString(String source) {
        String result =
            DELTA_START +
            source.substring(prefixLength, source.length() - suffixLength) +
            DELTA_END;
        if (prefixLength > 0)
            result = computeCommonPrefix() + result;
    }
}
```

```

    if (suffixLength > 0)
        result = result + computeCommonSuffix();
    return result;
}

...
private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength +
        contextLength, expected.length()
    );
    return
        expected.substring(expected.length() - suffixLength, end) +
        (expected.length() - suffixLength <
            expected.length() - contextLength ?
            ELLIPSIS : "");
}

```

Wir haben die `+1s` in `computeCommonSuffix` durch ein `-1` in `charFromEnd` ersetzt, wo es genau an der richtigen Stelle steht. Außerdem stehen die beiden `<=`-Operatoren in `suffixOverlapsPrefix` auch an passender Stelle. Deshalb konnten wir den Namen von `suffixIndex` in `suffixLength` ändern und so den Code viel lesbarer machen.

Es gibt jedoch ein Problem. Als ich die `+1s` entfernt habe, fiel mir die folgende Zeile in `compactString` auf:

```
if (suffixLength > 0)
```

Schauen Sie sich die Zeile in Listing 15.4 an. Genau genommen sollte ich jetzt den `>`-Operator in einen `>=`-Operator ändern, weil `suffixLength` nun um eins kleiner als vorher ist. Aber das ergibt keinen Sinn. *Jetzt* ergibt es einen Sinn! Dies bedeutet, dass es vorher keinen Sinn ergab und wahrscheinlich ein Bug war. Nun ja, nicht gerade ein Bug. Eine weitere Analyse zeigt uns, dass die `if`-Anweisung jetzt verhindert, dass ein Suffix mit der Länge null angehängt wird. Vor unserer Änderung war die `if`-Anweisung einfach funktionslos, weil `suffixIndex` nie kleiner als eins werden konnte!

Dies macht *beide* `if`-Anweisungen in `compactString` fragwürdig! Es sieht so aus, als könnten beide eliminiert werden. Wir wollen sie deshalb auskommentieren und die Tests ausführen. Alle werden bestanden! Deshalb wollen wir `compactString` umstrukturieren, um die überflüssigen `if`-Anweisungen zu entfernen und die Funktion erheblich zu vereinfachen [G9].

```

private String compactString(String source) {
    return
        computeCommonPrefix() +
        DELTA_START +
        source.substring(prefixLength, source.length() - suffixLength) +

```

```
        DELTA_END +  
        computeCommonSuffix();  
    }
```

Dies ist viel besser! Jetzt sehen wir, dass die `compactString`-Funktion einfach die Fragmente zusammenfügt. Wir können dies wahrscheinlich noch deutlicher machen. Tatsächlich sind zahlreiche kleine Bereinigungen möglich. Aber statt Sie mit den restlichen Änderungen zu langweilen, zeige ich Ihnen in Listing 15.5 einfach das Ergebnis.

Listing 15.5: ComparisonCompactor.java (Schlussversion)

```
package junit.framework;  
  
public class ComparisonCompactor {  
  
    private static final String ELLIPSIS = "...";  
    private static final String DELTA_END = "]";  
    private static final String DELTA_START = "[";  
  
    private int contextLength;  
    private String expected;  
    private String actual;  
    private int prefixLength;  
    private int suffixLength;  
  
    public ComparisonCompactor(  
        int contextLength, String expected, String actual  
    ) {  
        this.contextLength = contextLength;  
        this.expected = expected;  
        this.actual = actual;  
    }  
  
    public String formatCompactedComparison(String message) {  
        String compactExpected = expected;  
        String compactActual = actual;  
        if (shouldBeCompacted()) {  
            findCommonPrefixAndSuffix();  
            compactExpected = compact(expected);  
            compactActual = compact(actual);  
        }  
        return Assert.format(message, compactExpected, compactActual);  
    }  
  
    private boolean shouldBeCompacted() {  
        return !shouldNotBeCompacted();  
    }  
  
    private boolean shouldNotBeCompacted() {
```

```
        return expected == null ||
            actual == null ||
            expected.equals(actual);
    }

    private void findCommonPrefixAndSuffix() {
        findCommonPrefix();
        suffixLength = 0;
        for (; !suffixOverlapsPrefix(); suffixLength++) {
            if (charFromEnd(expected, suffixLength) !=
                charFromEnd(actual, suffixLength)
            )
                break;
        }
    }

    private char charFromEnd(String s, int i) {
        return s.charAt(s.length() - i - 1);
    }

    private boolean suffixOverlapsPrefix() {
        return actual.length() - suffixLength <= prefixLength ||
            expected.length() - suffixLength <= prefixLength;
    }

    private void findCommonPrefix() {
        prefixLength = 0;
        int end = Math.min(expected.length(), actual.length());
        for (; prefixLength < end; prefixLength++)
            if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
                break;
    }

    private String compact(String s) {
        return new StringBuilder()
            .append(startingEllipsis())
            .append(startingContext())
            .append(DELTA_START)
            .append(delta(s))
            .append(DELTA_END)
            .append(endingContext())
            .append(endingEllipsis())
            .toString();
    }

    private String startingEllipsis() {
        return prefixLength > contextLength ? ELLIPSIS : "";
    }
}
```

```
private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
}

private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
}

private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
        Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
}

private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
}
}
```

Dies ist eine ganze Menge. Das Modul ist in eine Gruppe von Analysefunktionen und eine weitere Gruppe von Synthesefunktionen unterteilt. Sie sind topologisch sortiert, damit die Definition jeder Funktion unmittelbar nach ihrer Verwendung steht. Zuerst werden alle Analysefunktionen, dann alle Synthesefunktionen gezeigt.

Wenn Sie sorgfältig hinschauen, werden Sie feststellen, dass ich mehrere Entscheidungen rückgängig gemacht habe, die ich weiter vorne in diesem Kapitel getroffen hatte. Beispielsweise verwende ich einige extrahierte Methoden wieder inline in `formatCompactedComparison`, und ich habe die Bedeutung des Ausdrucks `shouldNotBeCompacted` umgekehrt. Dies ist typisch. Oft führt ein Refactoring zu einem weiteren, das das erste rückgängig macht. Refactoring ist ein iterativer Versuch-und-Irrtum-Prozess, der nach und nach zu einem Ergebnis führt, das unseren professionellen Standards gerecht wird.

15.2 Zusammenfassung

Und so haben wir die Pfadfinder-Regel erfüllt. Wir haben dieses Modul ein wenig sauberer hinterlassen, als wir es vorgefunden haben. Nicht, dass es nicht bereits sauber war. Die Autoren hatten exzellente Arbeit geleistet. Aber kein Modul ist vor Verbesserungen gefeit, und jeder ist dafür verantwortlich, den Code etwas besser zu hinterlassen, als er ihn vorgefunden hat.

Refactoring von SerialDate



Unter <http://www.jfree.org/jcommon/index.php> finden Sie die Java-Library *JCommon*. Tief in ihrem Inneren enthält diese Library ein Package namens `org.jfree.date`. In diesem Package befindet sich eine Klasse namens `SerialDate`. Wir werden diese Klasse analysieren.

Der Autor von `SerialDate` ist David Gilbert. David ist sicherlich ein erfahrener und kompetenter Programmierer. Wie wir sehen werden, schreibt er seinen Code sehr professionell und diszipliniert. Im Großen und Ganzen ist dies »guter Code«. Und ich werde ihn in Stücke reißen.

Dies geschieht nicht aus Böswilligkeit. Ich halte mich auch nicht für so viel besser als David, dass ich mir irgendwie das Recht anmaße, ein Urteil über seinen Code abzugeben. Tatsächlich bin ich mir sicher, dass Sie, würden Sie meinen Code studieren, zahlreiche Dinge entdecken würden, die man bemängeln könnte.

Es geht also nicht um Böswilligkeit oder Arroganz. Was ich hier tun werde, ist nicht mehr und nicht weniger als eine professionelle Begutachtung. Es ist ein Vorgehen, das wir alle als selbstverständlichen Bestandteil unserer beruflichen Tätigkeit akzeptieren sollten. Wir sollten es gutheißen, wenn unsere Arbeit zur Begutachtung ausgewählt wird. Nur durch solche Kritiken können wir lernen. Ärzte tun es. Piloten tun es. Rechtsanwälte tun es. Und wir Programmierer müssen ebenfalls lernen, es zu tun.

Noch ein Wort zu David Gilbert: David ist mehr als nur ein guter Programmierer. David hatte den Mut, seinen Code der Öffentlichkeit kostenlos zur Verfügung zu

stellen. Er veröffentlichte ihn und lud andere Programmierer ein, den Code zu verwenden und zu prüfen. Dies war ein lobenswerter Schritt!

`SerialDate` (siehe Anhang B, Listing B.1) ist eine Klasse, die ein Datum in Java repräsentiert. Warum brauchen wir noch eine Klasse, die ein Datum repräsentiert, wenn Java bereits `java.util.Date`, `java.util.Calendar` und andere enthält? Der Autor schrieb diese Klasse, als er von der Quälerei mit den nativen Java-Klassen genug hatte. Ich habe diese Qualen selbst oft genug erlebt. Der Kommentar in seiner einleitenden Javadoc (Zeile 67) bringt seine Motive klar zum Ausdruck. Wir könnten über seine Absichten streiten, aber das beklagte Problem ist real; und ich schätze eine Klasse für Datumsangaben statt für Zeitangaben.

16.1 Zunächst bring es zum Laufen!

Es gibt einige Unit-Tests in einer Klasse namens `SerialDateTests` (Anhang B, Listing B.2). Die Tests werden alle bestanden. Leider zeigt eine kurze Untersuchung der Tests, dass sie nicht alles testen [T1]. Beispielsweise erbringt eine »Find Usages«-Suche nach der Methode `MonthCodeToQuarter` (Zeile 334), dass sie nicht benutzt wird [F4]. Deshalb wird sie von den Unit-Tests nicht getestet.

Deshalb startete ich Clover, um zu prüfen, was die Unit-Tests abdeckten und was nicht. Clover berichtete, dass die Unit-Tests nur 91 der 185 ausführbaren Anweisungen in `SerialDate` (~50 Prozent) ausführten [T2]. Die Abdeckungsdarstellung sieht wie eine Patchwork-Decke aus. Große Blöcke von nicht ausgeführtem Code waren über die ganze Klasse verstreut.

Ich wollte diese Klasse vollkommen verstehen und auch ein Refactoring durchführen. Ohne eine erheblich größere Testabdeckung würde mir das nicht möglich sein. Deshalb schrieb ich meine eigene Suite vollkommen unabhängiger Unit-Tests (Anhang B, Listing B.4).

Wenn Sie sich diese Tests anschauen, werden Sie feststellen, dass viele auskommentiert sind. Diese Tests wurden nicht bestanden. Sie enthalten Verhaltensweise, die `SerialDate` meiner Meinung nach haben sollte. Deshalb werde ich während des Refactorings von `SerialDate` diese Tests nach und nach zum Laufen bringen.

Selbst wenn einige Tests auskommentiert sind, berichtet Clover, dass die neuen Unit-Tests 170 (92 Prozent) der 185 ausführbaren Anweisungen ausführen. Dies ist ziemlich gut; und ich glaube, wir können diesen Wert noch verbessern.

Bei den ersten wenigen auskommentierten Tests (Zeilen 23–63) war ich wohl ein wenig verblendet. Das Programm war nicht dafür konzipiert, diese Tests zu bestehen, aber die Verhaltensweise schien mir offensichtlich zu sein [G2]. Ich bin nicht sicher, warum die `testWeekdayCodeToString`-Methode überhaupt geschrieben wurde; doch sie ist vorhanden und offensichtlich sollte die Groß-/Kleinschreibung bei ihr keine Rolle spielen. Diese Tests zu schreiben, war trivial [T3]. Ihr Bestehen

zu garantieren, war noch leichter; ich änderte einfach die Zeilen 259 und 263 so, dass sie `equalsIgnoreCase` verwendeten.

Ich ließ die Tests in Zeile 32 und Zeile 45 auskommentiert, weil mir nicht klar ist, ob die Abkürzungen »tues« und »thurs« unterstützt werden sollten.

Die Tests in Zeile 153 und Zeile 154 werden nicht bestanden. Sicher sollten sie jedoch bestanden werden [G2]. Wir können dies und die Tests in Zeile 163 bis Zeile 213 leicht korrigieren, indem wir die `stringToMonthCode`-Funktion wie folgt ändern:

```

457     if ((result < 1) || (result > 12)) {
458         result = -1;
459         for (int i = 0; i < monthNames.length; i++) {
460             if (s.equalsIgnoreCase(shortMonthNames[i])) {
461                 result = i + 1;
462                 break;
463             }
464             if (s.equalsIgnoreCase(monthNames[i])) {
465                 result = i + 1;
466                 break;
467             }
468         }

```

Der kommentierte Test in Zeile 318 zeigt einen Bug in der `getFollowingDayOfWeek`-Methode auf (Zeile 672). Der 25. Dezember 2004 war ein Samstag. Der folgende Samstag war der 1. Januar 2005. Doch wenn wir den Test ausführen, gibt `getFollowingDayOfWeek` den 25. Dezember als den Samstag zurück, dem der 25. Dezember folgt. Das ist sicher falsch [G3], [T1]. Wir erkennen das Problem in Zeile 685. Es ist ein typischer Grenzbedingungsfehler [T5]. Die Anweisung sollte wie folgt lauten:

```

685     if (baseDOW >= targetWeekday) {

```

Es ist interessant anzumerken, dass diese Funktion Gegenstand einer früheren Reparatur gewesen war. Die Änderungshistorie (Zeile 43) zeigt, dass »Bugs« in `getPreviousDayOfWeek`, `getFollowingDayOfWeek` und `getNearestDayOfWeek` behoben wurden [T6].

Der Unit-Test `testGetNearestDayOfWeek` (Zeile 329), der die `getNearestDayOfWeek`-Methode (Zeile 705) testet, war ursprünglich nicht so lang und erschöpfend wie in seiner jetzigen Form. Ich habe zahlreiche Testfälle hinzugefügt, weil meine ersten Testfälle nicht alle bestanden wurden [T6]. Sie können das Pattern des Scheiterns erkennen, wenn Sie sich die auskommentierten Testfälle anschauen. Das Pattern ist vielsagend [T7]. Es zeigt, dass der Algorithmus scheitert, wenn der nächste Tag in der Zukunft liegt. Offensichtlich liegt hier eine Art von Grenzbedingungsfehler [T5] vor.

Das Pattern der Testabdeckung, das von Clover berichtet wird, ist ebenfalls interessant [T8]. Zeile 719 wird niemals ausgeführt! Dies bedeutet, dass die `if`-Anweisung in Zeile 718 immer `false` ist. Ein Blick auf den Code zeigt, dass dies wahr sein muss. Die `adjust`-Variable ist immer negativ und kann deshalb nicht größer oder gleich 4 werden. Deshalb ist dieser Algorithmus einfach falsch.

Hier ist der korrekte Algorithmus:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
    adjust -= 7;

return SerialDate.addDays(adjust, base);
```

Schließlich können die Tests in Zeile 417 und Zeile 429 einfach dadurch bestanden werden, dass eine `IllegalArgumentException` ausgelöst wird, anstatt einen Fehler-String von `weekInMonthToString` und `relativeToString` zurückzugeben.

Mit diesen Änderungen werden alle Unit-Tests bestanden; und ich glaube, dass `SerialDate` jetzt funktioniert. Deshalb ist es an der Zeit, es »richtig« zu machen.

16.2 Dann mach es richtig!

Wir werden `SerialDate` von Anfang bis Ende durchgehen und den Code auf unserem Weg verbessern. Obwohl Sie dies in der Beschreibung nicht sehen, werde ich nach jeder Änderung alle `Jcommon`-Unit-Tests, einschließlich meines verbesserten Unit-Tests für `SerialDate`, ausführen. Deshalb können Sie davon ausgehen, dass jede hier gezeigte Änderung im gesamten Code von `Jcommon` funktioniert.

Gleich in Zeile 1 sehen wir umfangreiche Kommentare mit Lizenzinformationen, Copyrights, Autoren und einer Änderungshistorie. Ich akzeptiere, dass bestimmte rechtliche Vorschriften beachtet werden müssen. Deshalb müssen die Copyrights und Lizenzen bleiben. Andererseits ist die Änderungshistorie ein Überbleibsel aus den 1960er-Jahren. Wir verfügen heute über Sourcecode-Control-Systeme, die uns diese Aufgabe abnehmen. Deshalb sollte diese History gelöscht werden [C1].

Die Importliste beginnt in Zeile 61 und könnte mit `java.text.*` und `java.util.*` verkürzt werden [J1].

Bei der HTML-Formatierung in der Javadoc (Zeile 67) zuckte ich zusammen. Eine Quelldatei mit mehr als einer Sprache bereitet mir Sorgen. Dieser Kommentar enthält vier Sprachen: Java, Englisch, Javadoc und HTML [G1]. Wenn so viele Sprachen verwendet werden, ist es schwierig, den Überblick über die Aufgaben zu behalten. Beispielsweise geht die hübsche Positionierung der Zeilen 71 und Zeile 72 verloren, wenn die Javadoc generiert wird; doch wer möchte schon `` und `` in seinem

Sourcecode sehen? Besser wäre es, den gesamten Kommentar in `<pre>...</pre>` einzuschließen, damit die in dem Sourcecode offensichtliche Formatierung in der Javadoc erhalten bleibt. (Noch besser wäre es, wenn Javadoc alle Kommentare als vorformatiert behandeln würde, damit sie sowohl im Code als auch im Dokument in gleicher Form angezeigt würden.)

Zeile 86 ist die Klassendeklaration. Warum heißt diese Klasse `SerialDate`? Welche Bedeutung hat das Wort »serial«? Wurde es gewählt, weil die Klasse von `Serializable` abgeleitet wurde? Das scheint unwahrscheinlich zu sein.

Ich will nicht weiter raten. Ich weiß (oder wenigstens glaube ich zu wissen), warum das Wort »serial« verwendet wurde. Der Hinweis findet sich in den Konstanten `SERIAL_LOWER_BOUND` und `SERIAL_UPPER_BOUND` in Zeile 98 und Zeile 101. Ein noch besserer Hinweis steht in dem Kommentar, der in Zeile 830 beginnt. Diese Klasse heißt `SerialDate`, weil sie mit einer »seriellen Zahl« implementiert wurde, und zwar mit der Anzahl der Tage seit dem 30. Dezember 1899.

Ich sehe darin zwei Probleme. Erstens: Der Terminus »serielle Zahl (engl. *serial number*)« ist nicht wirklich korrekt. Dies mag eine Spitzfindigkeit sein, aber die Repräsentation ist eher ein relatives Offset als eine serielle Zahl. Der Terminus »serielle Zahl« hat mehr mit Zählnummern zur Identifikation von Produkten als mit Datumsangaben zu tun. Deshalb halte ich diesen Namen nicht für besonders aussagekräftig [N1]. Ein aussagekräftigerer Ausdruck wäre etwa »ordinal« gewesen.

Zweitens: Das zweite Problem wiegt schwerer. Der Name `SerialDate` impliziert eine Implementierung. Diese Klasse ist eine abstrakte Klasse. Es ist nicht erforderlich, irgendetwas davon zu implementieren. Tatsächlich gibt es einen guten Grund, die Implementierung zu verbergen! Deshalb steht dieser Name für mich auf der falschen Abstraktionsebene [N2]. Meiner Meinung sollte diese Klasse einfach `Date` heißen.

Leider enthält die Java-Library bereits zu viele Klassen namens `Date`; deshalb ist dieser Name wahrscheinlich nicht der beste. Weil diese Klasse mit Tagen, nicht mit Zeitangaben arbeitet, erwog ich auch den Namen `Day`, aber dieser Name wird ebenfalls an anderen Stellen intensiv genutzt. Schließlich wählte ich `DayDate` als den besten Kompromiss.

Ab jetzt werde ich in dieser Beschreibung die Bezeichnung `DayDate` verwenden. Ich überlasse es Ihnen, daran zu denken, dass die Listings, die Sie sich anschauen, immer noch `SerialDate` verwenden.

Ich verstehe, warum `DayDate` von `Comparable` und `Serializable` erbt. Aber warum erbt die Klasse nicht von `MonthConstants`? Die Klasse `MonthConstants` (Anhang B, Listing B.3) enthält nur eine Reihe von `static final` Konstanten, die die Monate definieren. Eine Ableitung von Klassen mit Konstanten ist zwar ein alter Trick von Java-Programmierern, um Ausdrücke wie `MonthConstants.JANUARY` zu vermeiden, aber keine gute Idee [J2]. `MonthConstants` sollte wirklich eine `enum` sein.

```
public abstract class DayDate implements Comparable, Serializable {
    public static enum Month {
        JANUARY(1),
        FEBRUARY(2),
        MARCH(3),
        APRIL(4),
        MAY(5),
        JUNE(6),
        JULY(7),
        AUGUST(8),
        SEPTEMBER(9),
        OCTOBER(10),
        NOVEMBER(11),
        DECEMBER(12);

        Month(int index) {
            this.index = index;
        }

        public static Month make(int monthIndex) {
            for (Month m : Month.values()) {
                if (m.index == monthIndex)
                    return m;
            }
            throw new IllegalArgumentException("Invalid month index " + monthIndex);
        }
        public final int index;
    }
}
```

Die Änderung von MonthConstants in diese enum zieht zahlreiche Änderungen der DayDate-Klasse und ihrer Benutzer nach sich. Ich brauchte eine Stunde für alle Änderungen. Doch jede Funktion, die vorher eine int für einen Monat verwendete, arbeitet jetzt mit einem Month-Enumerator. Dies bedeutet, dass wir die isValidMonthCode-Methode (Zeile 326) und den gesamten Code zur Monatsfehlerprüfung, wie etwa den Code in monthCodeToQuarter (Zeile 356), nicht mehr benötigen [G5].

Als Nächstes kommen wir zu Zeile 91, serialVersionUID. Mit dieser Variablen wird der Serializer kontrolliert. Wenn wir sie ändern, kann jedes DayDate, das mit einer älteren Version der Software geschrieben wurde, nicht mehr gelesen werden und löst eine InvalidClassException aus. Wenn Sie die Variable serialVersionUID nicht deklarieren, generiert der Compiler automatisch eine für Sie. Sie hat bei jeder Änderung des Moduls einen anderen Wert. Ich weiß, dass alle Dokumente eine manuelle Kontrolle dieser Variablen anraten, aber meiner Meinung nach scheint die automatische Kontrolle der Serialisierung viel sicherer zu sein [G4]. Schließlich debugge ich lieber eine InvalidClassException als das seltsame Verhalten, das eintreten würde, wenn ich vergessen würde, serialVersionUID zu ändern. Deswegen löschte ich die Variable – wenigstens für den Moment.

(Mehrere Begutachter dieses Textes waren mit dieser Entscheidung nicht einverstanden. Sie vertraten die Auffassung, dass es bei einem Open-Source-Framework besser wäre, die Serial-ID manuell zu kontrollieren, damit alte serialisierte Datumsangaben durch geringfügige Änderungen der Software nicht ungültig gemacht werden. Dies ist ein berechtigter Einwand. Doch wenigstens hat der Fehler, so unbequem er auch sein mag, eine klar erkennbare Ursache. Vergisst der Autor der Klasse dagegen, die ID zu aktualisieren, dann ist der Fehlermodus undefiniert und könnte auch unentdeckt bleiben. Ich meine, die eigentliche Lehre aus dieser Geschichte ist, dass man nicht erwarten darf, Daten versionsübergreifend zu deserialisieren.)

Ich halte den Kommentar in Zeile 93 für redundant. Redundante Kommentare sind genau die Stellen, an denen sich Lügen und Fehlinformationen [C2] ansammeln. Deshalb löschte ich ihn und ähnliche Kommentare.

Die Kommentare in Zeile 97 und Zeile 100 handeln von seriellen Zahlen, über die ich mich weiter vorne ausgelassen habe [C1]. Die Variablen, die sie beschreiben, geben das früheste und das späteste mögliche Datum an, das `DayDate` beschreiben kann. Dies könnte etwas deutlicher ausgedrückt werden [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2;    // 1. Januar 1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 31. Dezember 9999
```

Mir ist nicht klar, warum `EARLIEST_DATE_ORDINAL` den Wert 2 und nicht den Wert 0 hat. Es gibt einen Hinweis in dem Kommentar in Zeile 829, der vermuten lässt, dass dies etwas mit der Repräsentation von Datumsangaben in Microsoft Excel zu hat. Eine viel eingehendere Erklärung findet sich in einer abgeleiteten Klasse von `DayDate` namens `SpreadsheetDate` (Anhang B, Listing B.5). Der Kommentar in Zeile 71 beschreibt das Problem deutlich.

Mein Problem dabei ist, dass diese Frage mit der Implementierung von `SpreadsheetDate` und nicht mit `DayDate` zu tun hat. Ich schließe dies daraus, dass `EARLIEST_DATE_ORDINAL` und `LATEST_DATE_ORDINAL` nicht wirklich in `DayDate` gehören und nach `SpreadsheetDate` [G6] verlagert werden sollten.

Tatsächlich zeigt sich beim Durchsuchen des Codes, dass diese Variablen nur in `SpreadsheetDate` verwendet werden. Weder `DayDate` noch irgendeine andere Klasse in dem `Jcommon-Framework` nutzen sie. Deshalb verschob ich sie in die abgeleitete Klasse `SpreadsheetDate`.

Die nächsten Variablen, `MINIMUM_YEAR_SUPPORTED` und `MAXIMUM_YEAR_SUPPORTED` (Zeile 104 und Zeile 107), stellen uns vor ein Dilemma. Denn wenn `DayDate` eine abstrakte Klasse ist, die keine Implementierung vorwegnimmt, sollte sie keine Informationen über ein Mindest- oder Höchstjahr enthalten. Auch hier war ich versucht, diese Variablen in die abgeleitete Klasse `SpreadsheetDate` [G6] zu verschieben. Doch eine schnelle Suche nach den Benutzern dieser Variablen zeigte, dass sie von einer anderen Klasse verwendet werden: `RelativeDayOfWeekRule` (Anhang B, Listing B.6), und zwar in Zeile 177 und Zeile 178 in der `getDate`-Funktion, wo sie prüfen, ob das Argument von `getDate` ein gültiges Jahr enthält. Das Dilemma besteht

darin, dass ein Benutzer einer abstrakten Klasse Informationen über ihre Implementierung benötigt.

Wir müssen diese Informationen zur Verfügung stellen, ohne `DayDate` selbst zu verunreinigen. Normalerweise würden wir Implementierungsinformationen aus einer Instanz einer abgeleiteten Klasse abrufen. Doch der `getDate`-Funktion wird keine Instanz von `DayDate` übergeben. Sie gibt jedoch eine solche Instanz zurück, was bedeutet, dass sie diese irgendwo erstellt. Zeile 187 bis Zeile 205 liefern den Hinweis. Die `DayDate`-Instanz wird von einer der drei Funktionen `getPreviousDayOfWeek`, `getNearestDayOfWeek` oder `getFollowingDayOfWeek` erzeugt. Ein Blick zurück auf das `DayDate`-Listing zeigt uns, dass diese Funktionen (Zeilen 638–724) alle ein Datum zurückgeben, das mit `addDays` (Zeile 571) erstellt wird, die `createInstance` (Zeile 808) aufruft, die ihrerseits ein `SpreadsheetDate` erzeugt! [G7].

Im Allgemeinen sollten Basisklassen nichts über ihre abgeleiteten Klassen wissen. Um dieses Problem zu beheben, sollten wir mit dem *Abstract Factory*-Pattern ([GOF]) eine `DayDateFactory` erstellen. Diese Factory soll die Instanzen von `DayDate` erstellen, die wir benötigen, und kann außerdem Fragen über die Implementierung beantworten, wie etwa das Mindest- oder Höchstdatum.

```
public abstract class DayDateFactory {
    private static DayDateFactory factory = new SpreadsheetDateFactory();
    public static void setInstance(DayDateFactory factory) {
        DayDateFactory.factory = factory;
    }

    protected abstract DayDate _makeDate(int ordinal);
    protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);
    protected abstract DayDate _makeDate(int day, int month, int year);
    protected abstract DayDate _makeDate(java.util.Date date);
    protected abstract int _getMinimumYear();
    protected abstract int _getMaximumYear();

    public static DayDate makeDate(int ordinal) {
        return factory._makeDate(ordinal);
    }

    public static DayDate makeDate(int day, DayDate.Month month, int year) {
        return factory._makeDate(day, month, year);
    }

    public static DayDate makeDate(int day, int month, int year) {
        return factory._makeDate(day, month, year);
    }

    public static DayDate makeDate(java.util.Date date) {
        return factory._makeDate(date);
    }
}
```

```

public static int getMinimumYear() {
    return factory._getMinimumYear();
}

public static int getMaximumYear() {
    return factory._getMaximumYear();
}
}

```

Diese Factory-Klasse ersetzt die `createInstance`-Methoden durch `makeDate`-Methoden, die die Namen etwas verbessern [N1]. Standardmäßig wird eine `SpreadsheetDateFactory` verwendet, was aber jederzeit in eine andere Factory geändert werden kann. Die statischen Methoden, die an abstrakte Methoden delegieren, verwenden eine Kombination der Patterns *Singleton* ([GOF]), *Decorator* ([GOF]) und *Abstract Factory* ([GOF]).

Die `SpreadsheetDateFactory` sieht folgendermaßen aus:

```

public class SpreadsheetDateFactory extends DayDateFactory {
    public DayDate _makeDate(int ordinal) {
        return new SpreadsheetDate(ordinal);
    }

    public DayDate _makeDate(int day, DayDate.Month month, int year) {
        return new SpreadsheetDate(day, month, year);
    }

    public DayDate _makeDate(int day, int month, int year) {
        return new SpreadsheetDate(day, month, year);
    }

    public DayDate _makeDate(Date date) {
        final GregorianCalendar calendar = new GregorianCalendar();
        calendar.setTime(date);
        return new SpreadsheetDate(
            calendar.get(Calendar.DATE),
            DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
            calendar.get(Calendar.YEAR));
    }

    protected int _getMinimumYear() {
        return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
    }

    protected int _getMaximumYear() {
        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
    }
}

```


Wie Sie sehen können, habe ich bereits die Variablen `MINIMUM_YEAR_SUPPORTED` und `MAXIMUM_YEAR_SUPPORTED` nach `SpreadsheetDate` verschoben, wo sie hingehören [G6].

Das nächste Problem in `DayDate` sind die Tages-Konstanten, die in Zeile 109 beginnen. Sie sollten wirklich durch eine weitere `enum` [J3] ersetzt werden. Da ich dieses Pattern bereits beschrieben habe, will ich es hier nicht wiederholen. Sie finden es in den fertigen Listings.

Als Nächstes stoßen wir auf eine Reihe von Tabellen, die mit `LAST_DAY_OF_MONTH` in Zeile 140 beginnen. Zunächst einmal sind die Kommentare redundant, die sie beschreiben [C3]. Ihre Namen reichen aus. Deshalb löschte ich die Kommentare.

Es scheint keinen triftigen Grund dafür zu geben, warum diese Tabelle nicht `private` [G8] sein sollte, weil die statische Funktion `lastDayOfMonth` dieselben Daten liefert.

Die nächste Tabelle, `AGGREGATE_DAYS_TO_END_OF_MONTH`, ist etwas geheimnisvoller, weil sie nirgends in dem `Jcommon`-Framework benutzt wird [G9]. Deshalb löschte ich sie.

Dasselbe gilt für `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Die nächste Tabelle, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`, wird nur in `SpreadsheetDate` verwendet (Zeile 434 und Zeile 473). Natürlich wirft dies die Frage auf, warum sie nicht nach `SpreadsheetDate` verschoben werden sollte. Dagegen spricht, dass die Tabelle nichts mit einer speziellen Implementierung zu tun hat [G6]. Andererseits gibt es keine andere Implementierung als `SpreadsheetDate`; und deshalb sollte die Tabelle nahe an ihren Verwendungsort verschoben werden [G10].

Ausschlaggebend ist für mich schließlich die Forderung nach Konsistenz [G11]. Wir sollten die Tabelle als `private` deklarieren und sie mittels einer Funktion, etwa `julianDateOfLastDayOfMonth`, zugänglich machen. Niemand scheint eine derartige Funktion zu benötigen. Darüber hinaus kann die Tabelle leicht wieder nach `DayDate` verschoben werden, falls sie von einer neuen Implementierung von `DayDate` benötigt werden sollte. Deshalb verschob ich sie.

Dasselbe gilt für die Tabelle `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Als Nächstes finden wir drei Sätze von Konstanten, die wir in `enums` umwandeln können (Zeilen 162–205). Der erste Satz wählt eine Woche in einem Monat aus. Ich änderte ihn in eine `enum` namens `WeekInMonth`.

```
public enum WeekInMonth {
    FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
    public final int index;

    WeekInMonth(int index) {
```

```

    this.index = index;
}
}

```

Der zweite Satz von Konstanten (Zeilen 177–187) ist etwas undurchschaubarer. Die Konstanten `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND` und `INCLUDE_BOTH` beschreiben, ob die Endpunkte eines Datumsbereichs zu dem Bereich gehören sollen oder nicht. Mathematisch spricht man hier von einem »offenen«, einem »links-offenen« bzw. »rechts-geschlossenen«, einem »rechts-offenen« bzw. »links-geschlossenen« und einem »geschlossenen« Intervall. Meines Erachtens ist die mathematische Nomenklatur klarer [N3]. Deshalb änderte ich den Satz in eine `enum` namens `DateInterval` mit den Enumeratoren `CLOSED`, `CLOSED_LEFT`, `CLOSED_RIGHT` und `OPEN`.

Der dritte Satz von Konstanten (Zeilen 18–205) beschreibt, ob eine Suche nach einem speziellen Tag der Woche die letzte, die nächste oder die nächstliegende Instanz zurückgeben soll. Passende Bezeichnungen zu finden, ist hier schwierig. Schließlich entschloss ich mich für `WeekdayRange` mit den Enumeratoren `LAST`, `NEXT` und `NEAREST`.

Vielleicht gefällt Ihnen meine Namenswahl nicht. Für mich drücken sie die Absicht aus, für Sie vielleicht jedoch nicht. Wichtig ist jedoch, dass die Namen jetzt eine Form haben, in der sie leicht geändert werden können [J3]. Sie werden nicht mehr als Ganzzahlen, sondern als Symbole übergeben. Ich kann die Namen oder ihre Typen mit der Namensänderungsfunktion meiner IDE ändern, ohne mir Sorgen machen zu müssen, irgendwo im Code ein `-1` oder `2` zu übersehen oder eine unzureichend beschriebene `int`-Argumentdeklaration zu übergehen.

Das Beschreibungsfeld in Zeile 208 scheint von niemandem benutzt zu werden. Ich löschte es zusammen mit seinem Accessor und Mutator [G9].

Ich löschte ebenfalls den verkümmerten Standardkonstruktor in Zeile 213 [G12]. Der Compiler generiert ihn für uns.

Wir können die `isValidWeekdayCode`-Methode (Zeilen 216–238) überspringen, weil sie bei der Erstellung der `Day`-Enumeration gelöscht wurde.

Damit kommen wir zu der `stringToWeekdayCode`-Methode (Zeilen 242–270). Javadocs, die keine wesentlichen zusätzlichen Informationen liefern, sind nur Müll [C3], [G12]. Der einzige Mehrwert, den diese Javadoc liefert, ist die Beschreibung des Rückgabewertes `-1`. Doch weil wir jetzt die `Day`-Enumeration verwenden, ist der Kommentar tatsächlich falsch [C2]. Die Methode löst jetzt eine `IllegalArgumentException` aus. Deshalb löschte ich die Javadoc.

Ich löschte ebenfalls alle `final`-Schlüsselwörter in Deklarationen von Argumenten und Variablen. Soweit ich das beurteilen kann, liefern sie keinen echten Wert, machten aber den Code unübersichtlicher [G12]. Das Schlüsselwort `final` zu löschen, widerspricht der gängigen »Best Practice«. Beispielsweise rät Robert Simmons

([Simmonso4]) dringend dazu, `final` großzügig im gesamten Code zu verwenden. Ich bin entschieden anderer Meinung. Ich glaube, dass es einige gute Anwendungszwecke für `final` gibt, etwa die gelegentliche Verwendung von `final` Konstanten, aber ansonsten bietet dieses Schlüsselwort wenig Mehrwert und macht den Code sehr viel unübersichtlicher. Vielleicht bin ich zu dieser Auffassung gekommen, weil die Art von Fehlern, die `final` abfangen könnte, bereits von meinen Unit-Tests herausgefiltert wird.

Da ich keine doppelte `if`-Anweisungen [G5] in der `for`-Schleife (Zeile 259 und Zeile 263) haben wollte, verknüpfte ich sie mit dem `||`-Operator zu einer einzigen `if`-Anweisung. Außerdem verwendete ich die `Day`-Enumeration zur Steuerung der `for`-Schleife und nahm einige andere kosmetische Änderungen vor.

Es fiel mir auf, dass diese Methode nicht in `DayDate` gehört, sondern eigentlich die Parse-Funktion von `Day` ist. Deshalb verschob ich sie in die `Day`-Enumeration. Doch dadurch wurde die `Day`-Enumeration ziemlich groß. Weil das Konzept von `Day` nicht von `DayDate` abhängt, verlagerte ich die `Day`-Enumeration aus der `DayDate`-Klasse in eine separate Source-Datei [G13].

Außerdem verschob ich die nächste Funktion, `weekdayCodeToString` (Zeilen 272–286), in die `Day`-Enumeration und nannte sie `toString`.

```
public enum Day {
    MONDAY(Calendar.MONDAY),
    TUESDAY(Calendar.TUESDAY),
    WEDNESDAY(Calendar.WEDNESDAY),
    THURSDAY(Calendar.THURSDAY),
    FRIDAY(Calendar.FRIDAY),
    SATURDAY(Calendar.SATURDAY),
    SUNDAY(Calendar.SUNDAY);

    public final int index;
    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

    Day(int day) {
        index = day;
    }

    public static Day make(int index) throws IllegalArgumentException {
        for (Day d : Day.values())
            if (d.index == index)
                return d;
        throw new IllegalArgumentException(
            String.format("Illegal day index: %d.", index));
    }

    public static Day parse(String s) throws IllegalArgumentException {
        String[] shortWeekdayNames =
            dateSymbols.getShortWeekdays();
```

```

String[] weekDayNames =
    dateSymbols.getWeekdays();

s = s.trim();
for (Day day : Day.values()) {
    if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
        s.equalsIgnoreCase(weekDayNames[day.index])) {
        return day;
    }
}
throw new IllegalArgumentException(
    String.format("%s is not a valid weekday string", s));
}

public String toString() {
    return dateSymbols.getWeekdays()[index];
}
}

```

Es gibt zwei `getMonths`-Funktionen (Zeilen 288–316). Die erste ruft die zweite auf. Die zweite wird nur von der ersten und sonst von keiner anderen Funktion aufgerufen. Deshalb fasste ich beide Funktionen zu einer zusammen konnte sie so erheblich vereinfachen [G9], [G12], [F4]. Schließlich änderte ich den Namen, um ihn selbstbeschreibender zu machen [N1].

```

public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
}

```

Die `isValidMonthCode`-Funktion (Zeilen 326–346) war durch die `Month-Enum` überflüssig geworden. Deshalb löschte ich sie [G9].

Die `monthCodeToQuarter`-Funktion (Zeilen 356–375) riecht nach Funktionsneid (Feature Envy; [Refactoring]) [G14] und gehört wahrscheinlich als Methode namens `quarter` in die `Month-Enum`. Deshalb ersetzte ich sie.

```

public int quarter() {
    return 1 + (index-1)/3;
}

```

Damit wurde die `Month-Enum` groß genug für eine eigene Klasse. Deshalb verschob ich den Code – konsistent mit der `Day-Enum` [G11], [G13] – aus `DayDate` in eine separate Klasse.

Die nächsten zwei Methoden heißen `monthCodeToString` (Zeilen 377–426). Auch hier sehen wir das Pattern, dass eine Methode eine gleichnamige Methode mit einem Flag aufruft. Es ist normalerweise keine gute Idee, ein Flag als Argument an eine Funktion zu übergeben, besonders wenn dieses Flag einfach das Format des

Outputs auswählt [G15]. Ich benannte diese Funktionen um, vereinfachte sie, änderte ihre Struktur und verschob sie in die Month-Enum [N1], [N3], [C3], [G14].

```
public String toString() {
    return dateFormatSymbols.getMonths()[index - 1];
}

public String toShortString() {
    return dateFormatSymbols.getShortMonths()[index - 1];
}
```

Die nächste Methode ist `stringToMonthCode` (Zeilen 428–472). Ich benannte sie um, verschob sie in die Month-Enum und vereinfachte sie [N1], [N3], [C3], [G14], [G12].

```
public static Month parse(String s) {
    s = s.trim();
    for (Month m : Month.values())
        if (m.matches(s))
            return m;

    try {
        return make(Integer.parseInt(s));
    }
    catch (NumberFormatException e) {}
    throw new IllegalArgumentException("Invalid month " + s);
}

private boolean matches(String s) {
    return s.equalsIgnoreCase(toString()) ||
        s.equalsIgnoreCase(toShortString());
}
```

Die `isLeapYear`-Methode (Zeilen 495–517) kann etwas ausdrucksstärker gemacht werden [G16].

```
public static boolean isLeapYear(int year) {
    boolean fourth = year % 4 == 0;
    boolean hundredth = year % 100 == 0;
    boolean fourHundredth = year % 400 == 0;
    return fourth && (!hundredth || fourHundredth);
}
```

Die nächste Funktion, `leapYearCount` (Zeilen 519–536), gehört nicht in `DayDate`. Sie wird nur von zwei Methoden in `SpreadsheetDate` aufgerufen. Deshalb verschob ich sie in diese abgeleitete Klasse [G6].

Die `lastDayOfMonth`-Funktion (Zeilen 538–560) verwendet das Array `LAST_DAY_OF_MONTH`. Dieses Array gehört eigentlich in die Month-Enum [G17].

Deshalb verschob ich es dorthin. Außerdem vereinfachte ich die Funktion und machte sie etwas ausdrucksstärker [G16].

```
public static int lastDayOfMonth(Month month, int year) {
    if (month == Month.FEBRUARY && isLeapYear(year))
        return month.lastDay() + 1;
    else
        return month.lastDay();
}
```

Jetzt werden die Aufgaben etwas interessanter. Die nächste Funktion ist `addDays` (Zeilen 562–576). Erstens sollte diese Funktion nicht statisch sein, weil sie mit Variablen von `DayDate` arbeitet [G18]. Deshalb änderte ich sie in eine Instanzmethode. Zweitens ruft sie die Funktion `toSerial` auf. Diese Funktion sollte in `toOrdinal` [N1] umbenannt werden. Schließlich kann die Methode vereinfacht werden.

```
public DayDate addDays(int days) {
    return DayDateFactory.makeDate(toOrdinal() + days);
}
```

Dasselbe gilt für `addMonths` (Zeilen 578–602). Sie sollte eine Instanzmethode [G18] sein. Der Algorithmus ist etwas kompliziert. Deshalb verwendete ich *Explaining Temporary Variables* ([Beck97]) [G19], um ihn transparenter zu machen. Außerdem änderte ich den Namen der Methode `getYYY` in `getYear` [N1].

```
public DayDate addMonths(int months) {
    int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
    int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
    int resultYear = resultMonthAsOrdinal / 12;
    Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
    int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

Die `addYears`-Funktion (Zeilen 604–626) ist nicht überraschender als die anderen Funktionen.

```
public DayDate plusYears(int years) {
    int resultYear = getYear() + years;
    int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

Ein kleines Problem nagte noch in meinem Bewusstsein, und zwar die Änderung der statischen Methoden in Instanzmethoden. Macht der Ausdruck `date.addDays(5)` hinreichend klar, dass das `date`-Objekt nicht geändert wird und dass eine

neue Instanz von `DayDate` zurückgegeben wird? Oder suggeriert er fälschlicherweise, dass wir fünf Tage zu dem `date`-Objekt addieren? Vielleicht halten Sie dies nicht für ein großes Problem, aber Code, der wie der folgende aussieht, kann sehr täuschend sein [G20]:

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);  
date.addDays(7); // Datum um eine Woche weitersetzen
```

Jemand, der diesen Code liest, würde sehr wahrscheinlich einfach annehmen, dass `addDays` das `date`-Objekt ändert. Deshalb brauchen wir einen Namen, der diese Mehrdeutigkeit aufhebt [N4]. Aus diesem Grund änderte ich die Namen in `plusDays` und `plusMonths`. Mir scheint der Zweck der Methode durch

```
DayDate date = oldDate.plusDays(5);
```

deutlich zum Ausdruck gebracht zu werden, während sich die folgende Anweisung nicht flüssig genug liest, weshalb ein Leser einfach annehmen könnte, dass das `date`-Objekt geändert wird:

```
date.plusDays(5);
```

Die Algorithmen werden immer interessanter. Die Funktion `getPreviousDayOfWeek` (Zeilen 628–660) funktioniert, ist aber etwas kompliziert. Nach genauerem Studium des Ablaufs [G21] konnte ich sie vereinfachen und mit *Explaining Temporary Variables* [G19] verdeutlichen. Außerdem änderte ich sie von einer statischen Methode in eine Instanzmethode [G18] und löschte die doppelte Instanzmethode [G5] (Zeilen 997–1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {  
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;  
    if (offsetToTarget >= 0)  
        offsetToTarget -= 7;  
    return plusDays(offsetToTarget);  
}
```

Auf dieselbe Weise änderte ich `getFollowingDayOfWeek` (Zeilen 662–693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {  
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;  
    if (offsetToTarget <= 0)  
        offsetToTarget += 7;  
    return plusDays(offsetToTarget);  
}
```

Die nächste Funktion ist `getNearestDayOfWeek` (Zeilen 695–726), die wir weiter vorne im Abschnitt *Zunächst bring es zum Laufen!* korrigiert haben. Doch die dort vorgenommenen Änderungen sind nicht konsistent mit dem gegenwärtigen Pattern der letzten beiden Funktionen [G11]. Deshalb stellte ich die Konsistenz her und

verwendete einige *Explaining Temporary Variables* [G19], um den Algorithmus zu verdeutlichen.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
    int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
    int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
    int offsetToPreviousTarget = offsetToFutureTarget - 7;

    if (offsetToFutureTarget > 3)
        return plusDays(offsetToPreviousTarget);
    else
        return plusDays(offsetToFutureTarget);
}
```

Die `getEndOfCurrentMonth`-Methode (Zeilen 728–740) ist ein wenig seltsam, weil es sich um eine Instanzmethode handelt, die ihre eigene Klasse beneidet [G14], indem sie ein `DayDate`-Argument übernimmt. Ich machte daraus eine echte Instanzmethode und verbesserte einige Namen.

```
Public DayDate getEndOfMonth() {
    Month month = getMonth();
    int year = getYear();
    int lastDay = lastDayOfMonth(month, year);
    return DayDateFactory.makeDate(lastDay, month, year);
}
```

Das Refactoring von `weekInMonthToString` (Zeilen 742–761) war wirklich sehr interessant. Mit den Refactoring-Werkzeugen meiner IDE verschob ich die Methode zunächst in die `WeekInMonth`-Enum, die ich weiter oben erstellt hatte. Dann änderte ich den Namen der Methode in `toString`. Als Nächstes änderte ich sie aus einer statischen Methode in eine Instanzmethode. Alle Tests wurden immer noch bestanden. (Vermuten Sie, worauf ich hinauswill?)

Als Nächstes löschte ich die Methode komplett! Fünf Zusicherungen scheiterten (Zeilen 411–415, Anhang B, Listing B.4). Ich änderte diese Zeilen so, dass sie die Namen der Enumeratoren (`FIRST`, `SECOND`, ...) verwendeten. Alle Tests wurden bestanden. Verstehen Sie, warum? Verstehen Sie auch, warum jeder dieser Schritte erforderlich war? Das Refactoring-Werkzeug sorgte dafür, dass alle früheren Aufrufe von `weekInMonthToString` jetzt `toString` in dem `weekInMonth`-Enumerator aufrufen, weil alle Enumeratoren `toString` implementieren, um einfach ihren Namen zurückzugeben ...

Leider war ich ein wenig zu schlau. Diese wundervolle Kette von Refactorings war zwar sehr elegant, aber ich erkannte schließlich, dass die einzigen Benutzer dieser Funktion die Tests waren, die ich gerade geändert hatte. Deshalb löschte ich die Tests.

Legen Sie mich einmal rein, sollten Sie sich schämen. Legen Sie mich zweimal rein, sollte ich mich schämen! Nachdem ich also festgestellt hatte, dass niemand außer den Tests `relativeToString` (Zeilen 765–781) aufrief, löschte ich einfach die Funktion und ihre Tests.

Endlich haben wir die abstrakten Methoden dieser abstrakten Klasse erreicht. Gleich die erste Methode ist ganz typisch: `toSerial` (Zeilen 838–844). Weiter vorne hatte ich ihren Namen in `toOrdinal` geändert. Nachdem ich sie in diesem Kontext sah, beschloss ich, den Namen in `getOrdinalDay` zu ändern.

Die nächste abstrakte Methode ist `toDate` (Zeilen 838–844). Sie wandelt ein `DayDate` in ein `java.util.Date` um. Warum ist diese Methode abstrakt? Wenn wir uns ihre Implementierung in `SpreadsheetDate` (Anhang B, Listing B.5, Zeilen 198–207) anschauen, stellen wir fest, dass sie von keinem Element der Implementierung dieser Klasse abhängt [G6]. Deshalb verschob ich sie in die Basisklasse.

Die Methoden `getYYYY`, `getMonth` und `getDayOfMonth` sind schön abstrakt. Doch die `getDayOfWeek`-Methode sollte ebenfalls aus `SpreadsheetDate` herausgenommen werden, weil sie von nichts abhängt, was nicht in `DayDate` zu finden wäre [G6]. Oder etwa nicht?

Wenn Sie sorgfältig hinschauen (Anhang B, Listing B.5, Zeile 247), erkennen Sie, dass der Algorithmus implizit vom Ursprung des ordinalen Tages (anders ausgedrückt: dem Tag o der Woche) abhängt. Obwohl also diese Funktion keine physischen Abhängigkeiten hat, die nicht nach `DayDate` verschoben werden können, hat sie eine logische Abhängigkeit.

Derartige logische Abhängigkeiten beunruhigen mich [G22]. Wenn etwas logisch von der Implementierung abhängt, sollte es auch eine physische Abhängigkeit geben. Auch schien es mir, dass der Algorithmus selbst generisch formuliert werden könnte, und nur ein viel kleinerer Teil von ihm von der Implementierung abhängig sein müsste [G6].

Deshalb erstellte ich in `DayDate` eine abstrakte Methode namens `getDayOfWeekForOrdinalZero` und implementierte sie in `SpreadsheetDate`. Sie gibt `Day.SATURDAY` zurück. Dann verschob ich die `getDayOfWeek`-Methode nach oben nach `DayDate` und änderte sie so, dass sie `getOrdinalDay` und `getDayOfWeekForOrdinalZero` aufrief.

```
public Day getDayOfWeek() {
    Day startingDay = getDayOfWeekForOrdinalZero();
    int startingOffset = startingDay.index - Day.SUNDAY.index;
    return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Nebenbei bemerkt: Sie sollten sich den Kommentar in Zeile 895 bis Zeile 899 sorgfältig anschauen. War diese Wiederholung wirklich erforderlich? Wie gewöhnlich löschte ich diesen Kommentar zusammen mit allen anderen.

Die nächste Methode ist `compare` (Zeilen 902–913). Auch diese Methode ist unangemessen abstrakt [G6]. Deshalb verschob ich die Implementierung nach oben nach `DayDate`. Außerdem war der Name nicht aussagekräftig genug [N1]. Diese Methode gibt die Differenz in Tagen seit dem Argument zurück. Deshalb änderte ich den Namen in `daysSince`. Beachten Sie auch, dass es keine Tests für diese Methode gab. Deshalb schrieb ich sie.

Die nächsten sechs Funktionen (Zeilen 915–980) sind alle abstrakt und sollten in `DayDate` implementiert sein. Deshalb vorschob ich sie alle aus `SpreadsheetDate` nach oben nach `DayDate`.

Die letzte Funktion, `isInRange` (Zeilen 982–995), muss ebenfalls nach oben verschoben und ein Refactoring durchgeführt werden. Die `switch`-Anweisung ist hässlich [G23]. Sie kann ersetzt werden, indem die cases in die `DateInterval`-Enum verschoben werden.

```
Public enum DateInterval {
    OPEN {
        public boolean isIn(int d, int left, int right) {
            return d > left && d < right;
        }
    },
    CLOSED_LEFT {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d < right;
        }
    },
    CLOSED_RIGHT {
        public boolean isIn(int d, int left, int right) {
            return d > left && d <= right;
        }
    },
    CLOSED {
        public boolean isIn(int d, int left, int right) {
            return d >= left && d <= right;
        }
    };

    public abstract boolean isIn(int d, int left, int right);
}

public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
    int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
    int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
    return interval.isIn(getOrdinalDay(), left, right);
}
```

Damit kommen wir zum Ende von `DayDate`. Ein weiterer Durchgang über die gesamte Klasse soll uns zeigen, wie gut sie fließt.

Zunächst ist der einleitende Kommentar längst überholt; deshalb kürzte und verbesserte ich ihn [C2].

Als Nächstes fügte ich alle verbleibenden Enums in separate Dateien ein [G12].

Als Nächstes verschob ich die statische Variable (`dateFormatSymbols`) und drei statische Methoden (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) in eine neue Klasse namens `DateUtil` [G6].

Ich verschob die abstrakten Methoden nach oben in die Basisklasse, in die sie gehören [G24].

Ich änderte `Month.make` in `Month.fromInt` [N1] und tat dann bei allen anderen Enums dasselbe. Außerdem erstellte ich für alle Enums einen `toInt()`-Accessor und deklarierte das Indexfeld als `private`.

Es gab einige interessante Duplizierungen [G5] in `plusYears` und `plusMonths`, die ich eliminieren konnte, indem ich eine neue Methode namens `correctLastDayOfMonth` extrahierte und damit alle drei Methoden viel klarer machte.

Ich habe die magische Zahl 1 [G25] beseitigt, indem ich sie durch `Month.JANUARY.toInt()` bzw. `Day.SUNDAY.toInt()` ersetzte. Ich verbrachte etwas Zeit mit `SpreadsheetDate` und verbesserte die Algorithmen ein wenig. Das Endergebnis finden Sie in Anhang B, Listings B.7 bis B.16.

Interessanterweise *sank* die Code-Coverage in `DayDate` auf 84,9 Prozent! Der Grund liegt nicht darin, dass weniger Funktionalität getestet wurde, sondern dass die Klasse so viel kleiner geworden ist, dass die wenigen nicht abgedeckten Zeilen ein größeres Gewicht bekommen haben. In `DayDate` werden jetzt 45 von 53 ausführbaren Anweisungen durch Tests abgedeckt. Die nicht abgedeckten Zeilen sind so trivial, dass sich ein Test nicht lohnt.

16.3 Zusammenfassung

Auch hier haben wir die Pfadfinder-Regel befolgt. Wir haben den Code ein wenig sauberer eingesehen, als wir ihn ausgecheckt haben. Es brauchte etwas Zeit, aber es hat sich gelohnt. Die Test-Abdeckung wurde erweitert, einige Bugs wurden behoben, der Code wurde klarer strukturiert und verkleinert. Der nächste Leser des fertigen Codes wird sich hoffentlich leichter darin zurechtfinden als wir. Wahrscheinlich wird er ihn auch noch ein bisschen weiter säubern können als wir.

Smells und Heuristiken



In seinem wundervollen Buch *Refactoring* ([Refactoring]) identifiziert Martin Fowler viele »Code Smells«. Die folgende Liste umfasst viele von Fowlers Smells sowie viele meiner eigenen. Sie enthält auch andere Perlen und Heuristiken aus meiner Praxis.

Ich habe diese Liste zusammengestellt, indem ich verschiedene Programme studiert und ein Refactoring durchgeführt habe. Bei jeder Änderung habe ich nach dem Grund gefragt und ihn notiert. Das Ergebnis ist eine ziemlich lange Liste von Mängeln, die mir beim Lesen von Code auffallen.

Sie können diese Liste von Anfang bis Ende lesen oder als Referenz verwenden. Zu jeder Heuristik gibt es Querverweise, die auf Anwendungen im Text verweisen (siehe Anhang C).

17.1 Kommentare

C1: Ungeeignete Informationen

Kommentare sollten keine Informationen enthalten, die an anderen Stellen eines Systems besser aufgehoben wären (etwa in Ihrem Sourcecode-Control-System,

Ihrem Fehlerprotokoll-System oder einem anderen Aufzeichnungssystem). Änderungsverläufe (change histories) blähen Quellcode unnötig auf und enthalten nur historischen und uninteressanten Text. Im Allgemeinen sollten Metadaten, wie etwa Autoren, das Datum der letzten Änderung, SPR-Nummern (SPR = Software Problem Report) usw. nicht in Kommentaren erscheinen. Kommentare sollten technischen Anmerkungen zum Code und Design vorbehalten sein.

C2: Überholte Kommentare

Kommentare, die veraltet, irrelevant und nicht korrekt sind, sind überholt. Kommentare altern schnell. Am besten schreibt man keine Kommentare, die veralten können. Wenn Sie einen überholten Kommentar finden, sollten Sie ihn so schnell wie möglich aktualisieren oder entfernen. Häufig entfernen sich überholte Kommentare immer mehr von dem Code, den sie einmal beschrieben haben. Sie werden zu »schwimmenden Inseln« irrelevanter und irreführender Informationen im Code.

C3: Redundante Kommentare

Ein Kommentar ist redundant, wenn er etwas beschreibt, das sich hinreichend selbst beschreibt. Ein Beispiel:

```
i++; // i inkrementieren
```

Ein weiteres Beispiel ist eine Javadoc, die nicht mehr (oder sogar weniger) als die Signatur der Funktion aussagt:

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
    throws ManagedComponentException
```

Kommentare sollten Dinge ausdrücken, die der Code selbst nicht mitteilen kann.

C4: Schlecht geschriebene Kommentare

Ein Kommentar, den zu schreiben sich lohnt, sollte gut geschrieben sein. Wenn Sie einen Kommentar schreiben, sollten Sie sich die Zeit nehmen, ihn so gut wie möglich zu formulieren. Wählen Sie Ihre Worte sorgfältig. Verwenden Sie korrekte Grammatik und Zeichensetzung. Schwätzen Sie nicht. Schreiben Sie nichts, was offensichtlich ist. Fassen Sie sich kurz.

C5: Auskommentierter Code

Es macht mich verrückt, wenn ich lange Code-Bereiche sehe, die auskommentiert sind. Wer weiß, wie der Code ist? Wer weiß, ob der Code eine Bedeutung hat oder nicht? Dennoch löscht niemand diesen auskommentierten Code, weil jeder annimmt, jemand anderer würde ihn für seine Zwecke benötigen.

Dieser Code rottet einfach vor sich hin und wird mit jedem Tag irrelevanter. Er ruft Funktionen auf, die es nicht mehr gibt. Er verwendet Variablen, deren Namen sich geändert haben. Er folgt Konventionen, die längst passe sind. Er verunreinigt Module und lenkt Entwickler ab, die versuchen, sie zu lesen. Auskommentierter Code ist *abscheulich*.

Wenn Sie auskommentierten Code sehen, *löschen Sie ihn!* Keine Bange! Das Source-code-Control-System erinnert sich an den Code. Wenn er wirklich gebraucht wird, kann der betreffende Entwickler eine frühere Version auschecken. Lassen Sie sich von Auskommentiertem nicht die Freude an der Arbeit vergällen.

17.2 Umgebung

E1: Ein Build erfordert mehr als einen Schritt

Ein Projekt zu erstellen (ein Build), sollte eine einzige triviale Operation sein. Sie sollten nicht viele kleine Komponenten aus dem Sourcecode-Control-System auschecken müssen. Sie sollten keine Folge geheimnisvoller Befehle oder kontextabhängige Skripts eingeben müssen, um einzelne Elemente zu erstellen. Sie sollten nicht nah und fern nach verschiedenen kleinen zusätzlichen JARs, XML-Dateien und anderen Artefakten suchen müssen, die das System benötigt. Sie *sollten* das System mit einem einfachen Befehl auschecken und dann einen weiteren einfachen Befehl geben können, um es zu erstellen.

```
svn get mySystem  
cd mySystem  
ant all
```

E2: Tests erfordern mehr als einen Schritt

Sie sollten alle Unit-Tests mit nur einem Befehl ausführen können. Im besten Fall können Sie alle Tests ausführen, indem Sie einen Button in Ihrer DIE anklicken. Im schlimmsten Fall sollten Sie einen einzigen einfachen Befehl in einer Shell geben müssen. Alle Tests ausführen zu können, ist so grundlegend und so wichtig, dass es schnell, leicht und offensichtlich sein sollte.

17.3 Funktionen

F1: Zu viele Argumente

Funktionen sollten eine kleine Anzahl von Argumenten haben. Kein Argument ist am besten, gefolgt von einem, zwei und drei Argumenten. Mehr als drei Argumente sind sehr fragwürdig und sollten möglichst vermieden werden. (Siehe den Abschnitt *Funktionsargumente* in Kapitel 3.)

F2: Output-Argumente

Output-Argumente widersprechen der Intuition. Leser erwarten, dass Argumente keine Outputs, sondern Inputs sind. Wenn eine Funktion den Zustand einer Komponente ändern muss, soll sie den Status des Objekts ändern, für das sie aufgerufen wird. (Siehe den Abschnitt *Output-Argumente* in Kapitel 3.)

F3: Flag-Argumente

Boolesche Argumente bringen klar zum Ausdruck, dass die Funktion mehr als eine Aufgabe erfüllt. Sie sind verwirrend und sollten eliminiert werden. (Siehe den Abschnitt *Flag-Argumente* in Kapitel 3.)

F4: Tote Funktionen

Methoden, die nie aufgerufen werden, sollten gelöscht werden. Toten Code zu behalten, ist Platzverschwendung. Haben Sie keine Angst davor, die Funktion zu löschen. Denken Sie daran, dass sie immer noch in Ihrem Sourcecode-Control-System enthalten ist.

17.4 Allgemein

G1: Mehrere Sprachen in einer Quelldatei

Moderne Programmierumgebungen ermöglichen es, in einer einzigen Quelldatei verschiedene Sprachen zu verwenden. Beispielsweise kann eine Java-Quelldatei Abschnitte von XML, HTML, YAML, JavaDoc, Englisch/Deutsch, JavaScript usw. enthalten. Ein anderes Beispiel: Zusätzlich zu HTML kann eine JSP-Datei Java, eine Tag-Library-Syntax, englische Kommentare, Javadocs, XML, JavaScript usw. enthalten. Dies ist bestenfalls verwirrend und schlimmstenfalls äußerst nachlässig.

Idealerweise enthält eine Quelldatei eine und nur eine Sprache. Realistischerweise müssen wir wahrscheinlich mehr als eine verwenden. Aber wir sollten uns bemühen, sowohl die Anzahl als auch den Umfang zusätzlicher Sprachen in unseren Quelldateien auf ein Minimum zu beschränken.

G2: Offensichtliches Verhalten ist nicht implementiert

Nach dem Least-Surprise-Prinzip (http://en.wikipedia.org/wiki/Principle_of_least_astonishment; »Prinzip der kleinsten Überraschung«) sollte eine Funktion oder Klasse die Verhaltensweisen implementieren, die ein Programmierer vernünftigerweise erwarten darf. Betrachten Sie beispielsweise eine Funktion, die einen Tagesnamen in eine enum übersetzt, die den Tag repräsentiert:

```
Day day = DayDate.StringToDay(String dayName);
```

Wir würden erwarten, dass der String "Montag" in `Day.MONTAG` übersetzt wird. Wir würden auch erwarten, dass die gebräuchlichen Abkürzungen übersetzt werden, und wir würden erwarten, dass die Funktion die Groß/Kleinschreibung ignoriert.

Wenn eine offensichtliche Verhaltensweise nicht implementiert wird, können Leser und Benutzer des Codes Funktionsnamen nicht mehr intuitiv korrekt interpretieren. Sie verlieren ihr Vertrauen in den Autor des Codes und müssen wieder die Details des Codes genau studieren.

G3: Falsches Verhalten an den Grenzen

Offensichtlich sollte sich Code korrekt verhalten. Das Problem liegt darin, dass wir selten erkennen, wie kompliziert das korrekte Verhalten sein kann. Entwickler schreiben oft Funktionen, die ihrer Meinung nach korrekt arbeiten, und vertrauen dann ihrer Intuition, statt sich die Mühe zu machen zu testen, ob der Code auch in allen Grenzfällen funktioniert.

Es gibt keinen Ersatz für die gebotene Sorgfalt. Grenzbedingungen, Sonderfälle, Eigenheiten und Ausnahmen können einen eleganten und intuitiven Algorithmus verunstalten. *Verlassen Sie sich nicht auf Ihre Intuition.* Testen Sie alle Grenz- und Sonderfälle.

G4: Übergangene Sicherungen

Die Tschernobyl-Katastrophe trat ein, weil der Betriebsleiter alle Sicherheitsmechanismen nacheinander außer Kraft setzte. Die Sicherungen machten die Ausführung eines Experiments unbequem. Das Ergebnis ist bekannt: Das Experiment wurde nicht ausgeführt. Stattdessen erlebte die Welt ihre erste zivile Nuklearkatastrophe.

Es ist riskant, Sicherungen außer Kraft zu setzen. Die `serialVersionUID` manuell zu ändern, mag erforderlich sein, aber es ist immer riskant. Bestimmte (oder alle!) Compiler-Warnungen abzuschalten, mag Ihnen zu einem erfolgreichen Build verhelfen, aber Sie riskieren damit endlose Debugging-Sitzungen. Wenn Sie scheiternde Tests abschalten und sich einreden, Sie würden das Problem später beheben, ist das vergleichbar damit, Kreditkarten wie geschenktes Geld zu behandeln.

G5: Duplizierung

Dies ist eine der wichtigsten Regeln in diesem Buch. Sie sollten sie sehr ernst nehmen. Praktisch jeder Autor, der über Software-Design schreibt, erwähnt diese Regel. Dave Thomas und Andy Hunt bezeichnen sie als *DRY-Prinzip* (Don't Repeat Yourself. Wiederhole dich nicht. [PRAG]). Kent Beck erhebt sie zu den Kernprinzipien des Extreme Programming und drückt sie so aus: »Einmal, und nur einmal«. Bei Ron Jeffries kommt diese Regel an zweiter Stelle direkt nach der Regel, dass alle Tests bestanden werden müssen.

Jede Duplizierung von Code ist eine verpasste Gelegenheit zur Abstraktion. Sie könnten den Code wahrscheinlich in eine Subroutine oder vielleicht eine andere separate Klasse einfügen. Indem Sie die Duplizierung durch eine solche Abstraktion einkapseln, vergrößern Sie das Vokabular der Sprache Ihres Designs. Andere Programmierer können die abstrakten Funktionen nutzen, die Sie erstellen. Sie können schneller codieren und machen weniger Fehler, weil Sie das Abstraktionsniveau angehoben haben.

Die offensichtlichste Form der Duplizierung liegt vor, wenn Sie an mehreren Stellen eines Programms identischen Code verwenden. Es sieht so aus, als hätte der Programmierer ihn einfach per Cut&Paste überall eingefügt. Sie sollten diesen Code durch einfache Methoden ersetzen.

Eine subtilere Form der Duplizierung liegt vor, wenn dieselben `switch/case-` oder `if/else`-Strukturen wiederholt in verschiedenen Modulen auftauchen und immer dieselben Bedingungen testen. Sie sollten durch polymorphe Konstrukte ersetzt werden.

Eine noch subtilere Form der Duplizierung liegt vor, wenn Module ähnliche Algorithmen, aber mit unterschiedlichen Codezeilen verwenden. Dieses Problem kann mit dem *Template Method*- oder dem *Strategy*-Pattern (beide in [GOF]) gelöst werden.

Tatsächlich sind die meisten Design-Patterns, die im Laufe der letzten zwei Jahrzehnte veröffentlicht wurden, einfach wohlbekannte Lösungen, um die Duplizierung von Code zu vermeiden. So sind auch die Normalformen von Codd eine Strategie, um Duplizierung in Datenbankschemata zu vermeiden. OO selbst ist eine Strategie, um Module zu organisieren und Duplizierung zu vermeiden. Auch die strukturierte Programmierung hatte dies bereits zum Ziel.

Ich glaube, die Botschaft ist angekommen. Suchen und eliminieren Sie duplizierten Code, so weit dies möglich ist.

G6: Auf der falschen Abstraktionsebene codieren

Es ist wichtig, Abstraktionen zu erstellen, die allgemeinere Konzepte auf einer höheren Abstraktionsebene von konkreteren Konzepten auf einer niedrigeren Abstraktionsebene trennen. Manchmal verwenden wir zu diesem Zweck abstrakte Klassen für die allgemeineren Konzepte und abgeleitete Klassen für die konkreteren Kon-

zepte. Dabei müssen wir darauf achten, dass die Trennung vollständig ist. Alle konkreteren Konzepte müssen in den abgeleiteten Klassen stehen und alle allgemeineren Konzepte in der Basisklasse.

Beispielsweise sollten Konstanten, Variablen oder Utility-Funktionen, die nur für eine konkrete Implementierung gebraucht werden, nicht in der Basisklasse stehen. Die Basisklasse sollte nichts von ihnen wissen.

Diese Regel gilt auch für Quelldateien, Komponenten und Module. Gutes Software-Design verlangt, dass wir die Konzepte der verschiedenen Abstraktionsebenen trennen und in verschiedene Container einfügen. Manchmal sind diese Container Basisklassen oder abgeleitete Klassen und manchmal sind es Quelldateien, Module oder Komponenten. Auf jeden Fall muss die Trennung aber vollständig sein. Konkretere und allgemeinere Konzepte dürfen nicht vermischt werden.

Betrachten Sie den folgenden Code:

```
public interface Stack {  
    Object pop() throws EmptyException;  
    void push(Object o) throws FullException;  
    double percentFull();  
  
    class EmptyException extends Exception {}  
    class FullException extends Exception {}  
}
```

Die `percentFull`-Funktion befindet sich auf der falschen Abstraktionsebene. Obwohl es viele Stack-Implementierungen gibt, bei denen das Konzept der *fullness* (Füllung) sinnvoll anwendbar ist, gibt es andere Implementierungen, die einfach *nicht wissen können*, wie voll sie sind. Deshalb wäre die Funktion besser in einem abgeleiteten Klassen-Interface, etwa `BoundedStack`, aufgehoben.

Vielleicht denken Sie, die Implementierung könne einfach `null` zurückgeben, wenn der Stack unbegrenzt wäre. Das Problem ist, dass kein Stack wirklich unbegrenzt ist. Mit dem folgenden Test können Sie eine `OutOfMemoryException` nicht wirklich verhindern:

```
stack.percentFull() < 50.0.
```

Die Funktion so zu implementieren, dass sie 0 zurückgibt, wäre eine Lüge.

Fakt ist, dass man sich nicht mit einer Lüge oder einem erfundenen Wert aus einer falsch eingeordneten Abstraktion herauswinden kann. Abstraktionen zu isolieren, zählt zu den schwierigsten Aufgaben eines Software-Entwicklers; und es gibt keine schnelle Abhilfe, wenn Sie dabei einen Fehler machen.

G7: Basisklasse hängt von abgeleiteten Klassen ab

Der häufigste Grund, um Konzepte in Basisklassen und abgeleitete Klassen zu zerlegen, besteht darin, die Konzepte der höher angesiedelten Basisklassen unabhängig von den Konzepten der tiefer angesiedelten abgeleiteten Klassen verwenden zu können. Deshalb vermuten wir, wenn wir in Basisklassen Namen von abgeleiteten Klassen finden, ein Problem. Im Allgemeinen sollten Basisklassen nichts über ihre abgeleiteten Klassen wissen.

Natürlich gibt es Ausnahmen von dieser Regel. Manchmal ist die Anzahl der abgeleiteten Klassen streng festgelegt, und die Basisklasse enthält Code, der zwischen den abgeleiteten Klassen wählt. Dieser Fall tritt häufig bei Implementierungen von Finite State Machines (endlichen Automaten) auf. Doch in solchen Fällen sind die abgeleiteten Klassen und die Basisklasse stark gekoppelt und werden immer zusammen in derselben jar-Datei ausgeliefert. Im Allgemeinen sollten Basisklassen und abgeleitete Klassen in verschiedenen jar-Dateien geliefert werden können.

Indem wir Basisklassen und abgeleitete Klassen in verschiedenen jar-Dateien liefern und dafür sorgen, dass die jar-Dateien der Basisklassen nichts über den Inhalt der jar-Dateien der abgeleiteten Klassen wissen, können wir unsere Systeme in diskreten und unabhängigen Komponenten ausliefern. Werden einzelne Komponenten modifiziert, können sie separat ausgetauscht werden, ohne dass die Basiskomponenten ebenfalls ausgetauscht werden müssten. Dies bedeutet, dass die Auswirkungen einer Änderung erheblich eingeschränkt werden, die Wartung derartiger Systeme im Feld stark vereinfacht wird.

G8: Zu viele Informationen

Wohldefinierte Module haben sehr kleine Interfaces, die es Ihnen ermöglichen, mit wenig viel zu tun. Schlecht definierte Module haben breite und tiefe Interfaces, die Sie zwingen, viele verschiedene Handbewegungen auszuführen, um einfache Aufgaben zu erledigen. Ein wohldefiniertes Interface bietet nicht sehr viele Funktionen an, von denen man abhängig ist; deshalb ist die Kopplung gering. Ein schlecht definiertes Interface stellt zahlreiche Funktionen zur Verfügung, die Sie aufrufen müssen; deshalb ist die Kopplung stark.

Gute Software-Entwickler lernen, möglichst kleine Interfaces für ihre Klassen und Module zu erstellen. Je weniger Methoden eine Klasse hat, desto besser. Je weniger Variablen eine Funktion kennt, desto besser. Je weniger Instanzvariablen eine Klasse hat, desto besser.

Verbergen Sie Ihre Daten. Verbergen Sie Ihre Utility-Funktionen. Verbergen Sie Ihre Konstanten und Ihre temporären Elemente. Erstellen Sie keine Klassen mit zahlreichen Methoden oder Instanzvariablen. Erstellen Sie keine zahlreichen geschützten Variablen und Funktionen für Ihre Unterklassen. Konzentrieren Sie sich darauf, kleine und knappe Interfaces zu erstellen. Unterstützen Sie eine schwache Kopplung, indem Sie Informationen begrenzen.

G9: Toter Code

Toter Code ist Code, der nicht ausgeführt wird. Wo finden Sie toten Code? Im Body einer `if`-Anweisung, die eine Bedingung prüft, die nicht eintreten kann; im `catch`-Block einer `try`-Anweisung, die nie `throws`; in kleinen Utility-Methoden, die nie aufgerufen werden; oder in `switch/case`-Bedingungen, die niemals eintreten.

Das Problem ist: Toter Code fängt nach einiger Zeit an zu stinken. Je älter er ist, desto stärker und übler wird sein Geruch. Das liegt daran, dass toter Code nicht vollständig aktualisiert wird, wenn sich das Design ändert. Der Code wird zwar noch *kompiliert*, aber er folgt nicht den neueren Konventionen oder Regeln. Er wurde zu einer Zeit geschrieben, als das System *anders* war. Wenn Sie auf toten Code stoßen, sollten Sie das einzig Richtige tun: Begraben Sie ihn. Löschen Sie ihn aus dem System.

G10: Vertikale Trennung

Variablen und Funktion sollten nahe bei ihrem Anwendungsort definiert werden. Lokale Variablen sollten unmittelbar über ihrer ersten Verwendung deklariert werden und sollten über einen kleinen vertikalen Geltungsbereich verfügen. Lokale Variablen sollten nicht Hunderte von Zeilen entfernt von ihrem Verwendungsort deklariert werden.

Private Funktionen sollten direkt unter ihrer ersten Verwendung definiert werden. Private Funktionen gehören zum Geltungsbereich der gesamten Klasse, aber wir wollen dennoch den vertikalen Abstand zwischen den Aufrufen und den Definitionen minimieren. Um eine private Funktion zu finden, sollte man den Code nur von der ersten Anwendung an abwärts überfliegen müssen.

G11: Inkonsistenz

Wenn Sie etwas in einer gewissen Weise tun, sollten Sie alle ähnlichen Aufgaben auf dieselbe Weise erledigen. Diese geht auf das Prinzip der geringsten Überraschung zurück. Wählen Sie Ihre Konventionen sorgfältig aus; und wenn Sie sie gewählt haben, wenden Sie sie sorgfältig an.

Wenn Sie in einer speziellen Funktion eine Variable namens `response` zur Speicherung einer `HttpServletResponse` verwenden, sollten Sie denselben Variablennamen konsistent in den anderen Funktionen verwenden, die ebenfalls mit `HttpServletResponse`-Objekten arbeiten. Wenn Sie eine Methode `processVerificationRequest` nennen, sollten Sie einen ähnlichen Namen, wie etwa `processDeletionRequest`, für Methoden verwenden, die andere Arten von Anfragen verarbeiten.

Eine einfache Konsistenz wie diese kann Ihren Code bei zuverlässiger Anwendung viel lesbarer und änderungsfreundlicher machen.

G12: Müll

Welchen Nutzen hat ein Default-Konstruktor ohne Implementierung? Er dient allein dazu, den Code mit bedeutungslosen Artefakten zuzumüllen. Variablen, die nicht verwendet werden, Funktionen, die niemals aufgerufen werden, Kommentare, die keine Informationen hinzufügen, usw. Alle diese Dinge sind Müll und sollten beseitigt werden. Halten Sie Ihre Quelldateien sauber, gut organisiert und frei von Müll.

G13: Künstliche Kopplung

Dinge, die nicht voneinander abhängen, sollten nicht künstlich gekoppelt werden. Beispielsweise sollten allgemeine Enums nicht in speziellere Klassen eingeschlossen werden, weil dadurch die gesamte Anwendung gezwungen wird, diese spezielleren Klassen zu kennen. Dasselbe gilt für statische Allzweckfunktionen, die in speziellen Klassen deklariert werden.

Im Allgemeinen ist eine künstliche Kopplung eine Kopplung zwischen zwei Modulen, die keinem direkten Zweck dient. Sie ergibt sich, wenn Variablen, Konstanten oder Funktionen an einer zeitweilig bequemen, doch unpassenden Stelle deklariert werden. Dies zeugt von Faulheit und Nachlässigkeit.

Sie sollten sich die Zeit nehmen, herauszufinden, wo Funktionen, Konstanten und Variablen deklariert werden sollten. Fügen Sie sie nicht einfach an der gerade bequemsten Stelle ein, um sie dann dort zu lassen.

G14: Funktionsneid

Dieser Code Smell stammt von Martin Fowler ([Refactoring]). Die Methoden einer Klasse sollten sich für die Variablen und Funktionen der Klasse interessieren, zu der sie gehören, und nicht für die Variablen und Funktionen anderer Klassen. Wenn eine Methode Accessoren und Mutatoren eines anderen Objekts verwendet, um die Daten innerhalb dieses Objekts zu manipulieren, dann *beneidet* sie den Geltungsbereich der Klasse dieses anderen Objekts. Sie wünscht, sie würde sich innerhalb dieser anderen Klasse befinden, um direkt auf die Variablen zugreifen zu können, die sie manipuliert. Ein Beispiel:

```
public class HourlyPayCalculator {
    public Money calculateWeeklyPay(HourlyEmployee e) {
        int tenthRate = e.getTenthRate().getPennies();
        int tenthsWorked = e.getTenthsWorked();
        int straightTime = Math.min(400, tenthsWorked);
        int overTime = Math.max(0, tenthsWorked - straightTime);
        int straightPay = straightTime * tenthRate;
        int overTimePay = (int)Math.round(overTime*tenthRate*1.5);
```

```

    return new Money(straightPay + overtimePay);
}
}

```

Die `calculateWeeklyPay`-Methode greift auf das `HourlyEmployee`-Objekt zu, um die Daten abzurufen, mit denen sie arbeitet. Die `calculateWeeklyPay`-Methode *beneidet* den Geltungsbereich von `HourlyEmployee`. Sie »wünscht«, sie würde sich innerhalb von `HourlyEmployee` befinden.

Unter sonst gleichen Bedingungen sollten Sie Funktionsneid beseitigen, weil er die Interna einer Klasse einer anderen zugänglich macht. Doch manchmal ist Funktionsneid ein erforderliches Übel. Betrachten Sie den folgenden Code:

```

public class HourlyEmployeeReport {
    private HourlyEmployee employee ;

    public HourlyEmployeeReport(HourlyEmployee e) {
        this.employee = e;
    }

    String reportHours() {
        return String.format(
            "Name: %s\tHours:%d.%1d\n",
            employee.getName(),
            employee.getTenthsWorked()/10,
            employee.getTenthsWorked()%10);
    }
}

```

Offensichtlich beneidet die `reportHours`-Methode die `HourlyEmployee`-Klasse. Andererseits soll `HourlyEmployee` nichts über das Format des Berichts wissen. Würden wir diesen Format-String in die `HourlyEmployee`-Klasse einfügen, würden wir gegen mehrere Prinzipien des Object Oriented Designs verstoßen (insbesondere gegen das Single-Responsibility-Prinzip, das Open-Closed-Prinzip und das Common-Closure-Prinzip; siehe [PPP]). Wir würden damit `HourlyEmployee` an das Format des Berichts koppeln und somit für Änderungen dieses Formats anfällig machen.

G15: Selektor-Argumente

Es gibt kaum etwas Abscheulicheres als ein hängendes `false`-Argument am Ende eines Funktionsaufrufs. Was bedeutet es? Was würde passieren, wenn es den Wert `true` hätte? Es ist nicht nur schwierig, sich den Zweck eines Selektor-Arguments zu merken, sondern jedes Selektor-Argument fasst viele Funktionen zu einer zusammen. Selektor-Argumente werden von Entwicklern verwendet, die zu faul sind, eine

große Funktion in mehrere kleinere Funktionen zu zerlegen. Betrachten Sie folgendes Beispiel:

```
public int calculateWeeklyPay(boolean overtime) {
    int tenthRate = getTenthRate();
    int tenthsWorked = getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    double overTimeRate = overtime ? 1.5 : 1.0 * tenthRate;
    int overTimePay = (int) Math.round(overTime * overTimeRate);
    return straightPay + overTimePay;
}
```

Die Funktion wird mit dem Argument `true` aufgerufen, wenn Überstunden mit dem Anderthalbfachen bezahlt werden, und mit `false`, wenn Überstunden nach einfachem Zeitaufwand bezahlt werden. Es ist schlimm genug, dass Sie sich daran erinnern müssen, was `calculateWeeklyPay(false)` bedeutet, wenn Sie zufällig darauf stoßen. Schlimmer ist jedoch, dass der Autor die Gelegenheit versäumt hat, den folgenden Code zu schreiben:

```
public int straightPay() {
    return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
    int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
    int overTimePay = overTimeBonus(overTimeTenths);
    return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
    double bonus = 0.5 * getTenthRate() * overTimeTenths;
    return (int) Math.round(bonus);
}
```

Natürlich müssen Selektoren keine booleschen Werte sein. Es kann sich um Enums, Ganzzahlen oder Argumente anderen Typs handeln, mit denen das Verhalten der Funktion ausgewählt werden kann. Im Allgemeinen ist es besser, mehrere Funktionen zu verwenden, als einer Funktion einen Code zu übergeben, mit dem eine Verhaltensweise ausgewählt wird.

G16: Verdeckte Absicht

Code sollte so ausdrucksstark wie möglich sein. Run-on-Ausdrücke, Hungarian Notation (ungarische Notation) und magische Zahlen verbergen alle die Absicht des

Autors. Hier ist beispielsweise die `overTimePay`-Funktion, wie sie hätte aussehen können:

```
public int m_otCalc() {
    return iThsWkd * iThsRte +
        (int) Math.round(0.5 * iThsRte *
            Math.max(0, iThsWkd - 400)
        );
}
```

Dies sieht zwar klein und dicht aus, ist aber praktisch und verständlich. Es lohnt sich, die Zeit aufzuwenden, um dem Leser den Zweck dieses Codes verständlich darzustellen.

G17: Falsche Zuständigkeit

Eine der wichtigsten Entscheidungen eines Software-Entwicklers betrifft die Stelle, an der er Code einfügt. Wo sollte beispielsweise die Konstante `PI` gespeichert werden? In der `Math`-Klasse? Vielleicht in der `Trigonometry`-Klasse? Oder doch in der `Circle`-Klasse?

Hier kommt das Least-Surprise-Prinzip (Prinzip der geringsten Überraschung) ins Spiel. Code sollte dort eingefügt werden, wo ihn ein Leser natürlicherweise erwarten würde. Die Konstante `PI` sollte dort stehen, wo die trigonometrischen Funktionen deklariert werden. Die Konstante `OVERTIME_RATE` sollte in der `HourlyPayCalculator`-Klasse deklariert werden.

Manchmal wollen wir »besonders schlau« vorgehen. Wir fügen eine bestimmte Funktionalität hinzu, die für uns zwar bequem, aber für den Leser nicht unbedingt intuitiv ist. Ein Beispiel: Angenommen, wir müssten einen Bericht mit der Anzahl der gesamten Arbeitsstunden eines Mitarbeiters erstellen. Wir könnten diese Stunden in dem Code addieren, der den Bericht erstellt. Alternativ könnten wir versuchen, eine laufende Summe in dem Code zu berechnen, mit dem die Zeitkarten erfasst werden.

Eine Möglichkeit, diese Entscheidung zu treffen, besteht darin, die Namen der Funktionen zu studieren. Angenommen, unser Berichtsmodul habe eine Funktion namens `getTotalHours` und das Modul, mit dem die Zeitkarten erfasst werden, habe eine Funktion namens `saveTimeCard`. Welche dieser beiden Funktionen wird wohl, wenn man von ihrem Namen ausgeht, die Summe berechnen? Die Antwort sollte offensichtlich sein.

Sicher gibt es manchmal Performance-Gründe, warum die Summe nicht bei der Erstellung des Berichts, sondern bereits bei der Erfassung der Zeitkarten berechnet werden sollte. Das ist in Ordnung; aber die Namen der Funktionen sollten dieses Verhalten zum Ausdruck bringen. Beispielsweise sollte es dann in dem `timecard`-Modul eine `computeRunningTotalOfHours`-Funktion geben.

G18: Fälschlich als statisch deklarierte Methoden

`Math.max(double a, double b)` ist eine gute statische Methode. Sie arbeitet nicht mit einer einzelnen Instanz. Tatsächlich wäre es dummlich, wenn man `new Math().max(a, b)` oder auch nur `a.max(b)` sagen müsste. Alle Daten, die `max` verwendet, stammen aus ihren beiden Argumenten und daher, dass sie die »Eigentümerin« von Objekten ist. Noch präziser gesagt: Es ist *kaum vorstellbar*, dass `Math.max` polymorph sein sollte.

Doch manchmal schreiben wir statische Funktionen, die nicht statisch sein sollten. Betrachten Sie beispielsweise:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Auch diese Funktion sieht wie eine vernünftige statische Funktion aus. Sie bearbeitet kein spezielles Objekt und erhält alle ihre Daten von ihren Argumenten. Doch es besteht eine gewisse Möglichkeit, dass diese Funktion polymorph sein sollte. Es könnte andere Algorithmen geben, um den Stundenverdienst zu berechnen, etwa `OvertimeHourlyPayCalculator` oder `StraightTimeHourlyPayCalculator`. Deshalb sollte die Funktion in diesem Fall nicht statisch, sondern eine nicht-statische Mitgliedsfunktion von `Employee` sein.

Im Allgemeinen sollten Sie nicht-statische Methoden den statischen vorziehen. Im Zweifelsfall sollten Sie eine Funktion als nicht-statisch deklarieren. Wenn eine Funktion wirklich statisch sein soll, sollten Sie dafür sorgen, dass sie sich bei keiner Gelegenheit polymorph verhalten soll.

G19: Aussagekräftige Variablen verwenden

Kent Beck schrieb in seinem großartigen Buch *Smalltalk Best Practice Patterns* ([Beck97], S. 108) und dann vor jüngerer Zeit im wieder gleichermaßen großartigen *Implementation Patterns* ([Becko7]) über dieses Thema. Eine der wirksameren Methoden, ein Programm lesbarer zu machen, besteht darin, Berechnungen in Teilschritte zu zerlegen und die Zwischenwerte Variablen mit aussagekräftigen Namen zuzuweisen.

Betrachten Sie das folgende Beispiel aus `FitNesse`:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
    String key = match.group(1);
    String value = match.group(2);
    headers.put(key.toLowerCase(), value);
}
```

Einfache aussagekräftige Variablen machen es klar, dass die erste übereinstimmende Gruppe den *key* (Schlüssel) und die zweite den *value* (Wert) liefert.

Man kann diese Technik kaum übertreiben. Mehr aussagekräftige Variablen sind im Allgemeinen besser als weniger. Es ist erstaunlich, wie ein undurchsichtiges Modul plötzlich transparent wird, indem einfach die Berechnungen in wohlbenannte Zwischenwerte zerlegt werden.

G20: Funktionsname sollte die Aktion ausdrücken

Betrachten Sie folgenden Code:

```
Date newDate = date.add(5);
```

Würden Sie erwarten, dass damit fünf Tage zu dem Datum addiert werden? Oder Wochen, oder Stunden? Wird die `date`-Instanz geändert, oder gibt die Funktion nur ein neues Datum zurück, ohne das alte zu ändern? *Aus dem Code können Sie nicht ablesen, was die Funktion tut.*

Wenn die Funktion fünf Tage zu dem Datum addiert und das Datum ändert, dann sollte die Funktion `addDaysTo` oder `increaseByDays` heißen. Wenn die Funktion dagegen ein neues Datum zurückgibt, das fünf Tage später liegt, aber die `date`-Instanz nicht ändert, sollte sie `daysLater` oder `daysSince` heißen.

Wenn Sie die Implementierung (oder Dokumentation) der Funktion heranziehen müssen, um zu erfahren, was sie tut, sollten Sie sich bemühen, einen besseren Namen zu finden oder die Funktionalität so umzustrukturieren, dass sie in Funktionen mit besseren Namen eingefügt werden kann.

G21: Den Algorithmus verstehen

Es wird sehr viel seltsamer Code geschrieben, weil sich Entwickler nicht die Zeit nehmen, den Algorithmus zu verstehen. Sie bringen den Code zum Laufen, indem sie genügend if-Anweisungen und Flags in den Code einfügen, ohne sich zu überlegen, was wirklich passiert.

Programmierung ist oft eine Erforschung. Sie *glauben*, dass Sie den richtigen Algorithmus für ein Problem kennen, aber dann massieren Sie den Code hier und da und ändern dieses und jenes, bis er »funktioniert«. Woher wissen Sie, dass er »funktioniert«? Weil er die Testfälle bestanden hat, die Sie sich ausgedacht haben.

An diesem Ansatz ist nichts falsch. Tatsächlich ist er oft die einzige Methode, eine Funktion zur Erfüllung der gewünschten Aufgabe zu veranlassen. Doch es reicht nicht aus, hinter dem Wort »funktioniert« ein Fragezeichen stehen zu lassen.

Bevor Sie eine Funktion als fertig betrachten, sollten Sie sicher sein, dass Sie *verstehen*, wie sie funktioniert. Es ist nicht gut genug, dass sie alle Tests besteht. Sie müssen *wissen*, dass die Lösung korrekt ist. (Zu wissen, dass der Code funktioniert, ist etwas anderes, als zu wissen, dass der Algorithmus die Aufgabe erfolgreich löst. Unsicher zu sein, ob ein Algorithmus dem Problem angemessen ist, ist oft ein Faktum, mit dem man leben muss. Nicht zu wissen, was der Code tut, ist nur Faulheit.)

Oft kann man dieses Wissen und Verständnis am besten erlangen, indem man das Refactoring der Funktion so durchgeführt hat, dass der Code so sauber und ausdrucksstark ist, dass seine Arbeitsweise offensichtlich ist.

G22: Logische Abhängigkeiten in physische umwandeln

Wenn ein Modul von einem anderen abhängt, sollte diese Abhängigkeit nicht nur logisch, sondern physisch existieren. Das abhängige Modul sollte keine Annahme (anders ausgedrückt: logische Abhängigkeiten) über das Modul machen, von dem es abhängt. Stattdessen sollte es das Modul ausdrücklich nach allen Informationen fragen, von denen es abhängt.

Nehmen Sie beispielsweise an, Sie schreiben eine Funktion, die einen einfachen Textbericht der Arbeitsstunden der Mitarbeiter ausgibt. Eine Klasse namens `HourlyReporter` sammelt alle Daten in einer bequemen Form und übergibt sie dann an die Druckfunktion `HourlyReportFormatter` (siehe Listing 17.1).

```
public class HourlyReporter {
    private HourlyReportFormatter formatter;
    private List<LineItem> page;
    private final int PAGE_SIZE = 55;

    public HourlyReporter(HourlyReportFormatter formatter) {
        this.formatter = formatter;
        page = new ArrayList<LineItem>();
    }

    public void generateReport(List<HourlyEmployee> employees) {
        for (HourlyEmployee e : employees) {
            addLineItemToPage(e);
            if (page.size() == PAGE_SIZE)
                printAndClearItemList();
        }
        if (page.size() > 0)
            printAndClearItemList();
    }

    private void printAndClearItemList() {
        formatter.format(page);
        page.clear();
    }

    private void addLineItemToPage(HourlyEmployee e) {
        LineItem item = new LineItem();
        item.name = e.getName();
        item.hours = e.getTenthsWorked() / 10;
        item.tenths = e.getTenthsWorked() % 10;
        page.add(item);
    }
}
```

```
public class LineItem {  
    public String name;  
    public int hours;  
    public int tenths;  
}  
}  
HourlyReporter.java
```

Dieser Code enthält eine logische Abhängigkeit, die nicht in eine physische umgewandelt werden kann. Können Sie sie entdecken? Es ist die Konstante `PAGE_SIZE`. Warum sollte `HourlyReporter` die Seitengröße kennen? Die Seitengröße sollte Zuständigkeit von `HourlyReportFormatter` sein.

Die Tatsache, dass `PAGE_SIZE` in `HourlyReporter` deklariert wird, ist eine falsche Zuständigkeit (siehe [G17]), die `HourlyReporter` zu der Annahme veranlasst, die Seitengröße zu kennen. Eine solche Annahme ist eine logische Abhängigkeit. `HourlyReporter` hängt von der Tatsache ab, dass `HourlyReportFormatter` mit einer Seitengröße von 55 umgehen kann. Kann eine Implementierung von `HourlyReportFormatter` mit dieser Größe nicht umgehen, läge ein Fehler vor.

Wir können diese Abhängigkeit in eine physische umwandeln, indem wir eine neue Methode namens `getMaxPageSize()` in `HourlyReportFormatter` erstellen. `HourlyReporter` ruft dann diese Funktion auf, anstatt die `PAGE_SIZE`-Konstante zu verwenden.

G23: Polymorphismus statt If/Else oder Switch/Case verwenden

In Anbetracht des Themas von Kapitel 6 mag diese Regel etwas seltsam klingen. Schließlich plädiere ich in diesem Kapitel dafür, dass `switch`-Anweisungen wahrscheinlich in Teilen des Systems geeigneter sind, in denen eher neue Funktionen als neue Typen hinzugefügt werden.

Erstens: Die meisten Entwickler verwenden `switch`-Anweisungen, weil sie die offensichtliche Brute-Force-Lösung darstellen, und nicht, weil sie die richtige Lösung für die Situation repräsentieren. Deshalb soll uns diese Heuristik daran erinnern, dass wir zunächst eine polymorphe Lösung erwägen sollten, bevor wir eine `switch`-Anweisung verwenden.

Zweitens: Die Fälle, in denen Funktionen volatiler sind als Typen, sind relativ selten. Deshalb sollte *jede* `switch`-Anweisung mit Argwohn betrachtet werden.

Ich arbeite mit der folgenden »One Switch«-Regel: *Für eine bestimmte Art von Auswahl darf es nicht mehr als eine `switch`-Anweisung geben. Die Fälle in dieser `switch`-Anweisung müssen polymorphe Objekte erstellen, die an die Stelle der gleichartigen `switch`-Anweisungen im Rest des Systems treten.*

G24: Konventionen beachten

Jedes Team sollte einem Codierstandard folgen, der auf gebräuchlichen Normen der Branche basiert. Dieser Codierstandard sollte folgende Aufgaben spezifizieren: Wie sollen Instanzvariablen deklariert werden; wie sollten Klassen, Methoden und Variablen benannt werden; wo sollten Klammern stehen; usw. Das Team sollte kein Dokument benötigen, um diese Konventionen zu beschreiben, weil der Code die Beispiele liefert.

Jedes Teammitglied sollte diese Konventionen beachten. Dies bedeutet, dass jedes Teammitglied reifgenug sein muss zu erkennen, dass es absolut keine Rolle spielt, wo Klammern gesetzt werden, solange alle dieselbe Form verwenden.

Wenn Sie meine Konventionen kennen lernen wollen, sollten Sie den Code nach der Durchführung des Refactorings in den Listings B.7 bis B.14 studieren.

G25: Magische Zahlen durch benannte Konstanten ersetzen

Dies ist wahrscheinlich eine der ältesten Regeln der Software-Entwicklung. Ich erinnere mich, dass ich sie bereits in den späten 60er-Jahren in Einführungen in COBOL, FORTRAN und PL/I gelesen habe. Im Allgemeinen gilt es als Kunstfehler, nackte Zahlen in den Code einzufügen. Sie sollten hinter wohlbenannten Konstanten verborgen werden.

Beispielsweise sollte die Zahl 86.400 hinter der Konstanten SECONDS_PER_DAY verborgen werden. Wenn Sie 55 Zeilen pro Seite drucken, sollte die Konstante 55 hinter einer Konstanten namens LINES_PER_PAGE verborgen werden.

Einige Konstanten sind so leicht zu erkennen, dass man sie nicht immer hinter benannten Konstanten verbergen muss, wenn sie in Code verwendet werden, der selbsterklärend ist. Ein Beispiel:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Brauchen wir in diesen Beispielen wirklich die Konstanten FEET_PER_MILE (Fuß pro Meile) WORK_HOURS_PER_DAY (Arbeitsstunden pro Tag) und TWO? Gerade im letzten Fall wäre dies absurd. Es gibt einige Formeln, in denen Konstanten einfach besser als nackte Zahlen geschrieben werden. Man könnte über den Fall WORK_HOURS_PER_DAY streiten, weil sich die Gesetze oder Konventionen ändern könnten. Andererseits ist die Formel mit der 8 so gut zu lesen, dass ich zögern würde, den Leser mit 17 zusätzlichen Zeichen zu belasten. Und bei FEET_PER_MILE ist die Zahl 5280 als Konstante so gut bekannt und so eindeutig, dass (angelsächsische!) Leser sie selbst dann erkennen würden, wenn sie ohne umgebenden Text alleine auf einer Seite stehen würde.

Konstanten wie 3,141592653589793 sind ebenfalls sehr bekannt und leicht erkennbar. Jedoch ist die Fehlermöglichkeit zu groß, um den nackten Wert zu benutzen.

Liest jemand 3,141592753589793, denkt er automatisch an und prüft die Zahl nicht genauer. (Haben Sie den Fehler bemerkt? Eine Ziffer ist falsch.) Wir wollen auch nicht, dass Entwickler 3,14, 3,14159, 3,142 usw. verwenden. Deshalb ist es gut, dass der Wert mit *Math.PI* bereits definiert ist.

Der Terminus »Magische Zahl« bezieht sich nicht nur auf Zahlen, sondern auf alle Tokens, deren Wert nicht selbstbeschreibend ist. Ein Beispiel:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

Diese Zusicherung enthält zwei magische Zahlen. Die erste ist offensichtlich 7777, obwohl deren Bedeutung nicht offensichtlich ist. Die zweite ist "John Doe", und auch hier ist der Zweck unklar.

Es stellt sich heraus, dass "John Doe" der Name des Mitarbeiters mit der Personalnummer 7777 in einer gebräuchlichen Testdatenbank ist, die von unserem Team erstellt wurde. Jedes Teammitglied weiß, dass diese Datenbank bereits einige »vorgefertigte« Mitarbeiter mit bekannten Werten und Attributen enthält. Es stellt sich ebenfalls heraus, dass "John Doe" den einzigen Mitarbeiter in dieser Testdatenbank repräsentiert, der auf Stundenbasis arbeitet. Deshalb sollte dieser Test eigentlich wie folgt formuliert sein:

```
assertEquals(
    HOURLY_EMPLOYEE_ID,
    Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26: Präzise sein

Zu erwarten, dass der erste Treffer einer Abfrage der *einzige* Treffer ist, ist wahrscheinlich naiv. Fließkommazahlen zur Repräsentation von Geldbeträgen zu verwenden, ist fast kriminell. Locks und/oder Transaktionsverwaltung auszulassen, weil Sie nebenläufige Updates für unwahrscheinlich halten, ist bestenfalls Faulheit. Eine Variable als `ArrayList` zu deklarieren, wenn eine `List` ausreichen würde, ist zu einschränkend. Alle Variablen per Default als `protected` zu deklarieren, ist nicht einschränkend genug.

Wenn Sie in Ihrem Code eine Entscheidung treffen, müssen Sie so *präzise* wie möglich sein. Sie müssen wissen, warum Sie sie getroffen haben und wie Sie Ausnahmen umgehen wollen. Behandeln Sie die Präzision Ihrer Entscheidungen nicht als Nebensächlichkeit. Wenn Sie beschließen, eine Funktion aufzurufen, die `null` zurückgeben könnte, sollten Sie den Rückgabewert `null` prüfen. Wenn Sie den Ihrer Meinung nach einzigen Datensatz in der Datenbank abrufen, sollten Sie prüfen, ob es keine anderen Datensätze gibt. Wenn Sie mit Währungsbeträgen rechnen, sollten Sie Ganzzahlen und passende Rundungsfunktionen (oder noch besser: eine `Money`-Klasse) verwenden. Wenn die Möglichkeit nebenläufiger Updates besteht, sollten Sie einen Locking-Mechanismus implementieren.

Mehrdeutigkeiten und Ungenauigkeiten im Code sind entweder die Folge von Missverständnissen oder von Faulheit. Beides ist nicht akzeptabel.

G27: Struktur ist wichtiger als Konvention

Setzen Sie Design-Entscheidungen mit Hilfe der Struktur, nicht durch Konventionen durch. Namenskonventionen sind gut, aber nicht so gut wie Strukturen, die Konformität erzwingen. Beispielsweise sind `switch/case`-Anweisungen mit sauberen Namens-Enumerationen nicht so gut wie Basisklassen mit abstrakten Methoden. Niemand ist gezwungen, die `switch/case`-Anweisung jedes Mal auf dieselbe Weise zu implementieren; aber die Basisklassen erzwingen, dass konkrete Klassen alle abstrakten Methoden implementieren.

G28: Bedingungen einkapseln

Boolesche Logik ist auch ohne den Kontext einer `if`- oder `while`-Anweisung schwer genug zu verstehen. Extrahieren Sie Funktionen, die den Zweck der Bedingung zum Ausdruck bringen.

Ein Beispiel:

```
if (shouldBeDeleted(timer))
```

ist besser als:

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29: Negative Bedingungen vermeiden

Negativ formulierte Bedingungen sind schwerer zu verstehen als positiv formulierte. Wenn möglich, sollten Sie deshalb Bedingungen positiv formulieren. Ein Beispiel:

```
if (buffer.shouldCompact())
```

ist besser als:

```
if (!buffer.should-Not-Compact())
```

G30: Eine Aufgabe pro Funktion!

Es ist oft verlockend, Funktionen zu erstellen, die mehrere Abschnitte enthalten, die eine Reihe von Operationen ausführen. Derartige Funktionen erledigen *mehr als eine Aufgabe* und sollten in mehrere kleinere Funktionen zerlegt werden, die jeweils genau *eine Aufgabe* erfüllen.

Ein Beispiel:

```
public void pay() {
    for (Employee e : employees) {
        if (e.isPayday()) {
            Money pay = e.calculatePay();
            e.deliverPay(pay);
        }
    }
}
```

Dieser Code erledigt drei Aufgaben: Er durchläuft alle Mitarbeiter in einer Schleife; er prüft, ob ein Mitarbeiter bezahlt werden muss; und dann bezahlt er den Mitarbeiter. Besser wäre es, diesen Code wie folgt zu zerlegen:

```
public void pay() {
    for (Employee e : employees)
        payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
    if (e.isPayday())
        calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}
```

Jede dieser Funktionen erfüllt eine Aufgabe. (Siehe den Abschnitt *Erfülle eine Aufgabe* in Kapitel 1.)

G31: Verborgene zeitliche Kopplungen

Zeitliche Kopplungen sind oft erforderlich, aber Sie sollten die Kopplung nicht verbergen. Strukturieren Sie die Argumente Ihrer Funktionen so, dass die Reihenfolge, in der sie aufgerufen werden sollten, offensichtlich ist. Betrachten Sie den folgenden Code:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
        saturateGradient();
        reticulateSplines();
        diveForMoog(reason);
    }
    ...
}
```


Die Reihenfolge der drei Funktionen ist wichtig. Sie müssen den Gradienten saturieren, bevor Sie das Netz der Splines auslegen können, und erst dann kann sie nach dem Moog tauchen. Leider erzwingt der Code diese zeitliche Kopplung nicht. Ein anderer Programmierer könnte `reticulateSplines` vor `saturateGradient` aufrufen und so eine `UnsaturatedGradientException` auslösen. Folgende Lösung wäre besser:

```
public class MoogDiver {
    Gradient gradient;
    List<Spline> splines;

    public void dive(String reason) {
        Gradient gradient = saturateGradient();
        List<Spline> splines = reticulateSplines(gradient);
        diveForMoog(splines, reason);
    }
    ...
}
```

Dieser Code zeigt die zeitliche Kopplung, indem er eine so genannte *Bucket Brigade* (Eimerkette) erzeugt. Jede Funktion erzeugt ein Ergebnis, das von der nächsten Funktion benötigt wird; deshalb gibt es keine vernünftige Methode, die Funktionen in der falschen Reihenfolge aufzurufen.

Natürlich wird dadurch die Komplexität der Funktionen erhöht. Aber diese zusätzliche syntaktische Komplexität verdeutlicht die tatsächliche zeitliche Komplexität der Situation.

Beachten Sie, dass ich die Instanzvariablen nicht geändert habe. Ich nehme an, dass sie von privaten Methoden in der Klasse benötigt werden. Doch selbst dann möchte ich die Argumente lokal verwenden, um die zeitliche Kopplung zum Ausdruck zu bringen.

G32: Keine Willkür

Jede Struktur in Ihrem Code muss hinreichend begründet sein, und der Grund sollte aus der Struktur des Codes hervorgehen. Falls eine Struktur willkürlich wirkt, werden andere versucht sein, sie zu ändern. Wenn eine Struktur konsistent im ganzen System verwendet wird, werden andere sie ebenfalls benutzen und die Konvention bewahren. Beispielsweise führte ich vor Kurzem einen Merge von Änderungen an `FitNesse` durch und entdeckte dabei, dass einer unserer Committer Folgendes getan hatte:

```
public class AliasLinkWidget extends ParentWidget
{
    public static class VariableExpandingWidgetRoot {
        ...
    }
    ...
}
```

Das Problem dabei war, dass `VariableExpandingWidgetRoot` gar nicht im Geltungsbereich von `AliasLinkWidget` benötigt wurde. Darüber hinaus wurde die innere Klasse `AliasLinkWidget.VariableExpandingWidgetRoot` von anderen Klassen verwendet, die nichts über `AliasLinkWidget` wissen mussten.

Vielleicht hatte der Programmierer `VariableExpandingWidgetRoot` aus Gründen der Bequemlichkeit in `AliasWidget` eingefügt; vielleicht dachte er auch, sie würde wirklich im Geltungsbereich von `AliasWidget` benötigt werden. Aus welchem Grund auch immer: Das Ergebnis ist willkürlich. Öffentliche Klassen, die keine Hilfsklassen einer anderen Klasse sind, sollten nicht im Geltungsbereich einer anderen Klasse liegen. Per Konvention werden sie auf der obersten Ebene eines Packages als `public` deklariert.

G33: Grenzbedingungen einkapseln

Grenzbedingungen sind schwer zu kontrollieren. Fassen Sie ihre Verarbeitung an einer Stelle zusammen. Verteilen Sie sie nicht über Ihren gesamten Code. Wir wollen nicht überall auf `+1`- und `-1`-Ausdrücke stoßen. Betrachten Sie das folgende einfache Beispiel aus FIT:

```
if(level + 1 < tags.length)
{
    parts = new Parse(body, tags, level + 1, offset + endTag);
    body = null;
}
```

Beachten Sie, dass `level+1` zweimal auftaucht. Dies ist eine Grenzbedingung, die in einer Variablen mit dem Namen `nextLevel` (oder so ähnlich) einkapselt werden sollte.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
    parts = new Parse(body, tags, nextLevel, offset + endTag);
    body = null;
}
```

G34: In Funktionen nur eine Abstraktionsebene tiefer gehen

Die Anweisungen innerhalb einer Funktion sollten alle auf derselben Abstraktionsebene liegen, und zwar genau eine Ebene unter der Operation, die durch den Namen der Funktion beschrieben wird. Diese Heuristik zählt zu denen, die am schwersten umzusetzen sind. Auch wenn die Idee einfach genug ist, sind Menschen einfach zu gut darin, Abstraktionsebenen nahtlos zu vermengen. Betrachten Sie etwa den folgenden Code aus `FitNesse`:

```
public String render() throws Exception
{
    StringBuffer html = new StringBuffer("<hr");
    if(size > 0)
        html.append(" size=\"").append(size + 1).append("\"");
    html.append(">");

    return html.toString();
}
```

Ein kurzes Studium des Codes zeigt Ihnen, was passiert. Diese Funktion konstruiert das HTML-Tag, das eine horizontale Linie auf einer Seite anzeigt. Die Höhe der Linie wird durch die `size`-Variable festgelegt.

Schauen Sie noch einmal hin. Diese Methode vermengt wenigstens zwei Abstraktionsebenen. Die erste besteht darin, dass eine horizontale Linie eine `size`-Variable hat. Die zweite betrifft die Syntax des HR-Tags selbst. Dieser Code stammt aus dem `HrUleWidget`-Modul in `FitNesse`. Dieses Modul entdeckt eine Reihe aus vier oder mehr Bindestrichen und wandelt sie in das entsprechende HR-Tag um. Je mehr Bindestriche, desto länger die Linie.

Ich habe das Refactoring des Codes wie folgt durchgeführt. Beachten Sie, dass ich den Namen des `size`-Feldes so geändert habe, dass er dessen wahren Zweck ausdrückt. Es enthält die Anzahl der zusätzlichen Bindestriche.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (extraDashes > 0)
        hr.addAttribute("size", hrSize(extraDashes));
    return hr.html();
}

private String hrSize(int height)
{
    int hrSize = height + 1;
    return String.format("%d", hrSize);
}
```

Diese Änderung trennt die beiden Abstraktionsebenen sauber. Die `render`-Funktion konstruiert einfach ein HR-Tag, ohne dass sie etwas über die HTML-Syntax dieses Tags wissen muss. Das `HtmlTag`-Modul kümmert sich um alle lästigen Syntaxprobleme.

Tatsächlich habe ich durch diese Änderung einen subtilen Fehler entdeckt und beseitigt. Der ursprüngliche Code fügt keinen schließenden Schrägstrich in das HR-Tag ein, wie es von dem XHTML-Standard gefordert wird. (Anders ausgedrückt: Er gab `<hr>` statt `<hr />` aus.) Das `HtmlTag`-Modul wurde bereits vor langer Zeit an XHTML angepasst.

Die Trennung der Abstraktionsebenen ist eine der wichtigsten Funktionen des Refactorings. Zugleich gehört diese Aufgabe zu den schwierigsten. Ein Beispiel: Der folgende Code zeigt meinen ersten Versuch, die Abstraktionsebenen in der *Hrule-Widget.render*-Methode zu trennen.

```
public String render() throws Exception
{
    HtmlTag hr = new HtmlTag("hr");
    if (size > 0) {
        hr.addAttribute("size", ""+(size+1));
    }
    return hr.html();
}
```

Mein Ziel bestand zu diesem Punkt darin, die erforderliche Trennung herzustellen und die Tests zu bestehen. Ich erreichte dieses Ziel leicht, aber das Ergebnis war eine Funktion, in der *immer noch* Abstraktionsebenen vermengt waren. In diesem Fall wurden die Ebenen der Konstruktion des HR-Tags und der Interpretation und Formatierung der *size*-Variablen vermengt. Dies verweist auf ein weiteres Phänomen: Wenn Sie eine Funktion nach Abstraktionsebenen zerlegen, entdecken Sie oft neue Abstraktionsebenen, die durch die vorhergehende Struktur verdeckt worden waren.

G35: Konfigurierbare Daten hoch ansiedeln

Eine Konstante wie etwa ein Standardwert oder ein Konfigurationswert, der bekannt ist und auf einer hohen Abstraktionsebene erwartet wird, sollte nicht in einer niedrig angesiedelten Funktion vergraben werden. Stellen Sie sie so zur Verfügung, dass sie von allgemeineren Funktionen als Argument an konkretere Funktionen übergeben werden kann. Betrachten Sie den folgenden Code aus FitNesse:

```
public static void main(String[] args) throws Exception
{
    Arguments arguments = parseCommandLine(args);
    ...
}

public class Arguments
{
    public static final String DEFAULT_PATH = ".";
    public static final String DEFAULT_ROOT = "FitNesseRoot";
    public static final int DEFAULT_PORT = 80;
    public static final int DEFAULT_VERSION_DAYS = 14;
    ...
}
```

Die Befehlszeilen-Argumente werden in der allerersten ausführbaren Zeile von FitNesse geparkt. Die Standardwerte dieser Argumente werden am Anfang der Argu-

ment-Klasse definiert. Sie müssen nicht in den niedrigen Ebenen des Systems nach Anweisungen wie der folgenden suchen:

```
if (arguments.port == 0) // 80 per Default verwenden
```

Die Konfigurations-Konstanten sind auf einer sehr hohen Ebene angesiedelt und können leicht geändert werden. Sie werden an den Rest der Anwendung weitergeleitet. Die unteren Ebenen der Anwendung haben keinen Einfluss auf die Werte dieser Konstanten.

G36: Transitive Navigation vermeiden

Im Allgemeinen soll ein einzelnes Modul nicht viel über andere Module wissen, mit denen es zusammenarbeitet. Konkreter gesagt: Wenn A mit B zusammenarbeitet und B mit C zusammenarbeitet, müssen Module, die A verwenden, nichts über C wissen. (So sollte es beispielsweise keine Konstrukte der Form `a.getB().getC().doSomething()` geben.)

Dies wird manchmal als *Law of Demeter* (Gesetz von Demeter) bezeichnet. Die Pragmatic Programmer nennen es »Writing Shy Code« ([PRAG], S. 138; »Schüchternen Code schreiben«). In jedem Fall geht es darum, zu gewährleisten, dass Module nur ihre unmittelbaren Mitarbeiter kennen, aber nichts über die Navigationskarte des gesamten Systems wissen.

Verwendeten viele Module Anweisungen der Form `a.getB().getC()`, wäre es schwierig, das Design und die Architektur zu ändern, um ein *Q* zwischen *B* und *C* einzuschieben. Sie müssten jede Instanz von `a.getB().getC()` suchen und in `a.getB().getQ().getC()` umwandeln. So werden Architekturen immer unflexibler. Zu viele Module wissen zu viel über die Architektur.

Stattdessen sollten unsere unmittelbaren Mitarbeiter alle benötigten Services anbieten. Wir sollten nicht den Objektgraphen des Systems nach der aufzurufenden Methode durchsuchen müssen, sondern einfach sagen können:

```
myCollaborator.doSomething().
```

17.5 Java

J1: Lange Importlisten durch Platzhalter vermeiden

Wenn Sie zwei oder mehr Klassen eines Packages verwenden, importieren Sie das gesamte Package mit

```
import package.*;
```

Lange Importlisten sind für den Leser abschreckend. Wir wollen den Anfang unserer Module nicht mit 80 Zeilen langen Importanweisungen verstopfen. Stattdessen

sollen die Importanweisungen eine knappe Aussage über die Packages machen, mit denen wir zusammenarbeiten.

Spezielle Importanweisungen sind fest einprogrammierte Abhängigkeiten, Importanweisungen mit Platzhaltern dagegen nicht. Wenn Sie eine spezielle Klasse importieren, dann *muss* diese Klasse existieren. Doch wenn Sie ein Package mit einem Platzhalter importieren, muss keine spezielle Klassen existieren. Die Importanweisung fügt einfach das Package zu dem Suchpfad hinzu, auf dem nach Namen gesucht werden soll. Deshalb werden durch solche Importanweisungen keine echten Abhängigkeiten erzeugt; und deshalb dienen sie dazu, die Kopplung zwischen unseren Modulen zu verringern.

Manchmal kann eine lange Liste mit speziellen Importanweisungen auch nützlich sein. Wenn Sie beispielsweise mit Legacy-Code arbeiten und herausfinden wollen, welche Klassen Sie für die Erstellung von Mockups und Stubs benötigen, können Sie die Liste bis zu den speziellen Importanweisungen durchgehen, um die voll qualifizierten Namen all dieser Klassen zu ermitteln, und dann die passenden Stubs einfügen. Doch die Anwendungsmöglichkeiten für spezielle Importanweisungen sind sehr selten. Darüber hinaus ermöglichen es Ihnen die meisten modernen IDEs, Importanweisungen mit Platzhaltern mit einem einzigen Befehl in eine Liste mit speziellen Importanweisungen umzuwandeln. Deshalb ist es selbst im Legacy-Fall besser, Importanweisungen mit Platzhaltern zu verwenden.

Importanweisungen mit Platzhaltern können manchmal zu Namenskonflikten und Mehrdeutigkeiten führen. Zwei Klassen mit demselben Namen, aber in verschiedenen Packages, müssen speziell importiert werden oder wenigstens speziell qualifiziert werden, wenn sie verwendet werden. Dies kann ein Ärgernis sein, ist aber so selten, dass die Verwendung von Importanweisungen mit Platzhaltern im Allgemeinen immer noch besser ist als spezielle Importanweisungen.

J2: Keine Konstanten vererben

Mir ist dieses Problem mehrfach begegnet; und immer bekam ich Zahnschmerzen. Ein Programmierer fügt einige Konstanten in ein Interface ein und greift dann auf diese Konstanten zu, indem er das Interface vererbt. Betrachten Sie den folgenden Code:

```
public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    private double hourlyRate;

    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
        );
    }
}
```

```
}  
...  
}
```

Wo kommen die Konstanten `TENTHS_PER_WEEK` und `OVERTIME_RATE` her? Sie könnten aus der Klasse `Employee` stammen; werfen wir einen Blick darauf:

```
public abstract class Employee implements PayrollConstants {  
    public abstract boolean isPayday();  
    public abstract Money calculatePay();  
    public abstract void deliverPay(Money pay);  
}
```

Nada, nichts. Doch wo dann? Schauen Sie die Klasse `Employee` genauer an. Sie implementiert `PayrollConstants`.

```
public interface PayrollConstants {  
    public static final int TENTHS_PER_WEEK = 400;  
    public static final double OVERTIME_RATE = 1.5;  
}
```

Dies ist eine schreckliche Praxis! Die Konstanten sind in der Spitze der Vererbungshierarchie verborgen. Huch! Vererbung hat nicht den Zweck, die Sprachregeln für Geltungsbereiche zu umgehen. Verwenden Sie stattdessen einen statischen Import:

```
import static PayrollConstants.*;  
  
public class HourlyEmployee extends Employee {  
    private int tenthsWorked;  
    private double hourlyRate;  
  
    public Money calculatePay() {  
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);  
        int overTime = tenthsWorked - straightTime;  
        return new Money(  
            hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)  
        );  
    }  
    ...  
}
```

J3: Konstanten im Gegensatz zu Enums

Nachdem `enums` zu der Sprache (Java 5) hinzugefügt worden sind, sollten Sie sie auch verwenden! Bleiben Sie nicht bei dem alten Trick der `public static final ints`. Die Bedeutung von `ints` kann verloren gehen, die Bedeutung von `enums` nicht, weil sie zu einer benannten Enumeration gehört.

Darüber hinaus sollten Sie die Syntax für `enums` sorgfältig studieren. Sie können Methoden und Felder haben. Damit sind sie sehr leistungsstarke Tools, die Ihnen sehr viel mehr Ausdrucksmöglichkeiten und Flexibilität als `ints` bieten. Betrachten Sie die folgende Variation des Payroll-Codes:

```
public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    HourlyPayGrade grade;

    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
        );
    }
    ...
}

public enum HourlyPayGrade {
    APPRENTICE {
        public double rate() {
            return 1.0;
        }
    },
    LEUTENANT_JOURNEYMAN {
        public double rate() {
            return 1.2;
        }
    },
    JOURNEYMAN {
        public double rate() {
            return 1.5;
        }
    },
    MASTER {
        public double rate() {
            return 2.0;
        }
    }
};

public abstract double rate();
}
```


17.6 Namen

N1: Deskriptive Namen wählen

Wählen Sie einen Namen nicht zu schnell. Achten Sie darauf, dass der Name etwas aussagt. Denken Sie daran, dass sich oft Bedeutungen verschieben, während sich die Software entwickelt. Deshalb sollten Sie die Brauchbarkeit der gewählten Namen regelmäßig überprüfen.

Dies ist nicht nur eine Empfehlung, die Ihnen ein »angenehmes Gefühl« vermitteln soll. Die in der Software verwendeten Namen tragen 90 Prozent zur Lesbarkeit der Software bei. Sie müssen sich die Zeit nehmen, geeignete Namen zu wählen und ihre Relevanz zu erhalten. Namen sind zu wichtig, um nachlässig behandelt zu werden.

Betrachten Sie den folgenden Code. Was tut er? Wenn ich Ihnen den Code mit wohl-gewählten Namen zeigen würde, hätten Sie keine Probleme, ihn zu verstehen; doch in dieser Form ist er nur ein Mischmasch von Symbolen und magischen Zahlen.

```
public int x() {
    int q = 0;
    int z = 0;
    for (int kk = 0; kk < 10; kk++) {
        if (l[z] == 10)
        {
            q += 10 + (l[z + 1] + l[z + 2]);
            z += 1;
        }
        else if (l[z] + l[z + 1] == 10)
        {
            q += 10 + l[z + 2];
            z += 2;
        }
        else {
            q += l[z] + l[z + 1];
            z += 2;
        }
    }
    return q;
}
```

Hier ist der Code, wie er geschrieben werden sollte. Tatsächlich ist dieser Code-Ausschnitt weniger vollständig als der obige. Doch Sie können unmittelbar erkennen, was der Code bewirken soll, und Sie könnten sehr wahrscheinlich die fehlenden Funktionen anhand der erschlossenen Bedeutung schreiben. Die magischen Zahlen haben ihre »Magie« verloren, und die Struktur des Algorithmus geht aus dem Code klar hervor.

```
public int score() {
    int score = 0;
    int frame = 0;
```

```

for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
        score += 10 + nextTwoBallsForStrike(frame);
        frame += 1;
    } else if (isSpare(frame)) {
        score += 10 + nextBallForSpare(frame);
        frame += 2;
    } else {
        score += twoBallsInFrame(frame);
        frame += 2;
    }
}
return score;
}

```

Sorgfältig gewählte Namen sind deswegen so leistungsstark, weil sie die Struktur des Codes mit einer Beschreibung überlagern. Diese Überlagerung weckt im Leser bestimmte Erwartungen an die Aufgaben der anderen Funktionen in dem Modul. Sie können die Implementierung von `isStrike()` durch einen Blick auf den obigen Code erschließen. Wenn Sie die `isStrike`-Methode lesen, wird sie im Wesentlichen Ihren Erwartungen entsprechen. (Siehe das Zitat von Ward Cunningham im Abschnitt *Ward Cunningham* in Kapitel 1.)

```

private boolean isStrike(int frame) {
    return rolls[frame] == 10;
}

```

N2: Namen sollten der Abstraktionsebene entsprechen

Wählen Sie keine Namen, die etwas über die Implementierung aussagen. Wählen Sie Namen, die der Abstraktionsebene der Klasse oder Funktion entsprechen, auf der Sie arbeiten. Dies ist nicht leicht. Auch hier neigen Entwickler allzu leicht dazu, Abstraktionsebenen zu vermischen. Jedes Mal, wenn Sie Ihren Code überfliegen, werden Sie wahrscheinlich eine Variable entdecken, deren Name einer zu niedrigen Abstraktionsebene entspricht. Sie sollten diese Gelegenheit nutzen, den Namen zu ändern. Wenn Sie Ihren Code lesbar machen wollen, müssen Sie die laufende Verbesserung in Ihre Alltagspraxis einbeziehen. Betrachten Sie das folgende Modem-Interface:

```

public interface Modem {
    boolean dial(String phoneNumber);
    boolean disconnect();
    boolean send(char c);
    char recv();
    String getConnectedPhoneNumber();
}

```

Auf den ersten Blick sieht alles okay aus. Die Funktionen scheinen alle angemessen zu sein. Tatsächlich sind sie es für viele Anwendungen auch. Doch betrachten Sie jetzt eine Anwendung, bei der einige Modems nicht über eine Telefoneinwahl verbunden werden, sondern über eine Standleitung permanent mit einem Netz verbunden sind. (Einige Kabelmodems stellen auf diese Weise eine permanente Verbindung zum Internet zur Verfügung.). Vielleicht werden einige Modems verbunden, indem eine Port-Nummer über eine USB-Verbindung an einen Switch gesendet wird. Hier befindet sich das Konzept der Telefonnummer auf der falschen Abstraktionsebene. Folgender Code zeigt eine bessere Namensstrategie für dieses Szenario:

```
public interface Modem {  
    boolean connect(String connectionLocator);  
    boolean disconnect();  
    boolean send(char c);  
    char recv();  
    String getConnectedLocator();  
}
```

Jetzt enthalten die Namen keine Festlegungen auf Telefonnummern. Sie können immer noch für Telefonnummern, aber jetzt auch für andere Verbindungsstrategien verwendet werden.

N3: Möglichst die Standardnomenklatur verwenden

Namen sind leichter zu verstehen, wenn sie sich an bestehende Konventionen oder Nomenklaturen anlehnen. Verwenden Sie beispielsweise das *Decorator*-Pattern, sollten Sie das Wort *Decorator* in den Namen der dekorierenden Klassen verwenden. Beispielsweise könnte eine Klasse, die ein *Modem* mit der Fähigkeit dekoriert, am Ende einer Sitzung automatisch aufzulegen, den Namen *AutoHangupModemDecorator* bekommen.

Patterns sind nur eine Art von Standard. In Java heißen etwa Funktionen, die Objekte in String-Repräsentationen umwandeln, oft `toString`. Es ist besser, solchen Konventionen zu folgen, als eigene zu erfinden.

Teams erfinden oft ein eigenes Standardsystem für Namen in einem speziellen Projekt. Eric Evans bezeichnet dies als die *Ubiquitous Language* (»allgegenwärtige Sprache«) des Projekts ([DDD]). Ihr Code sollte vorwiegend die Ausdrücke dieser Sprache verwenden. Kurz gesagt: Je mehr Sie Namen verwenden können, die in Ihrem Projekt spezielle Bedeutungen haben, desto leichter verstehen in das Projekt eingeweihte Leser, worum es in Ihrem Code geht.

N4: Eindeutige Namen

Wählen Sie Namen, die die Aufgabe einer Funktion oder Variablen eindeutig ausdrücken. Betrachten Sie das folgende Beispiel aus FitNesse:

```
private String doRename() throws Exception
{
    if(refactorReferences)
        renameReferences();
    renamePage();

    pathToRename.removeNameFromEnd();
    pathToRename.addNameToEnd(newName);
    return PathParser.render(pathToRename);
}
```

Der Name dieser Funktion drückt nur *vage* und *allgemein* aus, was die Funktion tut. Dies wird durch die Tatsache unterstrichen, dass sie eine weitere Funktion namens `renamePage` enthält! Was sagen Ihnen die Namen über den Unterschied zwischen den beiden Funktionen? Nichts.

Ein besserer Name für diese Funktion ist `renamePageAndOptionallyAllReferences`. Er ist zwar lang, wird aber nur an einer Stelle des Moduls aufgerufen; deshalb wiegt der selbsterklärende Charakter dieses Namens seine Länge auf.

N5: Lange Namen für große Geltungsbereiche

Die Länge eines Namens sollte dem Umfang des Geltungsbereichs entsprechen. Sie können sehr kurze Variablennamen für winzige Geltungsbereiche verwenden, aber für umfangreiche Geltungsbereiche sollten Sie längere Namen benutzen.

Variablennamen wie `i` und `j` sind angemessen, wenn ein Geltungsbereich fünf Zeilen lang ist. Betrachten Sie den folgenden Code-Ausschnitt aus dem alten »Bowling Game«:

```
private void rollMany(int n, int pins)
{
    for (int i=0; i<n; i++)
        g.roll(pins);
}
```

Der Code ist klar verständlich und würde nur unlesbarer, wenn die Variable `i` durch etwas Unhandlicheres wie etwa `rollCount` ersetzt werden würde. Andererseits geht die Bedeutung von Variablen und Funktionen mit kurzen Namen über längere Distanzen verloren. Also: Je länger der Geltungsbereich des Namens, desto länger und präziser sollte der Name sein.

N6: Codierungen vermeiden

Namen sollten keine Informationen über ihren Typ oder Geltungsbereich in codierter Form enthalten. Präfixe wie etwa `m_` oder `f` sind in den heutigen Umgebungen nutzlos. Auch Projekt- und/oder Subsystem-Codes wie etwa `vis_` (für *Visual Imaging System*) sind ablenkend und redundant. Auch hier stellen die heutigen Umge-

bungen alle erforderlichen Informationen zur Verfügung, ohne dass ein solches »Name-Mangling« erforderlich wäre. Bewahren Sie Ihre Namen vor der Hungarian Pollution (»ungarischen Verschmutzung«, siehe Kapitel 2).

N7: Namen sollten Nebeneffekte beschreiben

Namen sollten alles ausdrücken, was eine Funktion, Variable oder Klasse ist oder tut. Verbergen Sie Nebeneffekte nicht mit einem Namen. Verwenden Sie kein einfaches Verb, um eine Funktion zu beschreiben, die mehr als nur diese einfache Aktion ausführt. Betrachten Sie beispielsweise den folgenden Code aus TestNG:

```
public ObjectOutputStream getOos() throws IOException {
    if (m_oos == null) {
        m_oos = new ObjectOutputStream(m_socket.getOutputStream());
    }
    return m_oos;
}
```

Diese Funktion leistet etwas mehr, als ein »oos« abzurufen. Sie erstellt das »oos«, wenn es noch nicht existiert. Deshalb wäre ein besserer Name etwa `createOrReturnOos`.

17.7 Tests

T1: Unzureichende Tests

Wie viele Tests sollte eine Test-Suite enthalten? Leider verwenden viele Programmierer das folgende Maß: »Das scheint mir genug zu sein.« Eine Test-Suite sollte alles testen, was möglicherweise schiefgehen könnte. Die Tests sind so lange unzureichend, wie es Bedingungen gibt, die nicht von den Tests geprüft werden, oder Berechnungen, die nicht validiert werden.

T2: Ein Coverage-Tool verwenden

Coverage-Tools decken Lücken in Ihrer Teststrategie auf. Sie machen es leicht, Module, Klassen und Funktionen zu finden, die unzureichend getestet werden. Die meisten IDEs liefern Ihnen ein visuelles Feedback, das die durch Tests abgedeckten Zeilen grün und die nicht abgedeckten rot anzeigt. So können Sie schnell und leicht `if`- oder `catch`-Anweisungen finden, deren Bodies nicht getestet worden sind.

T3: Triviale Tests nicht überspringen

Sie sind leicht zu schreiben und ihr dokumentarischer Wert übersteigt ihre Produktionskosten.

T4: Ein ignorierte Test zeigt eine Mehrdeutigkeit auf

Manchmal sind uns Verhaltensdetails unklar, weil die Anforderungen unklar sind. Wir können unsere Frage über die Anforderungen in Form eines auskommentierten Tests ausdrücken oder einen Test mit einer `@Ignore`-Annotation versehen. Welche Technik Sie wählen, hängt davon ab, ob die Mehrdeutigkeit kompilierbaren Code oder etwas anderes betrifft.

T5: Grenzbedingungen testen

Grenzbedingungen müssen besonders sorgfältig getestet werden. Oft wird die Mitte eines Algorithmus richtig programmiert, aber seine Grenzen werden falsch beurteilt.

T6: Bei Bugs die Nachbarschaft gründlich testen

Bugs treten oft gehäuft auf. Wenn Sie in einer Funktion einen Bug finden, ist es ratsam, diese Funktion erschöpfend zu testen. Wahrscheinlich stellen Sie fest, dass der Bug nicht allein war.

T7: Das Muster des Scheiterns zur Diagnose nutzen

Manchmal können Sie ein Problem diagnostizieren, indem Sie Patterns suchen und studieren, wie die Testfälle scheitern. Dies ist ein weiterer Grund dafür, die Testfälle so vollständig wie möglich auszugestalten. Vollständige Testfälle, die in einer vernünftigen Reihenfolge angeordnet sind, enthüllen Patterns.

Ein einfaches Beispiel: Angenommen, Sie stellten fest, dass alle Tests scheitern, deren Input länger als fünf Zeichen ist? Oder dass jeder Test scheitert, der eine negative Zahl als zweites Argument an eine Funktion übergibt? Manchmal löst schon das Muster der roten und grünen Markierungen auf dem Testbericht einen »Aha!«-Effekt aus, der zur Lösung führt. In Kapitel 16, *Refactoring von SerialDate*, finden Sie ein interessantes Beispiel dafür.

T8: Hinweise durch Coverage-Patterns

Ein Blick auf den Code, der von den bestandenen Tests ausgeführt oder nicht ausgeführt wird, kann Ihnen Hinweise geben, warum Tests scheitern.

T9: Tests sollten schnell sein

Ein langsamer Test ist ein Test, der nicht ausgeführt wird. Wenn die Zeit knapp wird, werden die langsamen Tests zuerst ausgelassen. Tun Sie deshalb *alles Erforderliche*, um Ihre Tests zu beschleunigen.

17.8 Zusammenfassung

Diese Liste der Heuristiken und Smells ist sicher nicht vollständig. Tatsächlich bin ich mir nicht sicher, ob eine solche Liste *jemals* vollständig sein kann. Aber vielleicht sollte Vollständigkeit gar nicht das Ziel sein, weil diese Liste ein Wertesystem impliziert.

Tatsächlich war dieses Wertesystem das Ziel und das Thema dieses Buches. Clean Code wird nicht dadurch geschrieben, dass man einen Satz von Regeln befolgt. Sie werden nicht zu einem Software-Könnner, indem Sie eine Liste von Heuristiken auswendig lernen. Professionalität und Könnerschaft basieren auf Werten, die einer Disziplin zugrunde liegen.

Nebenläufigkeit II

von Brett L. Schuchert

Dieser Anhang erweitert Kapitel 13, *Nebenläufigkeit*. Er besteht aus einer Reihe unabhängiger Themen, die Sie in beliebiger Reihenfolge lesen können. Damit dies möglich ist, sind einige Wiederholungen erforderlich.

A.1 Client/Server-Beispiel

Angenommen, Sie hätten eine einfache Client/Server-Anwendung. Der Server überwacht ein Socket und wartet auf eine Verbindung mit einem Client. Ein Client stellt eine Verbindung her und sendet eine Anfrage.

Der Server

Hier ist eine vereinfachte Version einer Server-Anwendung. Den kompletten Source für dieses Beispiel finden Sie am Ende dieses Anhangs im Abschnitt *Tutorial: kompletter Beispielcode, Client/Server ohne Threads*.

```
ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Diese einfache Anwendung wartet auf eine Verbindung, verarbeitet eine eingehende Nachricht und wartet dann erneut auf den Eingang der nächsten Client-Anfrage. Hier ist der Client-Code, der die Verbindung zu diesem Server herstellt:

```
private void connectSendReceive(int i) {
    try {
        Socket socket = new Socket("localhost", PORT);
        MessageUtils.sendMessage(socket, Integer.toString(i));
        MessageUtils.getMessage(socket);
        socket.close();
    } catch (Exception e) {
```



```
        e.printStackTrace();  
    }  
}
```

Wie gut ist das Leistungsverhalten dieses Client/Server-Paares? Wie können wir dieses Leistungsverhalten formal beschreiben? Hier ist ein Test, der prüft, ob das Leistungsverhalten »akzeptabel« ist:

```
@Test(timeout = 10000)  
public void shouldRunInUnder10Seconds() throws Exception {  
    Thread[] threads = createThreads();  
    startAllThreadsw(threads);  
    waitForAllThreadsToFinish(threads);  
}
```

Das Setup wurde weggelassen, um das Beispiel einfach zu halten (siehe *Listing 4: ClientTest.java* am Ende dieses Anhangs). Dieser Test sichert zu, dass die Ausführung maximal 10.000 Millisekunden dauert.

Dies ist ein klassisches Beispiel für die Validierung des Durchsatzes eines Systems. Dieses System soll eine Reihe von Client-Anfragen in zehn Sekunden beantworten. Solange der Server jede einzelne Client-Anfrage rechtzeitig beantwortet, wird der Test bestanden.

Was passiert, wenn der Test scheitert? Abgesehen von der Entwicklung einer Art von Event-Polling-Schleife kann man innerhalb eines einzelnen Threads kaum etwas tun, um den Code zu beschleunigen. Kann das Problem mit mehreren Threads gelöst werden? Vielleicht, aber wir müssen wissen, wo die Zeit verbraucht wird. Es gibt zwei Möglichkeiten:

- I/O – Verwendung eines Sockets, Verbindung mit einer Datenbank, Warten auf ein Swapping des virtuellen Speichers usw.
- Prozessor – numerische Berechnungen, Verarbeitung regulärer Ausdrücke, Garbage Collection usw.

Systeme führen üblicherweise mehrere dieser Funktionen gleichzeitig aus, aber bei einer bestimmten Aufgabe dominiert normalerweise eine Funktion. Wenn der Code prozessorabhängig ist, kann der Durchsatz mit einer leistungsstärkeren Hardware so verbessert werden, dass unser Test bestanden wird. Aber die Anzahl der CPU-Zyklen ist immer beschränkt. Deshalb kann der Code bei einem prozessorabhängigen Problem nicht durch zusätzliche Threads beschleunigt werden.

Ist der Prozess dagegen I/O-abhängig, kann die Effizienz durch Nebenläufigkeit verbessert werden. Wenn ein Teil des Systems auf I/O wartet, kann ein anderer Teil diese Wartezeit für andere Aufgaben verwenden und so die verfügbare Prozessorleistung effizienter nutzen.

Threading hinzufügen

Nehmen Sie für einen Moment an, der Test des Leistungsverhaltens scheitere. Wie können wir den Durchsatz so verbessern, dass der Test bestanden wird? Wenn die `process`-Methode des Servers I/O-abhängig ist, kann der Server durch folgende Methode veranlasst werden, Threads zu verwenden (ändern Sie einfach `processMessage`):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.getMessage(socket);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Nehmen Sie an, diese Änderung führe dazu, dass der Test bestanden wird. (Sie können dies selbst prüfen, indem Sie den Code vor und nach der Änderung ausführen. Sie finden den Code ohne und mit Threads am Ende dieses Anhangs.) Der Code ist fertig, richtig?

Server-Beobachtungen

Der aktualisierte Server beendet den Test erfolgreich in etwas über einer Sekunde. Leider ist diese Lösung ein wenig naiv und verursacht einige neue Probleme.

Wieviele Threads darf unser Server erstellen? Der Code setzt keine Grenze. Deshalb ist es möglich, dass wir an die von der Java Virtual Machine (JVM) gesetzte Grenze stoßen. Bei vielen einfachen Systemen mag dies ausreichen. Aber was passiert, wenn das System viele Benutzer im öffentlichen Netz unterstützen soll? Wenn sich zu viele Benutzer gleichzeitig verbinden, könnte das System zum Stillstand kommen.

Doch lassen wir das Verhaltensproblem im Moment beiseite. Die gezeigte Lösung hat Mängel bei der Sauberkeit und Struktur. Wie viele Verantwortlichkeiten hat der Server-Code?

- Verwaltung der Socket-Verbindung
- Client-Verarbeitung

- Threading-Regeln
- Server-Shutdown-Regeln

Leider befinden sich alle diese Verantwortlichkeiten in der `process`-Funktion. Außerdem umfasst der Code viele verschiedene Abstraktionsebenen. So klein die `process`-Funktion auch sein mag, sie muss neu partitioniert werden.

Es gibt mehrere Gründe, warum sich der Server ändern kann. Deshalb verstößt die Funktion gegen das Single-Responsibility-Prinzip. Sollen nebenläufige Systeme sauber bleiben, muss das Thread-Management auf wenige, wohlkontrollierte Stellen beschränkt werden. Darüber hinaus sollte der Code für die Threads-Verwaltung allein diese Aufgabe ausführen. Warum? Die Analyse von Nebenläufigkeitsproblemen ist schon allein schwierig genug. Sie sollte nicht durch andere Probleme, die nichts mit der Nebenläufigkeit zu tun haben, zusätzlich erschwert werden.

Wird für jede der oben genannten Verantwortlichkeiten, einschließlich des Thread-Managements, eine separate Klasse erstellt, hat eine Änderung der Thread-Management-Strategie weniger Einfluss auf den Code insgesamt und kontaminiert nicht die anderen Verantwortlichkeiten. Außerdem können Sie so die anderen Verantwortlichkeiten unabhängig von dem Threading testen.

Hier ist eine aktualisierte Version, die genau dies leistet:

```
public void run() {
    while (keepProcessing) {
        try {
            ClientConnection clientConnection = connectionManager.awaitClient();
            ClientRequestProcessor requestProcessor
                = new ClientRequestProcessor(clientConnection);
            clientScheduler.schedule(requestProcessor);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    connectionManager.shutdown();
}
```

In diesem Code wird alles, was mit Threads zu tun hat, an einer Stelle zusammengefasst: dem `clientScheduler`. Gibt es Nebenläufigkeitsprobleme, müssen Sie nur an einer Stelle nachschauen:

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

Die gegenwärtige Policy ist einfach zu implementieren:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        ...
    }
}
```

```

    }
};

Thread thread = new Thread(runnable);
thread.start();
}
}

```

Nachdem wir das Thread-Management an einer einzigen Stelle zusammengefasst haben, können wir die Threads viel einfacher kontrollieren. Wollen wir beispielsweise auf das Java 5 Executor Framework umstellen, müssen wir nur eine neue Klasse schreiben und einfügen (Listing A.1).

Listing A.1: ExecutorClientScheduler.java

```

import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;

    public ExecutorClientScheduler(int availableThreads) {
        executor = Executors.newFixedThreadPool(availableThreads);
    }

    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        executor.execute(runnable);
    }
}

```

Zusammenfassung

Die Einführung der Nebenläufigkeit in diesem speziellen Beispiel zeigt, wie Sie den Durchsatz eines Systems verbessern und diesen Durchsatz mit einem Test-Framework validieren können. Den gesamten Nebenläufigkeitscode in einer kleinen Menge von Klassen zusammenzufassen, ist ein Beispiel für die Anwendung des Single-Responsibility-Prinzips. Bei der nebenläufigen Programmierung ist es wegen der Komplexität der Aufgabe besonders wichtig.

A.2 Mögliche Ausführungspfade

Betrachten Sie noch einmal die Methode `incrementValue`, eine einzeilige Java-Methode ohne Schleifen oder Verzweigungen:

```

public class IdGenerator {
    int lastIdUsed;
}

```

```
public int incrementValue() {  
    return ++lastIdUsed;  
}  
}
```

Ignorieren Sie den Integer-Overflow und nehmen Sie an, dass nur ein Thread auf eine einzige Instanz von *IdGenerator* zugreifen kann. In diesem Fall gibt es einen einzigen Ausführungspfad und ein einziges garantiertes Ergebnis:

- Der zurückgegebene Wert entspricht dem Wert von `lastIdUsed`: Beide sind um eins größer als unmittelbar vor dem Aufruf der Methode.

Was passiert, wenn wir zwei Threads verwenden und die Methode nicht ändern? Welche Ergebnisse sind möglich, wenn jeder Thread `incrementValue` einmal aufruft? Wie viele mögliche Ausführungspfade gibt es? Zunächst die Ergebnisse (`lastIdUsed` startet mit einem angenommenen Wert von 93):

- Thread 1 erhält den Wert 94, Thread 2 erhält den Wert 95, und `lastIdUsed` ist jetzt 95.
- Thread 1 erhält den Wert 95, Thread 2 erhält den Wert 94, und `lastIdUsed` ist jetzt 95.
- Thread 1 erhält den Wert 94, Thread 2 erhält den Wert 94, und `lastIdUsed` ist jetzt 94.

Das letzte Ergebnis ist zwar überraschend, aber möglich. Um zu verstehen, wie diese verschiedenen Ergebnisse zustande kommen können, müssen Sie die Anzahl der möglichen Ausführungspfade kennen und verstehen, wie die Java Virtual Machine sie ausführt.

Anzahl der Pfade

Um die Anzahl der möglichen Ausführungspfade zu berechnen, beginnen wir mit dem generierten Byte-Code. Aus der einen Zeile Java-Code (`return ++lastIdUsed;`) werden acht Byte-Code-Anweisungen generiert. Zwei Threads können sich bei der Ausführung dieser acht Anweisungen ähnlich verzahnen, wie ein Kartengeber einen Stapel Spielkarten mischt. (Dies ist eine Vereinfachung. Doch für unseren Zweck reicht dieses vereinfachende Modell aus.) Selbst bei nur acht Karten auf jeder Hand gibt es eine bemerkenswert hohe Anzahl möglicher Mischungen.

Bei diesem einfachen Fall mit N Anweisungen in einer Folge, ohne Schleifen oder Bedingungen, und T Threads, beträgt die Gesamtzahl der möglichen Ausführungspfade

$$\frac{(NT)!}{N!^T}$$

Die möglichen Reihenfolgen berechnen

Der folgende Text stammt aus einer E-Mail von Uncle Bob an Brett:

Bei N Schritten und T Threads gibt es insgesamt $T * N$ Schritte. Vor jedem Schritt gibt es einen Kontextschalter, der einen der T Threads auswählt. Jeder Pfad kann deshalb durch einen String von Ziffern repräsentiert werden, der den jeweiligen Kontextschalter bezeichnet. Bei zwei Schritten A und B und zwei Threads 1 und 2 gibt es sechs mögliche Pfade: 1122, 1212, 1221, 2112, 2121 und 2211. Oder als Schritte ausgedrückt: A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 und A2B2A1B1. Bei drei Threads lautet die Folge: 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, ...

Diese Strings haben eine bestimmte Eigenschaft: Es muss immer N Instanzen von jedem T geben. Deshalb ist der String 111111 ungültig, weil er sechs Instanzen von 1 und null Instanzen von 2 und 3 enthält.

Deshalb müssen wir mit den Permutationen von N Einsen, N Zweien, ... und N Ts arbeiten. Dabei handelt es sich einfach um die Permutationen von $N * T$, wobei $N * T$ auf einmal genommen werden. Dies ist $(N * T)!$, doch ohne Duplikate. Der Trick besteht also darin, die Duplikate zu zählen und von $(N * T)!$ zu subtrahieren.

Wie viele Duplikate gibt es bei zwei Schritten und zwei Threads? Jeder Vier-Ziffern-String enthält zwei Einsen und zwei Zweien. Jedes dieser Paare könnte ausgetauscht werden, ohne die Bedeutung des Strings zu ändern. Sie könnten die Einsen oder die Zweien, beides oder nichts austauschen. Deshalb gibt es für jeden String vier Isomorphe, was bedeutet, dass es drei Duplikate gibt. Deshalb sind drei der vier Optionen Duplikate; alternativ ist eine der vier Permutationen KEIN Duplikat. $4! * 0,25 = 6$. Diese Überlegung scheint also richtig zu sein.

Wie viele Duplikate gibt es? Im Fall von $N = 2$ und $T = 2$ könnte ich die Einsen, die Zweien oder beide vertauschen. Im Fall von $N = 2$ und $T = 3$ könnte ich die Einsen, die Zweien, die Dreien, Einsen und Zweien, Einsen und Dreien oder Zweien und Dreien vertauschen. Ein Austausch entspricht einfach den Permutationen von N . Angenommen, es gäbe P Permutationen von N . Die Anzahl der verschiedenen Möglichkeiten, diese Permutationen anzuordnen, beträgt $P^{*}T$.

Deshalb beträgt die Anzahl der möglichen Isomorphen $N!^{*}T$. Und deshalb ist die Anzahl der Pfade $(T*N)!/(N!^{*}T)$. Auch hier erhalten wir bei $T = 2$ und $N = 2$ den Wert 6 ($24/4$).

Für $N = 2$ und $T = 3$ erhalten wir $720/8 = 90$.

Für $N = 3$ und $T = 3$ erhalten wir $9!/6^3 = 1.680$.

In unserem einfachen Fall mit einer Zeile Java-Code, der acht Zeilen Byte-Code und zwei Threads entspricht, beträgt die Gesamtzahl der möglichen Ausführungspfade 12.870. Hätte `lastIdUsed` den Typ `long`, würde jeder Lese/Schreib-Vorgang nicht eine, sondern zwei Operationen erfordern; die Anzahl der möglichen Reihenfolgen betrüge dann $2.704.156$.

Was passiert, wenn wir diese Methode an einer Stelle ändern?

```
public synchronized void incrementValue() {  
    ++lastIdUsed;  
}
```

Die Anzahl der möglichen Ausführungspfade ist für zwei Threads zwei und $N!$ im allgemeinen Fall.

Tiefer graben

Was ist mit dem überraschenden Ergebnis, dass zwei Threads beide die Methode einmal aufrufen (bevor wir `synchronized` hinzugefügt haben) und dasselbe numerische Ergebnis erhalten? Wie ist das möglich? Eins nach dem anderen.

Was ist eine atomare Operation? Wir können eine atomare Operation als eine Operation definieren, die nicht unterbrochen werden kann. Beispielsweise ist in dem folgenden Code Zeile 5, in der `o` der Variablen `lastId` zugewiesen wird, atomar, weil laut Java-Memory-Modell eine Zuweisung zu einem 32-Bit-Wert nicht unterbrochen werden kann.

```
01: public class Example {  
02:     int lastId;  
03:  
04:     public void resetId() {  
05:         value = 0;  
06:     }  
07:  
08:     public int getNextId() {  
09:         ++value;  
10:     }  
11:}
```

Was passiert, wenn wir den Typ von `lastId` von `int` in `long` ändern? Ist Zeile 5 immer noch atomar? Nicht laut JVM-Spezifikation. Sie könnte bei einem speziellen Prozessor atomar sein, doch laut JVM-Spezifikation erfordert die Zuweisung zu einem 64-Bit-Wert zwei 32-Bit-Zuweisungen. Dies bedeutet, dass zwischen der ersten und der zweiten 32-Bit-Zuweisung ein anderer Thread dazwischengehen und einen der Werte ändern könnte.

Was ist mit dem Prä-Inkrement-Operator, `++`, in Zeile 9? Der Prä-Inkrement-Operator kann unterbrochen werden; deswegen ist er nicht atomar. Wir wollen dies anhand des Byte-Codes dieser beiden Methoden im Detail erläutern.

Bevor wir fortfahren, müssen Sie drei wichtige Definitionen kennen:

- **Frame** – Jeder Methodenaufruf erfordert ein Frame. Das Frame enthält die Rückkehradresse, die Parameter, die an die Methode übergeben wurden, und die lokalen Variablen, die in der Methode definiert werden. Es handelt sich um eine Standardtechnik, um einen Aufruf-Stack (call stack) zu definieren. Sie wird in modernen Sprachen verwendet, um grundlegende Funktions- oder Methodenaufrufe, inklusive rekursiver Aufrufe, zu ermöglichen.
- **Lokale Variable** – Alle Variablen, deren Geltungsbereich auf die Methode beschränkt ist. Alle nicht-statischen Methoden verfügen über wenigstens eine Variable, `this`, die das gegenwärtige Objekt repräsentiert, das Objekt, das die (in dem gegenwärtigen Thread) letzte Nachricht empfangen hat, die den Aufruf der Methode ausgelöst hat.
- **Operanden-Stack** – Viele Anweisungen in der Java Virtual Machine haben Parameter. Der Operanden-Stack ist der Ort, an dem diese Parameter abgelegt werden. Der Stack hat eine Standard-LIFO-Struktur (last-in, first-out).

Hier ist der für `resetId()` generierte Byte-Code:

Mnemonic	Beschreibung	Operanden-Stack nachher
<code>ALOAD 0</code>	Lade die 0-te Variable auf den Operanden-Stack. Was ist die 0-te Variable? Es handelt sich um <code>this</code> , das gegenwärtige Objekt. Als die Methode aufgerufen wurde, wurde der Empfänger der Nachricht, eine Instanz von <code>Example</code> , in das lokale Variablen-Array des Frames eingefügt, das für Methodenaufrufe erstellt wird. Dies ist immer die erste Variable, die in jede Instanzmethode eingefügt wird.	<code>this</code>
<code>ICONST_0</code>	Lege den konstanten Wert 0 auf dem Operanden-Stack ab.	<code>this, 0</code>
<code>PUTFIELD lastId</code>	Speichere den oberen Wert (der 0 ist) auf dem Stack in den Feldwert des Objekts, das von dem zweiten Wert auf dem Stack, <code>this</code> , referenziert wird.	<code><leer></code>

Diese drei Anweisungen sind garantiert atomar, weil die Informationen für die `PUTFIELD`-Anweisung (der konstante Wert 0 am Anfang des Stacks und die Referenz auf `this` an der zweiten Stelle sowie der Feldwert) nicht von einem anderen Thread geändert werden können, obwohl der ausführende Thread nach jeder Anweisung unterbrochen werden könnte. Wenn also die Zuweisung stattfindet, ist garantiert, dass der Wert 0 in dem Feldwert gespeichert wird. Die Operation ist atomar. Die Operanden bearbeiten nur lokale Daten der Methode, weshalb es keine Konflikte zwischen mehreren Threads gibt.

Wenn diese drei Anweisungen von zehn Threads ausgeführt werden, gibt es $4,38679733629e+24$ mögliche Reihenfolgen. Es gibt jedoch nur ein mögliches Ergebnis. Deshalb sind die verschiedenen Reihenfolgen irrelevant. Zufällig ist in diesem Fall auch für longs dasselbe Ergebnis garantiert. Warum? Alle zehn Threads weisen einen konstanten Wert zu. Selbst wenn sie sich überlappen, bleibt das Endergebnis gleich.

Bei der ++-Operation in der getNextId-Methode gibt es Probleme. Angenommen, lastId enthielte am Anfang der Methode den Wert 42. Dann hat die Methode den folgenden Byte-Code:

Mnemonic	Beschreibung	Operanden-Stack nachher
ALOAD 0	Lade this auf den Operanden-Stack	this
DUP	Kopiere den Anfang des Stacks. Danach existieren zwei Kopien von this auf dem Operanden-Stack.	this, this
GETFIELD lastId	Rufe den Wert des Feldes lastId aus dem Objekt ab, auf das der Anfang des Stacks (this) verweist, und speichere diesen Wert wieder auf dem Stack.	this, 42
ICONST_1	Lege die ganzzahlige Konstante 1 auf dem Stack ab.	this, 42, 1
IADD	Führe eine ganzzahlige Addition der oberen beiden Werte des Operanden-Stacks durch und lege das Ergebnis wieder auf dem Operanden-Stack ab.	this, 43
DUP_X1	Dupliziere den Wert 43 und füge ihn vor this ein.	43, this, 43
PUTFIELD value	Speichere den oberen Wert des Operanden-Stacks, 43, in den Feldwert des gegenwärtigen Objekts, das von dem zweiten Wert auf dem Operanden-Stack, this, repräsentiert wird.	43
IRETURN	Gib den oberen (und einzigen) Wert auf dem Stack zurück.	<leer>

Stellen Sie sich den Fall vor, in dem der erste Thread die ersten drei Anweisungen bis einschließlich GETFIELD abschließt und dann unterbrochen wird. Ein zweiter Thread übernimmt, führt die gesamte Methode aus, erhöht lastId um eins und erhält 43 zurück. Dann macht der erste Thread an der Stelle weiter, an der er unterbrochen wurde; 42 befindet sich immer noch auf dem Operanden-Stack, weil dies der Wert von lastId war, als der Thread GETFIELD ausgeführt hat. Er fügt eins hinzu, was wieder 43 ergibt, und speichert das Ergebnis. Der Wert 43 wird auch an den ersten Thread zurückgegeben. Folglich geht eine Inkrementoperation verloren, weil der erste Thread dem zweiten Thread ins Gehege kam, nachdem der zweite Thread den ersten Thread unterbrochen hatte.

Wenn die `getNextId()`-Methode als `synchronized` deklariert wird, wird dieses Problem behoben.

Zusammenfassung

Eine genaue Kenntnis des Byte-Codes ist nicht erforderlich, um zu verstehen, wie sich Threads ins Gehege kommen können. Wenn Sie dieses eine Beispiel verstehen, sollte es Ihnen zeigen, wie Konflikte zwischen mehreren Threads entstehen können. Dieses Wissen reicht aus.

Davon abgesehen zeigt dieses triviale Beispiel, dass Sie das Speichermodell so gut verstehen müssen, dass Sie wissen, was sicher und nicht sicher ist. Es ist ein übliches Missverständnis, anzunehmen, dass der `++`-(Prä- oder Postinkrement-)Operator atomar ist. Er ist es nicht! Das bedeutet, Sie müssen wissen,

- wo Objekte/Werte gemeinsam genutzt werden,
- welcher Code Probleme bei nebenläufigen Lese/Update-Operationen auslösen kann,
- wie Sie solche Probleme der nebenläufigen Programmierung vermeiden können.

A.3 Lernen Sie Ihre Library kennen

Executor Framework

Das Listing A.1 (weiter vorne) mit dem `ExecutorClientScheduler.java` zeigt, dass das Executor Framework, das mit Java 5 eingeführt wurde, eine ausgefeilte Ausführung mit Thread-Pools ermöglicht. Dies ist eine Klasse aus dem Package `java.util.concurrent`.

Wenn Sie Threads erstellen und keinen oder einen handgeschriebenen Thread-Pool verwenden, sollten Sie den Einsatz des Executor Frameworks erwägen. Ihr Code wird dadurch sauberer, übersichtlicher und kleiner.

Das Executor Framework fasst Threads zu einem Pool zusammen, passt die Größe automatisch an und erstellt Threads nach Bedarf. Außerdem unterstützt es *Futures*, ein gebräuchliches Konstrukt der nebenläufigen Programmierung. Das Executor Framework arbeitet mit Klassen, die `Runnable` implementieren und auch mit Klassen, die das `Callable`-Interface implementieren. Ein `Callable` sieht wie ein `Runnable` aus, kann aber ein Ergebnis zurückgeben, was bei Multithreaded-Lösungen häufig benötigt wird.

Ein *Future* ist praktisch, wenn Code mehrere, unabhängige Operationen ausführen und darauf warten muss, dass alle beendet werden:

```
public String processRequest(String message) throws Exception {
    Callable<String> makeExternalCall = new Callable<String>() {
        public String call() throws Exception {
            String result = "";
            // make external request
            return result;
        }
    };

    Future<String> result = executorService.submit(makeExternalCall);
    String partialResult = doSomeLocalProcessing();
    return result.get() + partialResult;
}
```

In diesem Beispiel beginnt die Methode mit der Ausführung des `makeExternalCall`-Objekts. Die Methode setzt dann die Verarbeitung fort. In der letzten Zeile wird `result.get()` aufgerufen, die blockiert, bis das `Future` fertig ist.

Nicht blockierende Lösungen

Die Java-5-VM nutzt das Design moderner Prozessoren, das zuverlässige, nicht blockierende Updates unterstützt. Betrachten Sie beispielsweise eine Klasse, die Synchronisation (und deshalb Blocking) verwendet, um ein thread-sicheres Update eines Wertes zu garantieren:

```
public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

Java 5 verfügt über eine Reihe neuer Klassen für derartige Situationen: `AtomicBoolean`, `AtomicInteger`, `AtomicReference` und zahlreiche andere. Wir können den obigen Code so umschreiben, dass er einen nicht blockierenden Ansatz verwendet:

```
public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);

    public void incrementValue() {
        value.incrementAndGet();
    }
    public int getValue() {
        return value.get();
    }
}
```

Obwohl hierbei ein Objekt anstelle eines Primitives verwendet wird und Nachrichten wie `incrementAndGet()` anstelle des Operators `++` verwendet werden, ist das Leistungsverhalten dieser Klasse fast immer besser als das der vorherigen Version. In einigen Fällen ist der Code nur etwas schneller, aber es gibt praktisch keine Fälle, in denen er langsamer ist.

Wie ist dies möglich? Moderne Prozessoren verfügen über eine Operation, die üblicherweise als *Compare und Swap (CAS)* bezeichnet wird. Diese Operation entspricht dem optimistischen Locking bei Datenbanken, während die synchronisierte Version dem pessimistischen Locking entspricht.

Das Schlüsselwort `synchronized` setzt immer ein Lock, selbst wenn kein zweiter Thread versucht, denselben Wert zu aktualisieren. Auch wenn sich das Leistungsverhalten intrinsischer Locks von Version zu Version verbessert hat, sind sie immer noch teuer.

Die nicht blockierende Version beginnt mit der Annahme, dass mehrere Threads denselben Wert im Allgemeinen nicht so oft modifizieren, dass daraus ein Problem entsteht. Stattdessen entdeckt sie effizient, ob eine solche Situation eingetreten ist, und wiederholt ihre Versuche so lange, bis das Update erfolgreich ist. Diese Entdeckung ist selbst in Situationen mit vielen konkurrierenden Threads fast immer billiger als das Setzen eines Locks.

Wie geht die Virtual Machine dabei vor? Die CAS-Operation ist atomar. Logisch sieht die CAS-Operation etwa wie folgt aus:

```
int variableBeingSet;

void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
        variableBeingSet = newValue;
        return currentValue;
    }
    return variableBeingSet;
}
```

Wenn eine Methode versucht, eine gemeinsam genutzte Variable zu aktualisieren, prüft die CAS-Operation, ob die zu setzende Variable immer noch den letzten bekannten Wert hat. Ist dies der Fall, wird die Variable geändert. Ist dies nicht der Fall, wird die Variable nicht gesetzt, weil ein anderer Thread dazwischengegangen ist. Die Methode, die (mit der CAS-Operation) den Versuch unternimmt, sieht, dass die Änderung nicht erfolgt ist, und unternimmt einen weiteren Versuch.

Nicht thread-sichere Klassen

Einige Klassen sind von Natur aus nicht thread-sicher. Hier sind einige Beispiele:

■ SimpleDateFormat

- Datenbankverbindungen
- Container in `java.util`
- Servlets

Beachten Sie, dass einige Collection-Klassen über einzelne Methoden verfügen, die thread-sicher sind. Doch jede Operation, bei der mehr als eine Methode aufgerufen wird, ist es nicht. Wenn Sie etwa einen Wert in einer `HashTable` nicht ersetzen wollen, weil er bereits darin enthalten ist, könnten Sie den folgenden Code schreiben:

```
if(!hashTable.containsKey(someKey)) {  
    hashTable.put(someKey, new SomeValue());  
}
```

Jede einzelne Methode ist thread-sicher. Doch zwischen den Aufrufen von `containsKey` und `put` könnte ein anderer Thread einen Wert einfügen. Es gibt mehrere Optionen, um dieses Problem zu beheben.

- Sperren Sie zuerst die `HashTable` und sorgen Sie dafür, dass alle anderen Benutzer der `HashTable` dies ebenfalls tun – clientbasiertes Locking:

```
synchronized(map) {  
    if(!map.containsKey(key))  
        map.put(key, value);  
}
```

- Hüllen Sie die `HashTable` in ein eigenes Objekt ein und verwenden Sie ein anderes API – serverbasiertes Locking mit einem *Adapter*:

```
public class WrappedHashtable<K, V> {  
    private Map<K, V> map = new Hashtable<K, V>();  
  
    public synchronized void putIfAbsent(K key, V value) {  
        if (!map.containsKey(key))  
            map.put(key, value);  
    }  
}
```

- Verwenden Sie thread-sichere Collections:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer, String>();  
map.putIfAbsent(key, value);
```

Die Collections in `java.util.concurrent` haben Operationen wie `putIfAbsent()`, um sich an solche Operationen anzupassen.

A.4 Abhängigkeiten zwischen Methoden können nebenläufigen Code beschädigen

Hier ist ein triviales Beispiel für eine Möglichkeit, Abhängigkeiten zwischen Methoden einzuführen:

```
public class IntegerIterator implements Iterator<Integer> {
    private Integer nextValue = 0;

    public synchronized boolean hasNext() {
        return nextValue < 100000;
    }
    public synchronized Integer next() {
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }
    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

Hier ist Code, der diese `IntegerIterator` benutzt:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    // do something with nextValue
}
```

Wenn ein Thread diesen Code ausführt, gibt es kein Problem. Aber was passiert, wenn zwei Threads versuchen, eine einzige Instanz von `IntegerIterator` gemeinsam zu nutzen, wobei jeder Thread die Werte verarbeitet, die er bekommt, aber jedes Element der Liste nur einmal verarbeitet wird? Meistens passiert nichts Schlimmes. Ohne sich ins Gehege zu kommen, nutzen die Threads die Liste gemeinsam und verarbeiten die Elemente, die ihnen von dem Iterator geliefert werden, und halten an, wenn der Iterator die Liste abgearbeitet hat. Es gibt jedoch eine geringe Wahrscheinlichkeit, dass die beiden Threads am Ende der Iteration in Konflikt geraten und ein Thread über das Ende des Iterators hinausgeht und so eine Ausnahme auslöst.

Hier ist das Problem: Thread 1 fragt `hasNext()` und erhält `true` zurück. Thread 1 wird durch Präemption gestoppt; und dann stellt Thread 2 dieselbe Frage und erhält ebenfalls die Antwort `true`. Dann ruft Thread 2 `next()` auf und erhält erwartungsgemäß einen Wert zurück. Als Nebeneffekt wird aber der Rückgabewert von `hasNext()` auf `false` gesetzt. Thread 1 nimmt seine Arbeit im Glauben, `hasNext()` wäre immer noch `true`, wieder auf und ruft dann `next()` auf. Auch wenn die einzelnen Methoden synchronisiert sind, verwendet der Client *zwei* Methoden.

Dies ist ein echtes Problem und ein Beispiel für die Art von Problemen, die in nebenläufigem Code auftreten können. In dieser speziellen Situation ist dieses Problem besonders subtil, weil es nur bei der letzten Iteration des Iterators zu einem Fehler führt. Wenn die Threads zufällig an der passenden Stelle abbrechen, kann einer der Threads über das Ende des Iterators hinausgehen. Diese Art von Bug zeigt sich, lange nachdem ein System in Produktion gegangen ist, und ist nur schwer zu finden.

Sie haben drei Optionen:

- das Scheitern zu tolerieren,
- das Problem zu lösen, indem Sie den Client ändern: clientbasiertes Locking,
- das Problem zu lösen, indem Sie den Server ändern, wodurch zusätzlich der Client geändert wird: serverbasiertes Locking.

Das Scheitern tolerieren

Manchmal kann man es so einrichten, dass das Scheitern keinen Schaden verursacht. Beispielsweise könnte der obige Client die Ausnahme abfangen und aufräumen. Offen gesagt, ist diese Lösung etwas salopp. Es ähnelt ein wenig dem Rebooten eines Systems um Mitternacht, um Speicherlecks zu beheben.

Clientbasiertes Locking

Damit `IntegerIterator` korrekt mit mehreren Threads arbeitet, müssen Sie diesen Client (und jeden anderen Client) wie folgt ändern:

```
IntegerIterator iterator = new IntegerIterator();

while (true) {
    int nextValue;
    synchronized (iterator) {
        if (!iterator.hasNext())
            break;
        nextValue = iterator.next();
    }
    doSomethingWith(nextValue);
}
```

Jeder Client setzt mittels des Schlüsselworts `synchronized` eine Sperre. Diese Duplizierung verstößt gegen das DRY-Prinzip, kann aber erforderlich sein, wenn der Code Drittanbieter-Tools verwendet, die nicht thread-sicher sind.

Diese Strategie ist riskant, weil alle Programmierer, die den Server verwenden, daran denken müssen, ihn zu sperren, bevor sie ihn benutzen, und ihn zu entsperren, wenn sie fertig sind. Vor vielen (vielen!) Jahren arbeitete ich an einem System, das clientbasiertes Locking einer gemeinsam genutzten Ressource verwendete. Die Ressource wurde an Hunderten verschiedenen Stellen im Code verwendet. Ein armer Programmierer vergaß, die Ressource an einer dieser Stellen zu sperren.

Das System war ein Multi-Terminal-Time-Sharing-System, auf dem die Buchhaltungssoftware für eine ortsansässige Transportgewerkschaft lief. Der Computer befand sich in einem speziellen Gebäude mit klimatisierten Räumen 80 Kilometer nördlich der Hauptniederlassung der Gewerkschaft. Im Hauptquartier waren Dutzende von Datentypistinnen damit beschäftigt, an den Terminals Mitgliedsbeiträge einzugeben. Die Terminals waren mit dem Computer über Standleitungen verbunden und verwendeten 600bps-Half-Duplex-Modems. (Dies ist also schon sehr, *sehr* lange her.)

Etwa einmal pro Tag blieb eines der Terminals »stehen«. Der Grund war nicht ersichtlich. Es war kein spezielles Terminal betroffen; auch die Uhrzeit variierte. Es war, als würde jemand würfeln, um die Zeit und das Terminal auszuwählen. Manchmal froren auch mehrere Terminals ein. Und an manchen Tagen passierte gar nichts.

Anfangs bestand die Lösung in einem Reboot. Aber die Reboots waren schwer zu koordinieren. Wir mussten das Hauptquartier anrufen; und jeder musste seine laufende Arbeit beenden, und zwar auf allen Terminals. Dann konnten wir das System herunterfahren und neu starten. Wenn jemand mit einer wichtigen Arbeit beschäftigt war, die eine oder zwei Stunden dauerte, musste das eingefrorene Terminal einfach eingefroren bleiben.

Nachdem wir den Fehler einige Wochen gesucht hatten, stellten wir fest, dass die Ursache ein Ringpufferzähler war, der nicht mehr mit seinem Zeiger synchron lief. Dieser Puffer kontrollierte den Output zu dem Terminal. Der Zeigerwert zeigte an, dass der Puffer leer war, aber der Zähler sagte, dass der Puffer voll war. Weil er leer war, gab es nichts anzuzeigen; aber weil er auch voll war, konnte nichts mehr in den Puffer eingefügt werden, was auf dem Bildschirm angezeigt werden sollte.

Jetzt wussten wir, warum die Terminals einfroren, aber wir wussten nicht, warum der Ringpuffer nicht mehr synchron lief. Deshalb fügten wir einen Hack hinzu, um das Problem zu umgehen. Es war möglich, die Front-Panel-Schalter des Computers auszulesen. (Dies ist sehr, sehr, *sehr* lange her.) Wir schrieben eine kleine Fangfunktion, die entdeckte, wann einer dieser Schalter betätigt wurde, und dann nach einem Ringpuffer suchte, der sowohl leer als auch voll war. Wurde ein Puffer gefunden, wurde er auf leer zurückgesetzt. Voilà! Alle eingefrorenen Terminals nahmen ihre Anzeige wieder auf.

Jetzt mussten wir das System nicht mehr neu starten, wenn ein Terminal einfro. Der zuständige Mitarbeiter rief uns einfach an, wenn ein Terminal stehen blieb. Dann gingen wir einfach in den Computer-Raum und betätigten einen Schalter.

Natürlich wurde manchmal auch am Wochenende gearbeitet, wenn wir nicht da waren. Deshalb fügten wir eine Funktion zu dem Scheduler hinzu, die alle Ringpuffer einmal pro Minute prüfte und alle zurücksetzte, die sowohl leer als auch voll waren. Dadurch wurden die Displays schneller freigegeben, als der zuständige Mitarbeiter telefonieren konnte.

Es dauerte mehrere Wochen, in denen wir Seite für Seite mit monolithischem Assembler-Code studierten, bevor wir den Auslöser des Problems fanden. Laut unseren Berechnungen entsprach die Häufigkeit der Sperrungen einer einzelnen ungeschützten Nutzung des Ringpuffers. Wir mussten also nur diese eine fehlerhafte Nutzung finden. Leider gab es damals noch keine Suchwerkzeuge, Querverweise oder andere automatisierte Hilfen. Wir mussten einfach die Listings studieren.

Damals, in diesem kalten Winter in Chicago 1971, lernte ich eine wichtige Lektion. Clientbasiertes Locking hat es wirklich in sich.

Serverbasiertes Locking

Die Duplizierung kann durch folgende Änderungen von `IntegerIterator` behoben werden:

```
public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;
    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
}
```

Der Client-Code muss ebenfalls geändert werden:

```
while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // do something with nextValue
}
```

In diesem Fall ändern wir das API unserer Klasse, um sie Multithread-fähig zu machen. (Tatsächlich ist das `Iterator`-Interface nicht thread-sicher. Es wurde nie im Hinblick auf den Einsatz mit multiplen Threads konzipiert; deshalb ist dies nicht überraschend.) Der Client muss nicht `hasNext()` prüfen, sondern eine Null-Prüfung durchführen.

Im Allgemeinen sollten Sie serverbasiertes Locking aus den folgenden Gründen vorziehen:

- Es erfordert weniger Wiederholung von Code – clientbasiertes Locking zwingt jeden Client, den Server korrekt zu sperren. Wenn Sie den Locking-Code auf den Server verlagern, können Clients das Objekt verwenden und müssen sich nicht darum kümmern, zusätzlichen Locking-Code zu schreiben.

- Es ermöglicht ein besseres Leistungsverhalten – Sie können einen thread-sicheren Server gegen einen nicht thread-sicheren Server austauschen, falls die Umgebung mit einem einzigen Thread auskommt; dadurch können Sie den gesamten Verwaltungsaufwand einsparen.
- Es reduziert die Fehlermöglichkeiten – Ein einziger Programmierer, der vergisst, die Sperren korrekt zu setzen, kann das System nun nicht mehr blockieren.
- Es erzwingt eine einheitliche Regelung – Die Sperrvorschriften befinden sich an einer Stelle, auf dem Server, und nicht verteilt an vielen Stellen, bei mehreren Clients.
- Es reduziert die Reichweite der gemeinsam genutzten Variablen – Der Client kennt sie nicht oder weiß nicht, wie sie gesperrt werden. Der gesamte Mechanismus ist in dem Server verborgen. Im Fehlerfall ist die Anzahl der zu untersuchenden Stellen geringer.

Was machen Sie, wenn Ihnen der Server-Code nicht gehört?

- Verwenden Sie einen *Adapter*, um das API zu ändern und Locking hinzuzufügen

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();

    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- ODER noch besser: Verwenden Sie thread-sichere Collections mit extended Interfaces.

A.5 Den Durchsatz verbessern

Angenommen, wir wollten aus dem Internet die Inhalte einer Reihe von Seiten anhand einer Liste von URLs abrufen. Während eine Seite gelesen wird, parsen wir sie, um einige statistische Daten zu sammeln. Nachdem alle Seiten gelesen worden sind, geben wir eine Zusammenfassung aus.

Die folgende Klasse gibt den Inhalt einer Seite zurück, wenn der URL bekannt ist:

```
public class PageReader {
    //...
    public String getPageFor(String url) {
        HttpMethod method = new GetMethod(url);
```

```
try {
    httpClient.executeMethod(method);
    String response = method.getResponseBodyAsString();
    return response;
} catch (Exception e) {
    handle(e);
} finally {
    method.releaseConnection();
}
}
```

Die nächste Klasse ist der Iterator, der die Inhalte der Seiten anhand eines Iterators von URLs liefert:

```
public class PageIterator {
    private PageReader reader;
    private URLIterator urls;

    public PageIterator(PageReader reader, URLIterator urls) {
        this.urls = urls;
        this.reader = reader;
    }

    public synchronized String getNextPageOrNull() {
        if (urls.hasNext())
            getPageFor(urls.next());
        else
            return null;
    }

    public String getPageFor(String url) {
        return reader.getPageFor(url);
    }
}
```

Eine Instanz von `PageIterator` kann von vielen verschiedenen Threads gemeinsam genutzt werden, wobei jeder mit seiner eigenen Instanz von `PageReader` die Seiten liest und parst, die er von dem Iterator erhält.

Beachten Sie, dass wir den synchronisierten Block sehr klein gehalten haben. Er enthält nur den kritischen Abschnitt, der tief in dem `PageIterator` verborgen ist. Es ist immer besser, so wenig wie möglich, statt so viel wie möglich Code zu synchronisieren.

Single-Thread-Berechnung des Durchsatzes

Wir wollen jetzt einige einfache Berechnungen durchführen. Gehen Sie übungshalber von den folgenden Annahmen aus:

- I/O-Zeit, um eine Seite abzurufen (im Durchschnitt): 1 Sekunde
- Verarbeitungszeit, um eine Seite zu parsen (im Durchschnitt): 0,5 Sekunden
- I/O erfordert 0 Prozent der CPU, während die Verarbeitung 100 Prozent benötigt.

Bei N Seiten, die von einem einzigen Thread verarbeitet werden, beträgt die Gesamtzeit der Ausführung $1,5 \text{ Sekunden} * N$. Abbildung A.1 zeigt einen Schnappschuss von 13 Seiten oder etwa 19,5 Sekunden.

Einzelner Thread

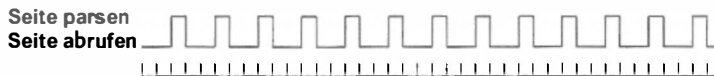


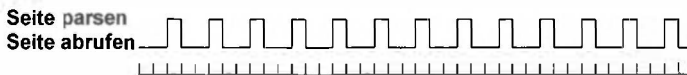
Abb. A.1: Einzelner Thread

Multithread-Berechnung des Durchsatzes

Wenn es möglich ist, Seiten in beliebiger Reihenfolge abzurufen und unabhängig voneinander zu verarbeiten, dann kann der Durchsatz durch Verwendung mehrerer Threads verbessert werden. Was passiert, wenn wir drei Threads verwenden? Wie viele Seiten können wir in derselben Zeit abrufen?

Abbildung A.2 zeigt, dass sich bei der nebenläufigen Lösung das Prozessor-gebundene Parsen der Seiten mit dem I/O-gebundenen Lesen der Seiten überlappen kann. In einer idealen Welt würde dies zu einer vollen Auslastung des Prozessors führen. Jedes eine Sekunde dauernde Lesen einer Seite würde sich mit zwei Parser-Durchläufen überlappen. Deshalb könnten wir zwei Seiten pro Sekunde verarbeiten, was dem dreifachen Durchsatz der Single-Threaded-Lösung entspräche.

Thread 1



Thread 2



Thread 3



Abb. A.2: Drei nebenläufige Threads

A.6 Deadlock

Angenommen, Sie hätten eine Web-Anwendung mit zwei gemeinsam genutzten Ressourcen-Pools endlicher Größe:

- einen Pool von Datenbankverbindungen zur Speicherung von Daten lokaler Verarbeitungsprozesse,
- einen Pool von MQ-Verbindungen zu einem Master Repository.

Nehmen Sie an, es gäbe zwei Operationen in dieser Anwendung, Create (Erstellen) und Update (Aktualisieren):

- Create – eine Verbindung zu dem Master Repository und einer Datenbank herstellen; mit dem Master Repository kommunizieren und dann Daten in der Datenbank für den lokalen Verarbeitungsprozess speichern.
- Update – eine Verbindung zu der Datenbank und zu dem Master Repository herstellen; Daten aus der Datenbank für den lokalen Verarbeitungsprozess lesen und dann an das Master Repository senden.

Was passiert, wenn es mehr Benutzer gibt, als der Pool groß ist? Nehmen Sie an, jeder Pool habe eine Größe von zehn.

- Zehn Benutzer versuchen, Create zu verwenden; deshalb sind alle zehn Datenbankverbindungen belegt; nach der Herstellung der Datenbankverbindung, aber vor der Aufnahme der Verbindung zu dem Master Repository erfolgt ein Thread-Interrupt.
- Zehn Benutzer versuchen, Update zu verwenden; deshalb sind alle zehn Verbindungen zu dem Master Repository belegt; nach der Herstellung der Verbindung zu dem Master Repository, aber vor der Herstellung der Datenbankverbindung erfolgt ein Thread-Interrupt.
- Jetzt müssen die zehn »erstellten« Threads warten, um eine Verbindung zu dem Master Repository zu erhalten, aber die zehn »Update«-Threads müssen warten, um eine Datenbankverbindung zu bekommen.
- Deadlock. Das System erholt sich niemals.

Dies mag sich nach einer unwahrscheinlichen Situation anhören, aber wer möchte schon mit einem System arbeiten, das alle zwei Wochen einfriert? Wer möchte ein System mit Symptomen debuggen, die sich so schwierig reproduzieren lassen? Dies ist die Art von Problemen, mit denen Sie auf diesem Gebiet rechnen müssen. Oft dauert es Wochen, eine Lösung zu finden.

Eine typische »Lösung« besteht darin, Debugging-Anweisungen einzufügen, um das Geschehen zu verfolgen. Natürlich ändern die Debug-Anweisungen den Code so weit, dass der Deadlock in anderen Situationen auftritt. Es kann Monate dauern,

bis die Fehlerkonstellation erneut eintritt. (Beispielsweise könnte jemand Debugging-Output einfügen, der das Problem »verschwinden« lässt. Da das Problem mit dem Debugging-Code nicht mehr auftritt, bleibt dieser einfach im System.)

Um das Deadlock-Problem wirklich zu lösen, müssen wir seine Ursache verstehen. Es müssen vier Bedingungen erfüllt sein, damit ein Deadlock eintreten kann:

- Gegenseitiger Ausschluss (mutual exclusion)
- Sperren & warten (lock & wait)
- Keine präemptive Aktion (no preemption)
- Zirkuläres Warten (circular wait)

Gegenseitiger Ausschluss

Gegenseitiger Ausschluss (mutual exclusion) tritt auf, wenn mehrere Threads dieselben Ressourcen benötigen und diese Ressourcen

- nicht von mehreren Threads gleichzeitig verwendet werden können,
- zahlenmäßig begrenzt sind.

Ein gebräuchliches Beispiel für solche Ressourcen sind Datenbankverbindungen, das Öffnen einer Datei, um Daten zu speichern, eine Datensatzsperrung oder eine Semaphore.

Sperren & warten

Nachdem ein Thread eine Ressource für sich reserviert hat, gibt er die Ressource nicht wieder frei, bis er nicht alle anderen erforderlichen Ressourcen reserviert und seine Arbeit beendet hat.

Keine präemptive Aktion

Ein Thread kann einem anderen Thread keine Ressourcen wegnehmen.

Nachdem ein Thread eine Ressource für sich reserviert hat, kann ein anderer Thread diese nur bekommen, wenn der jetzige Thread sie wieder freigibt.

Zirkuläres Warten

Dies wird auch als »tödliche Umarmung« (deadly embrace) bezeichnet. Stellen Sie sich zwei Threads, T₁ und T₂, und zwei Ressourcen, R₁ und R₂, vor. T₁ hat R₁, T₂ hat R₂. T₁ erfordert auch R₂, und T₂ erfordert auch R₁. Abbildung A.3 veranschaulicht die Situation:

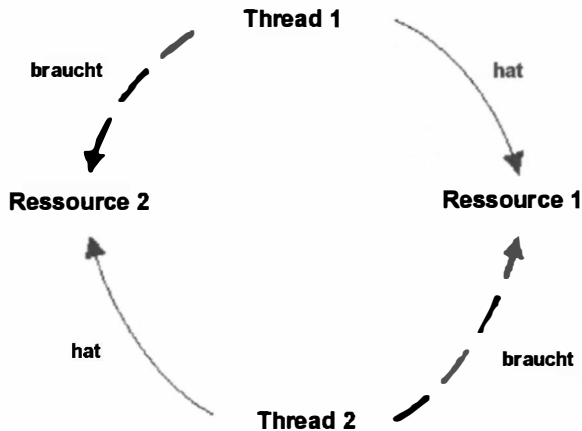


Abb. A.3: Deadlock

Alle vier Bedingungen müssen erfüllt sein, damit ein Deadlock möglich ist. Ist eine dieser Bedingungen nicht erfüllt, ist ein Deadlock nicht möglich.

Den gegenseitigen Ausschluss aufheben

Eine Strategie, um Deadlocks zu vermeiden, besteht darin, die Bedingung des gegenseitigen Ausschlusses zu umgehen. Dies könnten Sie beispielsweise wie folgt erreichen:

- indem Sie Ressourcen benutzen, die gleichzeitig verwendet werden können, wie etwa `AtomicInteger`;
- indem Sie die Anzahl der Ressourcen so vergrößern, dass sie die Anzahl der konkurrierenden Threads erreicht oder überschreitet;
- indem Sie alle Ressourcen prüfen, ob sie frei sind, bevor Sie sie reservieren.

Leider sind die meisten Ressourcen mengenmäßig begrenzt und erlauben keine gleichzeitige Nutzung. Und es ist nicht ungewöhnlich, dass die Identität der zweiten Ressource von den Ergebnissen der Nutzung der ersten abhängt. Doch lassen Sie sich nicht entmutigen; noch gibt es drei weitere Bedingungen.

Das Sperren & Warten aufheben

Sie können einen Deadlock auch beheben, wenn Sie es ablehnen, zu warten. Prüfen Sie jede Ressource, bevor Sie sie reservieren, und geben Sie alle Ressourcen wieder frei und beginnen Sie von vorne, wenn Sie auf eine Ressource stoßen, die belegt ist.

Dieser Ansatz bringt mehrere potenzielle Probleme mit sich:

- **Starvation (Verhungern)** – Ein Thread schafft es dauerhaft nicht, die erforderlichen Ressourcen für sich zu reservieren (vielleicht weil er eine einzigartige Kombination von Ressourcen benötigt, die selten alle gleichzeitig zur Verfügung stehen).
- **Livelock** – Mehrere Threads verhaken sich im Gleichschritt miteinander; alle reservieren immer wieder eine Ressource für sich und geben dann eine Ressource wieder frei. Dies kommt häufig bei simplen Algorithmen zum CPU-Scheduling vor (etwa bei eingebetteten Geräten oder simplen handgeschriebenen Algorithmen für das Thread-Balancing).

Beide Probleme haben einen schlechten Durchsatz zur Folge. Beim ersten Problem wird die CPU unterbeschäftigt, während die CPU beim zweiten stark, aber ergebnislos beschäftigt wird.

Auch wenn diese Strategie ineffizient sein mag, ist sie besser als nichts. Sie hat einen Vorteil: Sie kann fast immer implementiert werden, wenn alles andere scheitert.

Die Präemption umgehen

Eine weitere Strategie zur Vermeidung von Deadlocks besteht darin, Threads zu erlauben, anderen Threads Ressourcen wegzunehmen. Dazu wird normalerweise ein einfacher Anfragemechanismus verwendet. Wenn ein Thread feststellt, dass eine Ressource belegt ist, bittet er den Eigentümer, sie freizugeben. Wenn der Eigentümer ebenfalls auf eine andere Ressource wartet, gibt er alle Ressourcen frei und beginnt von vorne.

Dieser Ansatz ähnelt dem vorhergehenden, hat aber den Vorteil, dass ein Thread auf eine Ressource warten darf. Dadurch wird die Anzahl der Neuanfänge verringert. Seien Sie jedoch gewarnt: Die Verwaltung aller dieser Anfragen kann verzwickelt sein.

Das zirkuläre Warten umgehen

Dies ist der gebräuchlichste Ansatz, um Deadlocks zu vermeiden. Bei den meisten Systemen erfordert er nur eine gegenseitige Übereinkunft aller beteiligten Parteien.

In dem obigen Beispiel, in dem Thread 1 sowohl Ressource 1 als auch Ressource 2 und Thread 2 erst Ressource 2 und dann Ressource 1 benötigt, kann das zirkuläre Warten einfach dadurch umgangen werden, dass sowohl Thread 1 als auch Thread 2 gezwungen werden, ihre Ressourcen in derselben Reihenfolge zu allozieren.

Allgemeiner ausgedrückt: Wenn alle Threads auf eine globale Reihenfolge der Ressourcen festgelegt werden können und alle ihre Ressourcen in dieser Reihenfolge allozieren, ist ein Deadlock unmöglich.

Wie bei den anderen Strategien gibt es auch hier Probleme:

- Die Reihenfolge der Reservierung der Ressourcen entspricht möglicherweise nicht der Reihenfolge, in der sie verwendet werden. Deshalb wird eine Ressource, die bereits am Anfang eines Prozesses reserviert wird, möglicherweise erst am Ende benutzt.
- Dies kann dazu führen, dass Ressourcen länger gesperrt bleiben, als es unbedingt erforderlich wäre.
- Manchmal kann man die Reihenfolge der Reservierung der Ressourcen nicht vorschreiben. Wenn die ID der zweiten Ressource von der Nutzung der ersten Ressource abhängt, kann keine Reihenfolge festgelegt werden.

Es gibt also viele Möglichkeiten, einen Deadlock zu vermeiden. Einige führen zum Verhungern (starvation), während andere die CPU stark beanspruchen und die Reaktionszeit verlangsamen. (TANSTAAFL, »There ain't no such thing as a free lunch«, »Es gibt kein kostenloses Essen«)

Den Thread-bezogenen Teil einer Lösung zu isolieren, um Raum für Anpassungen und Experimente zu schaffen, ist eine leistungsstarke Methode, um die besten Strategien herauszuarbeiten.

A.7 Multithreaded-Code testen

Wie können wir einen Test schreiben, um zu zeigen, dass der folgende Code defekt ist?

```
01: public class ClassWithThreadingProblem {  
02:     int nextId;  
03:  
04:     public int takeNextId() {  
05:         return nextId++;  
06:     }  
07:}
```

Hier ist eine Beschreibung eines Tests, der beweist, dass der Code defekt ist:

- Speichere den gegenwärtigen Wert von nextId.
- Erstelle zwei Threads, die beide takeNextId() einmal aufrufen.
- Prüfe, ob nextId um zwei größer als der Ausgangswert ist.
- Führe den Test aus, bis er zeigt, dass nextId nur um eins statt um zwei vergrößert wurde.

Listing A.2 zeigt einen solchen Test:

Listing A.2: ClassWithThreadingProblemTest.java

```
01: package example;  
02:  
03: import static org.junit.Assert.fail;  
04:  
05: import org.junit.Test;
```

```

06:
07: public class ClassWithThreadingProblemTest {
08:     @Test
09:     public void twoThreadsShouldFailEventually() throws Exception {
10:         final ClassWithThreadingProblem classWithThreadingProblem
            = new ClassWithThreadingProblem();
11:
12:         Runnable runnable = new Runnable() {
13:             public void run() {
14:                 classWithThreadingProblem.takeNextId();
15:             }
16:         };
17:
18:         for (int i = 0; i < 50000; ++i) {
19:             int startingId = classWithThreadingProblem.lastId;
20:             int expectedResult = 2 + startingId;
21:
22:             Thread t1 = new Thread(runnable);
23:             Thread t2 = new Thread(runnable);
24:             t1.start();
25:             t2.start();
26:             t1.join();
27:             t2.join();
28:
29:             int endingId = classWithThreadingProblem.lastId;
30:
31:             if (endingId != expectedResult)
32:                 return;
33:         }
34:
35:         fail("Should have exposed a threading issue but it did not.");
36:     }
37: }

```

Zeile	Beschreibung
10	Eine einzelne Instanz von <code>ClassWithThreadingProblem</code> erstellen. Anmerkung: Wir müssen das Schlüsselwort <code>final</code> verwenden, weil wir sie weiter unten in einer anonymen inneren Klasse benutzen.
12–16	Eine anonyme innere Klasse erstellen, die die einzelne Instanz von <code>ClassWithThreadingProblem</code> verwendet.
18	Diesen Code »genügend oft« ausführen, um zu zeigen, dass er defekt ist, aber nicht so oft, dass der Test »zu lange« dauert. Dies ist ein Balanceakt; wir wollen nicht zu lange warten, um den Defekt zu zeigen. Die passende Anzahl auszuwählen, ist schwierig – obwohl Sie später sehen werden, dass wir diese Anzahl erheblich verringern können.

Zeile	Beschreibung
19	Den Ausgangswert festhalten. Dieser Test versucht nachzuweisen, dass der Code in <code>ClassWithThreadingProblem</code> defekt ist. Wird dieser Test bestanden, beweist er, dass der Code defekt ist. Scheitert dieser Test, konnte er nicht beweisen, dass der Code defekt ist.
20	Wir erwarten, dass der Endwert um zwei größer als der gegenwärtige Wert ist.
22–23	Zwei Threads erstellen, die beide das Objekt verwenden, das wir in den Zeilen 12–16 erstellt haben. Damit erhalten wir zwei mögliche Threads, die versuchen, auf unsere einzige Instanz von <code>ClassWithThreadingProblem</code> zuzugreifen, und sich gegenseitig beeinflussen.
24–25	Die beiden Threads starten.
26–27	Auf das Ende beider Threads warten, bevor die Ergebnisse geprüft werden.
29	Den tatsächlichen Endwert festhalten.
31–32	Weicht <code>endingId</code> von dem erwarteten Wert ab? Falls ja, den Test beenden – wir haben bewiesen, dass der Code defekt ist. Falls nicht, den nächsten Versuch ausführen.
35	Wenn wir hier ankommen, konnte unser Test innerhalb einer »vernünftigen« Zeitspanne nicht nachweisen, dass der Produktionscode defekt ist; unser Code ist gescheitert. Entweder ist der Produktionscode nicht defekt oder wir haben nicht genügend Iterationen ausgeführt, um die Fehlerbedingung herbeizuführen.

Dieser Test schafft mit Sicherheit die Bedingungen für ein nebenläufiges Update-Problem. Doch das Problem tritt so selten auf, dass dieser Test es in der überwiegenden Anzahl der Fälle nicht entdeckt.

Um das Problem wirklich aufzudecken, müssen wir die Anzahl der Iterationen auf über eine Million setzen. Selbst dann, bei zehn Ausführungen mit einer Schleifenanzahl von 1.000.000, trat das Problem nur einmal auf.

Das bedeutet, dass wir die Anzahl der Iterationen wahrscheinlich auf über hundert Millionen setzen sollten, um den Defekt zuverlässig zu entdecken. Wie viel Zeit wollen wir mit Warten verbringen?

Selbst wenn wir den Test so angepasst haben, dass er auf einem Rechner Defekte zuverlässig anzeigt, müssen wir ihn wahrscheinlich bei einem anderen Rechner, unter einem anderen Betriebssystem oder einer anderen Version der JVM erneut anpassen, um zuverlässige Ergebnisse zu erhalten.

Und dies ist ein *einfaches* Problem. Wenn wir einen Code-Defekt schon bei diesem Problem nicht leicht nachweisen können, wie soll uns dies bei wirklichen Problemen gelingen?

Also: Welche Optionen haben wir, um diesen einfachen Defekt nachzuweisen? Und was noch wichtiger ist: Wie können wir Tests schreiben, die Defekte in komplexe-

rem Code nachweisen? Wie können wir feststellen, ob unser Code Defekte hat, wenn wir nicht wissen, wo wir nachschauen müssen?

Hier sind einige Ideen:

- Monte-Carlo-Tests. Schreiben Sie flexible, anpassungsfähige Tests. Führen Sie dann den Tests – etwa auf einem Test-Server – immer wieder aus und ändern Sie dabei zufällig die Einstellungen. Falls der Test überhaupt scheitert, ist der Code defekt. Schreiben Sie diese Tests frühzeitig, damit sie so früh wie möglich von einem Continuous-Integration-Server ausgeführt werden können. Außerdem sollten Sie die Bedingungen, unter denen der Test gescheitert ist, sorgfältig protokollieren.
- Führen Sie den Test auf jeder Ziel-Plattform aus. Wiederholt. Laufend. Je länger der Test ohne Scheitern läuft, desto wahrscheinlicher ist es, dass
- der Produktionscode korrekt ist oder
- die Tests nicht geeignet sind, Probleme aufzudecken.
- Führen Sie die Tests auf einem Rechner unter verschiedenen Lasten aus. Wenn Sie Lasten simulieren können, die denen in der Produktionsumgebung ähneln, sollten Sie dies tun.

Doch selbst wenn Sie alle diese Maßnahmen ergreifen, stehen Ihre Chancen, Threading-Probleme in Ihrem Code aufzudecken, ziemlich schlecht. Die tückischsten Probleme treten nur in exotischen Kombinationen zahlreicher Faktoren, einmal in einer Milliarde Fälle und seltener, auf. Derartige Probleme sind der Fluch komplexer Systeme.

A.8 Threadbasierten Code mit Tools testen

IBM hat ein Tool namens ConTest entwickelt. Sie können damit Klassen so instrumentieren, dass die Wahrscheinlichkeit für ein Scheitern von Code erhöht wird, der nicht thread-sicher ist (<http://www.haifa.ibm.com/projects/verification/contest/index.html>).

Wir haben keine direkte Beziehung zu IBM oder dem Team, das ConTest entwickelt hat. Ein Kollege hat uns auf dieses Tool aufmerksam gemacht. Schon wenige Minuten nach Gebrauch des Tools stellten wir fest, dass sich unsere Fähigkeit, Threading-Probleme aufzudecken, erheblich verbessert hatte.

ConTest wird im Wesentlichen wie folgt benutzt:

- Sie schreiben Produktionscode und Tests, die, wie weiter vorne erwähnt, mehrere Benutzer unter verschiedenen Lasten simulieren.
- Sie instrumentieren den Produktionscode und die Tests mit ConTest.
- Sie führen die Tests aus.

Als wir Code mit ConTest instrumentierten, stieg unsere Erfolgsquote von etwa einem Scheitern in zehn Millionen Iterationen auf etwa ein Scheitern in *dreißig* Iterationen. Hier sind die Schleifen-Werte für mehrere Ausführungen des Tests nach der Instrumentierung: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Sie zeigen ganz deutlich, dass die instrumentierten Klassen viel früher und viel zuverlässiger scheiterten.

A.9 Zusammenfassung

Dieses Kapitel enthält einen sehr kurzen Überblick über das riesige und tückische Feld der nebenläufigen Programmierung. Er kratzt kaum die Oberfläche dieses Themas an und behandelt hauptsächlich Techniken, die Ihnen helfen, nebenläufigen Code sauber zu programmieren. Doch bevor Sie anfangen, selbst nebenläufige Systeme zu programmieren, müssen Sie noch sehr viel mehr lernen. Wir empfehlen Ihnen, mit dem wundervollen Buch *Concurrent Programming in Java: Design Principles and Patterns* von Doug Lea zu beginnen (siehe [Lea99], S. 191).

In diesem Kapitel werden nebenläufige Updates sowie die Techniken einer sauberen Synchronisation und des Lockings beschrieben, die sie verhindern können. Außerdem wird beschrieben, wie Threads den Durchsatz eines I/O-gebundenen Systems verbessern können und mit welchen sauberen Techniken Sie derartige Verbesserungen realisieren können. Es wird beschrieben, welche Deadlocks auftreten können und mit welchen Techniken Sie sie sauber verhindern können. Schließlich werden Strategien beschrieben, wie Sie Probleme in nebenläufigem Code aufdecken können, indem Sie diesen instrumentieren.

A.10 Tutorial: kompletter Beispielcode

Client/Server ohne Threads

Listing A.3: Server.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }
}
```

```

public void run() {
    System.out.printf("Server Starting\n");

    while (keepProcessing) {
        try {
            System.out.printf("accepting client\n");
            Socket socket = serverSocket.accept();
            System.out.printf("got client\n");
            process(socket);
        } catch (Exception e) {
            handle(e);
        }
    }
}

private void handle(Exception e) {
    if (!(e instanceof SocketException)) {
        e.printStackTrace();
    }
}

public void stopProcessing() {
    keepProcessing = false;
    closeIgnoringException(serverSocket);
}

void process(Socket socket) {
    if (socket == null)
        return;

    try {
        System.out.printf("Server: getting message\n");
        String message = MessageUtils.getMessage(socket);
        System.out.printf("Server: got message: %s\n", message);
        Thread.sleep(1000);
        System.out.printf("Server: sending reply: %s\n", message);
        MessageUtils.sendMessage(socket, "Processed: " + message);
        System.out.printf("Server: sent\n");
        closeIgnoringException(socket);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

private void closeIgnoringException(Socket socket) {
    if (socket != null)
        try {
            socket.close();
        } catch (IOException ignore) {
        }
}

```

```
private void closeIgnoringException(ServerSocket serverSocket) {
    if (serverSocket != null)
        try {
            serverSocket.close();
        } catch (IOException ignore) {
        }
    }
}
```

Listing A.4: ClientTest.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
    }
}
```

```

        closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
        if (socket == null)
            return;

        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}

```

Listing A.5: MessageUtils.java

```

package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
    public static void sendMessage(Socket socket, String message)
        throws IOException {

```



```
OutputStream stream = socket.getOutputStream();
ObjectOutputStream oos = new ObjectOutputStream(stream);
oos.writeUTF(message);
oos.flush();
}

public static String getMessage(Socket socket) throws IOException {
    InputStream stream = socket.getInputStream();
    ObjectInputStream ois = new ObjectInputStream(stream);
    return ois.readUTF();
}
}
```

Client/Server mit Threads

Den Server zur Verwendung von Threads zu veranlassen, erfordert nur die Änderung der Prozess-Message (neue Zeilen sind hervorgehoben):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                System.out.printf("Server: getting message\n");
                String message = MessageUtils.getMessage(socket);
                System.out.printf("Server: got message: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Server: sending reply: %s\n", message);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                System.out.printf("Server: sent\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

org.jfree.date.SerialDate

Listing B.1: SerialDate.java

```

1 /* =====
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * =====
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * SerialDate.java
29 * -----
30 * (C) Copyright 2001-2005, by Object Refinery Limited.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s):  -;
34 *
35 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
36 *
37 * Changes (from 11-Oct-2001)
38 * -----
39 * 11-Oct-2001 : Re-organised the class and moved it to new package
40 *               com.jrefinery.date (DG);
41 * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
42 *               class (DG);
43 * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
44 *               class is gone (DG); Changed getPreviousDayOfWeek(),
45 *               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
46 *               bugs (DG);
47 * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);

```

```

48 * 29-May-2002 : Moved the month constants into a separate interface
49 *             (MonthConstants) (DG);
50 * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
51 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
52 * 13-Mar-2003 : Implemented Serializable (DG);
53 * 29-May-2003 : Fixed bug in addMonths method (DG);
54 * 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
55 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
56 *
57 */
58
59 package org.jfree.date;
60
61 import java.io.Serializable;
62 import java.text.DateFormatSymbols;
63 import java.text.SimpleDateFormat;
64 import java.util.Calendar;
65 import java.util.GregorianCalendar;
66
67 /**
68 * An abstract class that defines our requirements for manipulating dates,
69 * without tying down a particular implementation.
70 * <P>
71 * Requirement 1 : match at least what Excel does for dates;
72 * Requirement 2 : class is immutable;
73 * <P>
74 * Why not just use java.util.Date? We will, when it makes sense. At times,
75 * java.util.Date can be too precise - it represents an instant in time,
76 * accurate to 1/1000th of a second (with the date itself depending on the
77 * time-zone). Sometimes we just want to represent a particular day (e.g. 21
78 * January 2015) without concerning ourselves about the time of day, or the
79 * time-zone, or anything else. That's what we've defined SerialDate for.
80 * <P>
81 * You can call getInstance() to get a concrete subclass of SerialDate,
82 * without worrying about the exact implementation.
83 *
84 * @author David Gilbert
85 */
86 public abstract class SerialDate implements Comparable,
87                                     Serializable,
88                                     MonthConstants {
89
90     /** For serialization. */
91     private static final long serialVersionUID = -293716040467423637L;
92
93     /** Date format symbols. */
94     public static final DateFormatSymbols
95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
96
97     /** The serial number for 1 January 1900. */
98     public static final int SERIAL_LOWER_BOUND = 2;
99
100    /** The serial number for 31 December 9999. */
101    public static final int SERIAL_UPPER_BOUND = 2958465;
102
103    /** The lowest year value supported by this date format. */
104    public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105

```

```

106  /** The highest year value supported by this date format. */
107  public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109  /** Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. */
110  public static final int MONDAY = Calendar.MONDAY;
111
112  /**
113   * Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY.
114   */
115  public static final int TUESDAY = Calendar.TUESDAY;
116
117  /**
118   * Useful constant for Wednesday. Equivalent to
119   * java.util.Calendar.WEDNESDAY.
120   */
121  public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123  /**
124   * Useful constant for Thursday. Equivalent to java.util.Calendar.THURSDAY.
125   */
126  public static final int THURSDAY = Calendar.THURSDAY;
127
128  /** Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. */
129  public static final int FRIDAY = Calendar.FRIDAY;
130
131  /**
132   * Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY.
133   */
134  public static final int SATURDAY = Calendar.SATURDAY;
135
136  /** Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. */
137  public static final int SUNDAY = Calendar.SUNDAY;
138
139  /** The number of days in each month in non leap years. */
140  static final int[] LAST_DAY_OF_MONTH =
141      {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143  /** The number of days in a (non-leap) year up to the end of each month. */
144  static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145      {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147  /** The number of days in a year up to the end of the preceding month. */
148  static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149      {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151  /** The number of days in a leap year up to the end of each month. */
152  static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153      {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155  /**
156   * The number of days in a leap year up to the end of the preceding month.
157   */
158  static final int[]
159      LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160      {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162  /** A useful constant for referring to the first week in a month. */
163  public static final int FIRST_WEEK_IN_MONTH = 1;

```

```
164
165 /** A useful constant for referring to the second week in a month. */
166 public static final int SECOND_WEEK_IN_MONTH = 2;
167
168 /** A useful constant for referring to the third week in a month. */
169 public static final int THIRD_WEEK_IN_MONTH = 3;
170
171 /** A useful constant for referring to the fourth week in a month. */
172 public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174 /** A useful constant for referring to the last week in a month. */
175 public static final int LAST_WEEK_IN_MONTH = 0;
176
177 /** Useful range constant. */
178 public static final int INCLUDE_NONE = 0;
179
180 /** Useful range constant. */
181 public static final int INCLUDE_FIRST = 1;
182
183 /** Useful range constant. */
184 public static final int INCLUDE_SECOND = 2;
185
186 /** Useful range constant. */
187 public static final int INCLUDE_BOTH = 3;
188
189 /**
190  * Useful constant for specifying a day of the week relative to a fixed
191  * date.
192  */
193 public static final int PRECEDING = -1;
194
195 /**
196  * Useful constant for specifying a day of the week relative to a fixed
197  * date.
198  */
199 public static final int NEAREST = 0;
200
201 /**
202  * Useful constant for specifying a day of the week relative to a fixed
203  * date.
204  */
205 public static final int FOLLOWING = 1;
206
207 /** A description for the date. */
208 private String description;
209
210 /**
211  * Default constructor.
212  */
213 protected SerialDate() {
214 }
215
216 /**
217  * Returns <code>true</code> if the supplied integer code represents a
218  * valid day-of-the-week, and <code>false</code> otherwise.
219  *
220  * @param code the code being checked for validity.
221  */
```

```

222  * @return <code>true</code> if the supplied integer code represents a
223  *      valid day-of-the-week, and <code>false</code> otherwise.
224  */
225  public static boolean isValidWeekdayCode(final int code) {
226
227      switch(code) {
228          case SUNDAY:
229          case MONDAY:
230          case TUESDAY:
231          case WEDNESDAY:
232          case THURSDAY:
233          case FRIDAY:
234          case SATURDAY:
235              return true;
236          default:
237              return false;
238      }
239
240  }
241
242  /**
243   * Converts the supplied string to a day of the week.
244   *
245   * @param s a string representing the day of the week.
246   *
247   * @return <code>-1</code> if the string is not convertible, the day of
248   *      the week otherwise.
249   */
250  public static int stringToWeekdayCode(String s) {
251
252      final String[] shortWeekdayNames
253          = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254      final String[] weekdayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256      int result = -1;
257      s = s.trim();
258      for (int i = 0; i < weekdayNames.length; i++) {
259          if (s.equals(shortWeekdayNames[i])) {
260              result = i;
261              break;
262          }
263          if (s.equals(weekdayNames[i])) {
264              result = i;
265              break;
266          }
267      }
268      return result;
269
270  }
271
272  /**
273   * Returns a string representing the supplied day-of-the-week.
274   * <P>
275   * Need to find a better approach.
276   *
277   * @param weekday the day of the week.
278   *
279   * @return a string representing the supplied day-of-the-week.

```

```
280     */
281     public static String weekdayCodeToString(final int weekday) {
282
283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284         return weekdays[weekday];
285     }
286 }
287
288 /**
289  * Returns an array of month names.
290  *
291  * @return an array of month names.
292  */
293 public static String[] getMonths() {
294
295     return getMonths(false);
296 }
297
298 /**
299  * Returns an array of month names.
300  *
301  * @param shortened a flag indicating that shortened month names should
302  *                 be returned.
303  *
304  * @return an array of month names.
305  */
306 public static String[] getMonths(final boolean shortened) {
307
308     if (shortened) {
309         return DATE_FORMAT_SYMBOLS.getShortMonths();
310     }
311     else {
312         return DATE_FORMAT_SYMBOLS.getMonths();
313     }
314 }
315
316 }
317
318 /**
319  * Returns true if the supplied integer code represents a valid month.
320  *
321  * @param code the code being checked for validity.
322  *
323  * @return <code>true</code> if the supplied integer code represents a
324  *         valid month.
325  */
326 public static boolean isValidMonthCode(final int code) {
327
328     switch (code) {
329         case JANUARY:
330         case FEBRUARY:
331         case MARCH:
332         case APRIL:
333         case MAY:
334         case JUNE:
335         case JULY:
336         case AUGUST:
337         case SEPTEMBER:
```

```

338         case OCTOBER:
339         case NOVEMBER:
340         case DECEMBER:
341             return true;
342         default:
343             return false;
344     }
345
346 }
347
348 /**
349  * Returns the quarter for the specified month.
350  *
351  * @param code the month code (1-12).
352  *
353  * @return the quarter that the month belongs to.
354  * @throws java.lang.IllegalArgumentException
355  */
356 public static int monthCodeToQuarter(final int code) {
357
358     switch(code) {
359         case JANUARY:
360         case FEBRUARY:
361         case MARCH: return 1;
362         case APRIL:
363         case MAY:
364         case JUNE: return 2;
365         case JULY:
366         case AUGUST:
367         case SEPTEMBER: return 3;
368         case OCTOBER:
369         case NOVEMBER:
370         case DECEMBER: return 4;
371         default: throw new IllegalArgumentException(
372             "SerialDate.monthCodeToQuarter: invalid month code.");
373     }
374
375 }
376
377 /**
378  * Returns a string representing the supplied month.
379  * <P>
380  * The string returned is the long form of the month name taken from the
381  * default locale.
382  *
383  * @param month the month.
384  *
385  * @return a string representing the supplied month.
386  */
387 public static String monthCodeToString(final int month) {
388
389     return monthCodeToString(month, false);
390
391 }
392
393 /**
394  * Returns a string representing the supplied month.
395  * <P>

```



```

396  * The string returned is the long or short form of the month name taken
397  * from the default locale.
398  *
399  * @param month the month.
400  * @param shortened if <code>true</code> return the abbreviation of the
401  *      month.
402  *
403  * @return a string representing the supplied month.
404  * @throws java.lang.IllegalArgumentException
405  */
406  public static String monthCodeToString(final int month,
407      final boolean shortened) {
408
409      // check arguments...
410      if (!isValidMonthCode(month)) {
411          throw new IllegalArgumentException(
412              "SerialDate.monthCodeToString: month outside valid range.");
413      }
414
415      final String[] months;
416
417      if (shortened) {
418          months = DATE_FORMAT_SYMBOLS.getShortMonths();
419      }
420      else {
421          months = DATE_FORMAT_SYMBOLS.getMonths();
422      }
423
424      return months[month - 1];
425  }
426
427  /**
428   * Converts a string to a month code.
429   * <P>
430   * This method will return one of the constants JANUARY, FEBRUARY, ...,
431   * DECEMBER that corresponds to the string. If the string is not
432   * recognised, this method returns -1.
433   *
434   * @param s the string to parse.
435   *
436   * @return <code>-1</code> if the string is not parseable, the month of the
437   *      year otherwise.
438   */
439  public static int stringToMonthCode(String s) {
440
441      final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
442      final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
443
444      int result = -1;
445      s = s.trim();
446
447      // first try parsing the string as an integer (1-12)...
448      try {
449          result = Integer.parseInt(s);
450      }
451      catch (NumberFormatException e) {
452          // suppress

```

```

454     }
455
456     // now search through the month names...
457     if ((result < 1) || (result > 12)) {
458         for (int i = 0; i < monthNames.length; i++) {
459             if (s.equals(shortMonthNames[i])) {
460                 result = i + 1;
461                 break;
462             }
463             if (s.equals(monthNames[i])) {
464                 result = i + 1;
465                 break;
466             }
467         }
468     }
469
470     return result;
471
472 }
473
474 /**
475  * Returns true if the supplied integer code represents a valid
476  * week-in-the-month, and false otherwise.
477  *
478  * @param code the code being checked for validity.
479  * @return <code>true</code> if the supplied integer code represents a
480  *         valid week-in-the-month.
481  */
482 public static boolean isValidWeekInMonthCode(final int code) {
483
484     switch(code) {
485         case FIRST_WEEK_IN_MONTH:
486         case SECOND_WEEK_IN_MONTH:
487         case THIRD_WEEK_IN_MONTH:
488         case FOURTH_WEEK_IN_MONTH:
489         case LAST_WEEK_IN_MONTH: return true;
490         default: return false;
491     }
492
493 }
494
495 /**
496  * Determines whether or not the specified year is a leap year.
497  *
498  * @param yyyy the year (in the range 1900 to 9999).
499  *
500  * @return <code>true</code> if the specified year is a leap year.
501  */
502 public static boolean isLeapYear(final int yyyy) {
503
504     if ((yyyy % 4) != 0) {
505         return false;
506     }
507     else if ((yyyy % 400) == 0) {
508         return true;
509     }
510     else if ((yyyy % 100) == 0) {
511         return false;

```

```

512     }
513     else {
514         return true;
515     }
516 }
517 }
518
519 /**
520  * Returns the number of leap years from 1900 to the specified year
521  * INCLUSIVE.
522  * <P>
523  * Note that 1900 is not a leap year.
524  *
525  * @param yyyy the year (in the range 1900 to 9999).
526  *
527  * @return the number of leap years from 1900 to the specified year.
528  */
529 public static int leapYearCount(final int yyyy) {
530
531     final int leap4 = (yyyy - 1896) / 4;
532     final int leap100 = (yyyy - 1800) / 100;
533     final int leap400 = (yyyy - 1600) / 400;
534     return leap4 - leap100 + leap400;
535 }
536 }
537
538 /**
539  * Returns the number of the last day of the month, taking into account
540  * leap years.
541  *
542  * @param month the month.
543  * @param yyyy the year (in the range 1900 to 9999).
544  *
545  * @return the number of the last day of the month.
546  */
547 public static int lastDayOfMonth(final int month, final int yyyy) {
548
549     final int result = LAST_DAY_OF_MONTH[month];
550     if (month != FEBRUARY) {
551         return result;
552     }
553     else if (isLeapYear(yyyy)) {
554         return result + 1;
555     }
556     else {
557         return result;
558     }
559 }
560 }
561
562 /**
563  * Creates a new date by adding the specified number of days to the base
564  * date.
565  *
566  * @param days the number of days to add (can be negative).
567  * @param base the base date.
568  *
569  * @return a new date.

```

```

570 */
571 public static SerialDate addDays(final int days, final SerialDate base) {
572     final int serialDayNumber = base.toSerial() + days;
573     return SerialDate.createInstance(serialDayNumber);
574 }
575
576 /**
577  * Creates a new date by adding the specified number of months to the base
578  * date.
579  * <P>
580  * If the base date is close to the end of the month, the day on the result
581  * may be adjusted slightly: 31 May + 1 month = 30 June.
582  *
583  * @param months the number of months to add (can be negative).
584  * @param base the base date.
585  * @return a new date.
586 */
587 public static SerialDate addMonths(final int months,
588                                     final SerialDate base) {
589     final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
590                     / 12;
591     final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
592                     % 12 + 1;
593     final int dd = Math.min(
594         base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
595     );
596     return SerialDate.createInstance(dd, mm, yy);
597 }
598
599 /**
600  * Creates a new date by adding the specified number of years to the base
601  * date.
602  *
603  * @param years the number of years to add (can be negative).
604  * @param base the base date.
605  * @return A new date.
606 */
607 public static SerialDate addYears(final int years, final SerialDate base) {
608     final int baseY = base.getYYYY();
609     final int baseM = base.getMonth();
610     final int baseD = base.getDayOfMonth();
611
612     final int targetY = baseY + years;
613     final int targetD = Math.min(
614         baseD, SerialDate.lastDayOfMonth(baseM, targetY)
615     );
616     return SerialDate.createInstance(targetD, baseM, targetY);
617 }

```

```

628  /**
629   * Returns the latest date that falls on the specified day-of-the-week and
630   * is BEFORE the base date.
631   *
632   * @param targetWeekday a code for the target day-of-the-week.
633   * @param base the base date.
634   *
635   * @return the latest date that falls on the specified day-of-the-week and
636   *         is BEFORE the base date.
637   */
638  public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                              final SerialDate base) {
640
641      // check arguments...
642      if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643          throw new IllegalArgumentException(
644              "Invalid day-of-the-week code."
645          );
646      }
647
648      // find the date...
649      final int adjust;
650      final int baseDOW = base.getDayOfWeek();
651      if (baseDOW > targetWeekday) {
652          adjust = Math.min(0, targetWeekday - baseDOW);
653      }
654      else {
655          adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656      }
657
658      return SerialDate.addDays(adjust, base);
659  }
660
661  /**
662   * Returns the earliest date that falls on the specified day-of-the-week
663   * and is AFTER the base date.
664   *
665   * @param targetWeekday a code for the target day-of-the-week.
666   * @param base the base date.
667   *
668   * @return the earliest date that falls on the specified day-of-the-week
669   *         and is AFTER the base date.
670   */
671  public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
672                                              final SerialDate base) {
673
674      // check arguments...
675      if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
676          throw new IllegalArgumentException(
677              "Invalid day-of-the-week code."
678          );
679      }
680
681      // find the date...
682      final int adjust;
683      final int baseDOW = base.getDayOfWeek();
684      if (baseDOW > targetWeekday) {
685          adjust = 7 + Math.min(0, targetWeekday - baseDOW);

```

```

687     }
688     else {
689         adjust = Math.max(0, targetWeekday - baseDOW);
690     }
691
692     return SerialDate.addDays(adjust, base);
693 }
694
695 /**
696  * Returns the date that falls on the specified day-of-the-week and is
697  * CLOSEST to the base date.
698  *
699  * @param targetDOW a code for the target day-of-the-week.
700  * @param base the base date.
701  *
702  * @return the date that falls on the specified day-of-the-week and is
703  *         CLOSEST to the base date.
704  */
705 public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                             final SerialDate base) {
707
708     // check arguments...
709     if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710         throw new IllegalArgumentException(
711             "Invalid day-of-the-week code."
712         );
713     }
714
715     // find the date...
716     final int baseDOW = base.getDayOfWeek();
717     int adjust = -Math.abs(targetDOW - baseDOW);
718     if (adjust >= 4) {
719         adjust = 7 - adjust;
720     }
721     if (adjust <= -4) {
722         adjust = 7 + adjust;
723     }
724     return SerialDate.addDays(adjust, base);
725 }
726
727 /**
728  * Rolls the date forward to the last day of the month.
729  *
730  * @param base the base date.
731  *
732  * @return a new serial date.
733  */
734 public SerialDate getEndOfCurrentMonth(final SerialDate base) {
735     final int last = SerialDate.lastDayOfMonth(
736         base.getMonth(), base.getYYYY()
737     );
738     return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
739 }
740
741 /**
742  * Returns a string corresponding to the week-in-the-month code.
743  * <P>
744  * Need to find a better approach.
745  */

```

```

746  *
747  * @param count  an integer code representing the week-in-the-month.
748  *
749  * @return a string corresponding to the week-in-the-month code.
750  */
751  public static String weekInMonthToString(final int count) {
752
753      switch (count) {
754          case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
755          case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
756          case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
757          case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
758          case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
759          default :
760              return "SerialDate.weekInMonthToString(): invalid code.";
761      }
762
763  }
764
765  /**
766   * Returns a string representing the supplied 'relative'.
767   * <P>
768   * Need to find a better approach.
769   *
770   * @param relative  a constant representing the 'relative'.
771   *
772   * @return a string representing the supplied 'relative'.
773   */
774  public static String relativeToString(final int relative) {
775
776      switch (relative) {
777          case SerialDate.PRECEDING : return "Preceding";
778          case SerialDate.NEAREST : return "Nearest";
779          case SerialDate.FOLLOWING : return "Following";
780          default : return "ERROR : Relative To String";
781      }
782
783  }
784
785  /**
786   * Factory method that returns an instance of some concrete subclass of
787   * {@link SerialDate}.
788   *
789   * @param day  the day (1-31).
790   * @param month  the month (1-12).
791   * @param yyyy  the year (in the range 1900 to 9999).
792   *
793   * @return An instance of {@link SerialDate}.
794   */
795  public static SerialDate createInstance(final int day, final int month,
796                                         final int yyyy) {
797      return new SpreadsheetDate(day, month, yyyy);
798  }
799
800  /**
801   * Factory method that returns an instance of some concrete subclass of
802   * {@link SerialDate}.
803   *

```

```

804  * @param serial the serial number for the day (1 January 1900 = 2).
805  *
806  * @return a instance of SerialDate.
807  */
808  public static SerialDate createInstance(final int serial) {
809      return new SpreadsheetDate(serial);
810  }
811
812  /**
813   * Factory method that returns an instance of a subclass of SerialDate.
814   *
815   * @param date A Java date object.
816   *
817   * @return a instance of SerialDate.
818   */
819  public static SerialDate createInstance(final java.util.Date date) {
820
821      final GregorianCalendar calendar = new GregorianCalendar();
822      calendar.setTime(date);
823      return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                calendar.get(Calendar.MONTH) + 1,
825                                calendar.get(Calendar.YEAR));
826  }
827  }
828
829  /**
830   * Returns the serial number for the date, where 1 January 1900 = 2 (this
831   * corresponds, almost, to the numbering system used in Microsoft Excel for
832   * Windows and Lotus 1-2-3).
833   *
834   * @return the serial number for the date.
835   */
836  public abstract int toSerial();
837
838  /**
839   * Returns a java.util.Date. Since java.util.Date has more precision than
840   * SerialDate, we need to define a convention for the 'time of day'.
841   *
842   * @return this as <code>java.util.Date</code>.
843   */
844  public abstract java.util.Date toDate();
845
846  /**
847   * Returns a description of the date.
848   *
849   * @return a description of the date.
850   */
851  public String getDescription() {
852      return this.description;
853  }
854
855  /**
856   * Sets the description for the date.
857   *
858   * @param description the new description for the date.
859   */
860  public void setDescription(final String description) {
861      this.description = description;

```



```
862     }
863
864     /**
865      * Converts the date to a string.
866      *
867      * @return a string representation of the date.
868      */
869     public String toString() {
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871             + "-" + getYYYY();
872     }
873
874     /**
875      * Returns the year (assume a valid range of 1900 to 9999).
876      *
877      * @return the year.
878      */
879     public abstract int getYYYY();
880
881     /**
882      * Returns the month (January = 1, February = 2, March = 3).
883      *
884      * @return the month of the year.
885      */
886     public abstract int getMonth();
887
888     /**
889      * Returns the day of the month.
890      *
891      * @return the day of the month.
892      */
893     public abstract int getDayOfMonth();
894
895     /**
896      * Returns the day of the week.
897      *
898      * @return the day of the week.
899      */
900     public abstract int getDayOfWeek();
901
902     /**
903      * Returns the difference (in days) between this date and the specified
904      * 'other' date.
905      * <P>
906      * The result is positive if this date is after the 'other' date and
907      * negative if it is before the 'other' date.
908      *
909      * @param other the date being compared to.
910      *
911      * @return the difference between this and the other date.
912      */
913     public abstract int compare(SerialDate other);
914
915     /**
916      * Returns true if this SerialDate represents the same date as the
917      * specified SerialDate.
918      *
919      * @param other the date being compared to.
```

```

920  *
921  * @return <code>true</code> if this SerialDate represents the same date as
922  *       the specified SerialDate.
923  */
924  public abstract boolean isOn(SerialDate other);
925
926  /**
927   * Returns true if this SerialDate represents an earlier date compared to
928   * the specified SerialDate.
929   *
930   * @param other The date being compared to.
931   *
932   * @return <code>true</code> if this SerialDate represents an earlier date
933   *         compared to the specified SerialDate.
934   */
935  public abstract boolean isBefore(SerialDate other);
936
937  /**
938   * Returns true if this SerialDate represents the same date as the
939   * ~ specified SerialDate.
940   *
941   * @param other the date being compared to.
942   *
943   * @return <code>true</code> if this SerialDate represents the same date
944   *         as the specified SerialDate.
945   */
946  public abstract boolean isOnOrBefore(SerialDate other);
947
948  /**
949   * Returns true if this SerialDate represents the same date as the
950   * ~ specified SerialDate.
951   *
952   * @param other the date being compared to.
953   *
954   * @return <code>true</code> if this SerialDate represents the same date
955   *         as the specified SerialDate.
956   */
957  public abstract boolean isAfter(SerialDate other);
958
959  /**
960   * Returns true if this SerialDate represents the same date as the
961   * ~ specified SerialDate.
962   *
963   * @param other the date being compared to.
964   *
965   * @return <code>true</code> if this SerialDate represents the same date
966   *         as the specified SerialDate.
967   */
968  public abstract boolean isOnOrAfter(SerialDate other);
969
970  /**
971   * Returns <code>true</code> if this {@link SerialDate} is within the
972   * specified range (INCLUSIVE). The date order of d1 and d2 is not
973   * important.
974   *
975   * @param d1 a boundary date for the range.
976   * @param d2 the other boundary date for the range.
977   *

```

```
978     * @return A boolean.
979     */
980     public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982     /**
983     * Returns <code>true</code> if this {@link SerialDate} is within the
984     * specified range (caller specifies whether or not the end-points are
985     * included). The date order of d1 and d2 is not important.
986     *
987     * @param d1 a boundary date for the range.
988     * @param d2 the other boundary date for the range.
989     * @param include a code that controls whether or not the start and end
990     *               dates are included in the range.
991     *
992     * @return A boolean.
993     */
994     public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995                                     int include);
996
997     /**
998     * Returns the latest date that falls on the specified day-of-the-week and
999     * is BEFORE this date.
1000     *
1001     * @param targetDOW a code for the target day-of-the-week.
1002     *
1003     * @return the latest date that falls on the specified day-of-the-week and
1004     *         is BEFORE this date.
1005     */
1006     public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007         return getPreviousDayOfWeek(targetDOW, this);
1008     }
1009
1010     /**
1011     * Returns the earliest date that falls on the specified day-of-the-week
1012     * and is AFTER this date.
1013     *
1014     * @param targetDOW a code for the target day-of-the-week.
1015     *
1016     * @return the earliest date that falls on the specified day-of-the-week
1017     *         and is AFTER this date.
1018     */
1019     public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020         return getFollowingDayOfWeek(targetDOW, this);
1021     }
1022
1023     /**
1024     * Returns the nearest date that falls on the specified day-of-the-week.
1025     *
1026     * @param targetDOW a code for the target day-of-the-week.
1027     *
1028     * @return the nearest date that falls on the specified day-of-the-week.
1029     */
1030     public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031         return getNearestDayOfWeek(targetDOW, this);
1032     }
1033 }
1034 }
```

Listing B.2: SerialDateTest.java

```

1 /* =====
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * =====
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * SerialDateTests.java
29 * -----
30 * (C) Copyright 2001-2005, by Object Refinery Limited.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s):  -;
34 *
35 * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Changes
38 * -----
39 * 15-Nov-2001 : Version 1 (DG);
40 * 25-Jun-2002 : Removed unnecessary import (DG);
41 * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
42 * 13-Mar-2003 : Added serialization test (DG);
43 * 05-Jan-2005 : Added test for bug report 1096282 (DG);
44 *
45 */
46
47 package org.jfree.date.junit;
48
49 import java.io.ByteArrayInputStream;
50 import java.io.ByteArrayOutputStream;
51 import java.io.ObjectInput;
52 import java.io.ObjectInputStream;
53 import java.io.ObjectOutput;
54 import java.io.ObjectOutputStream;
55
56 import junit.framework.Test;

```

```
57 import junit.framework.TestCase;
58 import junit.framework.TestSuite;
59
60 import org.jfree.date.MonthConstants;
61 import org.jfree.date.SerialDate;
62
63 /**
64  * Some JUnit tests for the {@link SerialDate} class.
65  */
66 public class SerialDateTests extends TestCase {
67
68     /** Date representing November 9. */
69     private SerialDate nov9Y2001;
70
71     /**
72      * Creates a new test case.
73      *
74      * @param name the name.
75      */
76     public SerialDateTests(final String name) {
77         super(name);
78     }
79
80     /**
81      * Returns a test suite for the JUnit test runner.
82      *
83      * @return The test suite.
84      */
85     public static Test suite() {
86         return new TestSuite(SerialDateTests.class);
87     }
88
89     /**
90      * Problem set up.
91      */
92     protected void setUp() {
93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
94     }
95
96     /**
97      * 9 Nov 2001 plus two months should be 9 Jan 2002.
98      */
99     public void testAddMonthsTo9Nov2001() {
100         final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101         final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102         assertEquals(answer, jan9Y2002);
103     }
104
105     /**
106      * A test case for a reported bug, now fixed.
107      */
108     public void testAddMonthsTo5Oct2003() {
109         final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110         final SerialDate d2 = SerialDate.addMonths(2, d1);
111         assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112     }
113
114     /**
```

```

115  * A test case for a reported bug, now fixed.
116  */
117  public void testAddMonthsTo1Jan2003() {
118      final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119      final SerialDate d2 = SerialDate.addMonths(0, d1);
120      assertEquals(d2, d1);
121  }
122
123  /**
124   * Monday preceding Friday 9 November 2001 should be 5 November.
125   */
126  public void testMondayPrecedingFriday9Nov2001() {
127      SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128          SerialDate.MONDAY, this.nov9Y2001
129      );
130      assertEquals(5, mondayBefore.getDayOfMonth());
131  }
132
133  /**
134   * Monday following Friday 9 November 2001 should be 12 November.
135   */
136  public void testMondayFollowingFriday9Nov2001() {
137      SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138          SerialDate.MONDAY, this.nov9Y2001
139      );
140      assertEquals(12, mondayAfter.getDayOfMonth());
141  }
142
143  /**
144   * Monday nearest Friday 9 November 2001 should be 12 November.
145   */
146  public void testMondayNearestFriday9Nov2001() {
147      SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148          SerialDate.MONDAY, this.nov9Y2001
149      );
150      assertEquals(12, mondayNearest.getDayOfMonth());
151  }
152
153  /**
154   * The Monday nearest to 22nd January 1970 falls on the 19th.
155   */
156  public void testMondayNearest22Jan1970() {
157      SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970);
158      SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(SerialDate.MONDAY,
159          jan22Y1970);
159      assertEquals(19, mondayNearest.getDayOfMonth());
160  }
161
162  /**
163   * Problem that the conversion of days to strings returns the right result. Actually, this
164   * result depends on the Locale so this test needs to be modified.
165   */
166  public void testWeekdayCodeToString() {
167
168      final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169      assertEquals("Saturday", test);
170  }
171  }

```

```
172
173 /**
174  * Test the conversion of a string to a weekday. Note that this test will fail if the
175  * default locale doesn't use English weekday names...devise a better test!
176  */
177 public void testStringToWeekday() {
178
179     int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180     assertEquals(SerialDate.WEDNESDAY, weekday);
181
182     weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183     assertEquals(SerialDate.WEDNESDAY, weekday);
184
185     weekday = SerialDate.stringToWeekdayCode("Wed");
186     assertEquals(SerialDate.WEDNESDAY, weekday);
187
188 }
189
190 /**
191  * Test the conversion of a string to a month. Note that this test will fail if the
192  * default locale doesn't use English month names...devise a better test!
193  */
194 public void testStringToMonthCode() {
195
196     int m = SerialDate.stringToMonthCode("January");
197     assertEquals(MonthConstants.JANUARY, m);
198
199     m = SerialDate.stringToMonthCode(" January ");
200     assertEquals(MonthConstants.JANUARY, m);
201
202     m = SerialDate.stringToMonthCode("Jan");
203     assertEquals(MonthConstants.JANUARY, m);
204
205 }
206
207 /**
208  * Tests the conversion of a month code to a string.
209  */
210 public void testMonthCodeToStringCode() {
211
212     final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213     assertEquals("December", test);
214
215 }
216
217 /**
218  * 1900 is not a leap year.
219  */
220 public void testIsNotLeapYear1900() {
221     assertTrue(!SerialDate.isLeapYear(1900));
222 }
223
224 /**
225  * 2000 is a leap year.
226  */
227 public void testIsLeapYear2000() {
228     assertTrue(SerialDate.isLeapYear(2000));
229 }
```

```

230
231 /**
232  * The number of leap years from 1900 up-to-and-including 1899 is 0.
233  */
234 public void testLeapYearCount1899() {
235     assertEquals(SerialDate.leapYearCount(1899), 0);
236 }
237
238 /**
239  * The number of leap years from 1900 up-to-and-including 1903 is 0.
240  */
241 public void testLeapYearCount1903() {
242     assertEquals(SerialDate.leapYearCount(1903), 0);
243 }
244
245 /**
246  * The number of leap years from 1900 up-to-and-including 1904 is 1.
247  */
248 public void testLeapYearCount1904() {
249     assertEquals(SerialDate.leapYearCount(1904), 1);
250 }
251
252 /**
253  * The number of leap years from 1900 up-to-and-including 1999 is 24.
254  */
255 public void testLeapYearCount1999() {
256     assertEquals(SerialDate.leapYearCount(1999), 24);
257 }
258
259 /**
260  * The number of leap years from 1900 up-to-and-including 2000 is 25.
261  */
262 public void testLeapYearCount2000() {
263     assertEquals(SerialDate.leapYearCount(2000), 25);
264 }
265
266 /**
267  * Serialize an instance, restore it, and check for equality.
268  */
269 public void testSerialization() {
270
271     SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272     SerialDate d2 = null;
273
274     try {
275         ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276         ObjectOutputStream out = new ObjectOutputStream(buffer);
277         out.writeObject(d1);
278         out.close();
279
280         ObjectInput in = new ObjectInputStream(
281             new ByteArrayInputStream(buffer.toByteArray()));
282         d2 = (SerialDate) in.readObject();
283         in.close();
284     } catch (Exception e) {
285         System.out.println(e.toString());
286     }

```



```

287     assertEquals(d1, d2);
288 }
289
290 /**
291  * A test for bug report 1096282 (now fixed).
292  */
293 public void test1096282() {
294     SerialDate d = SerialDate.createInstance(29, 2, 2004);
295     d = SerialDate.addYears(1, d);
296     SerialDate expected = SerialDate.createInstance(28, 2, 2005);
297     assertTrue(d.isOn(expected));
298 }
299
300 /**
301  * Miscellaneous tests for the addMonths() method.
302  */
303 public void testAddMonths() {
304     SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
305
306     SerialDate d2 = SerialDate.addMonths(1, d1);
307     assertEquals(30, d2.getDayOfMonth());
308     assertEquals(6, d2.getMonth());
309     assertEquals(2004, d2.getYYYY());
310
311     SerialDate d3 = SerialDate.addMonths(2, d1);
312     assertEquals(31, d3.getDayOfMonth());
313     assertEquals(7, d3.getMonth());
314     assertEquals(2004, d3.getYYYY());
315
316     SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
317     assertEquals(30, d4.getDayOfMonth());
318     assertEquals(7, d4.getMonth());
319     assertEquals(2004, d4.getYYYY());
320 }
321 }
322 }

```

Listing B.3: MonthConstants.java

```

1  /*
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  *
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info:  http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software

```

```

21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * MonthConstants.java
29 * -----
30 * (C) Copyright 2002, 2003, by Object Refinery Limited.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s):  -;
34 *
35 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Changes
38 * -----
39 * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
40 *
41 */
42
43 package org.jfree.date;
44
45 /**
46 * Useful constants for months. Note that these are NOT equivalent to the
47 * constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
48 * <P>
49 * Used by the SerialDate and RegularTimePeriod classes.
50 *
51 * @author David Gilbert
52 */
53 public interface MonthConstants {
54
55     /** Constant for January. */
56     public static final int JANUARY = 1;
57
58     /** Constant for February. */
59     public static final int FEBRUARY = 2;
60
61     /** Constant for March. */
62     public static final int MARCH = 3;
63
64     /** Constant for April. */
65     public static final int APRIL = 4;
66
67     /** Constant for May. */
68     public static final int MAY = 5;
69
70     /** Constant for June. */
71     public static final int JUNE = 6;
72
73     /** Constant for July. */
74     public static final int JULY = 7;
75
76     /** Constant for August. */
77     public static final int AUGUST = 8;
78

```

```

79  /** Constant for September. */
80  public static final int SEPTEMBER = 9;
81
82  /** Constant for October. */
83  public static final int OCTOBER = 10;
84
85  /** Constant for November. */
86  public static final int NOVEMBER = 11;
87
88  /** Constant for December. */
89  public static final int DECEMBER = 12;
90
91 }

```

Listing B.4: BobsSerialDateTest.java

```

1 package org.jfree.date.junit;
2
3 import junit.framework.TestCase;
4 import org.jfree.date.*;
5 import static org.jfree.date.SerialDate.*;
6
7 import java.util.*;
8
9 public class BobsSerialDateTest extends TestCase {
10
11     public void testIsValidWeekdayCode() throws Exception {
12         for (int day = 1; day <= 7; day++)
13             assertTrue(isValidWeekdayCode(day));
14         assertFalse(isValidWeekdayCode(0));
15         assertFalse(isValidWeekdayCode(8));
16     }
17
18     public void testStringToWeekdayCode() throws Exception {
19
20         assertEquals(-1, stringToWeekdayCode("Hello"));
21         assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22         assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23         //todo assertEquals(MONDAY, stringToWeekdayCode("monday"));
24         // assertEquals(MONDAY, stringToWeekdayCode("MONDAY"));
25         // assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27         assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28         assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29         // assertEquals(TUESDAY, stringToWeekdayCode("tuesday"));
30         // assertEquals(TUESDAY, stringToWeekdayCode("TUESDAY"));
31         // assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32         // assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34         assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35         assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36         // assertEquals(WEDNESDAY, stringToWeekdayCode("wednesday"));
37         // assertEquals(WEDNESDAY, stringToWeekdayCode("WEDNESDAY"));
38         // assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40         assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41         assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42         // assertEquals(THURSDAY, stringToWeekdayCode("thursday"));
43         // assertEquals(THURSDAY, stringToWeekdayCode("THURSDAY"));

```

```

44 // assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 // assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47 assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48 assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 // assertEquals(FRIDAY, stringToWeekdayCode("friday"));
50 // assertEquals(FRIDAY, stringToWeekdayCode("FRIDAY"));
51 // assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53 assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54 assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
55 // assertEquals(SATURDAY, stringToWeekdayCode("saturday"));
56 // assertEquals(SATURDAY, stringToWeekdayCode("SATURDAY"));
57 // assertEquals(SATURDAY, stringToWeekdayCode("sat"));
58
59 assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
60 assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
61 // assertEquals(SUNDAY, stringToWeekdayCode("sunday"));
62 // assertEquals(SUNDAY, stringToWeekdayCode("SUNDAY"));
63 // assertEquals(SUNDAY, stringToWeekdayCode("sun"));
64 }
65
66 public void testWeekdayCodeToString() throws Exception {
67     assertEquals("Sunday", weekdayCodeToString(SUNDAY));
68     assertEquals("Monday", weekdayCodeToString(MONDAY));
69     assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
70     assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
71     assertEquals("Thursday", weekdayCodeToString(THURSDAY));
72     assertEquals("Friday", weekdayCodeToString(FRIDAY));
73     assertEquals("Saturday", weekdayCodeToString(SATURDAY));
74 }
75
76 public void testIsValidMonthCode() throws Exception {
77     for (int i = 1; i <= 12; i++)
78         assertTrue(isValidMonthCode(i));
79     assertFalse(isValidMonthCode(0));
80     assertFalse(isValidMonthCode(13));
81 }
82
83 public void testMonthToQuarter() throws Exception {
84     assertEquals(1, monthCodeToQuarter(JANUARY));
85     assertEquals(1, monthCodeToQuarter(FEBRUARY));
86     assertEquals(1, monthCodeToQuarter(MARCH));
87     assertEquals(2, monthCodeToQuarter(APRIL));
88     assertEquals(2, monthCodeToQuarter(MAY));
89     assertEquals(2, monthCodeToQuarter(JUNE));
90     assertEquals(3, monthCodeToQuarter(JULY));
91     assertEquals(3, monthCodeToQuarter(AUGUST));
92     assertEquals(3, monthCodeToQuarter(SEPTEMBER));
93     assertEquals(4, monthCodeToQuarter(OCTOBER));
94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
95     assertEquals(4, monthCodeToQuarter(DECEMBER));
96
97     try {
98         monthCodeToQuarter(-1);
99         fail("Invalid Month Code should throw exception");
100     } catch (IllegalArgumentException e) {
101     }

```

```
102 }
103
104 public void testMonthCodeToString() throws Exception {
105     assertEquals("January", monthCodeToString(JANUARY));
106     assertEquals("February", monthCodeToString(FEBRUARY));
107     assertEquals("March", monthCodeToString(MARCH));
108     assertEquals("April", monthCodeToString(APRIL));
109     assertEquals("May", monthCodeToString(MAY));
110     assertEquals("June", monthCodeToString(JUNE));
111     assertEquals("July", monthCodeToString(JULY));
112     assertEquals("August", monthCodeToString(AUGUST));
113     assertEquals("September", monthCodeToString(SEPTEMBER));
114     assertEquals("October", monthCodeToString(OCTOBER));
115     assertEquals("November", monthCodeToString(NOVEMBER));
116     assertEquals("December", monthCodeToString(DECEMBER));
117
118     assertEquals("Jan", monthCodeToString(JANUARY, true));
119     assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120     assertEquals("Mar", monthCodeToString(MARCH, true));
121     assertEquals("Apr", monthCodeToString(APRIL, true));
122     assertEquals("May", monthCodeToString(MAY, true));
123     assertEquals("Jun", monthCodeToString(JUNE, true));
124     assertEquals("Jul", monthCodeToString(JULY, true));
125     assertEquals("Aug", monthCodeToString(AUGUST, true));
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
128     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129     assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132         monthCodeToString(-1);
133         fail("Invalid month code should throw exception");
134     } catch (IllegalArgumentException e) {
135     }
136
137 }
138
139 public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY, stringToMonthCode("1"));
141     assertEquals(FEBRUARY, stringToMonthCode("2"));
142     assertEquals(MARCH, stringToMonthCode("3"));
143     assertEquals(APRIL, stringToMonthCode("4"));
144     assertEquals(MAY, stringToMonthCode("5"));
145     assertEquals(JUNE, stringToMonthCode("6"));
146     assertEquals(JULY, stringToMonthCode("7"));
147     assertEquals(AUGUST, stringToMonthCode("8"));
148     assertEquals(SEPTEMBER, stringToMonthCode("9"));
149     assertEquals(OCTOBER, stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER, stringToMonthCode("12"));
152
153     //todo assertEquals(-1, stringToMonthCode("0"));
154     // assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1, stringToMonthCode("Hello"));
157
158     for (int m = 1; m <= 12; m++) {
159         assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160     }
161 }
```

```

160     assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161 }
162
163 // assertEquals(1,stringToMonthCode("jan"));
164 // assertEquals(2,stringToMonthCode("feb"));
165 // assertEquals(3,stringToMonthCode("mar"));
166 // assertEquals(4,stringToMonthCode("apr"));
167 // assertEquals(5,stringToMonthCode("may"));
168 // assertEquals(6,stringToMonthCode("jun"));
169 // assertEquals(7,stringToMonthCode("jul"));
170 // assertEquals(8,stringToMonthCode("aug"));
171 // assertEquals(9,stringToMonthCode("sep"));
172 // assertEquals(10,stringToMonthCode("oct"));
173 // assertEquals(11,stringToMonthCode("nov"));
174 // assertEquals(12,stringToMonthCode("dec"));
175
176 // assertEquals(1,stringToMonthCode("JAN"));
177 // assertEquals(2,stringToMonthCode("FEB"));
178 // assertEquals(3,stringToMonthCode("MAR"));
179 // assertEquals(4,stringToMonthCode("APR"));
180 // assertEquals(5,stringToMonthCode("MAY"));
181 // assertEquals(6,stringToMonthCode("JUN"));
182 // assertEquals(7,stringToMonthCode("JUL"));
183 // assertEquals(8,stringToMonthCode("AUG"));
184 // assertEquals(9,stringToMonthCode("SEP"));
185 // assertEquals(10,stringToMonthCode("OCT"));
186 // assertEquals(11,stringToMonthCode("NOV"));
187 // assertEquals(12,stringToMonthCode("DEC"));
188
189 // assertEquals(1,stringToMonthCode("january"));
190 // assertEquals(2,stringToMonthCode("february"));
191 // assertEquals(3,stringToMonthCode("march"));
192 // assertEquals(4,stringToMonthCode("april"));
193 // assertEquals(5,stringToMonthCode("may"));
194 // assertEquals(6,stringToMonthCode("june"));
195 // assertEquals(7,stringToMonthCode("july"));
196 // assertEquals(8,stringToMonthCode("august"));
197 // assertEquals(9,stringToMonthCode("september"));
198 // assertEquals(10,stringToMonthCode("october"));
199 // assertEquals(11,stringToMonthCode("november"));
200 // assertEquals(12,stringToMonthCode("december"));
201
202 // assertEquals(1,stringToMonthCode("JANUARY"));
203 // assertEquals(2,stringToMonthCode("FEBRUARY"));
204 // assertEquals(3,stringToMonthCode("MAR"));
205 // assertEquals(4,stringToMonthCode("APRIL"));
206 // assertEquals(5,stringToMonthCode("MAY"));
207 // assertEquals(6,stringToMonthCode("JUNE"));
208 // assertEquals(7,stringToMonthCode("JULY"));
209 // assertEquals(8,stringToMonthCode("AUGUST"));
210 // assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 // assertEquals(10,stringToMonthCode("OCTOBER"));
212 // assertEquals(11,stringToMonthCode("NOVEMBER"));
213 // assertEquals(12,stringToMonthCode("DECEMBER"));
214 }
215
216 public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {

```

```
218     assertTrue(isValidWeekInMonthCode(w));
219 }
220 assertFalse(isValidWeekInMonthCode(5));
221 }
222
223 public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236 }
237
238 public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254 }
255
256 public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270 }
271
272 public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
```

```

276 assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277 assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278 }
279
280 private static SpreadsheetDate d(int day, int month, int year) {
281     return new SpreadsheetDate(day, month, year);}
282
283 public void testAddMonths() throws Exception {
284     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
288     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
289     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
290
291     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
292     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
293
294     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
295 }
296
297 public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302 }
303
304 public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY,
309         2005)));
310
311     try {
312         getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
313         fail("Invalid day of week code should throw exception");
314     } catch (IllegalArgumentException e) {
315     }
316 }
317
318 public void testGetFollowingDayOfWeek() throws Exception {
319     // assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER,
320         2004)));
321     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER,
322         2004)));
323     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));
324
325     try {
326         getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
327         fail("Invalid day of week code should throw exception");
328     } catch (IllegalArgumentException e) {
329     }
330 }
331
332 public void testGetNearestDayOfWeek() throws Exception {
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));

```



```

331 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333 assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334 assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335 assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336 assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
339 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342 assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343 assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344 assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 // assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351 assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352 assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 // assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
358 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360 assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 // assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 // assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366 assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367 assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368 assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370 // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 // assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 // assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375 assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376 assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380 // assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383 // assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384 assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386 try {
387     getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388     fail("Invalid day of week code should throw exception");

```

```

389     } catch (IllegalArgumentException e) {
390     }
391 }
392
393 public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408 }
409
410 public void testWeekInMonthToString() throws Exception {
411     assertEquals("First", weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second", weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third", weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth", weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last", weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo try {
418 //    weekInMonthToString(-1);
419 //    fail("Invalid week code should throw exception");
420 // } catch (IllegalArgumentException e) {
421 // }
422 }
423
424 public void testRelativeToString() throws Exception {
425     assertEquals("Preceding", relativeToString(PRECEDING));
426     assertEquals("Nearest", relativeToString(NEAREST));
427     assertEquals("Following", relativeToString(FOLLOWING));
428
429 //todo try {
430 //    relativeToString(-1000);
431 //    fail("Invalid relative code should throw exception");
432 // } catch (IllegalArgumentException e) {
433 // }
434 }
435
436 public void testCreateInstanceFromDDMMYYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1, date.getDayOfMonth());
439     assertEquals(JANUARY, date.getMonth());
440     assertEquals(1900, date.getYYYY());
441     assertEquals(2, date.toSerial());
442 }
443
444 public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900), createInstance(2));
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));

```

```

447 }
448
449 public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),
451                 createInstance(new GregorianCalendar(1900,0,1).getTime()));
451     assertEquals(d(1, JANUARY, 2006),
452                 createInstance(new GregorianCalendar(2006,0,1).getTime()));
452 }
453
454 public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456 }
457 }

```

Listing B.5: SpreadsheetDate.java

```

1 /* =====
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * =====
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info: http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * SpreadsheetDate.java
29 * -----
30 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s):  -;
34 *
35 * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36 *
37 * Changes
38 * -----
39 * 11-Oct-2001 : Version 1 (DG);
40 * 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
41 * 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
42 *               Fixed a bug in calculating day, month and year from serial
43 *               number (DG);

```

```

44 * 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
45 *             month and year. Thanks to Trevor Hills for the report (DG);
46 * 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG);
47 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
48 * 13-Mar-2003 : Implemented Serializable (DG);
49 * 04-Sep-2003 : Completed isInRange() methods (DG);
50 * 05-Sep-2003 : Implemented Comparable (DG);
51 * 21-Oct-2003 : Added hashCode() method (DG);
52 *
53 */
54
55 package org.jfree.date;
56
57 import java.util.Calendar;
58 import java.util.Date;
59
60 /**
61 * Represents a date using an integer, in a similar fashion to the
62 * implementation in Microsoft Excel. The range of dates supported is
63 * 1-Jan-1900 to 31-Dec-9999.
64 * <P>
65 * Be aware that there is a deliberate bug in Excel that recognises the year
66 * 1900 as a leap year when in fact it is not a leap year. You can find more
67 * information on the Microsoft website in article Q181370:
68 * <P>
69 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
70 * <P>
71 * Excel uses the convention that 1-Jan-1900 = 1. This class uses the
72 * convention 1-Jan-1900 = 2.
73 * The result is that the day number in this class will be different to the
74 * Excel figure for January and February 1900...but then Excel adds in an extra
75 * day (29-Feb-1900 which does not actually exist!) and from that point forward
76 * the day numbers will match.
77 *
78 * @author David Gilbert
79 */
80 public class SpreadsheetDate extends SerialDate {
81
82     /** For serialization. */
83     private static final long serialVersionUID = -2039586705374454461L;
84
85     /**
86      * The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
87      * 2958465).
88      */
89     private int serial;
90
91     /** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
92     private int day;
93
94     /** The month of the year (1 to 12). */
95     private int month;
96
97     /** The year (1900 to 9999). */
98     private int year;
99
100    /** An optional description for the date. */
101    private String description;

```

```

102
103  /**
104   * Creates a new date instance.
105   *
106   * @param day the day (in the range 1 to 28/29/30/31).
107   * @param month the month (in the range 1 to 12).
108   * @param year the year (in the range 1900 to 9999).
109   */
110  public SpreadsheetDate(final int day, final int month, final int year) {
111
112      if ((year >= 1900) && (year <= 9999)) {
113          this.year = year;
114      }
115      else {
116          throw new IllegalArgumentException(
117              "The 'year' argument must be in range 1900 to 9999."
118          );
119      }
120
121      if ((month >= MonthConstants.JANUARY)
122          && (month <= MonthConstants.DECEMBER)) {
123          this.month = month;
124      }
125      else {
126          throw new IllegalArgumentException(
127              "The 'month' argument must be in the range 1 to 12."
128          );
129      }
130
131      if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132          this.day = day;
133      }
134      else {
135          throw new IllegalArgumentException("Invalid 'day' argument.");
136      }
137
138      // the serial number needs to be synchronised with the day-month-year...
139      this.serial = calcSerial(day, month, year);
140
141      this.description = null;
142  }
143
144  /**
145   * Standard constructor - creates a new date object representing the
146   * specified day number (which should be in the range 2 to 2958465.
147   *
148   * @param serial the serial number for the day (range: 2 to 2958465).
149   */
150
151  public SpreadsheetDate(final int serial) {
152
153      if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154          this.serial = serial;
155      }
156      else {
157          throw new IllegalArgumentException(
158              "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159      }

```

```

160
161     // the day-month-year needs to be synchronised with the serial number...
162     calcDayMonthYear();
163
164 }
165
166 /**
167  * Returns the description that is attached to the date. It is not
168  * required that a date have a description, but for some applications it
169  * is useful.
170  *
171  * @return The description that is attached to the date.
172  */
173 public String getDescription() {
174     return this.description;
175 }
176
177 /**
178  * Sets the description for the date.
179  *
180  * @param description the description for this date (<code>null</code>
181  * permitted).
182  */
183 public void setDescription(final String description) {
184     this.description = description;
185 }
186
187 /**
188  * Returns the serial number for the date, where 1 January 1900 = 2
189  * (this corresponds, almost, to the numbering system used in Microsoft
190  * Excel for Windows and Lotus 1-2-3).
191  *
192  * @return The serial number of this date.
193  */
194 public int toSerial() {
195     return this.serial;
196 }
197
198 /**
199  * Returns a <code>java.util.Date</code> equivalent to this date.
200  *
201  * @return The date.
202  */
203 public Date toDate() {
204     final Calendar calendar = Calendar.getInstance();
205     calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206     return calendar.getTime();
207 }
208
209 /**
210  * Returns the year (assume a valid range of 1900 to 9999).
211  *
212  * @return The year.
213  */
214 public int getYYYY() {
215     return this.year;
216 }
217

```

```
218  /**
219   * Returns the month (January = 1, February = 2, March = 3).
220   *
221   * @return The month of the year.
222   */
223  public int getMonth() {
224      return this.month;
225  }
226
227  /**
228   * Returns the day of the month.
229   *
230   * @return The day of the month.
231   */
232  public int getDayOfMonth() {
233      return this.day;
234  }
235
236  /**
237   * Returns a code representing the day of the week.
238   * <P>
239   * The codes are defined in the {@link SerialDate} class as:
240   * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241   * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
242   * <code>SATURDAY</code>.
243   *
244   * @return A code representing the day of the week.
245   */
246  public int getDayOfWeek() {
247      return (this.serial + 6) % 7 + 1;
248  }
249
250  /**
251   * Tests the equality of this date with an arbitrary object.
252   * <P>
253   * This method will return true ONLY if the object is an instance of the
254   * {@link SerialDate} base class, and it represents the same day as this
255   * {@link SpreadsheetDate}.
256   *
257   * @param object the object to compare (<code>null</code> permitted).
258   *
259   * @return A boolean.
260   */
261  public boolean equals(final Object object) {
262
263      if (object instanceof SerialDate) {
264          final SerialDate s = (SerialDate) object;
265          return (s.toSerial() == this.toSerial());
266      }
267      else {
268          return false;
269      }
270  }
271
272
273  /**
274   * Returns a hash code for this object instance.
275   *
```

```

276  * @return A hash code.
277  */
278  public int hashCode() {
279      return toSerial();
280  }
281
282  /**
283   * Returns the difference (in days) between this date and the specified
284   * 'other' date.
285   *
286   * @param other the date being compared to.
287   *
288   * @return The difference (in days) between this date and the specified
289   *         'other' date.
290   */
291  public int compare(final SerialDate other) {
292      return this.serial - other.toSerial();
293  }
294
295  /**
296   * Implements the method required by the Comparable interface.
297   *
298   * @param other the other object (usually another SerialDate).
299   *
300   * @return A negative integer, zero, or a positive integer as this object
301   *         is less than, equal to, or greater than the specified object.
302   */
303  public int compareTo(final Object other) {
304      return compare((SerialDate) other);
305  }
306
307  /**
308   * Returns true if this SerialDate represents the same date as the
309   * specified SerialDate.
310   *
311   * @param other the date being compared to.
312   *
313   * @return <code>true</code> if this SerialDate represents the same date as
314   *         the specified SerialDate.
315   */
316  public boolean isOn(final SerialDate other) {
317      return (this.serial == other.toSerial());
318  }
319
320  /**
321   * Returns true if this SerialDate represents an earlier date compared to
322   * the specified SerialDate.
323   *
324   * @param other the date being compared to.
325   *
326   * @return <code>true</code> if this SerialDate represents an earlier date
327   *         compared to the specified SerialDate.
328   */
329  public boolean isBefore(final SerialDate other) {
330      return (this.serial < other.toSerial());
331  }
332
333  /**

```



```
334 * Returns true if this SerialDate represents the same date as the
335 * specified SerialDate.
336 *
337 * @param other the date being compared to.
338 *
339 * @return <code>true</code> if this SerialDate represents the same date
340 * as the specified SerialDate.
341 */
342 public boolean isOnOrBefore(final SerialDate other) {
343     return (this.serial <= other.toSerial());
344 }
345
346 /**
347 * Returns true if this SerialDate represents the same date as the
348 * specified SerialDate.
349 *
350 * @param other the date being compared to.
351 *
352 * @return <code>true</code> if this SerialDate represents the same date
353 * as the specified SerialDate.
354 */
355 public boolean isAfter(final SerialDate other) {
356     return (this.serial > other.toSerial());
357 }
358
359 /**
360 * Returns true if this SerialDate represents the same date as the
361 * specified SerialDate.
362 *
363 * @param other the date being compared to.
364 *
365 * @return <code>true</code> if this SerialDate represents the same date as
366 * the specified SerialDate.
367 */
368 public boolean isOnOrAfter(final SerialDate other) {
369     return (this.serial >= other.toSerial());
370 }
371
372 /**
373 * Returns <code>true</code> if this {@link SerialDate} is within the
374 * specified range (INCLUSIVE). The date order of d1 and d2 is not
375 * important.
376 *
377 * @param d1 a boundary date for the range.
378 * @param d2 the other boundary date for the range.
379 *
380 * @return A boolean.
381 */
382 public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383     return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384 }
385
386 /**
387 * Returns true if this SerialDate is within the specified range (caller
388 * specifies whether or not the end-points are included). The order of d1
389 * and d2 is not important.
390 *
391 * @param d1 one boundary date for the range.
```

```

392  * @param d2 a second boundary date for the range.
393  * @param include a code that controls whether or not the start and end
394  *             dates are included in the range.
395  *
396  * @return <code>true</code> if this SerialDate is within the specified
397  *         range.
398  */
399  public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                          final int include) {
401      final int s1 = d1.toSerial();
402      final int s2 = d2.toSerial();
403      final int start = Math.min(s1, s2);
404      final int end = Math.max(s1, s2);
405
406      final int s = toSerial();
407      if (include == SerialDate.INCLUDE_BOTH) {
408          return (s >= start && s <= end);
409      }
410      else if (include == SerialDate.INCLUDE_FIRST) {
411          return (s >= start && s < end);
412      }
413      else if (include == SerialDate.INCLUDE_SECOND) {
414          return (s > start && s <= end);
415      }
416      else {
417          return (s > start && s < end);
418      }
419  }
420
421  /**
422   * Calculate the serial number from the day, month and year.
423   * <P>
424   * 1-Jan-1900 = 2.
425   *
426   * @param d the day.
427   * @param m the month.
428   * @param y the year.
429   *
430   * @return the serial number from the day, month and year.
431   */
432  private int calcSerial(final int d, final int m, final int y) {
433      final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434      int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435      if (m > MonthConstants.FEBRUARY) {
436          if (SerialDate.isLeapYear(y)) {
437              mm = mm + 1;
438          }
439      }
440      final int dd = d;
441      return yy + mm + dd + 1;
442  }
443
444  /**
445   * Calculate the day, month and year from the serial number.
446   */
447  private void calcDayMonthYear() {
448
449      // get the year from the serial date

```

```
450     final int days = this.serial - SERIAL_LOWER_BOUND;
451     // overestimated because we ignored leap days
452     final int overestimatedYYYY = 1900 + (days / 365);
453     final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454     final int nonleapdays = days - leaps;
455     // underestimated because we overestimated years
456     int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458     if (underestimatedYYYY == overestimatedYYYY) {
459         this.year = underestimatedYYYY;
460     }
461     else {
462         int ssl = calcSerial(1, 1, underestimatedYYYY);
463         while (ssl <= this.serial) {
464             underestimatedYYYY = underestimatedYYYY + 1;
465             ssl = calcSerial(1, 1, underestimatedYYYY);
466         }
467         this.year = underestimatedYYYY - 1;
468     }
469
470     final int ss2 = calcSerial(1, 1, this.year);
471
472     int[] daysToEndOfPrecedingMonth
473         = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475     if (isLeapYear(this.year)) {
476         daysToEndOfPrecedingMonth
477             = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478     }
479
480     // get the month from the serial date
481     int mm = 1;
482     int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483     while (sss < this.serial) {
484         mm = mm + 1;
485         sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486     }
487     this.month = mm - 1;
488
489     // what's left is d(+1);
490     this.day = this.serial - ss2
491         - daysToEndOfPrecedingMonth[this.month] + 1;
492
493 }
494
495 }
```

Listing B.6: RelativeDayOfWeekRule.java

```
1 /* =====
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * =====
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  * Project Info:  http://www.jfree.org/jcommon/index.html
8  *
9  * This library is free software; you can redistribute it and/or modify it
10 * under the terms of the GNU Lesser General Public License as published by
```

```

11 * the Free Software Foundation; either version 2.1 of the License, or
12 * (at your option) any later version.
13 *
14 * This library is distributed in the hope that it will be useful, but
15 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17 * License for more details.
18 *
19 * You should have received a copy of the GNU Lesser General Public
20 * License along with this library; if not, write to the Free Software
21 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301,
22 * USA.
23 *
24 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25 * in the United States and other countries.]
26 *
27 * -----
28 * RelativeDayOfWeekRule.java
29 * -----
30 * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
31 *
32 * Original Author: David Gilbert (for Object Refinery Limited);
33 * Contributor(s): -;
34 *
35 * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36 *
37 * Changes (from 26-Oct-2001)
38 * -----
39 * 26-Oct-2001 : Changed package to com.jrefinery.date.*;
40 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
41 *
42 */
43
44 package org.jfree.date;
45
46 /**
47 * An annual date rule that returns a date for each year based on (a) a
48 * reference rule; (b) a day of the week; and (c) a selection parameter
49 * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
50 * <P>
51 * For example, Good Friday can be specified as 'the Friday PRECEDING Easter
52 * Sunday'.
53 *
54 * @author David Gilbert
55 */
56 public class RelativeDayOfWeekRule extends AnnualDateRule {
57     /** A reference to the annual date rule on which this rule is based. */
58     private AnnualDateRule subrule;
59
60     /**
61      * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).
62      */
63     private int dayOfWeek;
64
65     /** Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). */
66     private int relative;
67
68

```

```
69  /**
70   * Default constructor - builds a rule for the Monday following 1 January.
71   */
72   public RelativeDayOfWeekRule() {
73       this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
74   }
75
76   /**
77   * Standard constructor - builds rule based on the supplied sub-rule.
78   *
79   * @param subrule the rule that determines the reference date.
80   * @param dayOfWeek the day-of-the-week relative to the reference date.
81   * @param relative indicates *which* day-of-the-week (preceding, nearest
82   *                   or following).
83   */
84   public RelativeDayOfWeekRule(final AnnualDateRule subrule,
85                               final int dayOfWeek, final int relative) {
86       this.subrule = subrule;
87       this.dayOfWeek = dayOfWeek;
88       this.relative = relative;
89   }
90
91   /**
92   * Returns the sub-rule (also called the reference rule).
93   *
94   * @return The annual date rule that determines the reference date for this
95   *         rule.
96   */
97   public AnnualDateRule getSubrule() {
98       return this.subrule;
99   }
100
101   /**
102   * Sets the sub-rule.
103   *
104   * @param subrule the annual date rule that determines the reference date
105   *               for this rule.
106   */
107   public void setSubrule(final AnnualDateRule subrule) {
108       this.subrule = subrule;
109   }
110
111   /**
112   * Returns the day-of-the-week for this rule.
113   *
114   * @return the day-of-the-week for this rule.
115   */
116   public int getDayOfWeek() {
117       return this.dayOfWeek;
118   }
119
120   /**
121   * Sets the day-of-the-week for this rule.
122   *
123   * @param dayOfWeek the day-of-the-week (SerialDate.MONDAY,
124   *               SerialDate.TUESDAY, and so on).
125   */
126   public void setDayOfWeek(final int dayOfWeek) {
127       this.dayOfWeek = dayOfWeek;
```

```

128     }
129
130     /**
131     * Returns the 'relative' attribute, that determines -which*
132     * day-of-the-week we are interested in (SerialDate.PRECEDING,
133     * SerialDate.NEAREST or SerialDate.FOLLOWING).
134     *
135     * @return The 'relative' attribute.
136     */
137     public int getRelative() {
138         return this.relative;
139     }
140
141     /**
142     * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
143     * SerialDate.FOLLOWING).
144     *
145     * @param relative determines *which* day-of-the-week is selected by this
146     *         rule.
147     */
148     public void setRelative(final int relative) {
149         this.relative = relative;
150     }
151
152     /**
153     * Creates a clone of this rule.
154     *
155     * @return a clone of this rule.
156     *
157     * @throws CloneNotSupportedException this should never happen.
158     */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167     * Returns the date generated by this rule, for the specified year.
168     *
169     * @param year the year (1900 &lt;= year &lt;= 9999).
170     *
171     * @return The date generated by the rule for the given year (possibly
172     *         <code>null</code>).
173     */
174     public SerialDate getDate(final int year) {
175
176         // check argument...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // calculate the date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186

```

```
187     if (base != null) {
188         switch (this.relative) {
189             case(SerialDate.PRECEDING):
190                 result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                     base);
192                 break;
193             case(SerialDate.NEAREST):
194                 result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                     base);
196                 break;
197             case(SerialDate.FOLLOWING):
198                 result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                     base);
200                 break;
201             default:
202                 break;
203         }
204     }
205     return result;
206
207 }
208
209 }
```

Listing B.7: DayDate.java (Final)

```
1  /* =====
2  * JCommon : a free general purpose class library for the Java(tm) platform
3  * =====
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 */
17 package org.jfree.date;
18
19 import java.io.Serializable;
20 import java.util.*;
21
22 /**
23  * An abstract class that represents immutable dates with a precision of
24  * one day. The implementation will map each date to an integer that
25  * represents an ordinal number of days from some fixed origin.
26  *
27  * Why not just use java.util.Date? We will, when it makes sense. At times,
28  * java.util.Date can be *too* precise - it represents an instant in time,
29  * accurate to 1/1000th of a second (with the date itself depending on the
30  * time-zone). Sometimes we just want to represent a particular day (e.g. 21
31  * January 2015) without concerning ourselves about the time of day, or the
32  * time-zone, or anything else. That's what we've defined DayDate for.
33  *
34  * Use DayDateFactory.makeDate to create an instance.
35  *
36  * @author David Gilbert
37  * @author Robert C. Martin did a lot of refactoring.
38  */
39
40 public abstract class DayDate implements Comparable, Serializable {
41     public abstract int getOrdinalDay();
42     public abstract int getYear();
43     public abstract Month getMonth();
44 }
```

```

64 public abstract int getDayOfMonth();
65
66 protected abstract Day getDayOfWeekForOrdinalZero();
67
68 public DayDate plusDays(int days) {
69     return DayDateFactory.makeDate(getOrdinalDay() + days);
70 }
71
72 public DayDate plusMonths(int months) {
73     int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74     int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75     int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
76     int resultYear = resultMonthAndYearAsOrdinal / 12;
77     int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
78     Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79     int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
80     return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81 }
82
83 public DayDate plusYears(int years) {
84     int resultYear = getYear() + years;
85     int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
86     return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87 }
88
89 private int correctLastDayOfMonth(int day, Month month, int year) {
90     int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
91     if (day > lastDayOfMonth)
92         day = lastDayOfMonth;
93     return day;
94 }
95
96 public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
97     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
98     if (offsetToTarget >= 0)
99         offsetToTarget -= 7;
100     return plusDays(offsetToTarget);
101 }
102
103 public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105     if (offsetToTarget <= 0)
106         offsetToTarget += 7;
107     return plusDays(offsetToTarget);
108 }
109
110 public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111     int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112     int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113     int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115     if (offsetToFutureTarget > 3)
116         return plusDays(offsetToPreviousTarget);
117     else
118         return plusDays(offsetToFutureTarget);
119 }
120
121 public DayDate getEndOfMonth() {
122     Month month = getMonth();

```



```
123     int year = getYear();
124     int lastDay = DateUtil.lastDayOfMonth(month, year);
125     return DayDateFactory.makeDate(lastDay, month, year);
126 }
127
128 public Date toDate() {
129     final Calendar calendar = Calendar.getInstance();
130     int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131     calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132     return calendar.getTime();
133 }
134
135 public String toString() {
136     return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137 }
138
139 public Day getDayOfWeek() {
140     Day startingDay = getDayOfWeekForOrdinalZero();
141     int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142     int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143     return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144 }
145
146 public int daysSince(DayDate date) {
147     return getOrdinalDay() - date.getOrdinalDay();
148 }
149
150 public boolean isOn(DayDate other) {
151     return getOrdinalDay() == other.getOrdinalDay();
152 }
153
154 public boolean isBefore(DayDate other) {
155     return getOrdinalDay() < other.getOrdinalDay();
156 }
157
158 public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
160 }
161
162 public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164 }
165
166 public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168 }
169
170 public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172 }
173
174 public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178 }
179 }
```

Listing B.8: Month.java (Final)

```

1 package org.jfree.date;
2
3 import java.text.DateFormatSymbols;
4
5 public enum Month {
6     JANUARY(1), FEBRUARY(2), MARCH(3),
7     APRIL(4), MAY(5), JUNE(6),
8     JULY(7), AUGUST(8), SEPTEMBER(9),
9     OCTOBER(10), NOVEMBER(11), DECEMBER(12);
10    private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11    private static final int[] LAST_DAY_OF_MONTH =
12        {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14    private int index;
15
16    Month(int index) {
17        this.index = index;
18    }
19
20    public static Month fromInt(int monthIndex) {
21        for (Month m : Month.values()) {
22            if (m.index == monthIndex)
23                return m;
24        }
25        throw new IllegalArgumentException("Invalid month index " + monthIndex);
26    }
27
28    public int lastDay() {
29        return LAST_DAY_OF_MONTH[index];
30    }
31
32    public int quarter() {
33        return 1 + (index - 1) / 3;
34    }
35
36    public String toString() {
37        return dateFormatSymbols.getMonths()[index - 1];
38    }
39
40    public String toShortString() {
41        return dateFormatSymbols.getShortMonths()[index - 1];
42    }
43
44    public static Month parse(String s) {
45        s = s.trim();
46        for (Month m : Month.values())
47            if (m.matches(s))
48                return m;
49
50        try {
51            return fromInt(Integer.parseInt(s));
52        }
53        catch (NumberFormatException e) {}
54        throw new IllegalArgumentException("Invalid month " + s);
55    }
56
57    private boolean matches(String s) {

```

```
58     return s.equalsIgnoreCase(toString()) ||
59         s.equalsIgnoreCase(toShortString());
60 }
61
62 public int toInt() {
63     return index;
64 }
65 }
```

Listing B.9: Day.java (Final)

```
1 package org.jfree.date;
2
3 import java.util.Calendar;
4 import java.text.DateFormatSymbols;
5
6 public enum Day {
7     MONDAY(Calendar.MONDAY),
8     TUESDAY(Calendar.TUESDAY),
9     WEDNESDAY(Calendar.WEDNESDAY),
10    THURSDAY(Calendar.THURSDAY),
11    FRIDAY(Calendar.FRIDAY),
12    SATURDAY(Calendar.SATURDAY),
13    SUNDAY(Calendar.SUNDAY);
14
15    private final int index;
16    private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18    Day(int day) {
19        index = day;
20    }
21
22    public static Day fromInt(int index) throws IllegalArgumentException {
23        for (Day d : Day.values())
24            if (d.index == index)
25                return d;
26        throw new IllegalArgumentException(
27            String.format("Illegal day index: %d.", index));
28    }
29
30    public static Day parse(String s) throws IllegalArgumentException {
31        String[] shortWeekdayNames =
32            dateSymbols.getShortWeekdays();
33        String[] weekdayNames =
34            dateSymbols.getWeekdays();
35
36        s = s.trim();
37        for (Day day : Day.values()) {
38            if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39                s.equalsIgnoreCase(weekdayNames[day.index])) {
40                return day;
41            }
42        }
43        throw new IllegalArgumentException(
44            String.format("%s is not a valid weekday string", s));
45    }
46
47    public String toString() {
48        return dateSymbols.getWeekdays()[index];
49    }
50 }
```

```

49 }
50
51 public int toInt() {
52     return index;
53 }
54 }

```

Listing B.10: DateInterval.java (Final)

```

1 package org.jfree.date;
2
3 public enum DateInterval {
4     OPEN {
5         public boolean isIn(int d, int left, int right) {
6             return d > left && d < right;
7         }
8     },
9     CLOSED_LEFT {
10        public boolean isIn(int d, int left, int right) {
11            return d >= left && d < right;
12        }
13    },
14    CLOSED_RIGHT {
15        public boolean isIn(int d, int left, int right) {
16            return d > left && d <= right;
17        }
18    },
19    CLOSED {
20        public boolean isIn(int d, int left, int right) {
21            return d >= left && d <= right;
22        }
23    };
24
25    public abstract boolean isIn(int d, int left, int right);
26 }

```

Listing B.11: WeekInMonth.java (Final)

```

1 package org.jfree.date;
2
3 public enum WeekInMonth {
4     FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
5     private final int index;
6
7     WeekInMonth(int index) {
8         this.index = index;
9     }
10
11    public int toInt() {
12        return index;
13    }
14 }

```

Listing B.12: WeekdayRange.java (Final)

```

1 package org.jfree.date;
2
3 public enum WeekdayRange {
4     LAST, NEAREST, NEXT
5 }

```

Listing B.13: DateUtil.java (Final)

```

1 package org.jfree.date;
2
3 import java.text.DateFormatSymbols;
4
5 public class DateUtil {
6     private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
7
8     public static String[] getMonthNames() {
9         return dateFormatSymbols.getMonths();
10    }
11
12    public static boolean isLeapYear(int year) {
13        boolean fourth = year % 4 == 0;
14        boolean hundredth = year % 100 == 0;
15        boolean fourHundredth = year % 400 == 0;
16        return fourth && (!hundredth || fourHundredth);
17    }
18
19    public static int lastDayOfMonth(Month month, int year) {
20        if (month == Month.FEBRUARY && isLeapYear(year))
21            return month.lastDay() + 1;
22        else
23            return month.lastDay();
24    }
25
26    public static int leapYearCount(int year) {
27        int leap4 = (year - 1896) / 4;
28        int leap100 = (year - 1800) / 100;
29        int leap400 = (year - 1600) / 400;
30        return leap4 - leap100 + leap400;
31    }
32 }

```

Listing B.14: DayDateFactory.java (Final)

```

1 package org.jfree.date;
2
3 public abstract class DayDateFactory {
4     private static DayDateFactory factory = new SpreadsheetDateFactory();
5     public static void setInstance(DayDateFactory factory) {
6         DayDateFactory.factory = factory;
7     }
8
9     protected abstract DayDate _makeDate(int ordinal);
10    protected abstract DayDate _makeDate(int day, Month month, int year);
11    protected abstract DayDate _makeDate(int day, int month, int year);
12    protected abstract DayDate _makeDate(java.util.Date date);
13    protected abstract int _getMinimumYear();
14    protected abstract int _getMaximumYear();
15
16    public static DayDate makeDate(int ordinal) {
17        return factory._makeDate(ordinal);
18    }
19
20    public static DayDate makeDate(int day, Month month, int year) {
21        return factory._makeDate(day, month, year);
22    }

```

```

23
24 public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26 }
27
28 public static DayDate makeDate(java.util.Date date) {
29     return factory._makeDate(date);
30 }
31
32 public static int getMinimumYear() {
33     return factory._getMinimumYear();
34 }
35
36 public static int getMaximumYear() {
37     return factory._getMaximumYear();
38 }
39 }

```

Listing B.15: SpreadsheetDateFactory.java (Final)

```

1 package org.jfree.date;
2
3 import java.util.*;
4
5 public class SpreadsheetDateFactory extends DayDateFactory {
6     public DayDate _makeDate(int ordinal) {
7         return new SpreadsheetDate(ordinal);
8     }
9
10    public DayDate _makeDate(int day, Month month, int year) {
11        return new SpreadsheetDate(day, month, year);
12    }
13
14    public DayDate _makeDate(int day, int month, int year) {
15        return new SpreadsheetDate(day, month, year);
16    }
17
18    public DayDate _makeDate(Date date) {
19        final GregorianCalendar calendar = new GregorianCalendar();
20        calendar.setTime(date);
21        return new SpreadsheetDate(
22            calendar.get(Calendar.DATE),
23            Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24            calendar.get(Calendar.YEAR));
25    }
26
27    protected int _getMinimumYear() {
28        return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29    }
30
31    protected int _getMaximumYear() {
32        return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33    }
34 }

```

Listing B.16: SpreadsheetDate.java (Final)

```

1 /* =====
2  * JCommon : a free general purpose class library for the Java(tm) platform

```

```

3  * =====
4  *
5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
6  *
7  *
8  *
9  *
10 *
11 *
12 *
13 *
14 *
15 *
16 *
17 *
18 *
19 *
20 *
21 *
22 *
23 *
24 *
25 *
26 *
27 *
28 *
29 *
30 *
31 *
32 *
33 *
34 *
35 *
36 *
37 *
38 *
39 *
40 *
41 *
42 *
43 *
44 *
45 *
46 *
47 *
48 *
49 *
50 *
51 *
52 *
53 */
54
55 package org.jfree.date;
56
57 import static org.jfree.date.Month.FEBRUARY;
58
59 import java.util.*;
60
61 /**
62  * Represents a date using an integer, in a similar fashion to the
63  * implementation in Microsoft Excel. The range of dates supported is
64  * 1-Jan-1900 to 31-Dec-9999.
65  * <p/>
66  * Be aware that there is a deliberate bug in Excel that recognises the year
67  * 1900 as a leap year when in fact it is not a leap year. You can find more
68  * information on the Microsoft website in article Q181370:
69  * <p/>
70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
71  * <p/>
72  * Excel uses the convention that 1-Jan-1900 = 1. This class uses the
73  * convention 1-Jan-1900 = 2.
74  * The result is that the day number in this class will be different to the
75  * Excel figure for January and February 1900...but then Excel adds in an extra
76  * day (29-Feb-1900 which does not actually exist!) and from that point forward
77  * the day numbers will match.
78  *
79  * @author David Gilbert
80  */
81 public class SpreadsheetDate extends DayDate {
82     public static final int EARLIEST_DATE_ORDINAL = 2; // 1/1/1900
83     public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
84     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
85     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
86     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
87         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
88     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
89         {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
90
91     private int ordinalDay;
92     private int day;
93     private Month month;
94     private int year;
95
96     public SpreadsheetDate(int day, Month month, int year) {
97         if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
98             throw new IllegalArgumentException(
99                 "The 'year' argument must be in range " +
100                 MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101         if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102             throw new IllegalArgumentException("Invalid 'day' argument.");
103     }

```

```

104     this.year = year;
105     this.month = month;
106     this.day = day;
107     ordinalDay = calcOrdinal(day, month, year);
108 }
109
110 public SpreadsheetDate(int day, int month, int year) {
111     this(day, Month.fromInt(month), year);
112 }
113
114 public SpreadsheetDate(int serial) {
115     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116         throw new IllegalArgumentException(
117             "SpreadsheetDate: Serial must be in range 2 to 2958465.");
118
119     ordinalDay = serial;
120     calcDayMonthYear();
121 }
122
123 public int getOrdinalDay() {
124     return ordinalDay;
125 }
126
127 public int getYear() {
128     return year;
129 }
130
131 public Month getMonth() {
132     return month;
133 }
134
135 public int getDayOfMonth() {
136     return day;
137 }
138
139 protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141 public boolean equals(Object object) {
142     if (!(object instanceof DayDate))
143         return false;
144
145     DayDate date = (DayDate) object;
146     return date.getOrdinalDay() == getOrdinalDay();
147 }
148
149 public int hashCode() {
150     return getOrdinalDay();
151 }
152
153 public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155 }
156
157 private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];

```



```

161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162         daysUpToMonth++;
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165 }
166
167 private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177 }
178
179 private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183         aMonth++;
184
185     return Month.fromInt(aMonth - 1);
186 }
187
188 private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190         return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192         return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193 }
194
195 private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198         aYear++;
199
200     return aYear - 1;
201 }
202
203 private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
205 }
206
207 public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                calendar.get(Calendar.YEAR));
213 }
214 }
215 }

```

Literaturverweise

[Alexander] – *A Timeless Way of Building*, Christopher Alexander, Oxford University Press, New York, 1979.

[AOSD] – Aspect-Oriented Software Development, <http://aosd.net>

[ASM] – ASM Home Page, <http://asm.objectweb.org/>

[AspectJ] – <http://www.eclipse.org/aspectj>

[Beck07] – *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.

[Beck94] – *JUnit Pocket Guide*, Kent Beck, O'Reilly, 1994.

[Beck97] – *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

[BeckTDD] – *Test Driven Development*, Kent Beck, Addison-Wesley, 2003.

[CGLIB] – Code Generation Library, <http://cglib.sourceforge.net/>

[Colyer] – *Eclipse AspectJ*, Adrian Colyer, Andy Clement, George Hurley, Mathew Webster, Pearson Education, Inc., Upper Saddle River, NJ, 2005.

[DDD] – *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

[DSL] – Domain-specific programming language, http://en.wikipedia.org/wiki/Domain-specific_programming_language; http://de.wikipedia.org/wiki/Domänenspezifische_Sprache

[Fowler] – Inversion of Control Containers and the Dependency Injection Pattern, <http://martinfowler.com/articles/injection.html>

[Goetz] – *Java Theory and Practice: Decorating with Dynamic Proxies*, Brian Goetz, <http://www.ibm.com/developerworks/java/Library/j-jtp08305.html>

[GOF] – *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[Javassist] – Javassist Home Page, <http://www.csg.is.titech.ac.jp/~chiba/javassist/>

[JBoss] – JBoss Home Page, <http://jboss.org>

[JMock] – JMock – A Lightweight Mock Object Library for Java, <http://jmock.org>

- [Knuth92] – *Literate Programming*, Donald E. Knuth, Center for the Study of Language and Information, Leland Stanford Junior University, 1992.
- [Kolence] – *Software physics and computer performance measurements, Proceedings of the ACM annual conference – Volume 2*, Kenneth W. Kolence, Boston, Massachusetts, pp. 1024–1040, 1972.
- [KP78] – *The Elements of Programming Style*, 2d. ed., by Kernighan und Plaugher, McGraw-Hill, 1978.
- [Lea99] – *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.
- [Martin] – *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- [Martino7] – *Professionalism and Test-Driven Development*, Robert C. Martin, Object Mentor, IEEE-Software, Mai/Juni 2007 (Vol. 24, No. 3), S. 32–36.
- [Mezzaroso7] – *XUnit Patterns*, Gerard Mezzaros, Addison-Wesley, 2007.
- [PPP] – *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.
- [PRAG] – *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.
- [RDD] – *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.
- [Refactoring] – *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.
- [RSpec] – *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesøy, David Chelimsky, Pragmatic Bookshelf, 2008.
- [Simmonso4] – *Hardcore Java*, Robert Simmons, Jr., O'Reilly, 2004.
- [SP72] – *Structured Programming*, O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, Academic Press, London, 1972.
- [Spring] – *The Spring Framework*, <http://www.springframework.org>
- [WELC] – *Working Effectively with Legacy Code*, Feathers, Michael, Addison-Wesley, 2004.
- [XPE] – *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison-Wesley, 1999.

Epilog

Auf der *Agile Conference*, 2005 in Denver, übergab mir Elisabeth Hedrickson (<http://www.qualitytree.com/>) ein grünes Armband, ähnlich der Art, die von Lance Armstrong gemacht worden war. Es trug die Aufschrift *Test Obsessed* (*Testbesessen*). Ich nahm es dankbar an und trug es stolz. Seit ich 1999 die Test-Driven Development von Kent Beck gelernt hatte, war ich zu einem überzeugten Anhänger der TDD geworden.

Aber dann passierte etwas Seltsames: Ich stellte fest, dass ich das Band nicht mehr ablegen konnte. Nicht, weil es physisch festsaß, sondern weil es *moralisch* an mir hing. Das Armband drückte meine professionelle Ethik aus. Es war ein sichtbares Zeichen meiner Entschlossenheit, den besten mir möglichen Code zu schreiben. Es abzunehmen, schien mir ein Verrat an dieser Ethik und an dieser Entschlossenheit zu sein.

Deshalb trage ich es noch immer am Handgelenk. Wenn ich Code schreibe, sehe ich es immer in meinem peripheren Blickfeld. Es erinnert mich ständig an das Versprechen, das ich mir selbst gegeben habe: sauberen Code zu schreiben.

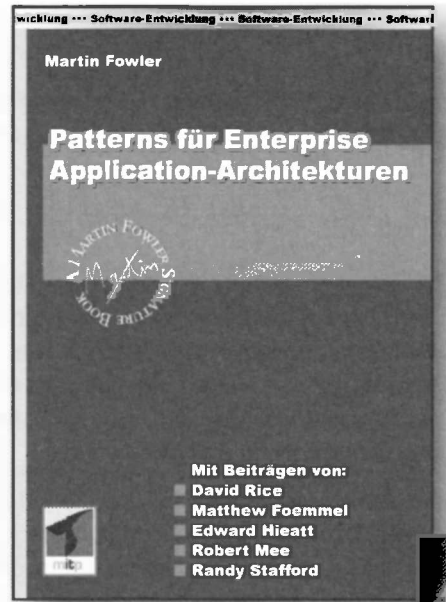


Martin Fowler

Patterns für Enterprise Application-Architekturen

Mit Beiträgen von:

- David Rice
- Matthew Foemmel
- Edward Hieatt
- Robert Mee
- Randy Stafford



Durch neue Technologien wurde die Erstellung von Unternehmensanwendungen revolutioniert. Plattformen wie Java oder .NET haben sich dabei etabliert. Mit Hilfe dieser neuen Technologien sind Sie zwar in der Lage, mächtige Applikationen zu entwickeln, diese sind aber häufig nur sehr schwierig zu implementieren. Viele Enterprise-Applikationen sind fehlerhaft, weil der dahinter stehenden Architektur nicht genügend Aufmerksamkeit gewidmet wurde.

Patterns für Enterprise Application-Architekturen ist exakt aus diesem Grund geschrieben worden. Der Herausgeber Martin Fowler, eine Koryphäe des objektorientierten Designs, stellte fest, dass trotz des rasanten Wandels in der IT-Technologie – von Smalltalk zu CORBA zu Java zu .NET – dieselben fundamentalen Design-Prinzipien angewandt werden können. Mit der Unterstützung vieler erfahrener Experten arbeitete Fowler mehr als 40 wiederver-

wendbare Lösungen in Patterns ein, die plattformunabhängig zur Erstellung von Unternehmensanwendungen genutzt werden können.

Dieses beeindruckende Handbuch beinhaltet eigentlich zwei Bücher:

Der erste Teil bietet dem Leser ein Tutorial zur Entwicklung von Unternehmensanwendungen. Im zweiten Teil finden Sie eine umfassende Referenz der behandelten Patterns. Zu jedem Pattern gibt es exakte Informationen bezüglich des Einsatzes und der Implementierung, sowie detaillierten Code in Java und C#. Zahlreiche erläuternde UML-Diagramme veranschaulichen den jeweiligen Lösungsweg. Dieses Buch vermittelt Ihnen das Wissen, welches Sie brauchen, um die passenden architektonischen Grundentscheidungen für eine Enterprise-Applikation zu treffen und das richtige, bewährte Pattern einzusetzen.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/1378

ISBN 978-3-8266-1378-4

Stichwortverzeichnis

Cross-Referenz der Heuristiken

C1 320, 323, 337
C2 323, 327, 336, 338
C3 326, 327, 330, 338
C4 338
C5 339
E1 339
E2 339
F1 278, 340
F2 340
F3 340
F4 318, 329, 340
G1 320, 340
G10 326, 345
G11 309, 326, 329, 332, 345
G12 327, 329, 330, 336, 346
G13 328, 329, 346
G14 329, 330, 333, 346
G15 330, 347
G16 330, 331, 348
G17 330, 349
G18 331, 332, 350
G19 331, 332, 333, 350
G2 318, 319, 341
G20 332, 351
G21 332, 351
G22 334, 352
G23 278, 335, 353
G24 336, 354
G25 336, 354
G26 355
G27 356
G28 307, 356
G29 308, 356
G3 319, 341
G30 309, 356
G31 310, 357
G32 310, 358
G33 312, 359
G34 359
G35 361

G36 362
G4 322, 341
G5 322, 328, 332, 336, 342
G6 323, 326, 330, 334, 335, 336, 342
G7 324, 344
G8 326, 344
G9 313, 326, 327, 329, 345
J1 320, 362
J2 321, 363
J3 326, 327, 364
N1 310, 321, 323, 325, 329, 330, 331, 335, 336, 366
N2 321, 367
N3 327, 330, 368
N4 308, 332, 368
N5 258, 369
N6 307, 369
N7 308, 370
T1 319, 370
T1, 16 318
T2 318, 370
T3 318, 370
T4 371
T5 319, 371
T6 319, 371
T7 319, 371
T8 320, 371
T9 371

A

Abhängigkeit
 logische vs. physische 352
Abhängigkeitsmagnet 79
 Error.java 79
Abschnitt
 kritischer 221
Abstand
 vertikaler 114
Abstract Factory 54, 69
Abstract Factory-Pattern 194, 324
Abstraktion 130

- Abstraktionsebene 26, 66, 192, 342, 359
 - Funktion 67
 - Namen 367
- Abteilung 192
- Acceptance-Test 256
- Active Record 137
- Adapter 226, 386
- Ad-hoc-Code 159
- Affinität
 - konzeptionelle 118
- Agile Conference 465
- Agilität 196
 - Praktiken 208
- Algorithmus 351
- Änderung
 - einplanen 185
 - isolieren 188
- Annotation 205
- Anordnung
 - vertikale 119
- Anti-Symmetrie 131
- Anwendungsinfrastruktur 202
- AOP siehe Aspect-oriented Programming
- Architektur 15
 - inkrementelle 196
 - naiv einfache 206
 - Software-Entwicklung 16
 - System 196
- Args 235
- Argument
 - Anzahl 71
 - Flag-Argumente 72, 340
 - hängendes 347
 - in Funktionen 340
 - Input-Argumente 71, 72
 - Listen 74
 - Objekte 74
 - Output-Argumente 71, 72, 76, 340
 - Reihenfolge 73
 - Selektor-Argumente 347
 - Whitespace 120
- ASM 200, 231
- AspectJ 205
- Aspect-Oriented Framework 231
- Aspect-oriented Programming 199
- Aspekt 199, 205
- Aspektorientierte Programmierung siehe Aspect-oriented Programming
- assertEquals 73
- AtomicBoolean 384
- AtomicInteger 384
- AtomicReference 384
- Aufgabe
 - eine 65
 - mehr als eine 193
- Aufruf-Stack 381
- Aufwärtsskalierung 196
- Ausdrucksstärke 214
- Ausfallsicherheit 202
- Ausnahme 139
 - checked 143
 - Klassifikation 144
 - mit Kontext auslösen 144
 - oder Fehler-Code? 77
 - unchecked 143
- Ausrichtung
 - horizontale 121
- Ausschluss
 - gegenseitiger 395
- Autor
 - Programmierer 41
- B**
- Basisklasse 344
- BDUF siehe Big Design Up Front
- Bean 136, 197, 203
- Beck, Kent
 - 27, 37, 64, 105, 210, 301, 342, 350, 465
- Bedingung
 - Kapselung 356
 - negative 356
- Befehlszeilenargument 235
- Belang siehe Concern
- Belang siehe Responsibility
- Benutzer-Story 206
- Big Design Up Front 206
- Booch, Grady 34
- Boundary-Interface 153
- Boundary-Test 157
- Bowling game 369
- Brooks, Fred 17
- Bucket Brigade 358
- Build 339
- Build-Operate-Check(Pattern) 165
- Byte-Code-Manipulation 200

C

Caching 202
 Call stack 381
 Callable 383
 CGLIB 200, 231
 COBOL 354
 Codd-Normalform 342
 Code 26
 Ad-hoc-Code 159
 als Erklärungsmedium 87
 als Spezifikation 26
 Ausdrucksstärke 214
 auskommentierter 102, 339
 Drittanbieter-Code 151
 duplizierter 61, 211
 Duplizierung 38
 Gründe für Verrotten 30
 Instrumentierung 230
 Lesbarkeit 34, 42
 literate 36
 Merkmale von sauberem 32
 minimal 35
 noch nicht existierender 157
 prozeduraler 133
 Redesign 29
 Rohentwurf 249
 sauberer 22
 schlechter 27
 schmutziger 243
 Shutdown-Code 227
 shy Code 362
 Threaded-Code 229
 toter 345
 und Design 19
 und Sprache 26
 vor Konsequenzen warnen 90
 Zusammenhang mit Kommentar 104
 Zweck erklären 88
 Code Smells 337
 Code-Coverage 336
 Code-Gefühl 31
 Codieren 22
 Collection
 thread-sichere 223
 Concern 192
 Cross-Cutting 199, 202
 Fehler-Handling 150
 Trennung 192
 Constantine, Larry 19
 ConTest 232, 401
 Convention over Configuration 203

Coplien, James O. 217
 Coverage-Analyse 304
 Coverage-Tool 370
 Cross-Cutting Concern 199, 202
 Cruft 17
 Cunningham, Ward 38, 70

D

DAO siehe Data Access Object
 Data Access Object 203
 Data Transfer Object 136, 199
 Daten
 Anti-Symmetrie 131
 gemeinsame Nutzung 222
 relationale 197
 Datenabstraktion 129
 Datenstruktur
 Unterschied zu Objekt 131
 Datentransfer-Objekt 136
 Datentransferobjekt siehe Data Transfer Object
 Datenzugriffs-Objekt 203
 DBMS 199
 Deadlock 224, 393, 396
 Decorator 368
 Denkschule 40
 Dependency Injection 195, 202, 210
 Dependency-Inversion-Prinzip 190
 Dependency-Management 195
 Deployment-Deskriptor 198
 Design 210
 und Code 19
 Design Pattern 38
 Details
 in der Software-Entwicklung 17
 DI siehe Dependency Injection
 Dichotomie
 fundamentale 133
 Dichte
 vertikale 113
 Dijkstra, Edsger 80
 Dining-Philosophers-Problem 225
 DIP 209
 DIP siehe Dependency-Inversion-Prinzip
 Domain-Specific Language 208
 Don't-Repeat-Yourself-Prinzip 80
 Drittanbieter-Code 151
 DRY siehe Don't-Repeat-Yourself-Prinzip
 DRY-Prinzip 342, 388
 DSL siehe Domain-Specific Language
 DTO 136

DTO siehe Data Transfer Object
 Dummy-Bereich 124
 Duplizierung 38, 211, 342
 Durchsatz
 berechnen 392
 Dyade 71

E

Eclipse 35, 37, 56
 Eimerkette 358
 Einhüllen 145
 Einkapselung 174
 Einmal, und nur einmal 342
 Einrückung 123
 durchbrechen 124
 EJB1 197
 EJB2 197, 198, 203, 205
 EJB3 203
 EJB-Container 218
 Emergenz 209
 Entity Bean 197
 Entscheidung
 aufschieben 207
 Entwicklung
 inkrementelle 256
 schrittweise 258
 Error.java
 Abhängigkeitsmagnet 79
 Erweiterbarkeit 110
 Erzeuger-Verbraucher-Problem 224
 Evans, Eric 368
 Event 72
 Exception siehe Ausnahme
 Exception-Klasse 144
 Executor Framework 383
 Explaining Temporary Variables 331
 Extreme Programming 37, 38, 342

F

F.I.R.S.T. 171
 Factory 194
 Faulheit 356
 Feathers, Michael 36, 139
 Feature Envy 135
 Fehler-Code 79
 oder Ausnahme? 77
 Fehler-Handling 32, 139
 Concern 150
 null zurückgeben 147
 Trennung von Geschäftslogik 146
 Fehler-Verarbeitung 79

FIT 38, 359
 FitNesse
 61, 63, 256, 270, 350, 358, 359, 361, 368
 Kommentar 88, 92
 FitNesseExpediter
 lange Variablenliste 122
 Flag-Argument 72, 340
 Formatierung 109
 Einrückung 123
 horizontale 119
 horizontale Ausrichtung 121
 Regeln 125
 Regeln im Team 125
 vertikale 110
 vertikale Anordnung 119
 vertikale Dichte 113
 vertikale Offenheit 112
 vertikaler Abstand 114
 Zeitungs-Metapher 111
 Zweck 109
 FORTRAN 354
 Fowler, Martin 337, 346
 Frame 381
 Funktion 61, 340
 abhängige 116
 Abschnitte 66
 Abstraktionsebene 67, 359
 Anweisung und Abfrage trennen 77
 Anzahl der Argumente 340
 Anzahl der Aufgaben 356
 Argumente 71
 Argument-Listen 74
 Aufgabenumfang 65
 Blöcke und Einrückungen 65
 drei Argumente 73
 dyadische 73
 ein Argument 72
 Flag-Argumente 72
 Größe 64
 Namen 70
 Objekte als Argument 74
 Output-Argumente 71, 76
 private 345
 tote 340
 zwei Argumente 73
 Funktions-Header
 Kommentar 104
 Funktionsname 351
 Funktionsneid 135, 346
 Future 383

G

Gabriel, Richard 20
 Gamma, Eric 301
 Gefühl für den Code 31
 Geltungsbereich
 Hierarchie 123
 vertikaler 345
 Geschäftslogik 198
 Trennung von Fehler-Handling 146
 Gesetz von Demeter 133, 362
 Getter 129
 Gilbert, David 317
 given-when-then-Konvention 169
 Global 75
 goto 80
 Gott-Klasse 174
 Grenning, James 151
 Grenzbedingung 341, 359
 testen 371
 Grenze 151
 Griffige Abstraktion 34

H

Hierarchie
 Geltungsbereiche 123
 HN siehe Notation, ungarische
 HTML 360
 HTML-Kommentar 102
 Hungarian Notation siehe Notation, ungarische
 Hunt, Andy 33, 342
 Hybrid 137

I

IBM 401
 Implementation Patterns 27
 Implementierung
 verbergen 130
 Implizität 46
 Importliste 362
 Injektion der Abhängigkeit siehe Dependency Injection
 Inkonsistenz 345
 Instanzvariable 115, 354
 Instrumentierung
 von Code 230
 IntelliJ 56
 Interface
 Boundary-Interface 153
 breites 344
 Intuition 341

Inversion of Control 195
 IoC siehe Inversion of Control
 Iterativ und inkrementell 196

J

jar-Datei 344
 Java-AOP-Framework 202
 Javadoc 92
 Kommentar 105
 Java-Proxy 200
 Javassist 200
 JBoss 202
 JBoss AOP 202, 205
 JCommon 317
 JDBC 203
 Jeffries, Ron 37, 342
 Jiggling 232
 JNDI 195
 JUnit 114, 256
 JUnit-Framework 301
 Just-in-Time-Compiler 220
 Just-in-Time-Entscheidung 207
 JVM 197

K

Kapselung
 Bedingung 356
 Kartesische Koordinaten 73
 Kernighan, Brian 85
 Klasse 173
 abgeleitete 344
 Abhängigkeit 344
 Basisklasse 344
 Größe 174
 Namen 55
 nicht thread-sichere 385
 Klassenaufbau 173
 Klassenname 354
 Knuth, Donald 179
 Kohäsion 73, 178, 210
 Kommentar 85, 337
 auskommentierter Code 102
 FitNesse 88, 92
 Funktions-Header 104
 Geschwätz 97, 99
 Grund für Kommentarer 86
 guter 87
 hinter schließenden Klammern 101
 HTML-Kommentare 102
 informierender 88
 irreführender 95

- Javadoc 105
- juristischer 87
- Nebenbemerkung 101
- nicht-lokale Informationen 103
- Positionsbezeichner 100
- redundanter 93, 338
- schlecht geschriebener 338
- schlechter 92
- Tagebuch-Kommentare 96
- TODO-Kommentare 91
- überholter 338
- und schlechter Code 86
- ungeeigneter 337
- ungenauer 86
- vorgeschriebener 96
- Zusammenhang mit Code 104
- Zweck des Codes 88
- Konfigurationswert 361
- Können 21
- Könnerschaft 21
- Konstante
 - Vererbung 363
 - Verwaltung auf Stufen 118
 - vs. Enums 364
- Kontext
 - globaler 193
- Konvention 356
 - Namen 354, 368
- Kopplung 210, 344
 - künstliche 346
 - zeitliche 75, 76, 310, 357

L

- Law of Demeter 133, 362
- Lazy Initialization 192
- Lea, Doug 223, 402
- Lean
 - Prinzipien 16
- Learning Tests 154
- Least-Surprise-Prinzip 349
- LeBlanc's Gesetz 28
- Lern-Test 154, 156
- Lesbarkeit 34, 42, 110
- Leser-Schreiber-Problem 225
- Literate Programming 36, 179
- Livelock 224, 397
- LocalHome 198
- Locking
 - clientbasiertes 226
 - serverbasiertes 226
- LoD 133, 362

- log4j 154
- Logger 154

M

- Map 152
 - clear 152
- Mehrdeutigkeit 356
 - Test 371
- Mentales Mapping 54
- Methode
 - falsche Deklaration 350
 - Namen 55
- Methoden
 - synchronisierte 226
- Methodenname 354
- Mies van der Rohe, Ludwig 15
- Minimaler Code 35
- Missverständnis 356
- Mock Object 193
- Modul 344
- Monade 71, 72
- Monte-Carlo-Test 400
- MQ-Verbindung 394
- Müll 346
- mutual exclusion 395
- Mutual-Exclusion 224

N

- Name 366
 - aussprechbarer 50
- Auswahl 366
- Codierungen 52
- eindeutiger 368
- Fehlinformationen 47
- Funktion 70, 351
- Humor vermeiden 55
- Implementierungen 54
- Interfaces 54
- Klasse 354
- Klassen 55
- Kontext 57, 60
- Konvention 354
- Länge 369
- Leerwörter 49
- Lösungsdomäne 57
- Member-Präfixe 53
- Methode 354
- Methoden 55
- mit Zahlenserien 49
- Nebeneffekte 370

- Problemdomäne 57
- Schlüsselwort 75
- suchbarer 51
- Unterschiede verdeutlichen 49
- Variable 354
- Wortspiele 56
- zweckbeschreibender 45
- Name-Mangling 370
- Navigation
 - transitive 362
- Nebeneffekt 75
- Nebenläufigkeit 218, 373
 - EJB-Container 218
 - Mythen 219
 - Probleme 220
- Newkirk, Jim 154
- Nilade 71
- Nomenklatur 368
- Normalform 342
- Notation
 - ungarische 52, 348, 369
- Null 147, 148
- Nutzung
 - gemeinsame Datennutzung 222

O

- Objekt
 - Anti-Symmetrie 131
 - Unterschied zu Datenstruktur 131
- Objekt Orientierung 38
- OCP siehe Open-Closed-Prinzip
- Offenheit
 - vertikale 112
- One-off 233
- One-Switch-Regel 353
- OO-Design 177, 188
- Open-Closed-Prinzip 69, 188
 - Checked Exceptions 143
- Operanden-Stack 381
- Operation
 - atomare 380
- OTI 35
- Ottinger, Tim 45
- Output-Argument 71, 76, 340
- Ozzie, Ray 191

P

- Persistenz 202
- Persistenzstrategie 199
- Pfadfinder-Regel 43, 307

- Philosophenproblem 225
- PI 349
- PL/I 354
- Platzhalter 362
- Plauger, P. J. 85
- POJO 201, 202, 205
- Polyade 71
- Polymorphismus 353
- Pragmatic Programmer 362
- Prinzip der geringsten Überraschung 349
- Producer-Consumer-Problem 224
- Produktionsumgebung
 - vs. Test-Umgebung 168
- Produktivität 28
- Professionalität 300
- Programmierer
 - als Autor 41
- Programmierung
 - Begriff 26
 - strukturierte 80
- Python 242

Q

- Quelldatei
 - Sprache 340
- Quick und dirty 161

R

- Readers-Writers-Problem 225
- Redesign 29
- Refactoring 196, 211, 256, 281
- Reihenfolge
 - Argumente 73
- Responsibility 174, 176
- Rohentwurf 243
- Ruby 242
- Rückgabe-Code 139
- Runnable 383
- Run-on-Ausdruck 348

S

- Sauberer Code 22
 - Wie schreiben? 31
- Scheren-Regel 115
- Schlechter Code 27
 - Kosten 28
- Schlüsselwort 75
- Schuchert, Brett L. 373
- Scissors-Regel 115
- Scrum 20

- Seiketsu 17
- Seiri 16
- Seiso 17
- Seiton 16
- Selektor-Argument 347
- SerialDate
 - Autor 317
 - Refactoring 318
- Server-Anwendung 373
- Servlet 222
- Setter 129, 195
- Setup-Idiom 193
- Shutdown-Code 227
- Shutsuke 17
- Shy Code 362
- Sicherheit 202
- Sicherung
 - übergangene 341
- Simple Design
 - Regeln 210
- Single-Responsibility-Prinzip
 - 68, 176, 186, 188, 193, 209, 221, 300, 377
- Smalltalk 38, 242
- Smells 337
- Software physics 206
- Software-Entwicklung
 - Architektur 16
 - Details 17
- Softwarephysik 206
- Software-Projekt
 - Hauptkosten 214
- Sonderfall 341
- Sparkle 64
- Special Case (Objekt) 147
- Special Case (Pattern) 147
- 5S-Philosophie 16
- Sprache
 - domänenspezifische 81
 - in Quelldatei 340
- Spring 195, 202, 203
- Spring AOP 202, 205
- SRP siehe Single-Responsibility-Prinzip
- Stack-Trace 144
- Standard 207
- Startup-Prozess 192
- Starvation 224, 397
- Stepdown-Regel 67
- Strategy (Pattern) 342
- Stroustrup, Bjarne 32

- Struktur 356
 - hybride 135
- Strukturierte Programmierung 80
- Switch-Anweisung 68
- Synchronisierte Methoden 226
- synchronized 221
- System 191
 - Anwendung 192
 - Konstruktion 192
- Systemarchitektur 196
 - testen 205

T

- Tabellenzeile 197
- TANSTAAFL 398
- TDD siehe Test Driven Development
- Team
 - Produktivität 28
- Team-Regel 125
- Template Method (Pattern) 169, 342
- Template Method-Pattern 213
- Test 370
 - Anzahl der assert-Anweisungen 168
 - Anzahl der Konzepte 170
 - assert-Anweisung 169
 - Doppelstandard 166
 - Einfachheit 163
 - F.I.R.S.T. 171
 - Geschwindigkeit 171
 - Grenzbedingung 371
 - Klarheit 163
 - Lesbarkeit 163
 - sauber halten 161
 - sauberer 163
 - selbst-validierender 171
 - trivialer 370
 - Unabhängigkeit 171
 - unzureichender 370
 - Wiederholbarkeit 171
 - zeitgerechter 171
- Test Driven Development
 - 35, 142, 159, 196, 256
 - drei Gesetze 160
- Testcode
 - Bedeutung 162
- Testfall
 - als Dokumentation 302
- Testgesteuerte Entwicklung siehe Test Driven Development
- TestNG 115, 370

Testsprache 166
 domänenspezifische 166
 Test-Umgebung
 vs. Produktionsumgebung 168
 Thomas, Dave 342
 Thomas, Dave (Big) 33, 35
 Thread
 Unabhängigkeit 222
 Threaded-Code 229
 Threading 218
 Durchsatz berechnen 392
 Thread-Management 377
 Tiger-Team 29
 TO-Absatz 66
 Tomcat 94
 Total Productive Maintenance 16
 TPM 16
 Train Wreck 134
 Transaktion 202
 Trennung
 Concerns 192, 196, 199, 206
 Geschäftslogik und Fehler-Handling
 146
 Verantwortlichkeiten 192
 vertikale 345
 Zuständigkeiten 192, 299
 Triade 71, 73
 Try/Catch-Block 78
 try-catch-finally-Anweisung 141

U

Ubiquitous language 368
 Umkehrung der Kontrolle siehe Inversion of
 Control
 UM-ZU-Absatz 66, 67
 Uncle Bob 40, 379
 Ungenauigkeit 356
 Unit-Test 141, 159, 256, 339
 automatisierter 160
 Unordnung 346

V

Variable
 Aussagekraft 350
 Instanzvariable 115
 Kontrollvariable für Schleifen 115
 lokale 114, 345, 381
 protected 114
 Variablendeklaration 114
 Variablenname 354
 Verantwortlichkeit 176
 Verantwortlichkeit siehe Concern
 Verb 75
 Verfeinerung
 schrittweise 235
 Verhalten
 offensichtliches 341
 Verhungern 397
 Vorausplanung 196

W

Wading-through-Code 28
 Wartbarkeit 110
 Whitespace
 Argumente 120
 horizontaler 120
 Wiki 38, 256
 Wirfs-Brock, Rebecca 464
 Wort
 leeres 49
 Wrapper 145

X

XHTML 360
 XP Immersion 105

Z

Zahl
 magische 348, 354
 Zeitungs-Metapher 111
 Zerlegung 299
 Zugkatastrophe 134
 Zusicherung 149
 Zuständigkeit
 falsche 349
 Zuständigkeit siehe Concern
 Zuständigkeit siehe Responsibility

Sebastian Kübeck

Software-Sanierung

Weiterentwicklung, Testen und Refactoring bestehender Software

- Weiterentwickeln bestehender Systeme ohne vorhandene Tests
- Bestehenden Code mit Tests absichern
- Maßnahmen zur Verbesserung der Qualität



Es erscheint möglicherweise seltsam, den Begriff »Sanierung« in Zusammenhang mit Software zu verwenden. Es gibt in der Softwareentwicklung jedoch ein Phänomen, das dem physischen Verschleiß in seiner Auswirkung nahekommt: Mit zunehmendem Alter wird es immer schwieriger und teurer, Software an geänderte Gegebenheiten anzupassen. Früher oder später führt kein Weg an einer Sanierung Ihres Bestandssystems vorbei – sofern keine ausreichende automatisierte Testabdeckung dafür vorhanden ist.

Mit den in diesem Buch beschriebenen Techniken können Sie die geforderte Testabdeckung nachziehen und den Code so weit verbessern, dass Sie diese ständigen Veränderungen auch längerfristig durchhalten können. Die beschriebenen Methoden haben nachweislich in vielen Projekten Verbesserungen in Bezug auf die Qualität und Produktivität der Entwicklung gebracht. Sie sind mit etwas Übung leicht zu erlernen, so dass sie auch von weniger erfahrenen Entwicklern angewendet werden können.

Dieses Buch ist vor allem für Praktiker geschrieben, so dass der Autor besonderes Augenmerk auf aussagekräftige Beispiele legt. Die Beispiele sind in

Java implementiert, so dass Programmierkenntnisse in Java vorausgesetzt werden.

Testen Sie, ob Ihre Software sanierungsbedürftig ist:

1. Treten häufig Fehler auf und ziehen Maßnahmen zur Fehlerbehebung öfter Folgefehler nach sich?
2. Verbringen Programmierer viel Zeit mit der Fehlersuche?
3. Ist der Quellcode für Programmierer schwer verständlich?
4. Sind Änderungen umständlich umzusetzen und treten dabei häufig Fehler auf?
5. Haben Sie keine automatisierten Tests oder eine geringe Testabdeckung?
6. Setzen Sie selten oder nie Refactoring ein, um die Software an neue Gegebenheiten anzupassen?

Wenn Sie mindestens eine dieser Fragen mit »Ja« beantworten müssen, ist Ihre Software wahrscheinlich sanierungsbedürftig. Dann sollten Sie unbedingt mit den im Buch beschriebenen Techniken die Qualität Ihrer Software verbessern, um langfristig produktiver arbeiten zu können.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/5072

ISBN 978-3-8266-5072-7

Chad Fowler

The PASSIONATE PROGRAMMER Der leidenschaftliche Programmierer

Wie Programmierer ihre berufliche Laufbahn erfolgreich gestalten

Dieses Buch ist ein inspirierender Wegweiser für Programmierer und Softwareentwickler. Chad Fowler zeigt Ihnen, wie Sie sich nachhaltig persönlich weiterentwickeln können, um Ihre eigene berufliche Laufbahn erfolgreich zu gestalten und Schritt für Schritt Ihre eigenen Ziele zu verfolgen und zu realisieren.

Wenn die Softwareentwicklung für Sie eine Leidenschaft ist, dann werden Sie mit diesem Buch lernen, wie es Ihnen gelingt, Ihre Fähigkeiten bestmöglich zu entfalten und in Ihrer beruflichen Laufbahn Anerkennung und Erfolg zu erlangen.

Chad Fowler berichtet aus eigener Erfahrung, worauf es im Beruf ankommt, und macht Ihnen deutlich, dass Sie Ihre berufliche Entwicklung nicht dem Zufall überlassen, sondern selbst in die Hand nehmen sollten. Dies erfordert Nachdenken, Handeln und die Bereitschaft, einen eingeschlagenen Weg zu ändern.

Wählen Sie den Markt und die Technologien, mit denen Sie sich beschäftigen, gezielt und bewusst aus. Investieren Sie in Ihre eigenen Fähigkeiten. Lernen Sie, wie Sie Ihre Fähigkeiten wie ein Produkt behandeln und vermarkten müssen, um damit erfolgreich zu sein. Sie werden erfahren, wie Sie Ihre Situation selber positiv beeinflussen und verbessern können.

Chad Fowler gibt Ihnen praktische Anleitungen und für jeden umsetzbare Methoden an die Hand und



erläutert Ihnen die notwendigen Schritte, die für Sie wichtig sind, um die eigenen Wünsche und Fähigkeiten zu erkennen, weiterzuentwickeln und diese auch gut verkaufen zu können.

Beispielhafte Laufbahnen erfolgreicher Softwareentwickler zeigen Ihnen, wie es andere geschafft haben.

Mit diesem Buch kann jeder seine persönliche Entwicklung ganz individuell gestalten. Es wird Ihr Leben, Ihre Einstellungen und Ihre Motivation positiv verändern! Und Sie werden Erfolg damit haben!

„Das Großartige an diesem Buch ist, dass es zahlreiche Handlungsanweisungen enthält – Dinge, die ich tun kann. Es macht deutlich, dass die Verantwortung für meine Situation dort liegt, wo sie hingehört – bei mir. Dieses Buch arbeitet heraus, was ich heute tun kann. Und morgen. Und im Rest meiner beruflichen Laufbahn.“

Kent Beck, Programmierer

„Knapp sechs Monate, bevor ich dieses Buch las, stand ich kurz davor, meinen Beruf zu wechseln. Mehrere Zufälle brachten mich dazu, nicht nur bei der Softwareentwicklung zu bleiben, sondern aus ihr eine Leidenschaft zu machen, die ich wirklich meistern wollte. Dabei diente mir dieses Buch mit seiner gesunden Prise Inspiration als Wegweiser zu meinen Zielen.“

Sammy Labri, Chief Spaghetti Coder

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/5885

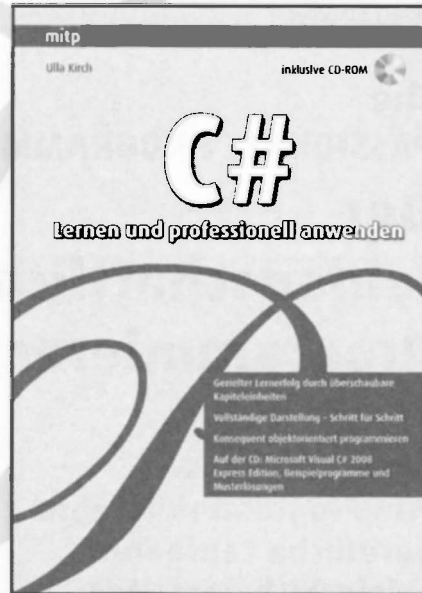
ISBN 978-3-8266-5885-3

Ulla Kirch

C#

Lernen und professionell anwenden

- Gezielter Lernerfolg durch überschaubare Kapiteleinheiten
- Vollständige Darstellung – Schritt für Schritt
- Konsequenter objektorientierter Programmieren
- Auf der CD: Microsoft Visual C# 2008 Express Edition, Beispielprogramme und Musterlösungen



Sie möchten die Programmiersprache C# als erste Programmiersprache erlernen oder als erfahrener Programmierer auf C# umsteigen? Dann ist dieses Buch genau richtig für Sie!

Sie lernen die elementaren Sprachkonzepte von C# und werden schrittweise bis zur Entwicklung professioneller C#-Programme geführt. Die Sprachbeschreibung basiert auf der ECMA/ISO-Spezifikation. Damit ist C# prinzipiell plattformunabhängig.

Beginnend mit den Grundlagen wie einfache Typen, Klassen und Arrays beschreibt die Autorin anschließend typische objektorientierte Themen wie Vererbung, Polymorphie und Exception Handling. Für den professionellen Einsatz sind in den hinteren Kapiteln auch Spracherweiterungen wie Nullable-Typen, anonyme Methoden und Generics detailliert

erklärt. Zahlreiche Anwendungsbeispiele veranschaulichen die unterschiedlichen Verwendungsmöglichkeiten.

Jede Doppelseite im Buch stellt eine Lerneinheit dar. Auf der rechten Seite sind die Sprachelemente erläutert, die auf der linken Seite durch C#-Programme und grafische Darstellungen illustriert werden. Die Beispielprogramme zeigen eine typische Anwendung für das jeweilige Sprachelement.

Jedes Kapitel bietet Ihnen die Gelegenheit, mit Übungen und Musterlösungen Ihre Kenntnisse direkt zu überprüfen und zu vertiefen. Damit Sie Ihre Programme unmittelbar testen können, ist auf der beiliegenden CD die Microsoft Visual C# 2008 Express Edition beigelegt. Die Beispielprogramme und Musterlösungen befinden sich ebenfalls auf der Buch-CD.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/5915

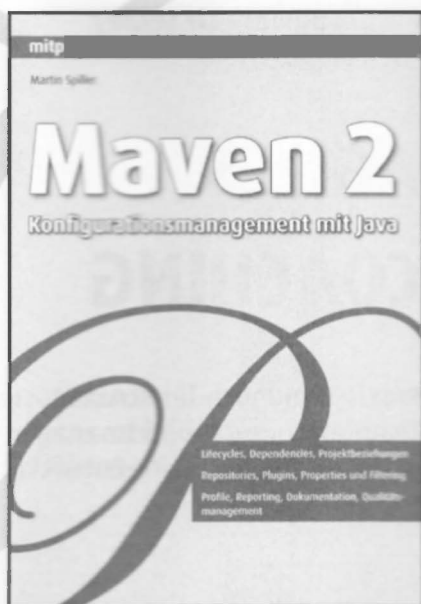
ISBN 978-3-8266-5915-7

Martin Spiller

Maven 2

Konfigurationsmanagement mit Java

- Lifecycles, Dependencies, Projektbeziehungen
- Repositories, Plugins, Properties und Filtering
- Profile, Reporting, Dokumentation, Qualitätsmanagement



Maven ist ein Build- und Konfigurationsmanagement-Tool der Apache Software Foundation und basiert auf Java. Mit Maven lassen sich Java-Projekte standardisiert erstellen und verwalten. Ziel hierbei ist die Automatisierung und Vereinfachung immer wieder anfallender Aufgaben.

Der Autor erläutert die grundlegenden Konzepte und Module von Maven und zeigt Ihnen, wie diese im Projektalltag eingesetzt werden können. Das Buch richtet sich an Softwareentwickler und -architekten, an technische Projektleiter und alle, die sich mit Konfigurationsmanagement beschäftigen.

Zunächst gibt Ihnen der Autor einen Schnelleinstieg in Maven und erläutert die elementaren Befehle und Konfigurationsschritte, so dass Sie sofort erste Projekte mit Maven erstellen und bearbeiten können. Die folgenden sechs Kapitel vermitteln Ihnen die grundlegenden Konzepte und Prinzipien z.B.

zu Verzeichnis- und Namenskonventionen, Lifecycles, Dependencies, Projektbeziehungen und zum Projektmodell. Alle weiteren Kapitel behandeln einzelne Themen, die im Verlaufe eines Projektes eine Rolle spielen können wie u.a. Repositories, Plugins, das Veröffentlichen von Software, Reporting und Dokumentation sowie Qualitätsmanagement. So erhalten Sie einen umfassenden Einblick in Maven.

Dieses Buch eignet sich sowohl als Einführung als auch als Referenz und Arbeitsbuch für die tägliche Praxis.

Über den Autor:

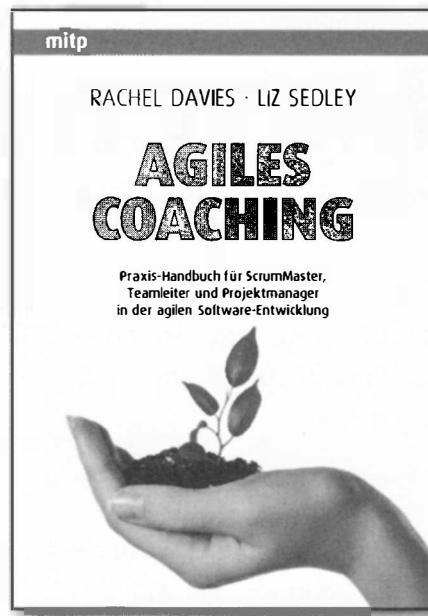
Martin Spiller ist Diplom-Mathematiker und arbeitet als Softwareentwickler und Berater im Java-Umfeld für die NEUSTAR-GmbH. Seine Schwerpunkte sind Softwarequalität, Konfigurationsmanagement, Performance-Tuning und Vorgehensmodelle.

Probekapitel und Infos erhalten Sie unter:
www.mitp.de/5937

ISBN 978-3-8266-5937-9

AGILES COACHING

**Praxis-Handbuch für ScrumMaster,
Teamleiter und Projektmanager
in der agilen Software-Entwicklung**



Das Coaching agiler Teams ist eine spannende Aufgabe und erfordert Soft Skills, die sich nicht auf der Basis eines Regelwerkes erlernen lassen. Vielmehr zählt hierbei insbesondere die Erfahrung im Umgang mit Teams. Mit diesem Buch können Sie von der Erfahrung zweier langjähriger Coaches lernen und auf diese Weise von Anfang an vieles richtig machen und viele Stolperstellen von vornherein vermeiden.

Rachel und Liz zeigen Ihnen anhand zahlreicher Beispiele, wie Sie als Coach agile Teams bestmöglich bei der Arbeit unterstützen können. In diesem Buch lernen Sie, wie Sie ein agiles Team aufbauen und bei der Arbeit begleiten können, so dass alle Teammitglieder Spaß daran haben, gute Software zu entwickeln. Die Autoren führen Sie durch den gesamten agilen Lebenszyklus und zeigen Ihnen, wie man Teams dazu befähigt, das Beste aus den agilen Methoden herauszuholen.

Sie werden erfahren, was bei der Zusammenarbeit mit Teammitgliedern gut funktioniert und was Sie vermeiden sollten. Anstatt feste Regeln vorzugeben, konzentrieren sich die Autoren auf praktische Ratschläge, Tipps und Erfahrungsberichte, die Ihnen helfen, in eigenen Situationen angemessen zu handeln. In jedem Kapitel listen sie typische Hürden auf mit Lösungsvorschlägen, die Ihnen verdeutlichen, wie Sie Schwierigkeiten meistern können. Sie finden zusätzlich hilfreiche Checklisten, die Ihnen Anregungen geben, worauf Sie achten sollten.

Mit diesem Buch sind Sie für Ihre Tätigkeit als Coach optimal vorbereitet.

Aus dem Inhalt:

- Lernen Sie, guten Code von schlechtem zu unterscheiden
- Sauberen Code schreiben und schlechten Code in guten umwandeln
- Aussagekräftige Namen sowie gute Funktionen, Objekte und Klassen erstellen
- Code so formatieren, strukturieren und kommentieren, dass er bestmöglich lesbar ist
- Ein vollständiges Fehler-Handling implementieren, ohne die Logik des Codes zu verschleiern
- Unit-Tests schreiben und Ihren Code testgesteuert entwickeln

Über den Autor:

Robert C. »Uncle Bob« Martin entwickelt seit 1970 professionell Software. Seit 1990 arbeitet er international als Software-Berater. Er ist Gründer und Vorsitzender von Object Mentor, Inc., einem Team erfahrener Berater, die Kunden auf der ganzen Welt bei der Programmierung in und mit C++, Java, C#, Ruby, OO, Design Patterns, UML sowie Agilen Methoden und eXtreme Programming helfen.

Selbst schlechter Code kann funktionieren. Aber wenn der Code nicht sauber ist, kann er ein Entwicklungsunternehmen in die Knie zwingen. Jedes Jahr gehen unzählige Stunden und beträchtliche Ressourcen verloren, weil Code schlecht geschrieben ist. Aber das muss nicht sein.

Mit *Clean Code* präsentiert Ihnen der bekannte Software-Experte Robert C. Martin ein revolutionäres Paradigma, mit dem er Ihnen aufzeigt, wie Sie guten Code schreiben und schlechten Code überarbeiten. Zusammen mit seinen Kollegen von Object Mentor destilliert er die besten Praktiken der agilen Entwicklung von sauberem Code zu einem einzigartigen Buch. So können Sie sich die Erfahrungswerte der Meister der Software-Entwicklung aneignen, die aus Ihnen einen besseren Programmierer machen werden – anhand konkreter Fallstudien, die im Buch detailliert durchgearbeitet werden.

Sie werden in diesem Buch sehr viel Code lesen. Und Sie werden aufgefordert, darüber nachzudenken, was an diesem Code richtig und falsch ist. Noch wichtiger: Sie werden herausgefordert, Ihre professionellen Werte und Ihre Einstellung zu Ihrem Beruf zu überprüfen.

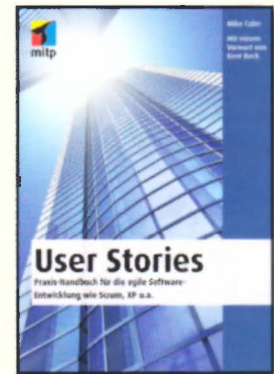
Clean Code besteht aus drei Teilen: Der erste Teil beschreibt die Prinzipien, Patterns und Techniken, die zum Schreiben von sauberem Code benötigt werden. Der zweite Teil besteht aus mehreren, zunehmend komplexeren Fallstudien. An jeder Fallstudie wird aufgezeigt, wie Code gesäubert wird – wie eine mit Problemen behaftete Code-Basis in eine solide und effiziente Form umgewandelt wird. Der dritte Teil enthält den Ertrag und den Lohn der praktischen Arbeit: ein umfangreiches Kapitel mit Best Practices, Heuristiken und Code Smells, die bei der Erstellung der Fallstudien zusammengetragen wurden. Das Ergebnis ist eine Wissensbasis, die beschreibt, wie wir denken, wenn wir Code schreiben, lesen und säubern.

Dieses Buch ist ein Muss für alle Entwickler, Software-Ingenieure, Projektmanager, Team-Leiter oder Systemanalytiker, die daran interessiert sind, besseren Code zu produzieren.

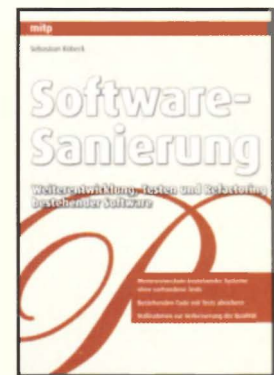
Außerdem zum Thema:



ISBN 978-3-8266-5885-3



ISBN 978-3-8266-5898-3



ISBN 978-3-8266-5072-7

Probekapitel und Infos
erhalten Sie unter:
www.mitp.de

Regalsystematik:
Programmierung, Software-Entwicklung

(D) € 39,95

ISBN 978-3-8266-5548-7

