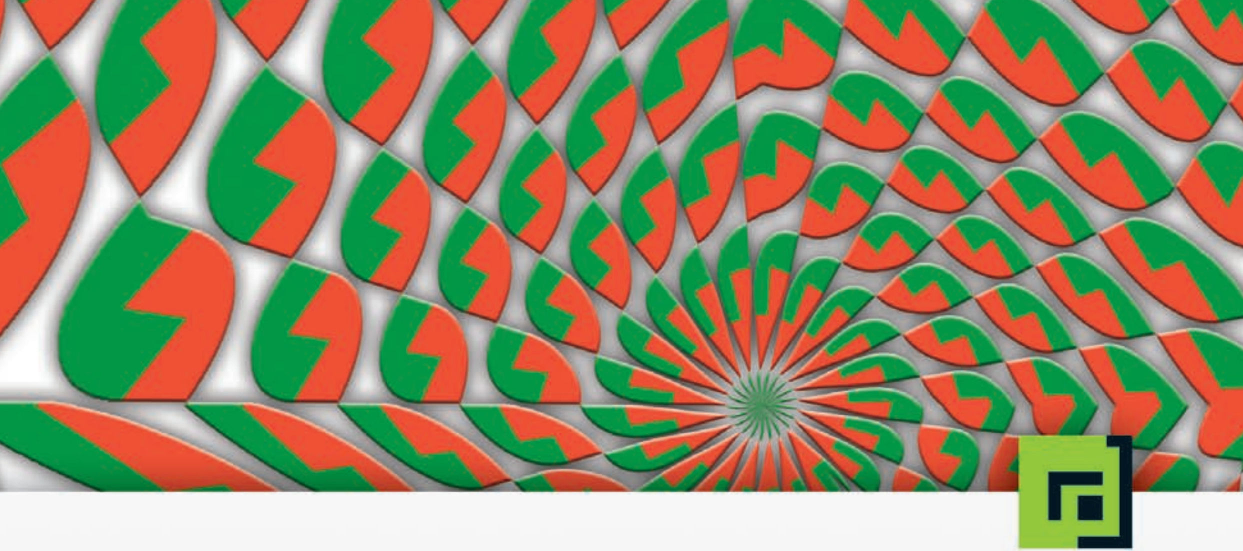


Frank Westphal

Testgetriebene Entwicklung mit JUnit & FIT

Wie Software änderbar bleibt

→ Mit einem Geleitwort von Johannes Link

dpunkt.verlag

Was sind dpunkt.ebooks?

Die dpunkt.ebooks sind Publikationen im PDF-Format, die es Ihnen erlauben, Inhalte am Bildschirm zu lesen, gezielt nach Informationen darin zu suchen und Seiten daraus auszudrucken. Sie benötigen zum Ansehen den Acrobat Reader (oder ein anderes adäquates Programm).

dpunkt.ebooks koennen Bücher (oder Teile daraus) sein, die es auch in gedruckter Form gibt (bzw. gab und die inzwischen vergriffen sind). (Einen entsprechenden Hinweis auf eine gedruckte Ausgabe finden Sie auf der entsprechenden E-Book-Seite.)

Es können aber auch Originalpublikationen sein, die es ausschließlich in E-Book-Form gibt. Diese werden mit der gleichen Sorgfalt und in der gleichen Qualität veröffentlicht, die Sie bereits von gedruckten dpunkt.büchern her kennen.

Was darf ich mit dem dpunkt.ebook tun?

Die Datei ist nicht kopiergeschützt, kann also für den eigenen Bedarf beliebig kopiert werden. Es ist jedoch nicht gestattet, die Datei weiterzugeben oder für andere zugänglich in Netzwerke zu stellen. Sie erwerben also eine Ein-Personen-Nutzungslizenz.

Wenn Sie mehrere Exemplare des gleichen E-Books kaufen, erwerben Sie damit die Lizenz für die entsprechende Anzahl von Nutzern.

Um Missbrauch zu reduzieren, haben wir die PDF-Datei mit einer PIN und einem Wasserzeichen (Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer) versehen.

Bitte beachten Sie, dass die Inhalte der Datei in jedem Fall dem Copyright des Verlages unterliegen.

Wie kann ich dpunkt.ebooks kaufen und bezahlen?

Legen Sie die E-Books in den Warenkorb. (Aus technischen Gruenden, können im Warenkorb nur gedruckte Bücher ODER E-Books enthalten sein.)

Downloads und E-Books können sie bei dpunkt per Paypal bezahlen. Wenn Sie noch kein Paypal-Konto haben, können Sie dieses in Minutenschnelle einrichten (den entsprechenden Link erhalten Sie während des Bezahlvorgangs) und so über Ihre Kreditkarte oder per Überweisung bezahlen.

Wie erhalte ich das dpunkt.ebook?

Sobald der Bestell- und Bezahlvorgang abgeschlossen ist, erhalten Sie an die von Ihnen angegebene E-Mail-Adresse eine Bestätigung von Paypal sowie eine E-Mail vom dpunkt.verlag mit dem folgenden Inhalt:

- Downloadlinks für die gekauften Dokumente
- PINs für die gekauften Dokumente
- eine PDF-Rechnung für die Bestellung

Die Downloadlinks sind zwei Wochen lang gültig. Die Dokumente selbst sind durch eine PIN geschützt und mit Ihrer E-Mail-Adresse und Ihrer Transaktionsnummer als Wasserzeichen versehen.

Wenn es Probleme gibt?

Bitte wenden Sie sich bei Problemen an den dpunkt.verlag:
hallo@dpunkt.de

Testgetriebene Entwicklung – mit JUnit & FIT

Frank Westphal ist freier Softwareentwicklungscoach, Berater und Trainer. Seine Schwerpunkte sind Extreme Programming, Lean und Test- Driven Development. Mit dem *Tonabnehmer* bietet er einen Podcast zum Thema Agile Softwareentwicklung.

Frank Westphal

Testgetriebene Entwicklung – mit JUnit & FIT

Wie Software änderbar bleibt

Frank Westphal
http://www.frankwestphal.de

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Herstellung: Birgit Bäuerlein
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: Koninklijke Wöhrmann B.V., Zutphen, Niederlande

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

ISBN 978-3-89864-996-4

1. Auflage 2006
Copyright © 2006 dpunkt.verlag GmbH
Ringstraße 19 B
69115 Heidelberg

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

Für Renate und Horst Westphal

Geleitwort von Johannes Link

Mein erster programmierbarer Computer hatte 3,5 Kilobyte verfügbaren Hauptspeicher, was in etwa 200 Basic-Zeilen entsprach. Mein Problem damals war, dass die erdachten Algorithmen und Prozeduren zwar problemlos in meinem Kopf Platz hatten, nicht jedoch im Rechner. Ich lechzte nach mehr Speicher, um all meine Ideen auch tatsächlich verwirklichen zu können. Irgendwo beim Übergang von den 3,5 Kilobyte zu den 2 Gigabyte, die heute das Innere meines PCs zieren, drehte sich der Spieß um. Wunderbare Programmiervisionen hatte ich immer noch, allein die Umsetzung gelang mir nicht, denn ich war nicht mehr fähig, mich an alle Details des Codes zu erinnern, und verlor mich im Sumpf fehlgeschlagener und doch nicht vollständig reversibler Programmierversuche.

Damals begann meine Suche nach der *skalierbaren* Softwareentwicklungsmethode: Wie gehen wir Systeme an, die weit größer sind, als dass einer der Beteiligten sie vollständig verstehen könnte? Wie gelingt es uns, den sich ständig ändernden Codemoloch im Griff zu behalten? Wie steuern wir mit anderen Programmierern auf ein gemeinsames Ziel zu, anstatt uns ständig ins Gehege zu kommen? Testgetriebene Entwicklung hat genau das geschafft: Zielgerichtete und qualitativ hochwertige Softwareentwicklung im großen Stil wird möglich. Und Frank hat in diesem Buch den elementaren Kern dieses Vorgehens festgehalten und weitergedacht.

Doch ich hatte noch mehr Glück, denn ich konnte das Buch bereits testen, bevor (und während) es geschrieben wurde. In zahlreichen Diskussionen, gemeinsamen Vorträgen und Programmiersitzungen hat mir Frank seine Sicht auf die Kunst der Softwareentwicklung gezeigt. Und nicht nur das: Immer wieder hat er neue Ideen eingebracht, die zum Umkrempeln meiner scheinbar so ausgereiften Ansichten geführt haben. Viele dieser in den letzten Jahren gewonnenen Einsichten stecken in diesem Buch und ich bin mir sicher, dass sowohl der Einsteiger

in Testgetriebene Entwicklung als auch der erfahrene Test-First-Entwickler hiervon inspiriert wird. Doch genug des Lobes, Frank hat auch schlechte Seiten: Nie ist er mit dem Erreichten zufrieden, ständig stellt er den bequemen Status quo in Frage. Das macht Angst und vertreibt mich vom gemütlichen Programmiersofa. Dank' dir dafür, Frank.

Johannes Link
andrena objects ag

Inhaltsverzeichnis

1 Einleitung 1

1.1 Was ist Testgetriebene Entwicklung? 2
1.2 Warum Testgetriebene Entwicklung? 4
1.3 Über dieses Buch 7
1.4 Merci beaucoup 8

2 Testgetriebene Entwicklung, über die Schulter geschaut 9

2.1 Eine Programmierepisode 10
2.2 Testgetriebenes Programmieren 11
2.3 Möglichst einfaches Design 12
2.4 Ein wenig Testen, ein wenig Programmieren 13
2.5 Evolutionäres Design 14
2.6 Natürlicher Abschluss einer Programmierepisode 16
2.7 Refactoring 17
2.8 Abschließende Reflexion 19
2.9 Häufige Integration 20
2.10 Rückblende 20

3 Unit Tests mit JUnit 21

3.1 Download und Installation 21
3.2 Ein erstes Beispiel 22
3.3 Anatomie eines Testfalls 23
3.4 Test-First 24
3.5 JUnit in Eclipse 26
3.6 Das JUnit-Framework von innen 27
3.7 »Assert« 27
3.8 »AssertionFailedError« 29

3.9 »TestCase« ... 31
3.10 Lebenszyklus eines Testfalls ... 34
3.11 »TestSuite« ... 36
3.12 »TestRunner« ... 38
3.13 Zwei Methoden, die das Testen vereinfachen ... 39
3.14 Testen von Exceptions ... 42
3.15 Unerwartete Exceptions ... 43
3.16 JUnit 4 ... 44

4 Testgetriebene Programmierung 51

4.1 Die erste Direktive ... 51
4.2 Der Testgetriebene Entwicklungszyklus ... 52
4.3 Die Programmierzüge ... 53
4.4 Beginn einer Testepisode ... 54
4.5 Ein einfacher Testplan ... 55
4.6 Erst ein neuer Test 56
4.7 ... dann den Test fehlschlagen sehen ... 57
4.8 ... schließlich den Test erfüllen ... 59
4.9 Zusammenspiel von Test- und Programmcode ... 60
4.10 Ausnahmebehandlung ... 61
4.11 Ein unerwarteter Erfolg ... 62
4.12 Ein unerwarteter Fehlschlag ... 64
4.13 Vorprogrammierte Schwierigkeiten ... 68
4.14 Kleine Schritte gehen ... 72

5 Refactoring 73

5.1 Die zweite Direktive ... 73
5.2 Die Refactoringzüge ... 74
5.3 Von übel riechendem Code 75
5.4 ... über den Refactoringkatalog ... 76
5.5 ... zur Einfachen Form ... 77
5.6 Überlegungen zur Refactoringroute ... 78
5.7 Substitution einer Implementierung ... 79
5.8 Evolution einer Schnittstelle ... 80
5.9 Teilen von Klassen ... 83
5.10 Verschieben von Tests ... 85
5.11 Abstraktion statt Duplikation ... 86
5.12 Die letzte Durchsicht ... 90
5.13 Ist Design tot? ... 91

5.14 Richtungswechsel ... 95
5.15 ... und der wegweisende Test 96
5.16 Fake it ('til you make it) 98
5.17 Vom Bekannten zum Unbekannten 99
5.18 Retrospektive 106
5.19 Tour de Design évolutionnaire 107
5.20 Durchbrüche erleben 108

6 Häufige Integration 109

6.1 Die dritte Direktive 109
6.2 Die Integrationszüge 110
6.3 Änderungen mehrmals täglich zusammenführen ... 111
6.4 ... das System von Grund auf neu bauen 113
6.5 ... und ausliefern 114
6.6 Versionsverwaltung (mit CVS oder Subversion) 116
6.7 Build-Skript mit Ant 116
6.8 Build-Prozess-Tuning 121
6.9 Integrationsserver mit CruiseControl 122
6.10 Aufbau einer Staging-Umgebung 124
6.11 Teamübergreifende Integration 124
6.12 Gesund bleiben 125

7 Testfälle schreiben von A bis Z 127

7.1 Aufbau von Testfällen 127
7.2 Benennung von Testfällen 129
7.3 Buchführung auf dem Notizblock 129
7.4 Der erste Testfall 130
7.5 Der nächste Testfall 130
7.6 Erinnerungstests 131
7.7 Ergebnisse im Test festschreiben, nicht berechnen 131
7.8 Erst die Zusicherung schreiben 132
7.9 Features testen, nicht Methoden 133
7.10 Finden von Testfällen 134
7.11 Generierung von Testdaten 134
7.12 Implementierungsunabhängige Tests 136
7.13 Kostspielige Setups 137
7.14 Lange Assert-Ketten oder mehrere Testfälle? 138
7.15 Lerntests 139

7.16 Minimale Fixture! 140
7.17 Negativtests 142
7.18 Organisation von Testfällen 143
7.19 Orthogonale Testfälle 144
7.20 Parameterisierbare Testfälle 150
7.21 Qualität der Testsuite 153
7.22 Refactoring von Testcode 160
7.23 Reihenfolgeunabhängigkeit der Tests 162
7.24 Selbsterklärende Testfälle 164
7.25 String-Parameter von Zusicherungen 165
7.26 Szenarientests 166
7.27 Testexemplare 166
7.28 Testsprachen 168
7.29 Umgang mit Defekten 169
7.30 Umgang mit externem Code 172
7.31 Was wird getestet? Was nicht? 173
7.32 Zufälle und Zeitabhängigkeiten 173

8 Isoliertes Testen durch Stubs und Mocks 175

8.1 Verflixte Abhängigkeiten! 176
8.2 Was ist die Unit im Unit Test? 177
8.3 Mikrointegrationstest versus strikter Unit Test 178
8.4 Vertrauenswürdige Komponenten 179
8.5 Austauschbarkeit von Objekten 182
8.6 Stub-Objekte 183
8.7 Größere Unabhängigkeit 184
8.8 Testen durch Indirektion 185
8.9 Stub-Variationen 189
8.10 Testen von Mittelsmännern 190
8.11 Self-Shunt 191
8.12 Testen von innen 193
8.13 Möglichst frühzeitiger Fehlschlag 194
8.14 Erwartungen entwickeln 195
8.15 Gebrauchsfertige Erwartungsklassen 197
8.16 Testen von Protokollen 198
8.17 Mock-Objekte 201
8.18 Wann verwende ich welches Testmuster? 204
8.19 Crashtest-Dummies 207

8.20 Dynamische Mocks mit EasyMock 208
8.21 Stubs via Record/Replay 210
8.22 Überspezifizierte Tests 211
8.23 Überstrapazierte Mocks 211
8.24 Systemgrenzen im Test 212

9 Entwicklung mit Mock-Objekten 215

9.1 Tell, don't ask 215
9.2 Von außen nach innen 216
9.3 Wer verifiziert wen? 217
9.4 Schnittstellen finden auf natürlichem Weg 218
9.5 Komponierte Methoden 220
9.6 Vom Mock lernen für die Implementierung 221
9.7 Viele schmale Schnittstellen 223
9.8 Kleine fokussierte Klassen 224
9.9 Tell und Ask unterscheiden 225
9.10 nereimmargorP sträwkcüR 226
9.11 Schüchterner Code und das Gesetz von Demeter 228
9.12 Fassaden und Mediatoren als Abstraktionsebene 229
9.13 Rekonstruktion 230

10 Akzeptanztests mit FIT 233

10.1 Von einer ausführbaren Spezifikation 234
10.2 Download Now 234
10.3 Schritt für Schritt für Schritt 235
10.4 ... zum ausführbaren Anforderungsdokument 242
10.5 Die drei Basis-Fixtures 243
10.6 »ActionFixture« 243
10.7 Richtung peilen, Fortschritt erzielen 247
10.8 Fixture wachsen lassen, dann Struktur extrahieren 249
10.9 Nichts als Fassade 251
10.10 Die Fixture als zusätzlicher Klient 253
10.11 Aktion: Neue Aktion 254
10.12 »ColumnFixture« 256
10.13 Fixture-Interkommunikation 258
10.14 Negativbeispiele 259
10.15 Transformation: Action ➟ Column 261
10.16 »RowFixture« 264
10.17 Einfacher Schlüssel 266

10.18 Mehrfacher Schlüssel 269
10.19 Abfragemethoden einspannen 270
10.20 »Summary« 271
10.21 »ExampleTests« 274
10.22 »AllFiles« 276
10.23 Setup- und Teardown-Fixtures 278
10.24 Das FIT-Framework von innen 278
10.25 »FileRunner« 279
10.26 »Parse« 279
10.27 »Fixture« 282
10.28 Annotationsmöglichkeiten in Dokumenten 284
10.29 »TypeAdapter« 289
10.30 »ScientificDouble« 293
10.31 Domänenspezifische Grammatiken 293
10.32 »ArrayAdapter« 294
10.33 »PrimitiveFixture« 295
10.34 Domänenspezifische Fixtures 297
10.35 Anschluss finden 299
10.36 Stichproben reiner Geschäftslogik 300
10.37 Integrationstests gegen Fassaden und Services 300
10.38 Oberflächentests 302
10.39 Kundenfreundliche Namen 304
10.40 FitNesse 304
10.41 FitLibrary 306
10.42 Akzeptanztesten aus Projektsicht 307

11 Änderbare Software 309

11.1 Konstantes Entwicklungstempo 310
11.2 Kurze Zykluszeiten 315
11.3 Neue Geschäftsmodelle 316

Literatur 321

Werkzeugkasten 323

Index 325

1 Einleitung

Testen ist eigentlich nicht cool. Nie hätte ich erwartet, einmal ein Buch übers Testen zu schreiben.

JUnit hat einen Wandel eingeläutet. JUnits grüner Balken macht süchtig, sorgt er doch für ein lang vermisstes Gefühl: das Vertrauen, dass die produzierte Software tatsächlich wie gewünscht funktioniert. In gleicher Weise schickt sich derzeit FIT an, das Vertrauensverhältnis der Kunden zur gelieferten Software zu stärken. Auf einmal ist Testen doch cool. Warum dieser Wandel?

Testen war typischerweise

- die letzte Phase in wasserfallartigen Projekten;
- das letzte Kapitel in Büchern über die Softwareentwicklung;
- worauf kaum ein Programmierer wirklich Lust hat;
- Aufgabe der Abteilung für Qualitätssicherung;
- schmerzhaft, wenn wir Fehler lange nach ihrer Entstehung finden;
- das Erste, was über Bord geht, wenn es eng wird;
- nur ad hoc, selten systematisch.

Ich möchte in diesem Buch einen anderen Weg gehen. Ich möchte zeigen, wie ein bisschen Testen einen großen Unterschied machen kann. Deshalb sollten Sie weiterlesen.

Testen ist wirklich wichtig. Wissen wir.

Dass Software meist nicht ausreichend getestet wurde, ist zum Teil gut verständlich, da die Gründe dafür in erster Linie im *Menschlichen* liegen. Damit das Testen im Programmieralltag und selbst bei zunehmendem Stress nicht vernachlässigt wird, muss es mit den natürlichen Instinkten der verantwortlichen Menschen gehen und der Natur der Softwareentwicklung folgen.

Wie könnte eine effektive Teststrategie zur Entwicklung technisch und geschäftlich hochwertiger Software aussehen?

Effektives Testen muss

- **möglichst zeitnah zur Programmierung** erfolgen, damit es nicht zu einem nachgelagerten Prozess wird, der unter Zeitdruck ausfällt;
- **automatisiert wiederholbar** sein, weil manuelle Tests nicht durchgeführt werden, wenn der Stress im Programmieralltag zunimmt;
- **Spaß machen –** Softwareentwicklung soll Spaß machen;
- **so häufig wie das Kompilieren** ausgeführt werden, damit Fehler und Seiteneffekte gleich bei ihrer Entstehung entdeckt werden, nicht erst vom Testteam oder gar vom Kunden;
- **so einfach wie das Kompilieren** sein, um Tests in der Programmiersprache selbst zu schreiben, nicht mit proprietären Werkzeugen;
- **pragmatisch** sein, um an erster Stelle funktionierende Software zu produzieren, nicht um Punkte einer Checkliste abzuhaken;
- **mehr bringen als kosten,** ansonsten wird es zum Selbstzweck.

1.1 Was ist Testgetriebene Entwicklung?

Testgetriebene Entwicklung (engl. *Test-Driven Development*) ist eine qualitätsbewusste Programmiertechnik, mit der wir ein Programm in kleinen Schritten entwickeln können. Als Entwickler schreiben wir automatisierte *Unit Tests* für die Anforderungen an unseren Code. Diese Tests prüfen ein Programm in kleinen unabhängigen Einheiten. Sie helfen uns, das *Programm richtig* zu entwickeln. Zudem schreiben wir automatisierte *Akzeptanztests* für die Anforderungen der Kunden. Diese Tests prüfen ein Programm als große integrierte Systemeinheit. Sie helfen uns, das *richtige Programm* zu entwickeln.

Unit Tests

Akzeptanztests

Testgetriebene Programmierung

Die Idee testgetriebenen Arbeitens ist, erst Testcode zu schreiben, bevor wir den eigentlich zu testenden Programmcode schreiben. Das heißt, dass jede funktionale Programmänderung durch einen gezielten Test motiviert wird [be$_{02a}$]. Diesen Test entwerfen wir so, dass er zunächst fehlschlägt. Er muss fehlschlagen, weil das Programm die gewünschte Funktionalität noch nicht besitzt. Erst anschließend schreiben wir den Code, der diesen Test erfüllt. Auf diese Weise treiben wir die gesamte Entwicklung inkrementell durch das unmittelbare Feedback konkreter Tests an. Somit lernen wir, wann unser Programm zu funktionieren beginnt und wann es zu funktionieren aufhört.

1. Direktive der Testgetriebenen Entwicklung:
Motivieren Sie jede Änderung des Programmverhaltens durch einen automatisierten Test.

Sie fragen sich vielleicht, was wir denn testen wollen, wenn wir noch überhaupt keinen Code geschrieben haben? Doch diese Frage lässt sich umdrehen: Woher wissen wir denn, was wir programmieren sollen, wenn wir noch nicht wissen, was denn überhaupt erforderlich ist? Zuerst die Tests zu schreiben, ermöglicht uns herauszufinden, was wir programmieren müssen und was nicht. Zuerst die Tests zu schreiben, stellt außerdem sicher, dass wir tatsächlich programmieren, was wir programmieren wollten.

Spezifizieren – Programmieren – Verifizieren

Wahrscheinlich kennen Sie das typische Testproblem, dass sich Code nur schwer testen lässt, der nicht gleich testbar entworfen wurde. Die klassische Testliteratur weiß für solche Fälle schweres Geschütz aufzufahren, weil das Kind leider schon in den Brunnen gefallen ist. Testgetriebene Entwicklung umgeht dieses Problem, weil das frühe Testen noch Auswirkungen auf das resultierende Design nehmen kann. Wenn Sie Ihre Tests zuerst schreiben, wird Ihr Code auch testbar sein. Den Test haben Sie ja schließlich gerade schon geschrieben.

Testbarkeit von Anfang an

Iterativ-inkrementelle Entwicklung ist nur mit gut strukturiertem Code möglich. Ohne ständige Pflege nimmt die Codequalität mit zunehmender Entwicklungsdauer für gewöhnlich ab, was jede Weiterentwicklung behindert, wenn nicht sogar verhindert.

Die einzige existierende Gegenmaßnahme ist stetiges *Refactoring*. Durch kleine funktionserhaltende Schritte zur Designverbesserung können wir diesen Trend umkehren: Wir können Code so entwickeln, dass er nicht von Tag zu Tag mehr und mehr degeneriert, sondern dass seine Qualität unaufhörlich steigt. *Einfache Form* zu erhalten, ist das Ziel des Refactorings. Indem wir nach jedem einzelnen Refactoringschritt wieder und wieder alle gesammelten Tests ausführen, können wir sicherstellen, dass durch unsere Umformung nicht ungewollt das vorhandene Funktionsverhalten des Programms beeinträchtigt wurde.

Refactoring

2. Direktive der Testgetriebenen Entwicklung:
Bringen Sie Ihren Code immer in die Einfache Form.

Sobald wir Software im Team entwickeln, wollen wir alle Änderungen am Code regelmäßig miteinander abgleichen. Die *Häufige Integration* der einzelnen Entwicklungstätigkeiten stellt sicher, dass Konflikte, wenn sie auftreten, noch klein und somit sehr leicht zu beheben sind, und dass wir jederzeit eine lauffähige Software demonstrieren können. Zur erfolgreichen Integration müssen natürlich alle Tests laufen.

Häufige Integration

3. Direktive der Testgetriebenen Entwicklung:
Integrieren Sie Ihren Code so häufig wie nötig.

1.2 Warum Testgetriebene Entwicklung?

Testgetriebene Entwicklung ist eine qualitätsbewusste Programmiertechnik. Doch was heißt Softwarequalität in diesem Zusammenhang?

Jede Software hat einen bestimmten Wert, der sich definiert durch

- **funktionale Qualität** im Hinblick auf Funktionalität und Fehlerfreiheit für die einwandfreie Benutzbarkeit einer Software;
- **strukturelle Qualität** im Hinblick auf Design und Codestruktur für die nahtlose Weiterentwicklung einer Software.

Testgetriebene Entwicklung ist qualitätsbewusst, weil sie die zwei Qualitäten gleichzeitig über weite Zeiträume aufrechterhalten kann.

- Automatisierte Tests bewahren die funktionale Qualität.
- Fortlaufendes Refactoring bewahrt die strukturelle Qualität.

Software, die wie gewünscht funktioniert, aber nicht wie gewünscht weiterentwickelt werden kann, hat nach dieser Definition keinen Wert. Die eine Qualität kommt nicht ohne die andere aus. Der Gesamtwert der Software wird wirklich durch das Minimum der beiden bestimmt. Testgetriebene Entwicklung ermöglicht uns erst, dass wir entwickeln können, was wir brauchen, wenn wir es brauchen, ohne die Qualität unserer Software zunehmend zu kompromittieren. Just-in-Time.

Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken.

Tests machen Fortschritt greifbar. Programmierte Software ist an und für sich nicht greifbar. Erst durch sich selbst prüfenden Code wird der programmierte Code greifbar. Programmierer, die ihren Programmcode durch solche Selbsttests fixieren, vergegenständlichen ihn damit. Sie lernen, indem sie etwas Greifbares entwickeln. Wie ist das gemeint?

Stellen Sie sich einen Kletterer vor, der jeden seiner Schritte durch einen Haken absichert. Mit jedem gesetzten Sicherheitshaken reduziert er ganz bewusst sein Risiko, wie tief er bei einem Fehltritt fallen kann. Die Länge seiner Sicherheitsleine bestimmt dabei maßgeblich die Lücke von dem Moment, wo der Kletterer den ersten ungesicherten Schritt probiert, bis zu dem Moment, wo er seine Kletterkünste durch einen weiteren Haken belohnt. Der bis zum Haken erkletterte Weg gehört ihm in jedem Fall, selbst wenn ihm ein Fehler unterläuft. Jeder gesetzte Haken bedeutet Fortschritt für ihn, auch wenn es ihm Mühe kostet, den Haken in die Wand zu schlagen. Auch hat der Kletterer es selbst in der Hand, wie viel er in einer Situation aufs Spiel setzen möchte. An steilen Hängen wird er seine Leine sicher kürzer halten wollen als an zugänglichen Passagen.

Programmierte Tests sind wie diese Sicherheitshaken. Sie schenken uns ein dichtes Sicherheitsnetz, während wir bei stetig wachsender Softwarekomplexität höher und höher klettern. Beim Programmieren gilt es, die Länge der Leine bewusst zu kontrollieren. Wie viele Tests Sie schreiben müssen, damit Sie Vertrauen in den programmierten Code haben, und wie groß Sie Ihre Programmierschritte dabei wählen, lernen Sie auf Ihrem Weg nach oben.

Tests sichern den Erhalt der vorhandenen Funktionalität bei Erweiterung und Überarbeitung.

Software ist änderbar. Zu leicht änderbar, wie sich herausstellt. Es ist schließlich kinderleicht, ein Programm zu ändern. Auch irrtümlich. Unter Umständen muss dazu nur ein einziges Bit kippen. Die erste Hürde der Softwareentwicklung besteht darin, ein Programm so zu entwickeln, dass es genau das tut, was es tun soll. Mit zunehmender Komplexität einer Software gewinnt diese Hürde jedoch gewöhnlich an Höhe. Unser Interesse muss es sein, diese Barriere während der gesamten Entwicklung möglichst niedrig zu halten. Tests können uns dabei helfen, Software so gut wie möglich zu entwickeln. Denn ohne Tests überlassen wir die Softwareentwicklung zum Teil dem Zufall, weil selbst kleine und sorgfältige Änderungen auch Auswirkungen auf das übrige Programmverhalten nehmen können. Dafür ist Software zu komplex und deshalb ist häufiges Testen so wichtig.

Die erste Hürde: die Entwicklung selbst

Erfolgreiche Software wird weiterentwickelt. Software wird nie ganz fertig. Praktisch ergeben sich immer noch Änderungen, die teils notwendig oder wenigstens wünschenswert sind. Entweder können sich die wirklichen Benutzeranforderungen erst aufgrund von schon ausgelieferter Software herauskristallisieren oder schnell wandelnde Märkte entscheiden über den neuen Geschäftswert einer Softwareidee. Die zweite Hürde besteht in der Softwareentwicklung darin, einmal entwickelte Software auch problemlos weiterentwickeln zu können.

Die zweite Hürde: die Weiterentwicklung

Software leidet unter Entropie. Obwohl die Änderbarkeit von Software in der Natur noch ihresgleichen sucht, verhärtet der Code doch in vielen Softwareprojekten mit der Zeit. Der Grund dafür ist die einsetzende Entropie. Das heißt, der Grad an Unordnung innerhalb eines Systems nimmt zu. Die Ursache lässt sich so erklären: Mit jeder weiteren Zeile Code schlägt die Geschichte, die unser Programm erzählt, neue Bögen. Gleichzeitig schränken wir jedoch den Rahmen der Geschichte immer weiter ein. Wenn wir nicht bald wieder Raum für neuen Code schaffen, ist die Geschichte schnell zu Ende erzählt.

Refactoring verlängert die produktive Lebensdauer einer Software.

Software muss soft bleiben. Der Code von gestern darf den Code von heute nicht behindern. Wenn wir nicht ständig den Kompromiss neu eingehen zwischen wertvoller weiterer Programmfunktionalität und den Aufwänden, um den bereits bestehenden Code in Schuss zu halten, steht die Weiterentwicklung bald vor ihrem Aus. Unsere Software gibt durch unzählige Änderungen und unter gleichzeitiger Missachtung der Softwareentropie zuerst ihre strukturelle Qualität auf und oft ihre funktionale Qualität gleich hinterher.

Software verrottet. Wenn die ursprünglichen Entwurfsabsichten im geschriebenen Programmcode immer schwieriger zu entziffern sind, verringert sich unsere Entwicklungsgeschwindigkeit. Niemand traut sich mehr, den verrottenden Code anzufassen. Gerade wenn wir aus Unachtsamkeit oder Zeitdruck nicht regelmäßig den Code aufräumen, degeneriert er so rapide, dass er bald keine Änderungen mehr zulässt, ohne immerzu neue Fehler ins Programm einzuführen. Zusätzlich steigt damit natürlich unsere Angst vor unkontrolliert entstehenden Seiteneffekten und die Geschichte nimmt ihren weiteren Verlauf.

Code lässt sich im Nachhinein oft nur schlecht testen.

Hochwertige Software erfordert automatisierte Tests. Je weniger Zeit zwischen der Entstehung und Entdeckung von Fehlern vergeht, desto schneller können wir sie eingrenzen und beheben. Wenn wir unsere Software jederzeit ändern und binnen Sekunden ihre Fitness prüfen können, haben wir beweglichen Code und damit Wettbewerbsvorteile. Unser Vertrauen in die Software steigt und damit unser Mut und die Fähigkeit, Änderungen vorzunehmen, die wir uns unter anderen Umständen nicht zugetraut hätten. Eine vollständige Automatisierung des Testprozesses ist notwendig, weil wir unsere Tests extrem häufig ausführen. Die Wiederholung manueller Tests wäre zu langwierig.

Testen muss in die Softwareentwicklung integriert sein. Nur wenn wir unsere Tests zeitnah zur Programmierung durchführen, haben wir die Chance, über die Tests die Einsicht in ein insgesamt einfacher zu testendes Design zu finden. Der Moment, in dem wir das Verständnis gewonnen haben, wie das Programm arbeiten soll, ist zudem ein guter Zeitpunkt, unser Wissen in einem Test festzuhalten. Aus diesem Grund liegt der ideale Zeitpunkt, um einen Test zu schreiben, unmittelbar vor der eigentlichen Programmierung. Der Compiler prüft die Programmsyntax, unsere Tests prüfen die Semantik.

1.3 Über dieses Buch

Dieses Buch führt Sie in die Testgetriebene Entwicklung ein, und zwar im Kleinen (mit Unit Tests) wie auch im Großen (mit Akzeptanztests). Ich habe das Buch für angehende und professionelle Softwareentwickler geschrieben. Mein Ziel ist es, Ihnen die Techniken und Werkzeuge zu vermitteln, die ich für das Handwerk der Softwareentwicklung für wertvoll halte.

Aufbau des Buches

Kapitel 2 gibt Ihnen einen Überblick über die Philosophie und Basistechniken der Testgetriebenen Entwicklung. Die darauf folgenden vier Kapitel umfassen das Grundhandwerkszeug: Kapitel 3 macht Sie vorab mit JUnit vertraut, Kapitel 4 widmet sich dem Testgetriebenen Entwicklungszyklus, Kapitel 5 der Rolle des Refactorings darin und Kapitel 6 der Häufigen Integration im Team. Die zwei Kapitel 7 und 8 vermitteln Ihnen dann fortgeschrittenere Techniken zum Schreiben automatisierter Tests und zum Meistern schwieriger Testsituationen. Kapitel 9 schlägt eine alternative Route zu Kapitel 8 ein. Überspringen Sie dieses Kapitel fürs Erste, falls Sie sich vom Stoff abgehängt fühlen – hier ballen sich viele fortgeschrittene Techniken in einem Buchkapitel. In Kapitel 10 treffen Sie schließlich auf FIT und Kapitel 11 beendet das Buch mit einigen Schlussgedanken.

Ich arbeite in diesem Buch mit einer großen Menge Codebeispiele. Achten Sie jedoch nicht so sehr auf den Code, sondern auf meine Bewegungen. Der Code dient mir nur als Vehikel. Viel wichtiger sind die Schritte. Aus diesem Grund möchte ich Sie auch dazu motivieren, die Beispiele live am Rechner mitzuverfolgen. Sie werden ungleich mehr davon haben. Den Quellcode aller Kapitel und Ergänzungen zum Buch finden Sie auf meinen Webseiten:

```
http://www.frankwestphal.de/
TestgetriebeneEntwicklungmitJUnitundFIT.html
```

Alle Codebeispiele basieren auf Java, lassen sich jedoch leicht in andere Sprachen übersetzen. Ich habe das Buchbeispiel, ein kleines DVD-Verleihsystem, bewusst einfach und technologieunabhängig gehalten. Wenn Sie *konkrete Teststrategien* suchen, wie Sie dies und das testen können, konsultieren Sie das Buch [li05] von Johannes Link. Ich habe dagegen versucht, mich auf die *allgemeinen Prinzipien* zu konzentrieren. In anderen Worten: Ich will Ihnen keine Testprobleme knacken; ich möchte Ihnen zeigen, wie Sie die Herausforderungen selber meistern. Die beiden Bücher sind sich gegenseitig also zwei gute Kompagnons.

1.4 Merci beaucoup

Insgesamt habe ich in der einen oder anderen Form über fünf Jahre an diesem Buch gearbeitet. Ich möchte an dieser Stelle den zahlreichen Menschen danken, die mir während dieser langen Zeit immer wieder Mut, Kraft und Inspiration geschenkt haben. Die Ehrenurkunden gehen an **Jutta Eckstein**, **Tammo Freese**, **Dierk König**, **Johannes Link** und **Christa Preisendanz**.

Für Review-Kommentare und Ideen danke ich (in chronologischer Reihenfolge): **Leah Striker**, **Michael Schürig**, **Meike Budweg**, **Tammo Freese**, **Ulrike Jürgens**, **Rolf F. Katzenberger**, **Hans Wegener**, **Marko Schulz**, **Antonín Andert**, **Manfred Lange**, **Julian Mack**, **Johannes Link**, **Karsten Menne**, **Martin Müller-Rohde**, **Stefan Roock**, **Andreas Schoolmann**, **Frankmartin Wiethüchter**, **Dierk König**, **Bastiaan Harmsen**, **Olaf Kock**, **Stefan Schmiedl**, **Martin Lippert**, **Etienne Studer**, **Ilja Preuß**, **Jens Uwe Pipka**, **Robert Wenner**, **Armin Röhrl**, **Eberhard Wolff**, **Peter Roßbach**, **Alexander Schmid**, **Mario Winter**, **Daniel Schweizer**, **Torsten Mumme**, **Eduard Bachner**, **Bernd Schiffer**, **Ali Natour**, **Jürgen Ahting**, **Ralf Stuckert**, **Markus Schramm**. Im Traum hätte ich nicht für möglich gehalten, wie viele unterschiedliche Fehler und Schwächen so viele Augen entdecken würden. Der Titel des Meisterrezensenten geht dabei an **Rolf F. Katzenberger** und **Robert Wenner**. Tausend Dank für eure Zeit und Gründlichkeit.

Besten Dank auch an meine Gastautoren: **Juan Altmayer Pizzorno**, **Ward Cunningham**, **Sabine Embacher**, **Michael Feathers**, **Steve Freeman**, **Tammo Freese**, **Bastiaan Harmsen**, **Michael Hill**, **Andy Hunt**, **Christian Junghans**, **Olaf Kock**, **Dierk König**, **Lasse Koskela**, **Steffen Künzel**, **Johannes Link**, **Tim Mackinnon**, **Ivan Moore**, **Moritz Petersen**, **Dave Thomas**, **Robert Wenner**.

Zu guter Letzt möchte ich meiner Lektorin meinen größten Dank aussprechen. Ohne **Christa Preisendanz** könnte sich dieses Buch heute sicherlich in die *Invisible Library* einreihen.

www.invisiblelibrary.com

Hamburg, Oktober 2005

Frank Westphal
http://www.frankwestphal.de

2 Testgetriebene Entwicklung, über die Schulter geschaut

»Test-infected«, so nennt Erich Gamma die sich rasch ausbreitende Begeisterung für die Entwicklung von Software mit automatisierten Tests. Die Wortschöpfung kennzeichnet die Fesslungskünste der Tests: Wenn Entwickler nicht früher vom Programmieren ablassen können, bis alle ihre Tests wieder erfolgreich durchlaufen. Der Funke springt gewöhnlich dann über, wenn Ihre Tests Ihnen zum ersten Mal eine lange Debugging-Session ersparen, wenn Sie bemerken, dass Sie mit einem dichten Sicherheitsnetz von Tests mit wenig Stress und mehr Flow hochwertige Software entwickeln können.

Bevor Sie sich anstecken lassen, möchten Sie sicherlich erfahren, worauf Sie sich mit diesem Buch möglicherweise einlassen. Sie wollen selbstverständlich herausfinden, ob und wie Sie von den vorgestellten Techniken in Ihrer täglichen Programmierarbeit profitieren können. Aus diesem Grund führe ich Sie in diesem Kapitel durch ein konkretes Beispiel. Um die ganze Breite der Testgetriebenen Entwicklung zu überblicken, werden Sie dann weiterlesen müssen, doch ein erster Überblick wird die Prinzipien klarer machen.

Ein gedrucktes Codebeispiel bringt jedoch ein Risiko mit sich. Konzentrieren Sie sich nur auf die mechanischen Programmierschritte, laufen Sie Gefahr, den vielleicht entscheidenden Aspekt zu verpassen. Testinfizierte Entwickler erzählen häufig, wie sie über ihre Tests eine neue Beziehung zum Code aufbauen. Sie können sich sicher vorstellen, welche Faszination davon ausgeht, auf Knopfdruck hunderte von Tests durchlaufen zu sehen, die die Software auf Herz und Nieren überprüfen. Ohne automatisierte Tests können wir nie genau sagen, wo die Entwicklung eigentlich gerade steht. Bei der Testgetriebenen Entwicklung können Sie am Ende des Tages auf die Menge neuer Tests zurückblicken, die Ihnen Vertrauen in die geleistete Arbeit schenken.

2.1 Eine Programmierepisode

Wie sieht die Testgetriebene Entwicklung in der Praxis aus? Nun, Sie werden wie gewohnt programmieren. Obwohl, das stimmt nicht ganz. Sie programmieren in *kleineren* Schritten als gewöhnlich und testen jeden *einzelnen* Schritt. Sie arbeiten in kurzen Programmierepisoden, an deren Ende ein funktionsfähiges getestetes System mit wertvoller neuer Funktionalität steht.

Ich möchte Sie dazu einladen, ein Gefühl für die Testgetriebene Entwicklung zu bekommen. Ich habe versucht, ihren Geist dadurch einzufangen, dass ich einen Dialog zweier Programmierer beschreibe. Wir begleiten die beiden beim Schreiben der ersten Zeilen Code für ein kleines DVD-Verleihsystem.

Ulrich: *Was ist unsere Aufgabe?*

Felix: *Wir müssen die Gebühren für ausgeliehene DVDs berechnen.*

Ulrich: *Wie berechnen wir das?*

Felix: *Der Preis ist davon abhängig, wie lange ein Film ausgeliehen wird.*

Ulrich: *Und wie?*

Felix: *Lass mich mal sehen ... Neuerscheinungen kosten für zwei Tage 2 Euro. Ab dem dritten Ausleihtag pro Tag zusätzliche 1,75 Euro.*

Ulrich: *Okay, wie machen wir das also?*

Felix: *Mmm, ich schlage vor, wir merken uns zunächst mal, welche Filme ein Kunde ausleiht, und berechnen später die Gebühr anhand der Ausleihtage?*

Ulrich: *Warum berechnen wir die Kosten nicht auf der Stelle?*

Felix: *Das ist später vielleicht notwendig, momentan ist das aber nicht gefordert.*

Ulrich: *Schön ... Wie soll unsere erste Klasse heißen?*

Felix: *Gute Frage ... Die Verantwortlichkeiten klingen fast danach, als sollte die Klasse `Customer` heißen.*

Ulrich: *Fangen wir mit dem Test an.*

Felix: *Dazu müssen wir unsere Testklasse von `TestCase` aus dem Test-Framework ableiten.*

```
public class CustomerTest extends junit.framework.TestCase {
}
```

2.2 Testgetriebenes Programmieren

Unserer Einstellung nach existiert eine Funktionalität so lange nicht, bis sie einen automatisierten Test besitzt. Tatsächlich entsteht keinerlei Programmcode, bevor es nicht einen korrespondierenden Testfall gibt, der fehlschlägt. Das bedeutet, Sie schreiben den Test, noch bevor Sie den Code schreiben, der diesen Test erfüllt. Sie erstellen inkrementell eine umfassende Menge von Unit Tests, die das gesamte Programm in kleinen isolierten Einheiten auf die erwartete Funktion hin überprüfen. Sie verwenden ein *Test-Framework*, das Ihnen hilft, automatisierte Tests zu schreiben und auszuführen. Sie sammeln und pflegen diese Tests, damit Sie nach jeder Änderung am Code sicherstellen können, dass alle Tests zu 100% laufen.

Ulrich: *Was ist der erste Testfall?*

Felix: *Am einfachsten fangen wir mit dem Ausleihen eines Films an.*

Ulrich: *Und wie?*

Felix: *Es ist so, dass wir für jede Methode, die funktionieren soll, Zusicherungen schreiben, um die Funktion abzusichern.*

Ulrich: *Dafür ist die `assertTrue`-Methode aus der `TestCase`-Klasse zuständig.*

Felix: *Genau. Als Parameter erwartet sie eine Bedingung, damit der Testfall als erfüllt gilt. Den Rest erledigt das Framework.*

Ulrich: *Und wohin schreiben wir nun unseren Testcode?*

Felix: *Wir schreiben eine Testfallmethode `testRentingOneMovie`, die dann die Mietkosten für den Film testet. Das Framework findet diese `test...`-Methoden automatisch und führt sie aus.*

Ulrich: *Schreiben wir mal auf, was wir bisher wissen: Wir benötigen zunächst ein `Customer`-Exemplar. Und dann muss ich so tun, als gäbe es einfach alle Methoden schon, die ich mir wünsche.*

Felix: *Richtig. Der Kunde leiht eine DVD für einen Tag und es soll ihn anschließend 2 Euro kosten.*

Ulrich: *Das ist einfach.*

Änderungen im Code und andere Hinweise sind durch Fettschrift hervorgehoben

```
public class CustomerTest extends junit.framework.TestCase {
  public void testRentingOneMovie() {
    Customer customer = new Customer();
    customer.rentMovie(1);
    assertTrue(customer.getTotalCharge() == 2);
  }
}
```

2.3 Möglichst einfaches Design

Unsere Designstrategie besteht darin, mit einem schlichten Design zu starten und dieses unaufhörlich zu verbessern. Tatsächlich werden kompliziertere Designelemente, die nicht unbedingt notwendig sind, zunächst aufgeschoben, selbst für nur wenige Minuten. Das bedeutet, Sie wählen von verschiedenen Wegen denjenigen, der am einfachsten erscheint, um den aktuellen Test zu erfüllen. Sie programmieren nur, was Sie jetzt *tatsächlich* benötigen, nicht, was Sie später *vielleicht* benötigen. Sie gehen sogar so weit, dass Sie unnötige Flexibilität wieder aus dem Code entfernen. Falls notwendig, treten Sie den Beweis an, dass die aktuelle Lösung noch zu einfach ist, indem Sie einen zusätzlichen Test schreiben, der ein komplexeres Design rechtfertigt.

Ulrich: *Also gut, du willst, dass ich nur den Test zum Laufen bringe und alles andere für einen Moment vergesse.*

Felix: *Ganz genau. Was würdest du tun, wenn du nur diesen einen Test erfüllen müsstest?*

Ulrich: *Hah, auch das ist einfach.*

```
public class Customer {
  public void rentMovie(int daysRented) {
  }

  public int getTotalCharge() {
    return 2;
  }
}
```

Felix: *Wie extrem! Aber gut ...*

2.4 Ein wenig Testen, ein wenig Programmieren ...

Das Zusammenspiel von Testgetriebenem Programmieren und einfachem Design ergibt den Zyklus des minutenweisen Programmierens. Tatsächlich wird nie länger als ein paar Minuten programmiert, ohne die Feedbackschleife unmittelbar durch konkrete Tests zu schließen. Das bedeutet, Sie schreiben das Programm in so winzigen Schritten, dass Ihr Code gerade einmal den aktuellen Testfall erfüllt. Sie testen ein wenig, Sie programmieren ein wenig. Dann testen Sie wiederum und programmieren ... Minute für Minute feiern Sie kleine Erfolge. Sie schreiben keine Klasse in einem Rutsch. Vielmehr schreiben Sie nur ein paar Zeilen Code, maximal eine Methode auf einmal.

Ulrich: *Als Nächstes könnten wir zwei Filme testen.*

Felix: *Der zweite wird für zwei Tage entliehen. Die Summe beträgt 4 Euro. Lass uns dazu die `assertEquals`-Methode benutzen. Als Parameter erhält sie den von uns erwarteten Wert sowie das tatsächlich berechnete Resultat.*

Drei Punkte hinter einem Klassennamen bedeuten, dass Sie hier nur einen Ausschnitt der Klasse sehen, also Details (weitere Methoden etc.) weggelassen wurden

```
public class CustomerTest...
  public void testRentingTwoMovies() {
    Customer customer = new Customer();
    customer.rentMovie(1);
    customer.rentMovie(2);
    assertEquals(4, customer.getTotalCharge());
  }
}
```

Ulrich: *Okay, dieser Test wird nicht laufen.*

Felix: *Woher weißt du das? Mach den Test, du weißt nie!*

Ulrich: *Also gut ... Der Test schlägt fehl.*

Felix: *Diesmal kommst du auch nicht mehr so einfach davon ...*

```
public class Customer {
  private int totalCharge = 0;

  public void rentMovie(int daysRented) {
    totalCharge += 2;
  }

  public int getTotalCharge() {
    return totalCharge;
  }
}
```

2.5 Evolutionäres Design

Organisches Wachstum ist eine Strategie, um auf Veränderung und Ungewissheit reagieren zu können. Tatsächlich werden Anforderungsänderungen als Chance, nicht als Problem betrachtet. Das bedeutet, Sie können sich im Design so verhalten, als wüssten Sie wirklich nicht, was die nächsten Anforderungen sein würden. Sie entwerfen stets die einfachste Lösung und bringen Ihren Code anschließend sofort wieder in eine Einfache Form. Sie vertrauen darauf, dass sauber strukturierter Code in jede gewünschte Richtung mitziehen kann und dass der Code, den Sie gestern geschrieben haben, Sie heute und morgen dabei unterstützen wird, weiterhin Code zu schreiben, und Sie nicht zunehmend daran hindert.

Ulrich: *Was ist der nächste Testfall?*

Felix: *Ein dritter Film, der drei Tage entliehen wird.*

Ulrich: *Wie viel kostet der Film, wenn er erst nach drei Tagen zurückgegeben wird?*

Felix: *Jeder weitere Tag kommt auf 1,75 Euro.*

Ulrich: *Also 3,75 Euro am Tag drei. Macht zusammen 7,75 Euro. Außerdem akzeptieren wir beim Vergleich der Fließkommazahlen eine Toleranz von 0,001.*

```
public class CustomerTest...
  public void testRentingThreeMovies() {
    Customer customer = new Customer();
    customer.rentMovie(1);
    customer.rentMovie(2);
    customer.rentMovie(3);
    assertEquals(7.75, customer.getTotalCharge(), 0.001);
  }
}
```

Felix: *Ab jetzt müssen wir auf den Preis 1,75 Euro draufschlagen.*

Ulrich: *Das bedeutet auch, dass* `totalCharge` *ab sofort Fließkommazahl sein möchte.*

```
public class Customer {
  private double totalCharge = 0;

  public void rentMovie(int daysRented) {
    totalCharge += 2;
    if (daysRented > 2) {
      totalCharge += 1.75;
    }
  }

  public double getTotalCharge() {
    return totalCharge;
  }
}
```

Felix: *Nun meckert aber der Compiler ... Wir müssen wohl auch den vorherigen Testfall zum Vergleich von Fließkommazahlen bringen.*

```
public class CustomerTest...
  public void testRentingTwoMovies() {
    Customer customer = new Customer();
    customer.rentMovie(1);
    customer.rentMovie(2);
    assertEquals(4.00, customer.getTotalCharge(), 0.001);
  }
}
```

2.6 Natürlicher Abschluss einer Programmierepisode

Bewusst aufhören zu können ist einer der stärksten Programmierzüge. Tatsächlich ist eine Aufgabe fertig, wenn alle Bedingungen getestet sind, die dazu führen könnten, dass etwas schief geht. Das bedeutet, Sie müssen nicht für jede Methode einen Test schreiben, sondern nur für solche, die unter Umständen fehlschlagen könnten. Sie halten Ihr Wissen über den Code in den Tests fest. Sie wissen, dass Sie erreicht haben, was Sie sich vorgenommen hatten, wenn alle Tests erfüllt sind.

Ulrich: *Ein Film, der vier Tage entliehen wird?*

Felix: *Kostet 5,50 Euro und macht dann insgesamt 13,25 Euro.*

```
public class CustomerTest...
  public void testRentingFourMovies() {
    Customer customer = new Customer();
    customer.rentMovie(1);
    customer.rentMovie(2);
    customer.rentMovie(3);
    customer.rentMovie(4);
    assertEquals(13.25, customer.getTotalCharge(), 0.001);
  }
}
```

Felix: *Lässt du mich mal tippen?*

Ulrich: *Okay!*

```
public class Customer...
  public void rentMovie(int daysRented) {
    totalCharge += 2;
    if (daysRented > 2) {
      totalCharge += (daysRented - 2) * 1.75;
    }
  }
}
```

Felix: *So, das hätten wir. Haben wir irgendwelche Tests vergessen?*

Ulrich: *Müsste mit dem Teufel zugehen ...*

2.7 Refactoring

Softwaredesign ist nie einfach und niemand bekommt die Dinge im ersten Versuch in den Griff. Tatsächlich entsteht ein Design durch *schrittweises Wachstum* und *ständige Überarbeitung*. Das bedeutet, dass Sie alle Erfahrungen in das Design zurückfließen lassen und das Design so fortlaufend verbessern. Sie refaktorisieren, um den Code so einfach und so verständlich wie möglich zu machen und jede Art von Redundanz zu beseitigen.

Ulrich: *Wenn ich mir unseren Code so ansehe, frage ich mich, was die vielen Zahlen bedeuten. Wir sollten ihnen Namen geben!*

Felix: *Vorschläge?*

```
public class Customer...
  private static final double BASE_PRICE = 2.00;  // Euro
  private static final double PRICE_PER_DAY = 1.75;  // Euro
  private static final int DAYS_DISCOUNTED = 2;
}
```

Ulrich: *Eigentlich finde ich die Änderung ja zu klein, um nach jedem Schritt zu testen ...*

Felix: *Du hast Mut, aber dafür haben wir ja die Tests ...*

```
public class Customer...
  public void rentMovie(int daysRented) {
    totalCharge += BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      totalCharge += (daysRented - DAYS_DISCOUNTED)
                     * PRICE_PER_DAY;
    }
  }
}
```

Ulrich: *Irgendwie passen die Konstantennamen nun aber doch nicht mehr so recht zu unserer Klasse.*

Felix: *Stimmt ... Das hat jetzt schon sehr viel mit der eigentlichen Preisberechnung für den Film zu tun.*

Ulrich: *Ziehen wir eine neue Klasse* `Movie` *raus!?*

Felix: *Einverstanden, aber zuerst die Tests ...*

```
public class MovieTest extends junit.framework.TestCase {
  public void testBasePrice() {
    assertEquals(2.00, Movie.getCharge(1), 0.001);
    assertEquals(2.00, Movie.getCharge(2), 0.001);
  }

  public void testPricePerDay() {
    assertEquals(3.75, Movie.getCharge(3), 0.001);
    assertEquals(5.50, Movie.getCharge(4), 0.001);
  }
}
```

Felix: *Die Preisberechnung verschieben wir einfach auf die neue Klasse ...*

```
public class Movie {
  private static final double BASE_PRICE = 2.00;  // Euro
  private static final double PRICE_PER_DAY = 1.75;  // Euro
  private static final int DAYS_DISCOUNTED = 2;

  public static double getCharge(int daysRented) {
    double result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      result += (daysRented - DAYS_DISCOUNTED) * PRICE_PER_DAY;
    }
    return result;
  }
}
```

Felix: *Unser* `Customer` *wird dadurch wieder ganz schlank ...*

```
public class Customer...
  public void rentMovie(int daysRented) {
    totalCharge += Movie.getCharge(daysRented);
  }
}
```

2.8 Abschließende Reflexion

Als Softwareentwickler sind wir Problemlöser. Tatsächlich können wir Software nicht schneller entwickeln, als wir dazulernen. Das bedeutet, dass Sie zum Abschluss auf Ihre Programmierepisode zurückblicken und über Ihre Arbeit reflektieren: Ist der Code klar und verständlich? Fehlen Tests? Was können wir zukünftig besser machen?

Ulrich: *Sollten sich die Tests nicht mit der Klasse ändern, nachdem wir unseren* `Customer` *so stark vereinfachen konnten?*

Felix: *Gute Idee! Was schlägst du vor?*

Ulrich: *Naja, der* `Customer` *ist jetzt nur noch für die Summierung zuständig. Das sollten wir auch in den Tests dokumentieren.*

Felix: *Bei der Gelegenheit können wir auch das* `customer`*-Objekt als Feld definieren, weil wir es eh in all unseren Tests benötigen. Initialisieren können wir es in der* `setUp`*-Methode, die läuft, bevor eine* `test...`*-Methode ausgeführt wird.*

Ulrich: *Mir fällt gerade noch auf, dass wir den Fall übersehen haben, wenn Kunden überhaupt keine DVDs ausleihen.*

```
public class CustomerTest extends junit.framework.TestCase {
  private Customer customer;

  protected void setUp() {
    customer = new Customer();
  }

  public void testRentingNoMovie() {
    assertEquals(0, customer.getTotalCharge(), 0.001);
  }

  public void testRentingOneMovie() {
    customer.rentMovie(1);
    assertEquals(2.00, customer.getTotalCharge(), 0.001);
  }

  public void testRentingThreeMovies() {
    customer.rentMovie(2);
    customer.rentMovie(3);
    customer.rentMovie(4);
    assertEquals(11.25, customer.getTotalCharge(), 0.001);
  }
}
```

2.9 Häufige Integration

Bei der Arbeit im Team wird neuer Code so häufig wie nötig integriert. Tatsächlich wird mehrmals täglich ein getesteter *Build* des kompletten Systems erstellt. Das bedeutet, dass Sie am Ende Ihrer Programmierepisode und wenigstens einmal am Tag Ihren lokalen Entwicklungsstand mit allen Änderungen aus der Versionsverwaltung abgleichen. Sie lösen die unter Umständen entstandenen Integrationskonflikte auf, führen die Unit Tests des Projekts aus und versionieren Ihren Stand, sobald alle Tests erfolgreich laufen.

Felix: *Lass uns den Code einchecken und Mittag machen!*

Ulrich: *Milchkaffee oder Chinese?*

2.10 Rückblende

Das Beispiel sollte Ihnen vermittelt haben, wie sich die Testgetriebene Entwicklung anfühlt. Ich hoffe, es ist deutlich geworden, wie das enge Zusammenspiel automatisierter Tests, programmierten Codes und evolutionären Designs zu einer sehr *methodischen Entwicklungsweise* führt, zu *stetigem Programmierfortschritt* und *allzeit sauberem Code*. Da in kleinen Schritten und vielen kurzen »Test-Code-Refactoring«-Zyklen programmiert wird, erfordert die Testgetriebene Entwicklung anfangs etwas Disziplin. Wer aber einmal vom Virus angesteckt ist, programmiert nicht mehr ohne JUnit- und Refactoringunterstützung.

Aus dem Bauch

von Sabine Embacher, Brockmann Consult GmbH

Die für mich prägendste Erfahrung und der Grund, warum ich Testgetriebene Entwicklung nicht mehr missen will, sind die Veränderungen, die Testgetriebene Entwicklung in meinem Leben bewirkt hat. Ich hatte noch nie zuvor so wenig Angst, bestehenden Code erneut anzufassen, zu erweitern, zu verändern, zu refaktorisieren. Ich ging nie zuvor nach getaner Arbeit so entspannt und zufrieden nach Hause. Mir fiel es noch nie so leicht, nach der Arbeit abzuschalten.

3 Unit Tests mit JUnit

Eine Art von Tests, die wir zeitnah zur Programmierung erstellen und nach jeder Programmmodifikation ausführen wollen, sind Unit Tests. Im Unit Test werden kleinere Programmteile in Isolation von anderen Programmteilen getestet. Die Granularität der unabhängig getesteten Einheit kann dabei von einzelnen Methoden über Klassen bis hin zu Komponenten reichen. Unit Tests sind in den überwiegenden Fällen *White-Box-Tests*. Das bedeutet, der Test kennt die Implementierungsdetails seines Prüflings und macht von diesem Wissen Gebrauch.

Basis für die Unit Tests in diesem Buch stellt JUnit in Version 3.8.1 dar. JUnit ist ein kleines mächtiges Java-Framework zum Schreiben und Ausführen automatisierter Unit Tests. Die Software ist frei und im Kern von Kent Beck und Erich Gamma geschrieben. Die Tests werden direkt in Java kodiert, sind selbstüberprüfend und damit wiederholbar. Testfälle können mit JUnit sehr einfach organisiert und über eine Bedienungsoberfläche ausgeführt werden. JUnit hat sich seit 1998 schnell zum *De-facto-Standard für Entwicklertests* gemausert und ist inzwischen in praktisch alle modernen Java-Entwicklungsumgebungen integriert.

JUnit 4 wird in Kapitel 3.16 besprochen

3.1 Download und Installation

JUnit ist *Open-Source-Software* unter *IBM Common Public License*. Aktuelle Version und Release-Kandidaten finden Sie auf *SourceForge*:

```
http://sourceforge.net/projects/junit/
```

Entsprechende Frameworks sind auch für alle anderen heute gängigen Programmiersprachen frei erhältlich:

```
http://www.xprogramming.com/software.htm
```

Wahrscheinlich bringt Ihre Entwicklungsumgebung schon JUnit mit. Wenn Ihr Compiler sich nicht darüber beschwert, dass er die JUnit-Klassen im Klassenpfad nicht finden kann, bleibt Ihnen die Installation also erspart. Den Test dafür werden wir in zwei Minuten antreten.

junit3.8.1.zip

Die vollständige JUnit-Distribution besteht gegenwärtig aus einem ZIP-Archiv, in dem neben dem Test-Framework (in `junit.jar`) auch seine Quellen (in `src.jar`), seine Tests, einige Beispiele, die Javadoc-Dokumentation, die FAQs, ein Kochbuch und zwei Artikel beiliegen.

Machen Sie sich mit der JUnit-Distribution vertraut, lesen Sie die beigefügten Artikel und spielen Sie die Beispiele durch.

Zur Installation entpacken Sie bitte das ZIP-Archiv und übernehmen `junit.jar` in Ihren `CLASSPATH`. Fertig!

Sollte Sie JUnit jemals vor scheinbar unlösbare Probleme stellen, versuchen Sie Ihr Glück in der *JUnit Yahoo! Group*:

```
http://groups.yahoo.com/group/junit/
```

Wenn Sie in die Welt der zahlreichen JUnit-Erweiterungen eintauchen wollen, werden Sie auf der *JUnit-Website* fündig:

```
http://www.junit.org
```

3.2 Ein erstes Beispiel

Wir wollen eine Klasse `Euro` ins Leben testen, die Geldbeträge akurater repräsentieren kann als der primitive Java-Typ `double`, den wir im vorherigen Kapitel verwendet haben. Anhand dieses kleinen Beispiels können Sie den Aufbau eines Testfalls kennen lernen und Ihre ersten kleinen Erfolge mit JUnit feiern. In der JUnit-Dokumentation finden Sie ein vergleichbares Beispiel.

Eine Warnung jedoch: Weder JUnits `Money`-Klasse noch die hier entwickelte Klasse genügt den Anforderungen der Finanzwirtschaft. Wenn es auf hohe Präzision ankommt, benutzen Sie bitte `BigDecimal`. Für unseren kleinen DVD-Verleih tut es jedoch auch weniger.

Wertsemantik

Unsere Klasse `Euro` stellt Wertobjekte für geldliche Beträge dar. Das heißt, das Objekt wird eindeutig durch seinen Wert beschrieben. Im Unterschied zu einem Referenzobjekt können wir ein Wertobjekt nicht verändern. Bei einer Operation auf einem Wertobjekt erhalten wir immer ein *neues* Objekt als Ergebnis der Operation zurück. Bekanntestes Beispiel dieser Wertsemantik in Java ist wahrscheinlich die Klasse `String`.

Einen Test für eine Klasse zu schreiben bietet immer auch eine gute Gelegenheit dazu, über ihre öffentliche Schnittstelle nachzudenken. Was also erwarten wir von unserer Klasse? Nun, zunächst möchten wir sicherlich ein `Euro`-Objekt instanziieren können, indem wir dem Konstruktor einen Geldbetrag mitgeben. Wenn wir ein `Euro`-Objekt zu einem anderen hinzuaddieren, möchten wir, dass unsere Klasse mit einem neuen `Euro`-Objekt antwortet, das die Summe der beiden Beträge enthält. Eurobeträge unterliegen dabei einer besonderen Auflösung in ihrer numerischen Repräsentation. Zum Beispiel erwarten wir, dass auf den Cent genau gerundet wird und dass 100 Cents einen Euro ergeben. Aber fangen wir mit der einfachen Datenhaltung an. Hier sehen Sie den ersten Test:

Entwurf der Klassenschnittstelle

```
import junit.framework.TestCase;

public class EuroTest extends TestCase {
  public void testAmount() {
    Euro two = new Euro(2.00);
    assertTrue(2.00 == two.getAmount());
  }

  public static void main(String[] args) {
    junit.swingui.TestRunner.run(EuroTest.class);
  }
}
```

JUnit ist wirklich einfach zu verwenden und es wird nicht schwerer.

3.3 Anatomie eines Testfalls

Sie erkennen, dass wir unsere Tests *getrennt* von der Klasse `Euro` in einer Klasse namens `EuroTest` definieren. Um unsere Testklasse in JUnit einzubinden, leiten wir sie von dessen Framework-Basisklasse `TestCase` ab.

TestCase

Unser erster Testfall verbirgt sich hinter der Methode `testAmount`. Das Framework erkennt und behandelt diese Methode automatisch als Testfall, weil ihre Signatur der Konvention für Testfallmethoden folgt: `public void test...()`

In diesem Testfall erzeugen wir uns zunächst ein Objekt mit dem Wert »zwei Euro«. Der eigentliche Test erfolgt mit dem Aufruf der `assertTrue`-Methode, die unsere Testklasse aus ihrer Oberklasse erbt. Die `assertTrue`-Anweisung formuliert eine Annahme oder Forderung, die für den Code gilt und die JUnit automatisch für uns verifiziert.

Die `assertTrue`-Methode dient dazu, eine Bedingung zu testen. Als Argument akzeptiert sie einen booleschen Wert bzw. einen Ausdruck, der einen solchen liefert. Der Test ist erfolgreich, wenn die Bedingung erfüllt ist, das heißt der Ausdruck zu `true` ausgewertet werden konnte. Ist der Ausdruck dagegen `false`, protokolliert JUnit einen Fehlschlag. In unserem Beispiel testen wir, ob das Objekt `two` als Ergebnis der `getAmount`-Abfrage die erwarteten »zwei Euro« liefert.

Ältere JUnit-Tests

Wenn Ihnen Testcode begegnet, der mit der JUnit-Version 3.7 und früher geschrieben wurde, werden Sie sehen, dass jede Testklasse einen Konstruktor für den Namen des auszuführenden Testfalls benötigte. Ab JUnit 3.8 können Sie sich diesen Konstruktor sparen:

```
public class EuroTest...
  public EuroTest(String name) {
    super(name);
  }
}
```

Direkte Testausführung in Eclipse und IntelliJ IDEA

Durch Aufruf der `main`-Methode können wir unseren ersten JUnit-Test ausführen. Der `junit.swingui.TestRunner` stellt dafür eine grafische Oberfläche auf Basis von Java Swing dar. In Entwicklungsumgebungen wie Eclipse [ec] und IntelliJ IDEA [int] lassen sich Tests auch direkt ausführen, womit wir uns die `main`-Methode sparen können. Beide IDEs bringen eigene JUnit-Oberflächen mit und verfügen über eine hervorragende JUnit-Integration. Wenn Sie zwei Seiten weiterblättern, erfahren Sie, was Eclipse im Detail zu bieten hat.

Im Moment lässt sich unsere Testklasse noch nicht kompilieren. Der Compiler bemängelt, keine Klasse mit Namen `Euro` zu kennen.

3.4 Test-First

Test-Last

Unser Ziel ist es, ein Programm inkrementell in kleinen Schritten zu entwickeln. In diesem Kapitel werden Sie sehen, wie wir immer zuerst einen Test schreiben, bevor wir die Klasse weiterentwickeln, die diesen Test erfüllt. Natürlich können Sie mit JUnit aber auch Tests für schon bestehenden Code schreiben. Allerdings sind Klassen häufig schlecht testbar, wenn sie nicht von vornherein unter diesem Aspekt entworfen wurden.

Möglicherweise werden Sie bei der Lektüre dieses Kapitels denken, dass die Programmierschritte ja unglaublich winzig sind, um dann im nächsten Kapitel zu lernen, wie viel winziger unsere Schritte noch sein können. Für den Anfang wollen wir uns jedoch erst mal allein auf das Testen mit JUnit konzentrieren. Lassen Sie uns direkt den Test erfüllen.

Was müssen wir dafür tun?

```
public class Euro {
  private final double amount;

  public Euro(double amount) {
    this.amount = amount;
  }

  public double getAmount() {
    return this.amount;
  }
}
```

Wenn Sie Ihren Code kompilieren und `EuroTest` ausführen, sollte ein Fenster mit JUnits grafischer Oberfläche und darin der grün strahlende Testbalken erscheinen. Erfolg! Keinen Lärm zu machen, solange die Welt in Ordnung ist, gehört zur Philosophie von JUnit. Der Test läuft:

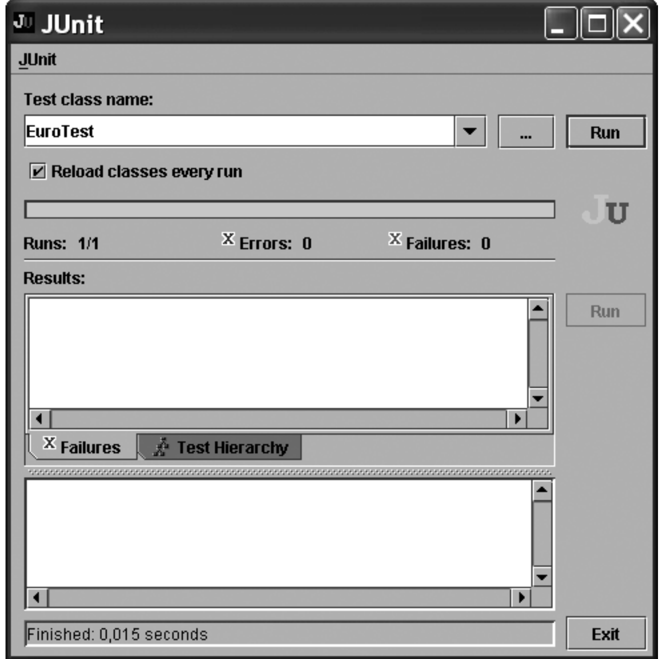


Abb. 3–1
Der legendäre grüne JUnit-Testbalken

Ein erfolgreicher Testlauf wird im Buch so am Seitenrand dargestellt:

***JUnit:** OK*

Achtung, Suchtgefahr!

Klicken Sie ruhig noch einmal auf den »Run«-Knopf. Dieser Knopf macht süchtig: Sie werden sich später ganz sicher mal dabei ertappen, die Tests zwei- oder dreimal nacheinander auszuführen, nur wegen des zusätzlichen Vertrauens, wenn hunderte von Tests ablaufen und der Fortschrittbalken dabei langsam mit Grün voll läuft.

3.5 JUnit in Eclipse

»Run As ... JUnit Test«-Kommando

In Eclipse (und ebenso IDEA) hat JUnit eine hervorragende Integration gefunden. Selektieren Sie einfach Ihre Testklasse im *Package Explorer*, öffnen Sie das »Run«-Menü, wählen Sie »Run As« und »JUnit Test«. Alternativ können Sie »Run As« auch über das Kontextmenü Ihrer Klasse im Package Explorer sowie über ein Tastaturkürzel erreichen. In jedem Fall sollte sich der JUnit-View zeigen:

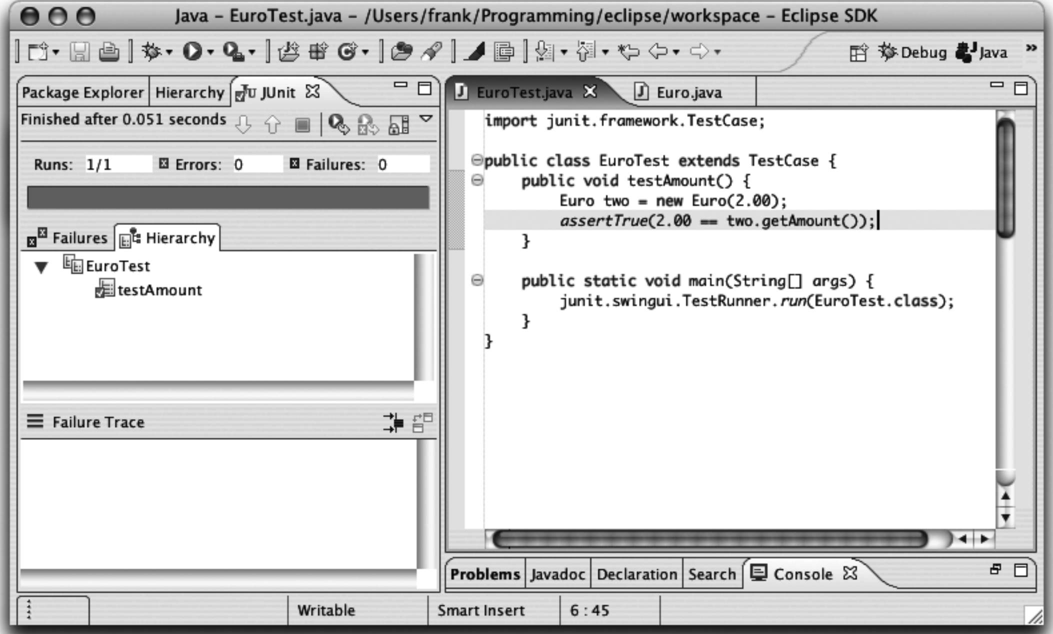


Abb. 3–2
JUnit-View in Eclipse

Für jeden weiteren Testlauf können Sie im View auf den »Run«-Knopf klicken oder auf Ihre Run-Konfiguration (das »Play«-Symbol neben dem Debugger) zurückgreifen. Sie müssen also nicht jedes Mal das »Run As«-Kommando neu bemühen.

Im Kontextmenü des Views können Sie Eclipse über die Auswahl des Menüpunktes »Activate on Error/Failure only« auch so einrichten, dass sich der View nur noch öffnet, sobald Tests fehlschlagen – aber wer will schon auf den grünen Balken verzichten?

JUnit-Klassen generieren lassen

Außerdem lohnt sich ein Blick in den »File > New«-Menüpunkt, wo Sie sich auf Knopfdruck JUnit-Klassen generieren lassen können (unter »Other...« und dann im Wizard unter »Java > JUnit« versteckt).

3.6 Das JUnit-Framework von innen

Mit unserem ersten grünen Balken im Rücken wenden wir uns nun dem Aufbau von JUnit zu. Ich empfehle Ihnen, sich den Code von JUnit einmal anzusehen. Wenn Sie so sind wie ich, dann lernen Sie ein Framework am besten, indem Sie dessen Code studieren. JUnit ist ein gutes Beispiel für ein kleines fokussiertes Framework mit einer hohen Dichte säuberlich verwendeter Entwurfsmuster. JUnit ist vor allem deshalb ein gutes Beispiel, weil es inklusive seines eigenen Testcodes kommt.

Eines guten Tages sollten Sie sich die Zeit nehmen und den `junit.framework`-Code studieren und gegebenenfalls um Ihre eigenen Anforderungen erweitern.

Der Framework-Kern besteht aus wenigen Klassen und circa tausend Zeilen Code. Ich werde im Folgenden die zentralen Klassen vorstellen, mit denen Sie am häufigsten in Berührung kommen werden. Damit Sie sehen, wie die Puzzlestücke zum Ganzen zusammengelegt werden, wenden wir uns hin und wieder unserer `Euro`-Klasse zu.

3.7 »Assert«

Wie testen wir mit JUnit?

JUnit erlaubt uns, Werte und Bedingungen zu testen, die jeweils erfüllt sein müssen, damit der Test erfolgreich durchläuft. Die Klasse `Assert` definiert dazu eine Familie spezialisierter `assert...`-Methoden, die unsere Testklassen aus JUnit erben und mit deren Hilfe wir in unseren Testfällen eine Reihe unterschiedlicher zu testender Behauptungen über unseren Code aufstellen können:

assert...-Methoden

- `assertTrue(boolean condition)` ist die allgemeinste Zusicherung. Sie verifiziert, ob eine logische Bedingung wahr ist:

  ```
  assertTrue(jungleBook.isChildrensMovie());
  ```

- `assertFalse(boolean condition)` verifiziert, ob eine Bedingung falsch ist. Diese Zusicherung ist erst seit JUnit 3.8 dabei:

  ```
  assertFalse(store.hasMovie("Buffalo 66"));
  ```

- `assertEquals(Object expected, Object actual)` verifiziert die *Gleichheit* zweier Objekte. Der Vergleich erfolgt in JUnit über die `equals`-Methode.

  ```
  assertEquals(new Euro(2.00), rental.getCharge());
  ```

Der Vorteil dieser und der noch folgenden `assertEquals`-Varianten gegenüber dem Check mit `assertTrue` liegt darin, dass JUnit uns nützliche Zusatzinformationen anbieten kann, wenn der Test tatsächlich fehlschlägt. JUnit benutzt in diesem Fall die `toString`-Repräsentation Ihres Objekts, um den erwarteten Wert darzustellen.

- `assertEquals(String expected, String actual)` verifiziert, ob zwei Zeichenketten gleich sind. Der Vergleich der `Strings` erfolgt wie gewohnt über die `equals`-Methode. Sind die `Strings` ungleich, wird nur das betreffende Segment dargestellt. Die Teile, die sich nicht unterscheiden, werden durch »...« ersetzt.

  ```
  assertEquals("Pulp Fiction", movie.getTitle());
  ```

- `assertEquals(int expected, int actual)` verifiziert, ob zwei ganze Zahlen gleich sind. Der Vergleich erfolgt für die primitiven Java-Typen über den `==`-Operator.

  ```
  assertEquals(40, xpProgrammer.workingHours());
  ```

- `assertEquals(double expected, double actual, double delta)` verifiziert, ob zwei Fließkommazahlen gleich sind. Da Fließkommazahlen nicht mit unendlicher Genauigkeit verglichen werden können, wird als drittes Argument noch eine Toleranz erwartet:

  ```
  assertEquals(3.1415, Math.PI, 1e-4);
  ```

- `assertNull(Object object)` sichert zu, dass eine Referenz `null` ist:

  ```
  assertNull(Prices.getPrice("unknown"));
  ```

- `assertNotNull(Object object)` sichert zu, dass eine Objektreferenz nicht `null` ist:

  ```
  assertNotNull(store.getMovie(1));
  ```

- `assertSame(Object expected, Object actual)` sichert die *Identität* zweier Objekte zu:

  ```
  assertSame(movie, store.getMovie(1));
  ```

- `assertNotSame(Object expected, Object actual)` sichert zu, dass zwei Objekte nicht identisch sind:

  ```
  assertNotSame(Price.REGULAR, Price.NEWRELEASE);
  ```

Die `assertEquals`-Methode ist neben den genannten Argumenttypen auch für die primitiven Datentypen `float`, `long`, `boolean`, `byte`, `char` und `short` überladen. Der Fantasie beim Testen sind somit also keine Grenzen gesetzt.

3.8 »AssertionFailedError«

Was passiert, wenn ein Test fehlschlägt?

Die im Test durch `assert...`-Anweisungen kodierten Behauptungen werden von der Klasse `Assert` automatisch verifiziert. Im Fehlerfall bricht JUnit den laufenden Testfall sofort ab und wirft den Fehler `AssertionFailedError` mit entsprechendem Protokoll aus.

Fehlschläge

Sehen wir uns dazu ein Beispiel an. Um den Test des centweisen Rundens nachzuholen, könnten wir folgenden Testfall schreiben:

```
public class EuroTest...
  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertTrue(2.00 == rounded.getAmount());
  }
}
```

Der Test läuft natürlich nicht, weil unsere Klasse noch kein Konzept für das Runden hat. Der Fortschrittbalken verfärbt sich rot:

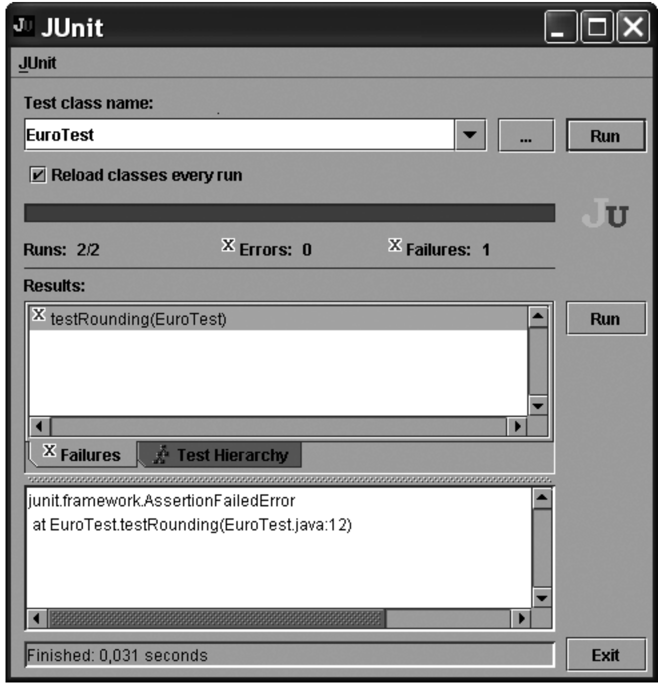


Abb. 3–3
Ein fehlschlagender Test ist ein wertvoller Test, da er uns mit wertvollen Informationen versorgt

Die JUnit-Oberfläche klärt uns sofort über den fehlschlagenden Test auf und protokolliert ihn zur Rückmeldung im unteren Textfenster.

»Der Testfall testRounding schlägt fehl«, sagt uns JUnit. »Werfen Sie einen Blick auf Zeile 12 der Klasse EuroTest.«

JUnit: *Failure*

```
junit.framework.AssertionFailedError
at EuroTest.testRounding(EuroTest.java:12)
```

Wenn Ihnen diese Fehlermeldung nicht ausreichen sollte, bietet Ihnen JUnit für alle assert...-Methoden an, optional einen Erklärungstext einzutippen, der im Fehlerfall mitprotokolliert wird. Ich rate Ihnen, diese Variante zur Dokumentation langer Testfälle auszunutzen:

```
assertTrue("amount not rounded", 2.00 == rounded.getAmount());
```

Der Unterschied liegt im ausdrucksstärkeren Begleittext, den JUnit im Fehlerfall zusätzlich ausgibt, statt nur den Ort des Fehlschlags:

JUnit: *Failure*

```
junit.framework.AssertionFailedError: amount not rounded
at EuroTest.testRounding(EuroTest.java:12)
```

Viel praktischer ist hier allerdings die assertEquals-Variante:

```
assertEquals("amount not rounded",
             2.00, rounded.getAmount(), 0.001);
```

Diese Zusicherung vergleicht erwartete und tatsächliche Werte:

JUnit: *Failure*

```
junit.framework.AssertionFailedError: amount not rounded
  expected:<2.0> but was:<1.995>
at EuroTest.testRounding(EuroTest.java:12)
```

Zum Vergleich der intern verwendeten Fließkommarepräsentation tolerieren wir in diesem Beispiel ein Delta von 0.001. Das bedeutet, dass die verglichenen Werte sich nicht um einen Betrag unterscheiden dürfen, der größer ist als ein Tausendstel Euro.

Mittlerweile wird es aber Zeit, den Test endlich einmal zu erfüllen. Was müssen wir also schreiben? Voilà!

JUnit: *OK*

```
public class Euro {
  private final long cents;

  public Euro(double euro) {
    cents = Math.round(euro * 100.0);
  }

  public double getAmount() {
    return cents / 100.0;
  }
}
```

3.9 »TestCase«

Wie gruppieren wir unsere Testfälle um eine gemeinsame Menge von Testobjekten?

Ein Testfall sieht in der Regel so aus, dass zunächst eine bestimmte Konfiguration von Objekten aufgebaut wird, gegen die der Test läuft. Diese Menge von Testobjekten wird auch als *Test-Fixture* bezeichnet. Pro Testfallmethode wird meist nur eine bestimmte Operation und oft sogar nur eine bestimmte Situation im Verhalten der Objekte getestet. Betrachten wir auch dazu ein Beispiel:

```
public class EuroTest...
  public void testAdding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", 4.00, sum.getAmount(), 0.001);
    assertEquals("two", 2.00, two.getAmount(), 0.001);
  }
}
```

In diesem Testfall erzeugen wir uns ein `Euro`-Objekt und addieren es mit sich selbst. Das Resultat dieser Addition soll ein `Euro`-Objekt sein, dessen Wert »vier Euro« beträgt. Unser ursprüngliches »zwei Euro«-Objekt soll durch die Addition nicht verändert werden. Um diesen Test zu erfüllen, verpassen wir unserer Klasse einen *privaten* Konstruktor:

JUnit: *OK*

```
public class Euro...
  private Euro(long cents) {
    this.cents = cents;
  }

  public Euro add(Euro other) {
    return new Euro(this.cents + other.cents);
  }
}
```

Der JUnit-Balken ist zurück auf Grün – doch was würde passieren, wenn wir hier ein `Euro`-Objekt mit negativem Betrag definiert hätten? Wäre damit noch die Intention des Addierens ausgedrückt oder sollte es dann besser `subtract` heißen? Verschieben wir die Antwort auf diese Frage für ein paar Momente. Damit die Idee aber nicht verloren geht, notieren wir uns »Wie geht `add()` mit negativen Beträgen um?«

Memo:
– add(): negative Beträge?

Halten Sie Ideen sofort fest, verlieren Sie sie nicht wieder.

Wir sind dabei stehen geblieben, eine geeignete Test-Fixture herauszuziehen. Werfen wir dazu noch einen Blick auf unsere drei Testfälle:

```
public class EuroTest...
  public void testAmount() {
    Euro two = new Euro(2.00);
    assertTrue(2.00 == two.getAmount());
  }

  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertEquals("amount not rounded",
                 2.00, rounded.getAmount(), 0.001);
  }

  public void testAdding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", 4.00, sum.getAmount(), 0.001);
    assertEquals("two", 2.00, two.getAmount(), 0.001);
  }
}
```

Da sich der Code zum Aufbau einer bestimmten Testumgebung häufig wiederholt, sollten Sie alle Testfälle, die gegen die gleiche Menge von Objekten laufen, unter dem Dach einer Testklasse zusammenfassen. Im ersten und dritten Testfall verwenden wir beispielsweise jeweils ein »zwei Euro«-Objekt. Wir könnten damit anfangen, dieses Objekt in die Test-Fixture zu übernehmen. Damit würden wir die duplizierten Testobjekte in einer gemeinsamen Testumgebung zusammenfassen und könnten so bestehende Testfälle vereinfachen. Der zweite Testfall hat keine Verwendung für unsere »zwei Euro«. Wir notieren uns also »`EuroTest: testRounding()` refaktorisieren« und kommen später auf die Notiz zurück.

Memo:
– add(): negative Beträge?
– EuroTest: testRounding() refaktorisieren

Allgemein gilt, dass alle Testfälle einer Testklasse von der gemeinsamen Fixture Gebrauch machen sollten. Hat eine Testfallmethode keine Verwendung für die Fixture-Objekte, so ist dies meist ein gutes Indiz dafür, dass die Methode auf eine andere Testklasse verschoben werden will. Generell sollten Testklassen um die Fixture organisiert werden, nicht um die getestete Klasse. Somit kann es auch durchaus vorkommen, dass zu einer Klasse mehrere korrespondierende Testfallklassen existieren, von denen jede ihre individuelle Fixture besitzt.

Testklasse per Fixture

Organisieren Sie Testklassen um eine gemeinsame Fixture von Testobjekten, nicht um die getestete Klasse.

Damit fehlerhafte Testfälle nicht andere Testfälle beeinflussen können, läuft jeder Testfall im Kontext seiner eigenen Fixture. Eine Test-Fixture erhalten Sie auf folgende Weise:

- Erklären Sie die Objekte Ihrer Testumgebung zu Instanzvariablen.
- Initialisieren Sie diese Variablen in der `setUp`-Methode, um so eine definierte Testumgebung zu schaffen.
- Geben Sie wertvolle Testressourcen wie zum Beispiel Datenbank- oder Netzwerkverbindungen in der `tearDown`-Methode wieder frei.

setUp()

tearDown()

JUnit behandelt Ihre Testklasse dann wie folgt:

- `setUp` wird gerufen, bevor ein Testfall ausgeführt wird.
- `tearDown` wird gerufen, nachdem ein Testfall ausgeführt wurde.

Hier sehen Sie unsere Testklasse mit extrahierter Test-Fixture:

***JUnit:** OK*

```
public class EuroTest...
  private Euro two;

  protected void setUp() {
    two = new Euro(2.00);
  }

  public void testAmount() {
    Euro two = new Euro(2.00);
    assertTrue(2.00 == two.getAmount());
  }

  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertEquals("amount not rounded",
                 2.00, rounded.getAmount(), 0.001);
  }

  public void testAdding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", 4.00, sum.getAmount(), 0.001);
    assertEquals("two", 2.00, two.getAmount(), 0.001);
  }
}
```

Bitte beachten Sie, dass Sie Fixture-Variablen im `setUp` initialisieren, nicht etwa im Deklarationsteil oder im Konstruktor der Testfallklasse. Hätten wir in diesem Beispiel wertvolle Testressourcen freizugeben, würden wir symmetrisch zum `setUp` auch `tearDown` definieren.

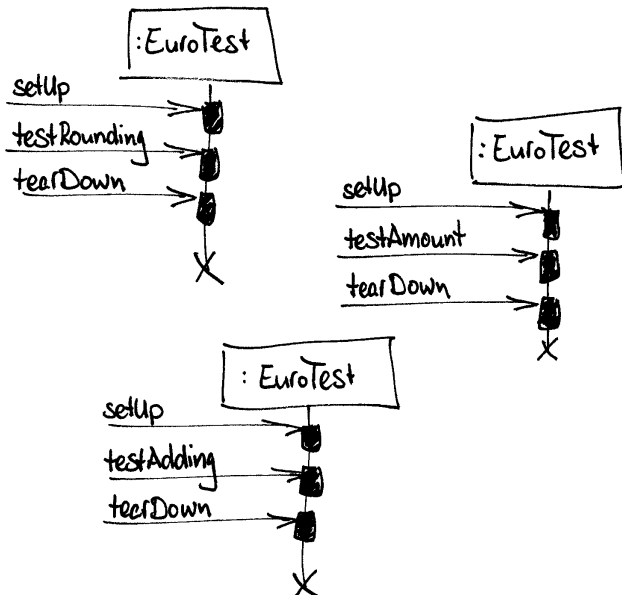
3.10 Lebenszyklus eines Testfalls

Wie führt JUnit die Tests unserer Klasse aus? Im Prinzip unterscheidet JUnit zwischen zwei Phasen:

1. **Testfallerzeugung:** Das Framework durchsucht Ihre Testklasse nach `test...`-Methoden und erzeugt für sie jeweils ein eigenes Objekt der Klasse.
2. **Testlauf:** JUnit führt die gesammelten Testfälle voneinander isoliert aus. Die Reihenfolge, in der Testfälle vom Framework ausgeführt werden, ist dabei prinzipiell undefiniert.

Damit zwischen einzelnen Testfällen keinerlei Seiteneffekte entstehen, erzeugt JUnit für jeden Testfall ein neues Exemplar Ihrer Testklasse. Das ist überaus wichtig zu verstehen. Ins Bild gesetzt, sieht es so aus:

Abb. 3–4
Für jede Testfallmethode existiert ein Testfallobjekt



Testlaufsequenz

Ihre Testfallklasse resultiert in so vielen Testfallobjekten, wie sie Testfallmethoden enthält. Auf diesen Objekten erfolgt dann der Testlauf:

1. Vor der Ausführung eines Testfalls wird jeweils erst die `setUp`-Methode ausgeführt, sofern diese redefiniert wurde.
2. Anschließend wird die den auszuführenden Testfall betreffende `test...`-Methode gerufen.
3. Abschließend wird noch die `tearDown`-Methode ausgeführt, falls diese in der Testklasse redefiniert wurde.

Es sollte klar geworden sein, dass Ihre `TestCase`-Klasse wirklich eine *Sammlung* von Testfällen darstellt, denn erst die instanziierten Objekte dieser Klasse repräsentieren jeweils einen Testfall. Für JUnit sind diese Testfälle vollkommen unabhängig voneinander: Sie besitzen keinen inhärenten Zustand, bauen sich selbst ihre Testumgebung auf, räumen hinter sich selbst auf und können in beliebiger Reihenfolge laufen.

Zum effektiven Testen müssen Testfälle isoliert voneinander ausführbar sein. Treffen Sie deshalb keine Annahmen über ihre Reihenfolge im Testlauf. Führen Sie voneinander abhängige Tests stattdessen gemeinsam in einem Testfall aus.

Damit sind wir an einer Stelle angekommen, an der es sich lohnt, JUnit im größeren Bild zu betrachten:

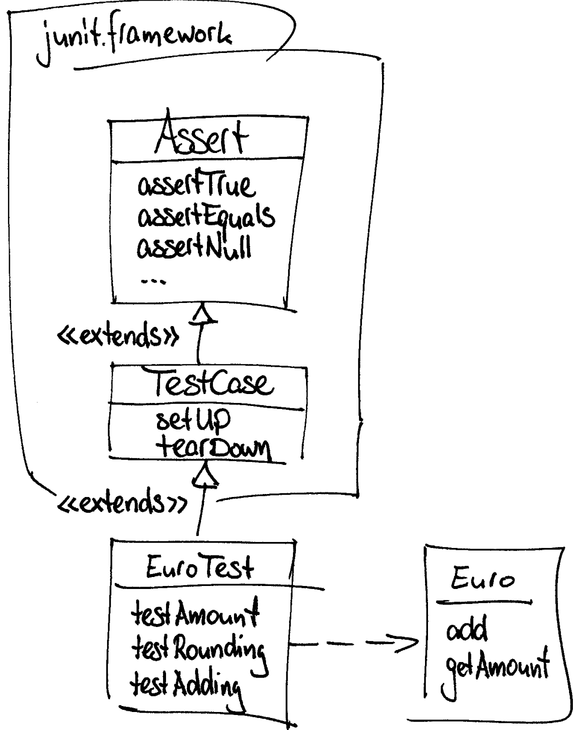


Abb. 3–5
Vererbungshierarchie von `Assert` und `TestCase`

- `Assert` testet Werte und Bedingungen.
- `TestCase` isoliert eine Reihe von Testfällen um eine gemeinsame Testumgebung.
- `EuroTest` testet das Verhalten unserer entwickelten `Euro`-Klasse.

3.11 »TestSuite«

Wie führen wir eine Reihe von Tests zusammen aus?

Unser Ziel ist es, den gesamten Testprozess so weit zu automatisieren, dass wir den Test ohne manuellen Eingriff wiederholt durchführen können. Wichtig ist schließlich, die Unit Tests möglichst häufig auszuführen, idealerweise nach jedem Kompilieren. Wenn wir dann immer nur winzig kleine Schritte nehmen, werden wir uns nie länger als wenige Minuten mit Programmieren aufhalten, ohne grünes Licht zum Weiterprogrammieren einzuholen. Eher selten wollen wir dabei nur einzelne Testfälle ausführen. Meistens wollen wir alle gesammelten Unit Tests in einem Testlauf zusammen ausführen, um so ungewollten Seiteneffekten möglichst frühzeitig zu begegnen.

Führen Sie nach jedem erfolgreichen Kompiliervorgang alle gesammelten Unit Tests aus.

suite()

AllTests

Mit JUnit können wir beliebig viele Tests in einer Testsuite zusammenfassen und gemeinsam ausführen. Dazu verlangt das Framework von uns, dass wir in einer statischen `suite`-Methode definieren, welche Tests zusammen ausgeführt werden sollen. Eine Suite von Tests wird durch ein `TestSuite`-Objekt definiert, dem wir beliebig viele Testfälle und selbst weitere Testsuiten hinzufügen können. Auf welcher Klasse Sie diese `suite`-Methode definieren, ist nebensächlich. In den meisten Fällen werden Sie jedoch spezielle `AllTests`-Klassen definieren wollen, um Ihre Testsuiten zu repräsentieren.

Wollen wir beispielsweise alle bisher geschriebenen Testfälle in einer Testsuite organisieren, muss unsere Testsuiteklasse die folgende Gestalt haben:

```
import junit.framework.Test;
import junit.framework.TestSuite;

public class AllTests {
  public static Test suite() {
    TestSuite suite = new TestSuite();

    suite.addTestSuite(CustomerTest.class);
    suite.addTestSuite(EuroTest.class);
    suite.addTestSuite(MovieTest.class);

    return suite;
  }
}
```

Um eine Testsuite zu erhalten, definieren wir ein TestSuite-Exemplar und fügen diesem mithilfe der addTestSuite-Methode die gewünschten Testfallklassen hinzu. Für jede Testfallklasse wird dabei implizit eine Testsuite definiert, in der alle Testfallmethoden eingebunden werden, die in der betreffenden Klasse definiert sind.

Test

Sie werden vielleicht über den Rückgabetyp der suite-Methode verwundert sein: Warum Test und nicht TestSuite? Hier begegnen wir erstmalig dem Test-Interface, das sowohl von der TestCase- als auch von der TestSuite-Klasse implementiert wird. Dieses Entwurfsmuster ist als *Composite* [ga95] bekannt geworden. Das Muster erlaubt uns, beliebig viele TestCase- und TestSuite-Objekte zu einer umfassenden Testsuite-Hierarchie zu kombinieren:

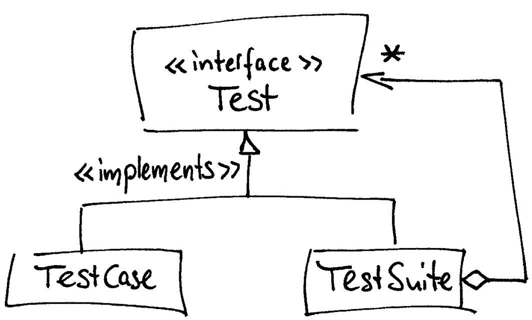


Abb. 3–6
Teil-und-Ganzes-Hierarchie von TestCase und TestSuite

- Test abstrahiert von Testfällen und Testsuiten.
- TestSuite führt eine Reihe von Tests zusammen aus.

Momentan liegen alle Klassen einfach im Default-Package. In der Regel verteilen wir unsere Klassen natürlich über verschiedene Pakete und müssen das Package beim Bündeln der Testsuite mit spezifizieren.

In den meisten Fällen ist es ohnehin praktisch, pro Package eine Testsuiteklasse zu definieren. Nach diesem Muster können Sie dann auch Hierarchien von Hierarchien von Testsuiten bilden:

```
suite.addTest(other.AllTests.suite());
```

Unsere main-Methode verschieben wir von der EuroTest-Testfallklasse in die AllTests-Testsuiteklasse, da sie hier viel besser aufgehoben ist:

```
public class AllTests...
  public static void main(String[] args) {
    junit.swingui.TestRunner.run(AllTests.class);
  }
}
```

3.12 »TestRunner«

Wie automatisieren wir den Test?

In einigen Entwicklungsumgebungen ist die JUnit-Integration so eng, dass Tests direkt ausgeführt werden können. Bietet Ihre IDE das nicht, können Sie zur Ausführung der Tests zwischen diesen drei Alternativen wählen:

- `junit.swingui.TestRunner` mit grafischer Swing-Oberfläche
- `junit.awtui.TestRunner` auf einfacher Basis des Java AWT
- `junit.textui.TestRunner` als Batch-Job auf der Textkonsole

Beide grafischen Varianten bieten den bekannten Fortschrittbalken und sind sich sehr ähnlich. Zur Ausführung können Sie Testfallklassen und Testsuiteklassen mit vollklassifiziertem Namen eingeben und unter Swing auch aus dem Klassenpfad oder Ihrer Historie auswählen.

Die GUI-Oberflächen haben gegenüber der textuellen Darstellung den Vorteil, dass sie modifizierte Klassen automatisch neu nachladen.

ClassLoader-Problem

In unerfreulichen Fällen kollidiert JUnits eigener Klassenlader jedoch mit anderen und Sie müssen das Laden über die Checkbox »reload classes every run« unterbinden oder sich in der JUnit-Doku anlesen, was über die `excluded.properties`-Konfiguration geschrieben steht.

Im mittleren Fenster werden die fehlschlagenden Tests aufgelistet, die Sie nach Mausklick im unteren Fenster mit Fehlermeldung und Stacktrace unter die Lupe nehmen können. Unter Swing können Sie im mittleren Fenster zusätzlich den Testbaum öffnen und dann Testfälle und Testsuiten daraus auch einzeln ausführen.

Die rein textuelle Variante bietet eine Alternative zur grafischen Oberfläche, die besonders für den Testlauf zum Integrationszeitpunkt oder für einen nächtlichen Build geeignet ist:

```
public class AllTests...
  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
}
```

Die Benutzungsoberfläche sieht entsprechend sparsam aus:

```
........
Time: 0,06

OK (8 tests)
```

Für jeden erfolgreichen Test tickt ein weiterer Punkt auf die Konsole. Fehlschläge dagegen hinterlassen weniger freundlich ihren Stacktrace.

3.13 Zwei Methoden, die das Testen vereinfachen

Abschließend wollen wir auf eine der beiden Fragen zurückkommen, die wir im Zuge unserer Programmierepisode offen gelassen haben, um den angefangenen Schritt zu Ende zu führen.

Erinnern Sie sich noch an unseren Notizzetteleintrag »`EuroTest: testRounding()` refaktorisieren«? Wir hatten die Absicht, uns erneut den Testfall `testRounding` vorzuknöpfen, da er keinerlei Verwendung für die »zwei Euro« aus der Test-Fixture hatte. Ist das nun wirklich ein Anzeichen dafür, dass wir eine neue Testfallklasse abspalten sollten? Nein, wir können noch guten Gebrauch von den »zwei Euro« machen, wenn wir den Testfall umschreiben:

Memo:
– add(): negative Beträge?
– ~~EuroTest: testRounding() refaktorisieren~~

```
public class EuroTest...
  public void testRounding() {
    Euro rounded = new Euro(1.995);
    assertEquals("amount not rounded", two, rounded);
  }
}
```

Für ein richtiges Wertobjekt gehört es sich ohnehin, dass wir zwei Objekte mittels der `equals`-Methode auf Gleichheit testen können. Lassen Sie uns diese Intention deshalb auch in unserem restlichen Code ausdrücken:

```
public class EuroTest...
  public void testAdding() {
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Aber warten Sie! Unsere Tests fangen an, sich lautstark zu beschweren:

JUnit: *Failure*

```
junit.framework.AssertionFailedError:
  sum expected:<Euro@8b456289> but was:<Euro@8b496289>
at EuroTest.testAdding(EuroTest.java:23)

junit.framework.AssertionFailedError: amount not rounded
  expected:<Euro@18d18634> but was:<Euro@19318634>
at EuroTest.testRounding(EuroTest.java:18)
```

Was geht hier vor? Die Ausgabe der Objektreferenzen, die Java hier standardmäßig als Hashcode ausgibt, den Java wiederum als Resultat der `toString`-Methode liefert, hilft uns nicht weiter.

toString()

Überschreiben wir also einfach das Standardverhalten mit einer ausdrucksstärkeren String-Repräsentation:

```
public class Euro...
  public String toString() {
    return "EUR " + getAmount();
  }
}
```

Ungetesteter Code? Tja, diese Methode ist so einfach, dass ein Test daran meines Erachtens verschwendet wäre. Die Methode getAmount hat ja bereits einen Test und mehr kann hier im schlimmsten Fall nicht schief gehen.

Schreiben Sie nur Tests für solchen Code, der unter Umständen fehlschlagen könnte.

Testen wir jetzt, erhalten wir folgende Rückmeldung:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  sum expected:<EUR 4.0> but was:<EUR 4.0>
at EuroTest.testAdding(EuroTest.java:23)

junit.framework.AssertionFailedError: amount not rounded
  expected:<EUR 2.0> but was:<EUR 2.0>
at EuroTest.testRounding(EuroTest.java:18)
```

Hmm, das wird ja immer seltsamer! Wissen Sie, was jetzt los ist?

Die assertEquals-Anweisung in den Tests greift intern auf die equals-Methode unseres Objekts zurück und die wiederum vergleicht zwei Objekte standardgemäß anhand ihrer Referenzen.

Damit unser Euro-Objekt als wirkliches Wertobjekt und nicht etwa als Referenzobjekt behandelt wird, müssen wir die equals-Methode überschreiben. Doch dafür schreiben wir zunächst wieder einen Test, denn hier will der *Equals-Vertrag* sichergestellt werden:

```
public class EuroTest...
  public void testEquality() {
    assertEquals(two, two);
    assertEquals(two, new Euro(2.00));
    assertEquals(new Euro(2.00), two);

    assertFalse(two.equals(new Euro(7.00)));
    assertFalse(two.equals(null));
    assertFalse(two.equals(new Object()));
  }
}
```

Die Implementierung der `equals`-Methode sieht so aus:

equals()

***JUnit:** OK*

```
public class Euro...
  public boolean equals(Object o) {
    if (o == null || !o.getClass().equals(this.getClass())) {
      return false;
    }

    Euro other = (Euro) o;
    return this.cents == other.cents;
  }
}
```

Unsere Tests laufen wieder.

Noch einmal nachdenken: Der *Hashcode-Vertrag* erwartet von uns, dass wir mit `equals` auch die `hashCode`-Methode redefinieren:

```
public class Euro...
  public int hashCode() {
    return (int) cents;
  }
}
```

Wenn unser Objekt niemals mit `HashMap` oder `HashSet` verwendet wird, könnten wir hier sogar eine `UnsupportedOperationException` werfen, doch der Hashcode ist in diesem einfachen Fall schneller getippt.

Wir schreiben hier zusätzliche Methoden, nur um unsere Klassen zu testen. Das ist interessanterweise nicht mal schlimm, wenn es unsere Klassen dahin lenkt, dass sie dadurch auch einfacher testbar sind.

Spendieren Sie Ihren Klassen zusätzliche öffentliche Methoden, wenn sie dadurch leichter getestet werden können.

Durch die Verwendung von `assertEquals` wandert hier lediglich ein Teil der Verantwortlichkeiten aus dem Testcode in den funktionalen Code. Das bedeutet, der Anwendungscode nimmt Komplexität auf, damit der Testcode Komplexität abgeben kann.

Wir haben jetzt schon neun Tests(!). Sicherlich können wir immer mehr Tests schreiben. Nach oben gibt es keine Grenzen. Wichtig ist nur, im Auge zu behalten, welche Tests ihren Aufwand wirklich wert sind und welche nicht.

Beobachten Sie Ihren Return-on-Investment. Schreiben Sie nur Tests, die wertvoll sind und aus denen Sie auch etwas lernen.

3.14 Testen von Exceptions

Wie testen wir, ob eine Exception wie erwartet geworfen wird?

Memo:
~~– add(): negative Beträge?~~

Ein anderer offener Punkt unserer Randnotizen war, »Wie geht `add()` mit negativen Beträgen um?« Sicherlich ist es schon gar nicht sinnvoll, negative Beträge im Konstruktor zu erlauben. Wir sollten ihm deshalb eine *Guard Clause* [be96] spendieren und entsprechend eine Exception werfen:

JUnit: *OK*

```
public class EuroTest...
  public void testNegativeAmount() {
    try {
      final double NEGATIVE_AMOUNT = -2.00;
      new Euro(NEGATIVE_AMOUNT);
      fail("amount must not be negative");
    } catch (IllegalArgumentException expected) {
    }
  }
}
```

Das Muster von JUnit, um erwartete Exceptions und Fehler zu testen, ist denkbar einfach, nachdem es einmal klar geworden ist:

Wir versuchen, ein `Euro`-Exemplar mit negativem Wert zu bilden. Was wir im Fall negativer Eurobeträge erwarten würden, wäre eine `IllegalArgumentException`. Wenn der Konstruktor diese Exception also auslöst, kommt der `catch`-Block zum Tragen. Der Variablenname `expected` drückt schon aus, dass wir eine `IllegalArgumentException` erwarten. Wir fangen diese Exception und alles ist gut.

Wird die Exception dagegen nicht geworfen, läuft der Test zu weit, die `fail`-Anweisung wird ausgeführt und JUnit protokolliert unsere Fehlermeldung in einem `AssertionFailedError`-Objekt.

fail()

Die `fail`-Methode wird von JUnits `Assert`-Klasse realisiert und wird verwendet, um einen Testfall durch Fehlschlag abzubrechen.

Um obigen Testfall zu erfüllen, fügen wir eine Guard Clause im Konstruktor ein:

```
public class Euro...
  public Euro(double euro) {
    if (euro < 0)
      throw new IllegalArgumentException("negative amount");

    cents = Math.round(euro * 100.0);
  }
}
```

3.15 Unerwartete Exceptions

Was passiert mit Exceptions, die uns im Test nicht interessieren?

JUnit unterscheidet zwischen einem *Failure* und einem *Error*:

- ***Failure* kennzeichnet eine fehlschlagende Zusicherung**, die wir im Test sicherstellen wollen.
- ***Error* kennzeichnet ein unerwartetes Problem** wie zum Beispiel eine `NullPointerException`, die wir auch nicht behandeln wollen.

Wenn *Unchecked Exceptions* wie `RuntimeException`, `Error` und ihre Untertypen in eine `setUp`-, `test...`- oder `tearDown`-Methode gelangen, werden sie von JUnit gefangen und protokolliert.

Checked Exceptions müssen Sie in Ihrem Testcode nicht fangen, sondern können sie direkt an das Framework weiterreichen, indem Sie die betreffende Methode um eine `throws Exception`-Klausel erweitern.

Lassen Sie unerwartete Exceptions einfach von JUnit fangen, denn auch diese stellen Fehler dar.

Das Handling von Exceptions geschieht durch JUnit dann wie folgt:

- Tritt im `setUp` ein Fehler auf, wird sowohl der Testfall als auch `tearDown` nicht mehr ausgeführt.
- Tritt in der `test...`-Methode ein Fehler auf, wird trotzdem noch `tearDown` ausgeführt.

Das bedeutet, dass die Testumgebung, die Sie im `setUp` aufgebaut haben, in `tearDown` selbst dann noch ordnungsgemäß abgerissen wird, wenn der Test schief geht. Testressourcen werden somit auf jeden Fall wieder freigegeben, es sei denn, `setUp` schlägt schon fehl.

Woran erkennt man, dass etwas testgetrieben entwickelt wurde?

von Johannes Link, andrena objects ag

Daran, dass es uns auch leicht fällt, darauf aufbauende Software in kleinen isolierten Units zu testen. Negativbeispiele gibt es ausreichend. Beinahe alle komplexeren Java-APIs widersetzen sich dem Testen mehr oder weniger hartnäckig. Mit unseren selbst geschriebenen Bibliotheken und Frameworks haben wir das Problem praktisch nie. Wenn doch, dann zeigt sich bei näherem Hinsehen, dass es sich um eine Stelle handelt, die selbst nur sehr oberflächlich getestet wurde (und damit nicht wirklich test-first entwickelt wurde).

3.16 JUnit 4

Kurz vor Drucklegung dieses Buches steht JUnit in Version 4 knapp vor der Freigabe. Das neue Release soll die Einstiegshürde zum Testen weiter senken und dieses Versprechen wird mit Sicherheit gehalten. Das einfachste Test-Framework der Welt ist noch einfacher geworden.

... für Java 1.5

Da sich dessen Neuerungen jedoch komplett auf Java 1.5 beziehen und viele Projekte noch eine kleine Weile mit Java 1.4 auskommen müssen, gehe ich auf das neue JUnit nur in diesem Abschnitt ein und verwende ansonsten die Version 3.8.1. Wenn Sie später auf JUnit 4 umsatteln, können Sie Ihre alten JUnit-Tests wie gewohnt weiter benutzen.

Mit dem Versionssprung haben Kent Beck und Erich Gamma nach mittlerweile sieben Jahren zum ersten Mal die Architektur ihres Test-Frameworks grundlegend geändert und alle Story-Karten auf die mit

Annotationen

Java 1.5 eingeführten *Annotationen* gesetzt. Annotationen sind ein neues Ausdrucksmittel der Metaprogrammierung. Per Annotation können Sie Code mit frei wählbaren Anmerkungen versehen und auf die so markierten Codeelemente über den Reflection-Mechanismus später wieder zugreifen. In JUnit 4 wird dieses Sprachkonstrukt nun dazu verwendet, jede x-beliebige Methode jeder x-beliebigen Klasse als ausführbaren Testfall kennzeichnen zu können. Hier ist ein Test der neuen Schule:

```
import junit.framework.TestCase;

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

public class EuroTest extends TestCase {
  @Test public void testadding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Sie erkennen, dass die Namenskonvention `public void test...()` wie auch das Ableiten der Klasse `TestCase` der Vergangenheit angehören. Sie kleben künftig einfach eine `@Test`-Annotation an Ihre Testfälle und können Ihre Methoden nennen, wie es Ihnen gerade gefällt. Betrachten wir aber die Neuigkeiten doch einmal mit dem Vergrößerungsglas – behalten Sie dabei jedoch im Hinterkopf, dass Unterschiede zwischen dieser Vorschau und dem tatsächlichen JUnit 4-Release möglich sind.

Mit JUnit 4 kommt ein neuer Namensraum: Im Package `org.junit` steckt der neue annotationsbasierte Code. Das `junit.framework`-Paket bleibt so weit bestehen und hat lediglich kleine Änderungen erfahren, um die Aufwärtskompatibilität herzustellen. Für Tests, die der neuen Schule folgen, benötigen wir von dem alten Zeugs jedoch nichts mehr:

org.junit-Package

```
~~import junit.framework.TestCase;~~
```

Stattdessen importieren wir jetzt die `@Test`-Annotation und Methoden der `Assert`-Klasse aus dem neuen JUnit-Package:

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

Falls Sie mit den neuen Sprachkonstrukten noch nicht vertraut sind: Ab Java 1.5 können Sie mithilfe von *statischen Imports* die statischen Methoden einer anderen Klasse einfach in den Namensraum Ihrer eigenen Klasse einblenden. Mit der Zeile:

Statischer Import der assert...-Methoden

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

... könnten wir beispielsweise die `assertEquals`-Methode importieren, als wäre sie eine `static`-Methode unserer Klasse. Der oben verwendete *-Joker holt einfach alle statischen Methoden auf einen Schwung.

Als Nächstes ist augenfällig, dass unsere Klasse nicht mehr von der Klasse `TestCase` abgeleitet ist. Ab sofort können Sie Ihre Tests nämlich in jede beliebige Klasse stecken. Die einzige Bedingung ist: Ihre Klasse muss über einen öffentlichen Defaultkonstruktor instanziierbar sein:

```
public class EuroTest ~~extends TestCase~~ {
```

Welche Methoden als Testfälle auszuführen sind, markieren wir jetzt mithilfe der `@Test`-Annotation. Den Klammeraffen nicht vergessen! Welche Namen Sie Ihren Methoden geben, ist egal. Das `test...`-Präfix können Sie als Zeichen alter Tradition oder aus guter Konvention beibehalten oder es auch lassen. Einzige Bedingung: Ihre Methode muss öffentlich sein, darf keine Parameter und keinen Rückgabewert haben:

@Test-Annotation

```
  @Test public void ~~test~~adding() {
    Euro two = new Euro(2.00);
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Das ist, auf einer Seite zusammengefasst, was sich grob geändert hat.

JUnit 4 führt sechs unterschiedliche Annotationen ein:

- `@Test` kennzeichnet Methoden als *ausführbare* Testfälle.
- `@Before` und `@After` markieren Setup- bzw. Teardown-Aufgaben, die für *jeden* Testfall wiederholt werden sollen.
- `@BeforeClass` und `@AfterClass` markieren Setup- bzw. Teardown-Aufgaben, die nur *einmal* pro Testklasse ausgeführt werden sollen.
- `@Ignore` kennzeichnet temporär *nicht auszuführende* Testfälle.

@Before und @After

`setUp`- und `tearDown`-Methoden werden wie Testfälle via Annotation gekennzeichnet:

- `@Before`-Methoden werden vor jedem Testfall ausgeführt,
- `@After`-Methoden nach jedem Testfall.

Auch diese Methoden können beliebige Namen tragen, müssen nun aber *öffentlich* zugänglich sein, parameterlos und ohne Rückgabewert. Der Fixture-Aufbau erfolgt auf dem gewohnten Weg:

```
import org.junit.Before;

public class EuroTest {
  private Euro two;

  @Before public void setUp() {
    two = new Euro(2.00);
  }

  @Test public void amount() {
    assertEquals(2.00, two.getAmount(), 0.001);
  }

  @Test public void adding() {
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Neu ist, dass auch mehrere `@Before`- und `@After`-Methoden pro Klasse laufen können. Eine bestimmte Ausführungsreihenfolge wird dabei jedoch nicht zugesagt. Vererbte und nicht überschriebene Methoden werden in symmetrischer Weise geschachtelt gerufen:

- `@Before`-Methoden der Oberklasse vor denen der Unterklasse,
- `@After`-Methoden der Unterklasse vor denen der Oberklasse.

@BeforeClass und @AfterClass

Für kostspieligere Test-Setups, die nicht für jeden einzelnen Testfall neu aufgebaut und danach gleich wieder abgerissen werden können, existieren zwei weitere Annotationen:

- `@BeforeClass` läuft für jede Testklasse nur ein einziges Mal und noch vor allen `@Before`-Methoden,
- `@AfterClass` entsprechend für jede Testklasse nur einmal und zwar nach allen `@After`-Methoden.

Mehrere `@BeforeClass`- und `@AfterClass`-Annotationen pro Klasse sind zulässig. Die so markierten Methoden müssen aber *statisch* sein:

```
import org.junit.BeforeClass;
import org.junit.AfterClass;

public class EuroTest...
  @BeforeClass public static void veryExpensiveSetup() { ... }
  @AfterClass public static void releaseAllResources() { ... }
}
```

Die `@BeforeClass`-Methoden einer Oberklasse würden entsprechend noch vorher ausgeführt, ihre `@AfterClass`-Methoden anschließend.

Erwartete Exceptions

Zum Testen von Exceptions können Sie der `@Test`-Annotation über ihren optionalen Parameter `expected` mitteilen, dass die Ausführung Ihres Testfalls zu einer gezielten Exception führen soll:

```
public class EuroTest...
  @Test(expected = IllegalArgumentException.class)
  public void negativeAmount() {
    final double NEGATIVE_AMOUNT = -2.00;
    new Euro(NEGATIVE_AMOUNT); // should throw the exception
  }
}
```

Wird keine Exception geworfen oder eine Exception anderen Typs, schlägt dieser Testfall eben fehl.

Wenn Sie das Exception-Objekt in dem Test noch weiter unter die Lupe nehmen wollen, sollten Sie den altbekannten Weg über den `try/catch`-Block nehmen. Das Gleiche gilt, wenn Sie zusichern wollen, dass sich der Prüfling nach einer Exception noch in einem konsistenten Zustand befindet. Ansonsten ist diese Annotation sehr elegant.

Timeouts

Ein für Performanztests interessanter optionaler Parameter ist `timeout`. Geben Sie Ihrem Testfall mit auf die Reise, in welcher Zeitspanne von Millisekunden er laufen sollte. Überschreitet er sein Zeitlimit, wird er zwecks Fehlschlags *abgebrochen*:

```
public class EuroTest...
  @Test(timeout = 100)
  public void performanceTest() { ... }
}
```

@Ignore

Wenn Sie einen Test *kurzzeitig* außer Gefecht setzen wollen, können Sie das folgendermaßen tun:

```
import org.junit.Ignore;

public class EuroTest...
  @Ignore("not today")
  @Test(timeout = 100)
  public void performanceTest() { ... }
}
```

Der `@Ignore`-Kommentar darf, sollte aber niemals fehlen! Der Test wird dann im Testlauf unter Protokollierung dieses Textes übergangen. Sorgen Sie jedoch dafür, dass ignorierte Tests schnellstens auf Grün kommen und im besten Fall gar nicht erst eingecheckt werden können!

Neue Tests mit altem Runner ausführen

Damit die existierenden und zum Teil in die Entwicklungsumgebungen direkt integrierten `TestRunner` unsere neuen Tests ausführen können, müssen wir einen kleinen Kunstgriff vornehmen:

```
import junit.framework.JUnit4TestAdapter;

public class EuroTest...
  public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(EuroTest.class);
  }
}
```

Etwas unschön muss uns die gute alte `suite`-Methode als Adapterfunktion herhalten, zumindest bis unsere Werkzeuge die Brücke zur neuen annotationsbasierten Form geschlagen haben.

Alte Tests mit neuem Runner ausführen

JUnit 4 liefert nur noch einen textuellen Runner mit. Die grafische Integration wird konsequent den Werkzeugherstellern überlassen. Ebenso wandert die Organisation von Testfällen zu den Entwicklungswerkzeugen über. Der neue Runner `JUnitCore` akzeptiert tatsächlich nur noch ein flaches Array von Testklassen:

```
import org.junit.runner.JUnitCore;

public class AllTests {
  public static void main(String[] args) {
    JUnitCore.run(CustomerTest.class,
                  EuroTest.class,
                  MovieTest.class);
  }
}
```

Testfallerzeugung und Testlauf

Erwähnenswert ist noch, dass JUnit 4 nicht mehr in zwei Phasen läuft: Es werden also nicht erst alle Testfallobjekte auf Halde erzeugt und dann ausgeführt. Die Erzeugung erfolgt *just-in-time* zum Testlauf.

Schlüsselwort assert

Wenn Sie mögen, können Sie in Ihren Tests jetzt auch das Schlüsselwort `assert` aus Java 1.4 verwenden. Sie müssen nur daran denken, die Zusicherungen zur Laufzeit auch mit der Option `-ea` zu aktivieren. JUnit ist dazu intern zur Verwendung von `AssertionError` gewechselt, was auch dazu geführt hat, dass nicht mehr zwischen möglichen und unerwarteten Fehlschlägen, *Failures* und *Errors*, unterschieden wird. Fehler sind Fehler sind Fehler!

Subtile Unterschiede existieren zwischen altem und neuem `Assert`: `junit.framework.Assert` vs. `org.junit.Assert`. Unter Java 1.5 sollten Sie nur noch die neue Klasse einsetzen. Da Arrays untereinander nun auch über `equals` vergleichbar sind, werden Sie sogar mit einer neuen `assert`-Methode beschenkt:

```
assertEquals(Object[] expected, Object[] actual)
```

JUnit 3.8.1 spuckt bei `String`-Vergleichen mit `assertEquals` manchmal nahezu unbrauchbare Fehlermeldungen aus. Dieser Bug ist mit dem neuen Release behoben.

Umstieg auf JUnit 4

Der Umstieg zur neuen Schule ist leicht. Tests der alten Schule müssen nicht zwangsläufig auf Annotationen umgestellt werden, sie können friedlich weiterexistieren. Wichtig ist lediglich, dass Sie die alte und neue Form nicht in einer Testklasse mischen können.

Das reicht erst einmal. Genug Rüstzeug gesammelt – tauchen wir nun tiefer in die Testgetriebene Entwicklung ein ...

4 Testgetriebene Programmierung

Die Testgetriebene Entwicklung basiert auf drei Grundtechniken und ihrem engen Zusammenspiel: Testgetriebene Programmierung, unser linker Fuß, Refactoring, unser rechter Fuß, und Häufige Integration, unsere Fußspur.

Mein Ziel ist es zu demonstrieren, wie wir mit diesen Techniken während der gesamten Entwicklungsdauer stetigen Fortschritt erzielen können, wie wir stets wissen, wo wir eigentlich gerade stehen, und wie wir die Angst verlieren, auch einmal einen Schritt zurückzunehmen. Nachdem Sie im vergangenen Kapitel gelernt haben sollten, wie Sie mit JUnit einfache Tests für einfachen Code schreiben können, befasst sich dieses Kapitel mit der ersten Grundtechnik: vor allem Tests.

4.1 Die erste Direktive

Die Testgetriebene Entwicklung kann, wie schon eingangs erwähnt, durch drei einfache Regeln definiert werden.

1. Direktive: *Motivieren Sie jede Änderung des Programmverhaltens durch einen automatisierten Test.*

Diese Regel erklärt, wie Testgetriebene Programmierung funktioniert: Wir gehen immer von einem Test aus. Wir entwerfen diesen Test so, dass er nur einen kleinen Schritt unternimmt, so klein, dass das Ziel in greifbare Nähe rückt. Gleich im Anschluss schreiben wir den Code, der diesen Test erfüllt. So steuern die Tests die Entwicklung:

- Bevor wir neuen Code schreiben, schreiben wir neue Tests.
- Bevor wir bestehenden Code ändern, ändern wir bestehende Tests.
- Bevor wir einen Fehler im Code suchen und reparieren, suchen und reparieren wir das dafür verantwortliche Loch in den Tests.

4.2 Der Testgetriebene Entwicklungszyklus

Unser Entwicklungsstil lässt sich beschreiben als ein Tanz zwischen erwartetem Testerfolg und erwartetem Fehlschlag. JUnits Testbalken spielt deshalb eine besondere Rolle: An seinem Zustand erkennen wir zum einen, ob der Programmcode die Anforderungen der Tests erfüllt. Zum anderen setzen wir durch seine gezielten *Zustandsübergänge* die Schritte der Testgetriebenen Entwicklung um:

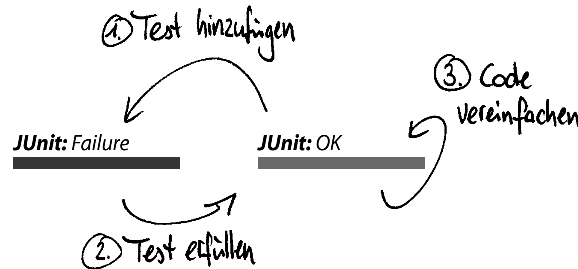


Abb. 4–1 *Der Testgetriebene Entwicklungszyklus aus Sicht der JUnit-Zustandsübergänge*

Die drei Schritte der Testgetriebenen Entwicklung:

1. **grün➠rot:** Der Programmierzyklus beginnt damit, dass wir neue Anforderungen an unseren Code stellen, indem wir ihnen einen Test voranstellen. Dieser Schritt führt dazu, dass der Test fehlschlägt, bis die geforderte Funktionalität implementiert ist.
2. **rot➠grün:** Der Test wird in kleinen Schritten und auf möglichst einfachem Weg erfüllt. Jederlei Vereinfachung ist willkommen, die dabei Zeit einspart. Sobald unser Code das gewünschte Verhalten implementiert, strahlt der JUnit-Balken wieder grün.
3. **grün➠grün:** Nachdem die Tests laufen, muss der Code in die Einfache Form gebracht werden. Kleine Programmiersünden, die wir vorher begangen haben, müssen wir jetzt gutmachen. Der grüne Balken sichert ab, dass uns keine Fehler unterlaufen.

Durch diesen Zyklus avanciert die Tätigkeit des Programmierens von einem meist ad hoc durchgeführten zu einem methodischen Prozess:

- Als Entwickler verwickeln wir uns gewöhnlich in zu viele Probleme gleichzeitig. Testgetriebene Entwicklung *ent-wickelt* den Prozess.
- Komplexe Probleme lassen sich in kleinen Schritten bewältigen.
- Regressionstests sichern, dass bewältigt bleibt, was bewältigt ist (!).

4.3 Die Programmierzüge

Kleine Schritte

Testgetriebene Programmierung bewegt sich in einem stark iterativen Arbeitsprozess, in dem jeder Schritt auch die Möglichkeit zum Lernen bietet: Jeder Test, den wir schreiben und erfüllen, kann uns konkretes Feedback darüber liefern, welchen Test wir als Nächstes hinzufügen. Ein solcher Programmierzyklus ist typischerweise sehr kurz: maximal einige Minuten. Ein Schlüsselaspekt dieser Arbeitsweise ist, gerade komplizierte Probleme in einfachere Teilprobleme zerlegen zu können: Größere Anforderungen müssen wir also in kleinen Schritten angehen. Je kürzer wir diese Etappen wählen, desto schneller lernen wir auch, wohin uns der Weg wirklich führt: Wichtig ist, dass Sie den Wechsel der Farben verinnerlichen: »grün➠rot➠grün➠rot➠grün➠...«

Die Programmierzüge in Kürze:

1. Schreiben Sie einen Test, der zunächst fehlschlagen sollte.
2. Schreiben Sie gerade so viel Code, dass der Test fehlschlägt.
3. Schreiben Sie gerade so viel Code, dass alle Tests laufen.

Diesen Prozess wiederholen wir, solange uns weitere Tests einfallen, die unter Umständen fehlschlagen könnten. Das heißt, wir drücken in den Tests so lange neue Anforderungen aus, bis die Todo-Liste leer ist. Anschließend refaktorisieren wir den Code in seine Einfache Form. Oft beginnt das Refactoring jedoch schon früher, um Platz für weiteren Code zu schaffen und neue Tests dadurch auch einfacher erfüllen zu können.

Arbeiten Sie von einem grünen Testbalken aus immer zunächst auf einen roten Balken hin. Arbeiten Sie dann auf kürzestem Wege zurück auf den grünen Testbalken. Dort angekommen, nehmen Sie die Möglichkeit zum Refactoring wahr.

Mit dieser Arbeitsweise verfolgen wir die Idee, das konkrete Feedback, das uns nur geschriebener Code liefern kann, unmittelbar als Basis für unseren nächsten Programmierzug verwenden zu können. Erfolgreich angewendet mündet der Fluss Testgetriebenen Programmierens in eine Serie kleiner Erfolge positiven Feedbacks: Sie testen ein wenig, lassen sich von einem fehlschlagenden Test demonstrieren, wo Änderungen notwendig sind, programmieren ein wenig, testen erneut und feiern einen kleinen Erfolg. *High Five!* Und dann von vorne ...

Entscheidend ist, eine Idee des kleinstmöglichen Inkrementschritts zu haben und diesen möglichst schnell und möglichst einfach in einer kurzen Programmieriteration in ausführbaren Code zu verwandeln.

4.4 Beginn einer Testepisode

Ich möchte das Prinzip des unmittelbaren Feedbacks an einer kurzen Geschichte verdeutlichen. Starten wir dazu mit einer neuen Episode: Unsere Aufgabe soll es sein, eine neue Preiskategorieklasse für Filme mit regulärem Preis zu entwickeln. Nennen wir sie `RegularPrice`.

Natürlich fangen wir mit dem Test an:

```
import junit.framework.TestCase;

public class RegularPriceTest extends TestCase {
}
```

Der Clou an dieser leeren Klasse sind zwei Dinge: Zum einen erkennt JUnit eine Klasse erst als Testfallklasse an, wenn sie mindestens eine `test...`-Methode enthält. Zum anderen hätte ich vielleicht mitzählen sollen, wie oft ich schon eine neue Testklasse angelegt, dann jedoch vergessen habe, sie in die passende Testsuite einzuhängen.

Besonders im Team bemerkt man den Fortschritt in der Anzahl von Testfällen im Projekt nicht mehr so deutlich. Wir finden uns dann unter Umständen in der Situation wieder, dass wir testen, was das Zeug hält, bis uns irgendwann schlagartig klar wird, dass der Balken grün bleibt, was immer wir auch für einen Unsinn tippen.

Nachdem ich mich auf diese Weise mehrmals zum Narren gemacht hatte, dachte ich darüber nach, wie ich mir das Feedback von JUnit zunutze machen könnte. Schließlich kam mir die Idee, neue Testfallklassen mittels eines roten Balkens einzuläuten. Und so erwarte ich heute die Erinnerungsstütze durch den roten Balken und zwinge mich damit dazu, die neuen Tests in meine aktive Suite einzubinden:

```
public class AllTests...
  public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTestSuite(CustomerTest.class);
    suite.addTestSuite(EuroTest.class);
    suite.addTestSuite(MovieTest.class);
    suite.addTestSuite(RegularPriceTest.class);
    return suite;
  }
}
```

Ein kurzer Testlauf bestätigt unser Handeln. Der Test wirkt:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
No tests found in RegularPriceTest
```

4.5 Ein einfacher Testplan

Eine bewährte Idee ist, dass wir uns zu Beginn einer Episode zunächst die notwendigen Testfälle überlegen und sie zum Beispiel auf einer Karteikarte notieren. Das ist Analyse.

Unsere Aufgabe besteht im Entwickeln einer neuen Preiskategorie:

Filme kosten regulär für die ersten drei Tage insgesamt 1,50 Euro. Für jeden weiteren Ausleihtag kommen 1,50 Euro dazu.

Wir malen kurz ein Bild und benennen unsere ersten Testfälle:

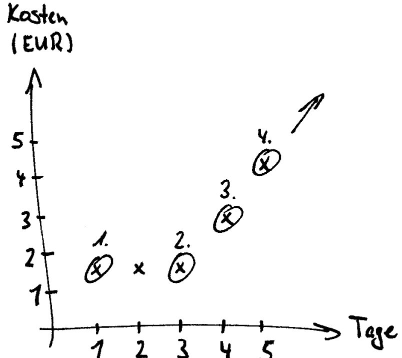


Abb. 4–2
Eine neue Preiskategorie: Brainstorming einer Testroute

Da erschöpfendes Testen unmöglich ist, besteht unsere Strategie darin, entlang der Fälle zu testen, die interessante Entwurfsentscheidungen aufweisen: Die repräsentativen Testfälle liegen für uns bei *einem*, *drei*, *vier* und *fünf Ausleihtagen*. Weitere Tests würden uns keine neuen Informationen liefern. Wir legen außerdem fest, dass Fehleingaben wie *null* oder *negative Werte* außerhalb unseres aktuellen Problems liegen. Die Verantwortung, gültige Eingabedaten sicherzustellen, müssen also die benutzenden Klassen übernehmen.

Tests:
– 1 Tag
– 3 Tage
– 4 Tage
– 5 Tage

In den meisten Fällen werden wir diese anfangs gewählte Liste von Testfällen während der Entwicklung anpassen: Neue Testfälle mögen hinzukommen und geplante fallen wieder weg. Das Wichtige ist nicht der Plan an sich, sondern die *Planung*. Während der Entwicklung dazuzulernen ist ein gutes Zeichen. Und tatsächlich ist es auch interessant, eine ganz andere Route einzuschlagen, als wir hier gehen werden. Wichtig ist allein, dass die Tests die Programmentwicklung antreiben.

Der erste Schritt

Für den Einstieg wählen wir immer einen Testfall aus unserer Liste, der ein möglichst kleines Inkrement für unsere bestehende Codebasis darstellt und durch den wir etwas Wesentliches und Wichtiges lernen. Starten wir also mit dem Testfall für den *ersten Ausleihtag*!

4.6 Erst ein neuer Test ...

Schritt 1: Schreiben Sie einen Test, der zunächst fehlschlagen sollte

Sie erinnern sich: Wir wollen uns der gewünschten Funktionalität schrittweise annähern, um den Feedbackzyklus möglichst häufig zu schließen. Wir wollen die Entwicklung eines evolutionären Designs antreiben, indem wir den Testfall so schreiben, dass er zunächst einmal fehlschlägt. Aufgrund der zu treffenden Entscheidungen kann dieser Schritt manchmal eine ernste Herausforderung darstellen.

Design aus der Verwendungsperspektive

Ferner entwerfen wir den Test so, wie wir uns die Verwendung der zu testenden Klasse wirklich wünschen. Das ist Design.

Wir müssen uns also erst einmal fragen, was die Klasse überhaupt tun soll, und uns anschließend überlegen, wie sie es denn tun könnte. Den Blickwinkel auf diese Weise abzulenken ist deshalb praktisch, weil die Schnittstelle einer Klasse wichtiger ist als ihre Implementierung. Ignorieren Sie deshalb stets für einen Moment die Implementierungsdetails, die sich schon in Ihrem Kopf zu Wort melden. Stellen Sie sich die Klassenschnittstelle erst einmal so einfach vor, wie sie im gewünschten Zusammenhang ideal verwendbar wäre.

Versetzen Sie sich geistig in den Programmierer, der Ihre Klasse verwendet, und entwerfen Sie die Klasse so, wie Sie sie intuitiv verwenden würden.

Los gehts! Die Ausleihgebühr beträgt für den ersten Tag *1,50 Euro*. Schreiben wir einfach mal auf, was wir vorhaben:

Tests:
– 1 Tag
– 3 Tage
– 4 Tage
– 5 Tage

```
public class RegularPriceTest...
  public void testChargingOneDayRental() {
    RegularPrice price = new RegularPrice();
    assertEquals(new Euro(1.50), price.getCharge(1));
  }
}
```

Durch diese zwei Zeilen Testcode haben wir bereits ein paar wesentliche Entwurfsentscheidungen getroffen:

- Unsere neue Klasse trägt den Namen `RegularPrice`.
- Exemplare der Klasse erzeugen wir mit dem Defaultkonstruktor.
- Die Ausleihgebühr berechnen wir mittels der Methode `getCharge`, die dazu die Anzahl von Ausleihtagen des Typs `int` erhält.
- Als Resultat der Operation erwarten wir ein Objekt vom Typ `Euro` zurück.

4.7 ... dann den Test fehlschlagen sehen

Schritt 2: Schreiben Sie gerade so viel Code, dass der Test fehlschlägt

Wir haben einen neuen Testfall geschrieben und versuchen diesen nun zu kompilieren. In dem Fall, dass wir durch den Test eine neue Klasse oder eine neue Methode motivieren, muss der Übersetzungsversuch natürlich schief gehen, doch auch dies stellt sich als nützlich heraus.

Typischerweise wollen wir auf diesem Weg in kleinen Schritten Fehlermeldungen der Art provozieren, dass eine Klasse oder Methode, die wir in unserem Test bereits benutzen, in der geforderten Form noch gar nicht existiert. Wir prüfen dadurch, ob unsere neue Klasse oder neue Methode mit einer bestehenden Klasse oder Methode kollidiert. Durch Polymorphie können schwer identifizierbare Seiteneffekte auftreten und es gibt kaum etwas Schlimmeres, als sich darin zu täuschen, welches Stück Code ausgeführt wird. Wenn Sie etwas Glück haben, hat ein Kollege die Klasse sogar schon implementiert.

In statisch typisierten Programmiersprachen wie Java macht der Compiler den ersten Test:

```
Class RegularPrice not found.
```

Die Fehlermeldung zwingt uns, mindestens eine leere Hülse anzulegen:

```
public class RegularPrice {
}
```

Dann versuchen wir erneut zu kompilieren. Wieder mit einem Fehler:

```
Method getCharge(int) not found in class RegularPrice.
```

Ganz typisch für diesen Entwicklungsstil ist, dass wir uns in kurzen Bewegungen von Übersetzungsfehler zu Übersetzungsfehler hangeln. Wir schreiben dabei nur gerade so viel Code, dass sich die Testklasse überhaupt übersetzen lässt. Das Ziel sind kleine schnelle Feedbackschleifen, die in einer iterativen Arbeitsweise münden, die durch die modernen interaktiven IDEs und inkrementellen Compiler auch ganz hervorragend unterstützt wird.

Schnelles Feedback durch iterative Arbeitsweise

Eclipse und IntelliJ IDEA beispielsweise gehen den neuen Weg und lassen uns nicht nur Codeelemente verwenden, die schon existieren, also der Vergangenheit entstammen. Sie lassen uns auch schon mit Klassen, Methoden und Variablen arbeiten, die es noch gar nicht gibt, also die Zukunft beschreiben. Beide IDEs unterkringeln unbekannte Codeelemente im Editor und können sie auf Knopfdruck aufgrund der aus dem Kontext schon verfügbaren Information einfach generieren.

... unterstützt durch interaktive IDEs

Lassen Sie sich bei der Fehlerbehebung von der Entwicklungsumgebung führen. Beheben Sie nur gerade, was der aktuelle Übersetzungsfehler von Ihnen verlangt.

Abschließend definieren wir die benötigte Methode:

```
public class RegularPrice {
  public Euro getCharge(int daysRented) {
    return null;
  }
}
```

Da unser Test als Rückgabewert ein `Euro`-Objekt erwartet, liefern wir vorerst einfach `null` zurück. Dieser Schritt mag Sie jetzt überraschen. Die Implementierung kann so natürlich nicht funktionieren, doch das genau ist unser Ziel, um den Test zunächst fehlschlagen zu sehen.

Wir rekompilieren. Unser Compiler ist zufrieden. Schlägt unser neuer Test aber auch tatsächlich fehl?

Ein neuer Test, und zwar nur ein fehlschlagender, sollte Anlass sein zum Schreiben neuer Programmfunktionalität: Mit jedem neuen Test stellen wir neue Behauptungen auf, an denen wir unsere bestehende Codebasis elegant scheitern lassen. Der einzige Weg, um den Test zu erfüllen, ist somit, das tatsächlich geforderte Verhalten zu realisieren.

Test für den Test

Wir wollen dabei immer vom grünen Balken ausgehend auf einen roten Balken hinarbeiten. Indem wir den Test zunächst fehlschlagen sehen, testen wir den Test selbst. Wenn der Test fehlschlägt, war der Test tatsächlich erfolgreich. Ein Fehlschlag gibt uns das Vertrauen, dass wir einen wertvollen Test geschrieben haben. Läuft der Test aber unvermutet durch, haben sich unsere Annahmen wohl als fehlerhaft entpuppt. Entweder haben wir dann gar nicht den Test geschrieben, den wir zum Antreiben weiterer Programmlogik schreiben wollten, oder unser Code implementiert die neue Funktion wirklich bereits.

Machen wir also den Test für den Test:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  expected:<EUR 1.5> but was:<null>
at RegularPriceTest.testChargingOneDayRental
  (RegularPriceTest.java:6)
```

Diesen kleinen Zwischenschritt aus Bequemlichkeit auszulassen ist kinderleicht und ich bekomme oft die Frage gestellt, ob ich wirklich so »umständlich und offensichtlich langsam« arbeite. Natürlich! Denn der Fehlschlag verrät den nächsten Programmierzug. Das Prinzip ist so einfach: Arbeiten Sie nur vom roten zum grünen Balken *vorwärts*!

4.8 ... schließlich den Test erfüllen

Schritt 3: Schreiben Sie gerade so viel Code, dass alle Tests laufen

Um neue Funktionalität in das Programm zu integrieren, schlagen wir immer den Weg ein, der am einfachsten erscheint. Wir programmieren wirklich nur, was zur Erfüllung des *aktuellen* Testfalls notwendig ist, und kein bisschen mehr.

In einigen Fällen ist die einfachste Lösung, feste Werte zu kodieren. Unfassbar, aber absolut sinnvoll, wie Sie noch sehen werden:

***JUnit:** OK*

```
public class RegularPrice {
  public Euro getCharge(int daysRented) {
    return new Euro(1.50);
  }
}
```

Tests:
~~– 1 Tag~~
– 3 Tage
– 4 Tage
– 5 Tage

Diese Implementierung kann natürlich nur eine Übergangslösung sein bis zum Zeitpunkt, da uns ein weiterer Test beweist, dass die aktuelle Lösung wirklich noch zu einfach ist. Auf diesem Weg treiben die Tests schrittweise die Entwicklung fortgeschrittener Logik an, die nicht mehr allein mit festkodierten Werten auskommt.

Damit sind wir am Ende eines Programmierzyklus angekommen. Jetzt erfolgt der Test, wo wir eigentlich wirklich stehen. Die Tests sagen uns, wenn wir erfolgreich waren: Wenn alle Tests laufen.

Kleine Erfolge

Wenn der Testbalken jetzt grün läuft, wissen wir, dass es der *gerade* geschriebene Code war, der den Unterschied ausgemacht hat. Es ist diese Art der Entwicklung, die uns ein kurzes Glücksgefühl schenken kann, das Ihnen vorenthalten bleibt, solange Sie ohne diese kurzen Feedbackzyklen programmieren. Der kleine Erfolg gibt uns zu Recht einen kurzen Moment zum Durchatmen und Freimachen (»Okay, geschafft!«) und bietet uns einen natürlichen Abschluss für den gegenwärtigen Gedankengang (»Was ist der nächste Testfall?«). Die Tests reduzieren so unseren Stress und erhöhen den Spaßfaktor.

Mehr Spaß, weniger Stress

Wenn der Testbalken dagegen rot bleibt, wissen wir, dass es der *gerade* geschriebene Code war, der keinen oder nicht den erwarteten Unterschied gebracht hat: Interessant wäre dann, ob nur der zuletzt geschriebene Test betroffen ist oder gar ältere Tests wieder zerbrochen sind, die schon einmal liefen. Da wir immer nur ein paar Zeilen Code und selten eine ganze Methode am Stück schreiben, lohnt es sich dafür in den meisten Fällen jedoch kaum noch, den Debugger zu öffnen.

Wir sind zurück auf Grün und damit schließt sich der Kreislauf der Testgetriebenen Programmierung. Was ist der nächste Schritt?

4.9 Zusammenspiel von Test- und Programmcode

Wenn Sie aufmerksam dem Programmierzyklus folgen, fällt Ihnen ein enges Wechselspiel zwischen den Tests und dem Programmcode auf:

- Der fehlschlagende Test entscheidet, welchen Code wir schreiben müssen, und treibt damit die Entwicklung der Programmlogik an.

Der nächste Schritt

- Wir entscheiden anhand des bisher geschriebenen Codes, welchen Test wir schreiben sollten, und treiben damit die Entwicklung des Softwaredesigns an.

Der zweite Punkt erfordert im Vergleich wesentlich mehr Kreativität, weil Tests oft schwieriger zu formulieren als zu erfüllen sind. Dies ist gleichzeitig der Schritt, in dem wir die Anforderungen an das System in ein testbares Softwaredesign übersetzen.

Nur ein Test auf einmal

Aufgrund des starken Zusammenspiels von Test- und Programmcode ist es nicht zu empfehlen, mehr als einen Test zur gleichen Zeit ins Rennen zu schicken oder sogar erst alle möglichen Tests zu schreiben und sie anschließend Stück für Stück zu erfüllen. Stellen wir während der Implementierung nämlich fest, dass unser Klassenentwurf eine ganz andere Richtung einschlagen möchte, müssten wir alle unsere geschriebenen Tests erneut anpassen.

Manchmal können wir tatsächlich erst bei der Programmierung erkennen, welchen Test wir denn eigentlich hätten schreiben sollen. Oft werden dann getroffene Entwurfsentscheidungen umgeworfen und in vielen Fällen ändert sich die Klassenschnittstelle noch einmal. Aus diesem Grund ist es angebracht, nur einen Test auf einmal zu schreiben und sofort zu implementieren. So lernen wir schrittweise, welche Tests uns wirklich voranbringen. Die Buchführung darüber, welche Tests wir noch schreiben müssen, können wir zum Beispiel auf einer einfachen Karteikarte vornehmen.

Schreiben Sie nur einen Testfall auf einmal und bringen Sie ihn auf Grün, bevor Sie die Arbeit am nächsten Testfall beginnen. Wenn Sie mehrere Testfälle schreiben wollen, sammeln Sie sie auf einer Karteikarte o.Ä.

Zu viel Code, zu wenig Tests

Eine kleine Warnung an dieser Stelle: Wenn Sie mehr Code schreiben, als Ihr Testfall in Wirklichkeit verlangt, schreiben Sie immer zu wenig Tests. Sie produzieren dann Code, der unter Umständen von keinem Test gesichert wird. Die Codeabdeckung Ihrer Tests wird in diesem Fall geringer sein und das Sicherheitsnetz, das durch die feingranularen Unit Tests für das anschließende Refactoring entsteht, wird dadurch grobmaschiger. Fehler können somit leichter durchschlüpfen.

4.10 Ausnahmebehandlung

Der erste Durchlauf sollte anschaulich machen, wie die Testgetriebene Programmierung praktisch umgesetzt wird:

1. Schreiben Sie einen Test, der zunächst fehlschlagen sollte.
2. Schreiben Sie gerade so viel Code, dass der Test fehlschlägt.
3. Schreiben Sie gerade so viel Code, dass alle Tests laufen.

Gewissermaßen habe ich Ihnen jedoch ein Idealbild projiziert und die Überraschungen vorenthalten, mit der die Realität aufwarten wird. Natürlich gibt es Ausnahmen in diesem Prozess. Oder anders gesagt: Es existieren Ausnahmen zu meiner Beschreibung, nicht jedoch im Prozess selbst. Genau genommen können diese »Bad Day«-Szenarien während der Programmierung ebenso häufig auftreten wie das zuvor beschriebene »Good Day«-Szenario:

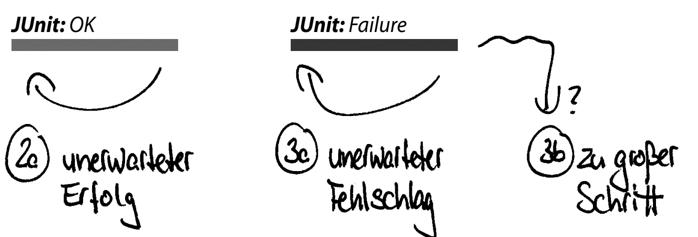


Abb. 4–3
Drei Ausnahmefälle

Die drei häufigsten und für uns interessantesten Ausnahmen kommen während der Schritte 2 und 3 vor:

2a. **grün➠grün:** Der Test läuft auf Anhieb.
3a. **rot➠rot:** Der Test schlägt weiterhin fehl.
3b. **rot➠???:** Der Test lässt sich nur unter erheblichem Aufwand erfüllen, wenn denn überhaupt.

Durch Ausnahmefälle zum Lernerlebnis

Wenn Sie könnten, sollten Sie nur Tests schreiben, bei denen Ihnen die obigen Überraschungen passieren. Und das ist wirklich ernst gemeint, denn in jedem der dargestellten Fälle wartet ein Lernerlebnis auf Sie:

2a. **grün➠grün:** Der Test sollte fehlschlagen. Warum läuft er nur?
3a. **rot➠rot:** Der Test sollte laufen. Warum schlägt er jetzt fehl?
3b. **rot➠???:** Der Test sollte sich in relativ kurzer Zeit mit relativ wenig Aufwand erfüllen lassen. Wie kommen wir dort hin?

Natürlich können Sie aber nicht nur Tests schreiben, die mit einer Überraschung auf Sie warten, denn dann wär's ja keine mehr, nicht?

4.11 Ein unerwarteter Erfolg

Ausnahme 2a: Der Test läuft, obwohl er eigentlich fehlschlagen sollte

Die natürliche Triebfeder der Testgetriebenen Programmierung ist es, Tests zu schreiben, die neue Funktionalität einfordern und unseren bestehenden Code auf intelligente Weise zum Scheitern bringen. Solche Tests schlagen fehl, bis unser Code die neuen Anforderungen erfüllt.

Manchmal lässt sich der Code jedoch nicht so elegant durch neue Tests erzwingen. Manchmal schreiben wir einfach Tests, die schon auf Anhieb laufen. Wieso das?

Hier sehen Sie ein Beispiel: die Ausleihgebühr für *drei Tage*.

Wir benötigen erneut ein `RegularPrice` Objekt, weshalb es günstig erscheint, zunächst eine Test-Fixture zu gewinnen:

***JUnit:** OK*

```
public class RegularPriceTest...
  private RegularPrice price;

  protected void setUp() {
    price = new RegularPrice();
  }

  public void testChargingOneDayRental() {
    assertEquals(new Euro(1.50), price.getCharge(1));
  }
}
```

Nach diesem kleinen Refactoring können wir den zweiten Test gleich viel einfacher hinschreiben und müssen nicht unnötig Code kopieren. Vorher lassen wir jedoch zur Sicherheit die Tests durchlaufen.

Dann schreiben wir den Test:

Tests:
~~– 1 Tag~~
– 3 Tage
– 4 Tage
– 5 Tage

```
public class RegularPriceTest...
  public void testChargingThreeDaysRental() {
    assertEquals(new Euro(1.50), price.getCharge(3));
  }
}
```

***JUnit:** OK*

Keine Überraschung: Der Test läuft natürlich auf Anhieb. Schließlich werden überhaupt keine neuen Anforderungen gestellt.

Hier hätten wir einfach eine andere Testroute einschlagen müssen, denn Tests dieser Art sind für die Testgetriebene Entwicklung zunächst einmal wertlos, weil sie uns mit dem Programm nicht weiterbringen. Wertvoll sind sie dennoch, weil sie uns eine Geschichte über unseren Testprozess selbst erzählen, aus der wir für die Zukunft lernen können.

Ein unerwartet grüner Balken kann nur zwei Gründe haben:

- Der Code erfüllt die Anforderung bereits.
- Der Test selbst enthält bereits einen Fehler.

Unter Umständen sind wirklich die Behauptungen fraglich, die wir im letzten Test aufgestellt haben. Die Wahrscheinlichkeit dafür ist zwar meist relativ gering, wenn wir uns aber vor Augen halten, dass sich der Fehler nur in den Code eingeschlichen haben kann, den wir gerade getippt haben, geht eine genaue Nachprüfung schnell vonstatten.

Es gibt auch Momente, wo wir dem grünen Balken nicht mehr trauen: Wenn uns das Gefühl befällt, dass es nicht mehr mit rechten Dingen zugeht. Die Geschichte vom nicht ausgeführten Test kennen Sie ja bereits. Ich streue dann absichtlich Fehler in Tests oder Code ein, um zu sehen, ob ich eigentlich noch alles unter Kontrolle habe.

Rückschritt machen

Manchmal kann es auch ratsam sein, den getippten Unfug der paar letzten Minuten über den Haufen zu werfen und in kleineren Schritten nochmals von vorne zu starten. Ehrlich, fassen Sie Mut und gehen Sie zum letzten grünen oder auch roten Balken zurück, mit dem die Welt noch in Ordnung war. Probieren Sie's aus, Sie können sich unter Umständen viel Zeit und Mühe ersparen.

Prozess-Check

Der andere Grund für einen unerwarteten Testerfolg ist, dass die vermeintlich neue Funktion wirklich schon realisiert und getestet ist. Dieser Umstand gibt dann tatsächlich Auskunft über unseren Testprozess und bietet eine gute Gelegenheit zum Reflektieren:

- Haben wir für einen der vergangenen Testfälle geringfügig mehr Code geschrieben, als unbedingt notwendig gewesen wäre?
- Ist dieser Fall schon anderswo mitgetestet? Vielleicht nur indirekt? Wie stark unterscheiden sich die Tests? Können wir sie geeignet zusammenfassen?
- Duplizieren wir gerade Testcode und wissen es vielleicht nicht?
- Haben die Tests zu wenig »Kick« dafür, dass sie interessante neue Entwurfsentscheidungen aufwerfen? Könnten wir solche Tests zukünftig ans Ende der Episode stellen? Ist es dann überhaupt noch notwendig, sie zu schreiben?

In diesem Fall hatten wir nur Pech mit der Wahl unseres Inkrements. Eine bessere Route wäre gewesen, erst *vier* oder *fünf Tage* zu testen.

Ernsthaft problematisch wird es, sobald Sie mehr Schwierigkeiten haben, den korrekten Testcode zu schreiben als den Programmcode. Die gesamte Arbeitsweise beruht auf der stillen Annahme, dass die Tests einfach zu schreiben sind. Überlegen Sie bei schwierigen Tests, wie das Design geändert werden müsste, um die Tests zu vereinfachen.

4.12 Ein unerwarteter Fehlschlag

Ausnahme 3a: Der Test schlägt fehl, obwohl er eigentlich laufen sollte

Die Eleganz des Testgetriebenen Programmieransatzes liegt in seinen schnellen und konkreten Feedbackzyklen. Zwischen dem Zeitpunkt einer getroffenen Entwurfsentscheidung und dem bewiesenen Erfolg oder Misserfolg dieser Idee liegen nur wenige Minuten und immer nur ein paar Zeilen Code: Wenn wir in sehr kleinen Schritten vorgehen und uns stets am einfachsten Weg orientieren, vergehen zwischen den zwei grünen Testbalken gerade einmal ein bis drei Minuten.

Manchmal jedoch sind unsere Inkremente zu groß gewählt und wir verzetteln uns einfach. Manchmal machen wir an sich vermeidbare Programmierfehler. Manchmal enthält der Test selbst einen Fehler. Manchmal schreiben wir Code, der anders funktioniert als erwartet.

Hier ist ein Beispiel, wie es sich tatsächlich zugetragen hat:

Tests:
~~– 1 Tag~~
~~– 3 Tage~~
– 4 Tage
– 5 Tage

```
public class RegularPriceTest...
  public void testChargingFourDaysRental() {
    assertEquals(new Euro(3.00), price.getCharge(4));
  }
}
```

Die Kontrolle, ob der Test auch wirklich fehlschlägt:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  expected:<EUR 3.0> but was:<EUR 1.5>
at RegularPriceTest.testChargingFourDaysRental
  (RegularPriceTest.java:19)
```

Die Implementierung, die uns als Erstes in den Kopf schoss:

```
public class RegularPrice {
  public Euro getCharge(int daysRented) {
    Euro result = new Euro(1.50);
    if (daysRented == 4) result.add(new Euro(1.50));
    return result;
  }
}
```

Und JUnits Kommentar dazu:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  expected:<EUR 3.0> but was:<EUR 1.5>
at RegularPriceTest.testChargingFourDaysRental
  (RegularPriceTest.java:19)
```

Oops, wohl vergessen zu rekompilieren. Also kompilieren wir neu ..., aber ohne Resultat.

Wir kratzen uns am Kopf: Wie sind wir hierher gekommen?

Die Regel lautet, dass der Fehler nur in dem Code stecken kann, den wir vor einer Minute getippt haben. Sehen Sie den Fehler?

Wenn wir uns aus dem Geschehen keinen Reim machen können, verwerfen wir unsere letzte Änderung. Zeitlich verlieren wir dadurch eine weitere Minute. Niemand weiß, wie viele unnötige Minuten wir uns tatsächlich ersparen. Also schnell Steuerung-Z (oder Apfel-Z) gedrückt:

```
public class RegularPrice {
  public Euro getCharge(int daysRented) {
    return new Euro(1.50);
  }
}
```

Wenn wir nur gerade den aktuellen Testfall erfüllen sollten, wäre diese Lösung eigentlich noch einfacher gewesen:

JUnit: *OK*

```
public class RegularPrice {
  public Euro getCharge(int daysRented) {
    if (daysRented <= 3) return new Euro(1.50);

    return new Euro(3.00);
  }
}
```

Unsere Tests belohnen uns mit der ersehnten Farbe. Trotzdem sind wir mit der Lösung noch nicht zufrieden.

In unserem ersten Versuch wollten wir bereits ausdrücken, dass der Preis an sich aus einem fixen und einem variablen Anteil besteht. Holen wir das nach:

JUnit: *OK*

```
public class RegularPrice {
  public Euro getCharge(int daysRented) {
    if (daysRented <= 3) return new Euro(1.50);

    return new Euro(1.50).add(new Euro(1.50));
  }
}
```

Brrr... Bei dem Anblick läuft uns ein kleiner Schauer über den Rücken. Erstens: Warum tauchen hier drei »ein Euro fünfzig«-Schnipsel auf? Zweitens: Was bedeuten diese drei Schnipsel? Drittens: Können wir sie geeignet zusammenfassen oder haben sie nichts miteinander zu tun?

Aber: Moment! Wieso funktioniert die add-Operation plötzlich an dieser Stelle, nicht aber vor einer Minute im ersten Versuch?

Ein kurzer Blick auf den Testfall testAdding zeigt ein konkretes Beispiel zur korrekten Verwendung der Euro-Klasse:

```
public class EuroTest...
  public void testAdding() {
    Euro sum = two.add(two);
    assertEquals("sum", new Euro(4.00), sum);
    assertEquals("two", new Euro(2.00), two);
  }
}
```

Wir erinnern uns: Objekte der Klasse Euro waren nicht veränderbar. Stattdessen wird ein neues Objekt mit dem neuen Wert zurückgegeben. Vermutlich hat uns der Name add in die Irre geführt. Aus dem Grund beschließen wir, die Umbenennung zu plus als Refactoringkandidaten zu notieren.

Refactorings:
– Euro: add in plus umbenennen

Führen Sie ein Refactoring durch, wenn Sie etwas gelernt haben und dieses Wissen Bestandteil des Programms selbst sein sollte.

Zunächst müssen wir jedoch unsere angefangene Maßnahme beenden. Zur Erinnerung: Die unzähligen »1,50 Euro«-Objekte störten uns. Guter Code braucht gute Namen:

JUnit: *OK*

```
public class RegularPrice {
  private static final Euro BASEPRICE = new Euro(1.50);

  public Euro getCharge(int daysRented) {
    if (daysRented <= 3) return BASEPRICE;

    return BASEPRICE.add(new Euro(1.50));
  }
}
```

Ein kurzer Testlauf und weiter geht’s mit den anderen »1,50 Euro«:

JUnit: *OK*

```
public class RegularPrice...
  private static final Euro PRICE_PER_DAY = new Euro(1.50);

  public Euro getCharge(int daysRented) {
    if (daysRented <= 3) return BASEPRICE;

    return BASEPRICE.add(PRICE_PER_DAY);
  }
}
```

Ebenso gehen wir mit der Anzahl Tage um, für die der reduzierte Preis gilt:

JUnit: *OK*

```
public class RegularPrice...
  private static final int DAYS_DISCOUNTED = 3;

  public Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;

    return BASEPRICE.add(PRICE_PER_DAY);
  }
}
```

Durch den letzten Test und anschließendes Refactoring hat unser Code erste Form angenommen. Der Punkt aber war, dass es sich immer lohnt, kleinere Schritte zu wählen. Fehler wie in diesem Beispiel sind menschlich. Wann immer Sie so ins Straucheln geraten, fangen Sie sich.

Rückschritt für den Fortschritt

von Tammo Freese, freier Berater

Jeder Wechsel von rot nach grün gibt uns die Sicherheit, dass wir Fortschritt machen. Wenn wir Probleme haben, zum grünen Balken zurückzufinden, stagniert unser Fortschritt. Dann können wir entweder versuchen, den Fehler im Code aufzuspüren, oder wir gehen zurück auf einen sicheren Stand und versuchen einen anderen Weg.

Tatsächlich wählen wir meist den Rückschritt, da er uns schnell zu einem Punkt zurückführt, von dem aus wir wieder Fortschritt machen können. Das unangenehme Gefühl, Änderungen rückgängig zu machen, ist bei kurzen Schritten praktisch nicht mehr vorhanden.

Wohin wir zurückspringen, orientiert sich an der Schwierigkeit, auf die wir treffen. Erkennen wir die Ursache für einen Fehlschlag nicht sofort, gehen wir zurück zu dem Zeitpunkt direkt nach der Erstellung des Tests und schlagen eine andere Route ein. Merken wir beim Erfüllen eines Tests, dass er zu viel fordert und wir uns deshalb verzetteln, machen wir auch den Test selbst rückgängig und starten mit einem grünen Balken neu.

Werkzeuge helfen uns bei der Durchführung von Rückschritten. Während Editoren *Undo*-Funktionalitäten anbieten, gehen aktuelle Entwicklungsumgebungen einen Schritt weiter: Sie bieten eine *lokale Historie*, mit der ältere Versionen wiederhergestellt werden können. Dadurch sinkt die Hemmschwelle, einen Rückschritt durchzuführen, da rückgängig gemachte Änderungen erhalten bleiben und notfalls wiederhergestellt werden können.

4.13 Vorprogrammierte Schwierigkeiten

Ausnahme 3b: Die Implementierung des Tests erfordert neue Methoden ... und damit weitere Tests

Seine schnellen Feedbackzyklen erfährt der Testansatz dadurch, dass jeder Test zunächst auf einfache Weise erfüllt wird und keine unnötige Komplexität entsteht. Sinnvolle Generalisierungen und Abstraktionen bilden sich erst nach Richtungsweisung weiterer Tests heraus. Test für Test entwickeln wir so, was unser Design für heute wirklich benötigt.

Manchmal können wir einen Test jedoch beim besten Willen nicht in einem Schritt erfüllen. Manchmal wäre vorab ein vereinfachendes Refactoring angebracht. Manchmal fehlen uns zur Implementierung eigentlich noch andere Methoden oder Objekte. Manchmal ist der Schritt, den unser Testfall nimmt, einfach viel zu groß und wir bohren uns weiter und weiter in die Tiefe, bis wir schließlich das Tageslicht verlieren und abbrechen müssen.

Auch dazu ein Beispiel: Unser letzter Testfall testet die Progression für Ausleihen von mehr als *vier Tagen*:

Tests:
– ~~1 Tag~~
– ~~3 Tage~~
– ~~4 Tage~~
– 5 Tage

```
public class RegularPriceTest...
  public void testChargingFiveDaysRental() {
    assertEquals(new Euro(4.50), price.getCharge(5));
  }
}
```

Erneut lassen wir uns von unserer Testsuite führen:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  expected:<EUR 4.5> but was:<EUR 3.0>
at RegularPriceTest.testChargingFiveDaysRental
  (RegularPriceTest.java:23)
```

Für diesen Testfall müssen wir für alle Zeiträume von mehr als drei Tagen den tagesabhängigen Eurobetrag mit den Tagen multiplizieren, die über die drei Tage hinausgehen:

***JUnit:** OK*

```
public class RegularPrice...
  public Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;

    int additionalDays = daysRented - DAYS_DISCOUNTED;
    return BASEPRICE.add(new Euro(PRICE_PER_DAY.getAmount()
                                  * additionalDays));
  }
}
```

Diese Implementierung ist hässlich: Um den Test möglichst einfach zu erfüllen, haben wir unser `Euro`-Objekt `PRICE_PER_DAY` erst wieder in den primitiven Typ `double` wandeln müssen, um es dort mit der Anzahl von Tagen zu multiplizieren, um dann sofort wieder ein `Euro`-Objekt zu bauen. Die Berechnung des Resultats lässt sich kaum noch lesen.

Wann immer ein Programm so aus der Form gerät wie jetzt gerade, will es uns etwas mitteilen: Hier sieht es danach aus, als fehle der `Euro`-Klasse nur die Möglichkeit, Geldbeträge zu vervielfachen. Wir müssen dieses Verständnis folglich in das Programm selbst zurückführen und können dadurch den Code wieder vereinfachen.

Erst zurück auf Grün, dann refaktorisieren

Definitiv richtig ist jedoch die Entscheidung, erst die einfache, unter Umständen hässliche Lösung hinzuschreiben, damit den Test zu erfüllen und dann zu refaktorisieren. Nachdem der Test grün ist, haben wir eine viel bessere Ausgangsbasis für eine funktionserhaltende Verbesserung des Designs.

Refaktorisieren Sie nur mit einem grünen Balken.

Die Disziplin dafür aufzubringen und durchzuhalten, das zu verschieben, was sich schon vor Ihrem geistigen Auge präsentiert, wenn auch nur für wenige Momente, ist ein Schlüsselaspekt dieses Prozesses. Bleiben Sie in der Nähe des grünen Lichts und versuchen Sie nicht, zu viele Bälle gleichzeitig zu jonglieren.

Arbeiten Sie nicht an verschiedenen Enden zur gleichen Zeit. Versuchen Sie stattdessen, Probleme nacheinander zu lösen, nicht gleichzeitig. Brechen Sie die Probleme dazu in kleinere auf und holen Sie sich öfter Feedback ein.

Unsere Verantwortung als Programmierer ist, Code so zu hinterlassen, dass er unsere Intention für andere Programmierer kommuniziert.

Die Aufgabe, den Eurobetrag mit einem Skalar zu multiplizieren, gehört zur `Euro`-Klasse selbst. Zeit also für einen weiteren Test:

```
public class EuroTest...
  public void testMultiplying() {
    Euro result = two.times(7);
    assertEquals(new Euro(14.00), result);
    assertEquals(new Euro(2.00), two);
  }
}
```

Wie es weitergeht, wissen Sie ja schon: Wir kompilieren und müssen erkennen, dass uns noch niemand die Arbeit für die `times`-Methode abgenommen hat:

```
Method times(int) not found in class Euro.
```

Solange sich der Code nicht fehlerfrei übersetzen lässt, können wir nicht die Tests ausführen, und solange sich die Tests nicht ausführen lassen, können wir nicht sagen, was der nächste Programmierzug ist. Ein leerer Methodenrumpf muss her:

```
public class Euro...
  public Euro times(int factor) {
    return null;
  }
}
```

Sofort gibt uns JUnit die rote Karte:

***JUnit:** Failure*

```
junit.framework.AssertionFailedError:
  expected:<14.0> but was:<null>
at EuroTest.testMultiplying(EuroTest.java:27)
```

Die Lösung dazu ist einfach:

***JUnit:** OK*

```
public class Euro...
  public Euro times(int factor) {
    return new Euro(cents * factor);
  }
}
```

Jetzt können wir die neue `times`-Methode in unsere Preisberechnung einbeziehen, wodurch sich der Code erheblich vereinfachen lässt:

***JUnit:** OK*

```
public class RegularPrice...
  public Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;

    int additionalDays = daysRented - DAYS_DISCOUNTED;
    return BASEPRICE.add(PRICE_PER_DAY.times(additionalDays));
  }
}
```

Meine Erfahrung sagt mir, dass es immer möglich ist, wie in diesem Beispiel zunächst die Abkürzung einzuschlagen und hinterher sofort zu refaktorisieren. Machen Sie Ihr eigenes Experiment!

Top-down-Design, Bottom-up-Implementierung

Wenn wir aber meinen, den aktuellen Testfall tatsächlich nur mit einer neuen Methode oder einer neuen Klasse möglichst einfach zum Laufen zu bekommen, dann müssen wir uns an den langsameren iterativen Abstieg machen. Dazu fangen wir eine neue Testfallmethode an und implementieren die neue Methode oder die neue Klasse auf die gewohnte testgetriebene Art und Weise. So hätten wir auch in dem Beispiel vorgehen können. Wir müssen dabei nur Acht geben, dass wir nicht länger tauchen gehen, als uns der Sauerstoff reicht. Ständiges Feedback erhalten wir bei der Bottom-up-Implementierung nur dann, wenn wir häufig genug auftauchen und die Top-Level-Tests ausführen können. Es ist sonst sehr leicht möglich, das eigentliche Ziel aus den Augen zu verlieren.

Indirekte Tests

Wenn wir zur Erfüllung des Testfalls zum Beispiel sofort die `times`-Methode verwendet und mitentwickelt hätten, wäre diese Methode schlussendlich nur indirekt durch einen Testfall der `RegularPriceTest`-Klasse mitgetestet worden. Hätten wir außerdem die Arbeit an dem Testfall `testMultiplying` begonnen, würden wir uns mit zwei fehlschlagenden Testfällen gleichzeitig herumschlagen. Das wäre Murks!

Offene Tests

Offene Tests, die wegen inkompatibler Umstellungen ruhig gestellt werden müssen, sich für eine Zeit lang gar nicht übersetzen lassen oder aus anderen Gründen fehlschlagen, deuten in diesem Prozess auf eine ungeschickte Zerlegung unserer Programmieraufgabe in Teilaufgaben und Testfälle hin. Unsere Absicht ist es aber, nur eine Methode vom grünen Balken entfernt zu bleiben, da uns sonst der Prozess zerbricht.

Zwei offene Tests sind einer zu viel

von Tammo Freese, freier Berater

Wir wollen mit nicht mehr als einem fehlschlagenden Test zur gleichen Zeit arbeiten. Warum eigentlich?

Jeder Test beweist seinen Wert dadurch, dass er den Testbalken von grün nach rot färbt. Jede Änderung am Code beweist ihren Wert dadurch, dass sie den einen, gerade fehlschlagenden Test erfüllt und damit den Balken wieder grün färbt.

Erfüllen wir mehrere fehlschlagende Tests auf einmal, beweist der einzelne Test nicht mehr seinen Wert. Unnötige Tests, die unseren Testcode aufblähen, fallen uns nicht mehr auf. Schlimmer noch ist, dass die Änderungen am Code zum Erfüllen mehrerer Tests auch umfangreicher ausfallen. Auf diese Weise entsteht leicht mehr Code, als unsere Tests fordern. Wir handeln uns also ungetesteten Code ein!

4.14 Kleine Schritte gehen

Nun haben Sie den Testgetriebenen Programmierzyklus kennen gelernt und werden zukünftig nur noch in kleinen Schritten programmieren, nicht wahr? – Ich hoffe nicht.

Entscheidend ist nicht, immer in winzigen Programmierschritten vorwärts zu gehen. Entscheidend ist zu verinnerlichen, wie klein die Schritte tatsächlich sein können.

In Ihrer täglichen Arbeit werden Sie die Schrittlänge ganz bewusst nachführen: Geht es gut voran, werden Sie mutiger, Sie machen einen größeren Hopser ... und fallen dadurch irgendwann einmal auf die Nase. Darauf reagieren Sie, werden wieder bescheidener, gehen kleinere Schritte ... und fangen irgendwann an, sich zu langweilen.

Eigenes Tempo finden

Tatsächlich hat jede Person und jede Situation ihr bevorzugtes Eigentempo: Finden Sie es heraus und seien Sie gewarnt, dass Sie sich dabei ab und an blutige Knie holen müssen. Anders lernen Sie niemals, wie weit Sie wirklich gehen können.

Meistern werden Sie diese Technik, wenn Sie Ihr Schritttempo zuversichtlich den aktuellen Bedürfnissen und örtlichen Begebenheiten anzupassen wissen.

Halten Sie Ihre Füße trocken

von Michael Feathers, Object Mentor

In meinen Trainingskursen sage ich gewöhnlich:

Unser Ziel ist es, einen Fluss mit trockenen Schuhen zu überqueren. Sie können von Stein zu Stein gehen oder zu den weiter entfernten Steinen springen. Wenn wir das andere Ufer erreicht haben, können wir alle Schuhe vergleichen.

5 Refactoring

Als Refactoring bezeichnen wir die Verbesserungen am Design eines Programms, die es verständlicher und einfacher änderbar machen, ohne dabei das existierende Verhalten zu ändern. Aus vorangehenden Kapiteln sollten Sie schon einen ersten Eindruck von dieser Technik gewonnen haben. Dieses Kapitel soll Ihnen zeigen, was Sie heute für Ihr Programm tun können, damit es morgen etwas für Sie tun kann. Refactoring investiert in den Wert der Software: allzeit sauberer Code.

5.1 Die zweite Direktive

Es liegt in der Natur von Software, dass sie weiterentwickelt und modifiziert wird. Mit wachsender Programmgröße wird dabei jedoch häufig die Codequalität in Mitleidenschaft gezogen. Insbesondere bei inkrementeller Entwicklung muss dem Verfall der Programmstruktur kontinuierlich entgegengewirkt werden. Ohne die fürsorgliche Pflege verkrustet Software, wird zu Hardware. Jeder Programmierer muss daher jede Möglichkeit zur Codeverbesserung jederzeit wahrnehmen:

2. Direktive: *Bringen Sie Ihren Code immer in die Einfache Form.*

Diese Regel sagt uns, wann Refactoring ins Spiel kommt: Wir wollen *sauberen* Code, der *funktioniert*. Also verbessern wir den Code durch zahlreiche Refactorings und führen nach jedem Schritt alle Tests aus. So vereinfacht Refactoring den geschriebenen Code:

- Um einen Test möglichst einfach zu erfüllen, können wir den Code vorher in Form bringen (refaktorisieren).
- Unmittelbar nachdem der Test läuft, müssen wir refaktorisieren und den Code in eine Einfache Form bringen.

Ein vorausschauendes Refactoring ist dabei als Kürteil anzusehen, wohingegen das abschließende Refactoring zum Pflichtteil gehört.

5.2 Die Refactoringzüge

In Verbindung mit der Testgetriebenen Programmierung geht die Rolle von Refactoring weit über das übliche Aufräumen von Code hinaus. Der Testgetriebene Entwicklungszyklus beschreibt dazu, wie Software in zahllosen kleinen Schritten entwickelt wird, die abwechselnd darauf abzielen, a) das geforderte Programmverhalten zu implementieren und b) die Codestruktur zu verbessern, ohne das Verhalten zu berühren.

Refactoring als fortlaufende Aktivität

Refactoring ist eine *fortlaufende* Aktivität. Als Wegbereiter findet es während der gesamten Entwicklungsdauer in kleinen Zügen statt: Entweder Sie refaktorisieren, um Raum für neuen Code zu schaffen, oder Sie bringen einen Test zum Laufen und refaktorisieren direkt im Anschluss. Stoßen Sie auf Codeteile, die schwer verständlich sind oder unnötig kompliziert erscheinen, verbessern und vereinfachen Sie sie.

Die Refactoringzüge in Kürze:

1. Spüren Sie schlecht riechenden Code auf.
2. Planen Sie eine Refactoringroute, um das Design zu verbessern.
3. Refaktorisieren Sie in kleinen Schritten und testen Sie häufig.

Das Ziel von Refactoring ist Code in einer möglichst einfachen Form. Wenn Ihnen während der Programmierung also eine Möglichkeit zur Vereinfachung auffällt, notieren Sie die Idee, bevor sie verloren geht. Halten Sie immerzu Ausschau nach Codeteilen, die ein Refactoring vertragen könnten. Oft offenbaren sie sich in Form von Codegerüchen, wie Indizien für nötiges Refactoring metaphorisch bezeichnet werden.

Refaktorisieren Sie Ihren Code, wenn Sie die Erkenntnis in ein einfacheres Design gefunden haben, nicht jedoch inmitten der Programmierung. Machen Sie sich dann lieber eine Notiz.

Organische Natur

Wahre Aha-Momente werden Sie erleben, wenn Sie Ihren Code als etwas Organisches begreifen, das Sie durchkneten und formen können. Lernen Sie, Ihrem Code ein offenes Ohr zu schenken: Denn tatsächlich erzählen Sie dem Code mit jedem Testfall, was Sie von ihm verlangen. Sie stellen ihm die Frage, wie sich der neue Code ins bestehende System einfügt. Ein roter Testbalken wäre eine klare Antwort. Aber auch sonst gibt Ihnen Ihr Code häufig kleine Hinweise, die Sie mental als roten Balken interpretieren sollten: »Ich bin hässlich!«, »Ich werde zu fett!« oder »Niemand versteht mich!«. Es mag eine Spur esoterisch klingen, aber Sie müssen Ihrem Programm zuhören: Dass uns der Code erzählt, wie er strukturiert werden möchte, ist eine überaus nützliche Illusion.

5.3 Von übel riechendem Code ...

Schritt 1: Spüren Sie schlecht riechenden Code auf

Objektive Kriterien für gutes Design aufzustellen fällt relativ schwer. Zwar orientieren wir uns während des Refactorings auch an allgemein anerkannten Designprinzipien und wohlbekannten Entwurfsmustern, doch leichter fällt es, wenn wir uns auf schlechtes Design verständigen: auf subjektive Indizien für verbesserungsbedürftige Codestrukturen. Wir bezeichnen diese als *schlechte Codegerüche* und nehmen sie als Anlass zum Refactoring. Ihre Merkmale sind mit Bedacht unscharf gewählt, damit Sie ganz und gar Ihrer menschlichen Intuition und Ihrem feinen Geruchssinn folgen können.

Entwickeln Sie eine sensible Nase für Code, der Ihnen stinkt, und legen Sie sich einen Maßstab zu, was guten Code ausmacht.

Code Smells

Kent Beck und Martin Fowler beschreiben im Refactoringbuch [fo99] über zwanzig verschiedene »Code Smells«, die Sie beizeiten in Ihren Wortschatz übernehmen könnten. Damit Sie wissen, worauf besonders zu achten ist, möchte ich hier einen Überblick über die am häufigsten anzutreffenden Duftnoten geben:

- **Duplizierte Logik** ist der unbestrittene Anführer der Gestanksliga und die häufigste Ursache, um Projekte zum Erlahmen zu bringen. Das Problem damit ist, dass jede Kopie, ob sie nun bewusst oder unbewusst entstanden ist, bald ihr eigenes Leben beginnt und Information dadurch über mehrere Orte verstreut und künftig schwer zu ändern ist.
- **Große Klassen und lange Methoden** verschlechtern die Mobilität, Lokalisation, Verwendung und Dokumentation von Codestücken und sind zudem noch Nährboden für schleichende Duplikation.
- **Schlechte Namen und unpassende Codestrukturen** erhöhen die Lernkurve, um die Funktionsweise des Programms zu verstehen, und ziehen so nach und nach weitere Codegerüche nach sich.
- **Spekulative Verallgemeinerungen** sind Designelemente, die früher einmal zur Flexibilisierung eingeplant waren, jedoch nie benötigt wurden und damit nur im Weg stehen.
- **Kommentare** werden häufig als Deodorant für schlecht riechenden Code missbraucht und nach gründlichem Refactoring überflüssig. Nicht entbehrlich sind Kommentare, die Entwurfsentscheidungen festhalten, veröffentlichte Schnittstellen in Javadoc dokumentieren oder Literaturhinweise geben.

5.4 ... über den Refactoringkatalog

Schritt 2: Planen Sie eine Refactoringroute, um das Design zu verbessern

Design ist etwas Organisches, das im Fluss bleiben und sich unaufhörlich neuen Anforderungen und Erkenntnissen anpassen muss. Design umfasst unsere Bemühungen, alle konkurrierenden Kräfte so weit auszubalancieren, dass wir zu jedem Zeitpunkt sauberen Code haben. Entwurfsentscheidungen von gestern können sich im heutigen Licht jedoch immer auch als unvorteilhaft erweisen. Dabei wäre es falsch, am überholten Verständnis haften zu bleiben.

Jede Ihrer Entwurfsentscheidungen ist umkehrbar.

Software ist änderbar. Ohne ihre Semantik zu verändern, können Sie

- Klassen teilen oder zusammenführen,
- Methoden und Attribute in andere Klassen verschieben,
- Klassen, Methoden, Parameter und Variablen umbenennen,
- Methoden extrahieren oder integrieren,
- Vererbung durch Delegation ersetzen oder andersherum sowie
- Schnittstellen einziehen.

Erneut möchte ich auf das hervorragende Refactoringbuch verweisen, in dem Martin Fowler für über 70 Refactorings detailliert beschreibt, wie Sie Ihren Code diszipliniert in kleinen Schritten umstellen können. Obwohl Sie auf Ihre Testsuite als Sicherheitsnetz zurückfallen können, müssen Sie wissen, wie Sie die einzelnen Refactorings systematisch in sicheren Schritten durchführen und welche Risiken Sie damit eingehen. Fürs Erste können Sie auch in den Onlinekatalog hineinschauen:

`http://www.refactoring.com`

Wenn Sie eine unangenehme Duftnote in Ihrem Code feststellen, suchen Sie nach einem Refactoring, das diesen Codegeruch bekämpft. Als sehr hilfreich erweisen sich dabei Was-wäre-wenn-Betrachtungen, um verschiedene Entwurfsalternativen in Erwägung zu ziehen.

Stellen Sie sich vor, wie ein Refactoring das Design beeinflussen würde.

Werkzeugunterstütztes Refactoring erlaubt Ihnen ferner, verschiedene Refactoringzüge live am Code zu testen. Denken Sie dran: Jeder Zug ist reversibel. Experimentieren Sie einfach, spielen Sie mit Ihrem Code!

5.5 ... zur Einfachen Form

Schritt 3: Refaktorisieren Sie in kleinen Schritten und testen Sie häufig

Refactoring zielt stets auf eine kleine Verbesserung in Richtung eines insgesamt einfacheren Designs. Unsere Kriterien für Einfachheit sind dabei wiederum bewusst subjektiv ausgelegt. Nach Priorität geordnet: **Ein Design hat die Einfache Form, wenn der Code**

1. **alle seine Tests erfüllt.** Refaktorisieren Sie nur Code, der so gut getestet ist, dass Sie voller Vertrauen ans Werk gehen können. Ein missglücktes Refactoring, das nicht sofort erkannt wird, kann die Arbeit von Stunden zunichte machen. Aus diesem Grund müssen Sie sich mitunter bremsen und erst mal weitere Tests schreiben, um dann mit den geplanten Refactoringzügen fortschreiten zu können.
2. **seine Intention klar ausdrückt.** Sie sind dafür verantwortlich, alle Ihre Ideen durch den Programmcode zu kommunizieren: Code wird öfter gelesen und geändert als geschrieben. Deshalb sind verständliche Namen selbst für kleinste Codeteile wichtig. Entscheidend ist, dass Sie die semantische Distanz vergrößern zwischen der Information, was der Code tut und wie er es tut: Drücken Sie in den Namen stets Ihre Intention aus, nicht die Implementierung. Indem Sie die Implementierungsdetails dann nur wiederum durch die Intention auf dem nächstniedrigeren Abstraktionsniveau ausdrücken, können Sie Code schreiben, der sich zum größten Teil selbst dokumentiert.
3. **keine duplizierte Logik enthält.** Schreiben Sie nie die gleiche Zeile Code zweimal. Selbst kleine Codeduplikate riechen übel, da Sie den Code später an mehreren Stellen ändern müssten. Stattdessen sollten Sie Duplikation als Lernerlebnis betrachten: Eine neue Abstraktion wartet in Ihrem Code darauf, entdeckt und ins Leben refaktorisiert zu werden.
4. **möglichst wenig Klassen und Methoden umfasst.** Das bedeutet nicht, dass Sie Schweizertaschenmesserklassen programmieren, sondern kurioserweise werden Sie viele, sehr kleine Klassen mit vielen kurzen Methoden haben. Anders könnten Sie auch gar nicht die vorher genannten Kriterien erfüllen. Der einzige Weg, um wirklich zu weniger Klassen und Methoden zu kommen, besteht darin, unnötige Designelemente wieder zu entfernen: Es existiert eine Grauzone, wo eine größere Methode unsere Absicht besser vermittelt als zwei kleinere Methoden oder wo Abstraktion die Intention verbirgt und Duplikation okay ist.

5.6 Überlegungen zur Refactoringroute

Refactoring sollte sich wie ein Lauffeuer über weite Systemteile ausbreiten: Im Idealfall machen wir eine Änderung an einer Klasse und können darauf den gesamten abhängigen Code vereinfachen. Bingo!

Weitreichendes Refactoring bringt aber auch Probleme mit sich: Das Refactoringziel muss durch eine Folge kleiner Schritte erreichbar sein, sonst sehen wir unseren Code tatsächlich in Flammen aufgehen. Refaktorisieren wir stets in kleinen Schritten und testen jeden Schritt, dann geben sich Fehler unmittelbar zu erkennen.

Ohne Disziplin und ein wenig Voraussicht gelingt es jedoch nicht, Refactoring in Babyschritten zu verwirklichen: Einige Refactorings verändern die öffentliche Schnittstelle der refaktorisierten Klasse und ziehen so zwangsläufig Anpassungen des abhängigen Codes nach sich. Wenn wir während solcher Refactorings allerdings Tests und Code gleichzeitig modifizieren, würden wir die Möglichkeit verschenken, Veränderungen am beobachtbaren Verhalten des Codes festzustellen. Außerdem würden wir leicht das eigentliche Refactoringziel aus den Augen verlieren. Eine bessere Strategie ist, Refactorings zu halbieren:

Refactoring aufteilen

- **Implementierungssubstitution** berührt die Implementierungsdetails einer Klasse. Tests und Verwender bleiben davon unbetroffen.
- **Schnittstellenevolution** berührt primär die öffentliche Schnittstelle. Tests und Verwender sind deshalb von Anpassungen betroffen.

Der Rest des Kapitels wird sich hauptsächlich mit dieser Fragestellung beschäftigen: Wie refaktorisieren wir Code in so winzigen Schritten, dass wir in kürzester Zeit wieder kompilierbaren und somit testbaren Code erhalten. Wir werden uns dazu einiger Codegerüche annehmen, die wir im geschriebenen Code zurückgelassen haben:

- Die `add`-Methode der `Euro`-Klasse sollte `plus` heißen.
- Die `Customer`- und `Movie`-Objekte rechnen mit `double` statt `Euro`.
- Die `Movie`-Klasse hat eine Preiskategorie verschluckt.
- Die Klassen `Movie` und `RegularPrice` enthalten Duplikation.

Refactorings:
– ~~Euro: add in plus umbenennen~~
– Customer und Movie: auf Euro umstellen
– Movie: Preis extrahieren
– Movie und RegularPrice: Code faktorisieren

Unser erstes Refactoring ist so einfach, dass ich den Code dafür nicht extra abdrucken werde: Die Umbenennung von `add` nach `plus` können Sie entweder von Hand oder durch *Suchen und Ersetzen* durchführen. Vorsicht jedoch, dass Sie die `additionalDays` nicht in `plusitionalDays` umbenennen. Schneller geht Refactoring mit *Werkzeugunterstützung* vonstatten: Einfach `add` markieren, »Rename« wählen, `plus` eintippen und das Werkzeug ersetzt automatisch die drei Vorkommen im Code. Da das Tool mitdenkt, passieren dabei kaum noch Fehler.

5.7 Substitution einer Implementierung

Als Nächstes werden wir die Innereien der Movie-Klasse austauschen. Wenn Sie sich die Klasse ansehen, erkennen Sie an den Kommentaren der zwei Preiskonstanten, dass der Typ Euro unsere Absicht viel besser dokumentiert als der primitive Datentyp double:

```
public class Movie {
  private static final double BASE_PRICE = 2.00;  // Euro
  private static final double PRICE_PER_DAY = 1.75;  // Euro
  private static final int DAYS_DISCOUNTED = 2;

  public static double getCharge(int daysRented) {
    double result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      result += (daysRented - DAYS_DISCOUNTED) * PRICE_PER_DAY;
    }
    return result;
  }
}
```

Code Smell:
– Primitive Obsession

Hier am Seitenrand gebe ich Ihnen einige Hinweise auf Code Smells und Refactorings aus dem empfohlenen Buch von Martin Fowler [fo99]

Stellen Sie sicher, dass die Tests Ihnen ausreichend Sicherheit fürs Refactoring geben. Fügen Sie sonst nötige Tests hinzu.

Da unsere Klassen gut getestet sind, können wir uns sofort an die Arbeit machen. Dazu ersetzen wir die Konstanten durch Wertobjekte, löschen die Kommentare und schwenken die Implementierung unserer getCharge-Methode in einem Rutsch auf die Euro-Klasse um. Auf dem Weg dahin ziehen wir additionalDays als *erklärende Variable* heraus. Die Schnittstelle der Klasse bleibt vorerst unberührt:

Refactoring:
– Replace Data Value with Object
– Introduce Explaining Variable

JUnit: *OK*

```
public class Movie {
  private static final Euro BASE_PRICE = new Euro(2.00);
  private static final Euro PRICE_PER_DAY = new Euro(1.75);
  private static final int DAYS_DISCOUNTED = 2;

  public static double getCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(PRICE_PER_DAY.times(additionalDays));
    }
    return result.getAmount();
  }
}
```

5.8 Evolution einer Schnittstelle

Bisher wird die Euro-Klasse nur im Verborgenen verwendet. Damit sich unser Refactoring möglichst weit über den Code erstrecken kann, müssen wir die Änderung auch in der Schnittstelle der Movie-Klasse reflektieren.

Eine Änderung an den Schnittstellen wirkt sich unmittelbar auf alle abhängigen Systemteile aus. Einerseits wollen wir diesen Effekt, andererseits fürchten wir ihn. Typischerweise sind unsere Aufwände, die zur Umstellung aller Verwender auf die neue Benutzungsweise entstehen, relativ hoch. Aus diesem Grund hat sich eine recht änderungsunfreundliche Kultur entwickelt, die von den zwei Ideen getrieben ist, Schnittstellen möglichst wenig zu ändern und bei Änderung Abwärtskompatibilität zu erreichen.

Diese Richtlinien sind wichtig, wenn wir nicht über die Quellen des abhängigen Codes verfügen oder gar Schnittstellen veröffentlichen, doch in allen anderen Fällen sind auch Schnittstellen leicht änderbar. Dennoch können uns die beiden Regeln bei unserer Routenplanung behilflich sein: Ein jeder Refactoringschritt sollte für sich genommen

- Schnittstellen möglichst wenig ändern und
- bei Änderung Abwärtskompatibilität erreichen.

Refactoring:
– Extract Method

Hier ist eine Refactoringroute, die ich von Tammo Freese gelernt habe: Wir extrahieren zunächst den gesamten Inhalt der getCharge-Methode außer der return-Anweisung in der letzten Zeile in eine temporäre Methode, die wir zum Beispiel tmpCharge nennen. Auf diese Weise erhalten wir mit der neuen Methode die gewünschte neue Signatur:

***JUnit:** OK*

```
public class Movie...
  public static double getCharge(int daysRented) {
    Euro result = tmpCharge(daysRented);
    return result.getAmount();
  }

  public static Euro tmpCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }
}
```

Anschließend ersetzen wir die Variable in der getCharge-Methode mit dem Ausdruck selbst. Die Methode erfüllt so eine *Adapterfunktion*, um die Abwärtskompatibilität zum bestehenden Code herzustellen:

Refactoring:
– Inline Temp

```
public class Movie...
  public static double getCharge(int daysRented) {
    return tmpCharge(daysRented).getAmount();
  }
}
```

***JUnit:** OK*

Im nächsten und entscheidenden Schritt betten wir den Rumpf der getCharge-Methode überall dort ein, wo die Methode verwendet wird. Dadurch können wir die Adapterfunktion aus der Movie-Klasse in den abhängigen Code ausrollen. Refactoringtools helfen, alle Vorkommen zu ersetzen: vier in der MovieTest-Klasse, eines in der Customer-Klasse:

Refactoring:
– Inline Method

```
public class MovieTest extends TestCase {
  public void testBasePrice() {
    assertEquals(2.00, Movie.tmpCharge(1).getAmount(), 0.001);
    assertEquals(2.00, Movie.tmpCharge(2).getAmount(), 0.001);
  }

  public void testPricePerDay() {
    assertEquals(3.75, Movie.tmpCharge(3).getAmount(), 0.001);
    assertEquals(5.50, Movie.tmpCharge(4).getAmount(), 0.001);
  }
}

public class Customer...
  public void rentMovie(int daysRented) {
    totalCharge += Movie.tmpCharge(daysRented).getAmount();
  }
}
```

***JUnit:** OK*

Coding Standards bewusst verletzen

von Tammo Freese, freier Berater

Bei einigen Refactorings werden in Zwischenschritten Methoden oder Klassen verwendet, die nach Abschluss des Refactorings wieder verschwunden sein sollen. Diesen temporären Strukturen gebe ich gerne Namen, die den Coding Standard verletzen. Ein hässlicher Name wie getCharge_OLD fällt gut ins Auge und damit ist es sehr wahrscheinlich, dass ich ihn entferne. Und sollte ich es vergessen, informiert der Name den Nächsten, der den Code liest.

Refactoring:
– Rename Method

Durch die Integration der `getCharge`-Methode ist die `Movie`-Klasse von der alten Signatur losgekommen und der Name `getCharge` wieder frei. Als Nächstes benennen wir die Methode `tmpCharge` in `getCharge` um und haben damit die wesentliche Schnittstellenänderung vollzogen:

***JUnit:** OK*

```
public class Movie...
  public static Euro getCharge(int daysRented)...
}
```

Übrig bleibt die Vereinfachung der Codeteile, die die Adapterfunktion aufgenommen haben. Um die `Movie`-Klasse überhaupt vereinfachen zu können, mussten ihre Verwender komplizierter werden. So ist es oft: Damit eine Klasse schöner wird, muss eine andere hässlicher werden:

```
public class MovieTest extends TestCase {
  public void testBasePrice() {
    assertEquals(2.00, Movie.getCharge(1).getAmount(), 0.001);
    assertEquals(2.00, Movie.getCharge(2).getAmount(), 0.001);
  }

  public void testPricePerDay() {
    assertEquals(3.75, Movie.getCharge(3).getAmount(), 0.001);
    assertEquals(5.50, Movie.getCharge(4).getAmount(), 0.001);
  }
}
```

Abschließend stellen wir die Verwender einen nach dem anderen um, so dass die Adapterfunktion `getAmount` am Ende nicht mehr verwendet wird. Die Tests lassen sich umformen wie algebraische Gleichungen:

***JUnit:** OK*

```
public class MovieTest...
  public void testBasePrice() {
    assertEquals(new Euro(2.00), Movie.getCharge(1));
    assertEquals(2.00, Movie.getCharge(2).getAmount(), 0.001);
  }
}
```

Refactorings:
– ~~Customer und Movie: auf Euro umstellen~~
– Movie: Preis extrahieren
– Movie und RegularPrice: Code faktorisieren

Auf entsprechende Art und Weise können Sie auch die `Customer`-Klasse von der `double`- auf die `Euro`-Verwendung umstellen. Entscheidend ist, dass Sie kleine Schritte finden und möglichst nach jedem Schritt testen.

Schnittstellenänderungen bleiben trotz Refactoringunterstützung in den Entwicklungswerkzeugen vorerst eine aufwändige Maßnahme. Dennoch sollte diese Refactoringroute anschaulich gemacht haben, wie klein unsere Schritte sein können, um wirklich jede Minute einen testbaren Stand zu erreichen.

5.9 Teilen von Klassen

Unser drittes Refactoring ist äußerst typisch für die organische Natur der Programmevolution. Der Code wächst mit seinen Anforderungen: Klassen und Methoden werden häufig modifiziert, um noch weitere Fälle abzudecken. Auf diese Weise kann eine Klasse zu groß oder eine Methode zu lang werden. Indem wir ein Programm jedoch in kleinere Teile spalten, entsteht neuer Raum, in den neuer Code hineinwachsen kann ... bis dieser Zyklus schließlich aufs Neue beginnt.

Nur indem wir den Code daran hindern, ins Uferlose zu wuchern, kommen wir überhaupt dazu, die nächsten Refactorings zu entdecken. Tatsächlich ist es oft so, dass wir erst nach einem Refactoring weitere Vereinfachungsmöglichkeiten erkennen, die im unstrukturierten Code gar nicht sichtbar waren. So auch hier: Die `Movie`-Klasse dupliziert die Geschäftsregeln zur Berechnung der Ausleihgebühr. Wirkliche Ursache dafür ist, dass die Klasse eine andere Klasse in ihrem Bauch trägt:

```
public class Movie...
  public static Euro getCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }
}
```

Code Smell:
– Duplicated Code

Die Geburt einer neuen Klasse beginnt:

***JUnit:** OK*

```
public class Movie...
  public static Euro getCharge(int daysRented) {
    return tmpCharge(daysRented);
  }

  public static Euro tmpCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }
}
```

Refactoring: *– Extract Class*

Anschließend lösen wir die neue Klasse heraus: Dazu verschieben wir alle Konstanten und die temporär umbenannte tmpCharge-Methode in ihre neue Heimat. Der extrahierte Preis gilt für Neuerscheinungen:

JUnit: *OK*

```
public class NewReleasePrice {
  private static final Euro BASE_PRICE = new Euro(2.00);
  private static final Euro PRICE_PER_DAY = new Euro(1.75);
  private static final int DAYS_DISCOUNTED = 2;

  public static Euro tmpCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }
}
```

Und nun können wir auch die getCharge-Methode zurückbenennen:

JUnit: *OK*

```
public class NewReleasePrice...
  public static Euro getCharge(int daysRented)...
}
```

Da in der Movie-Klasse jetzt nur noch die Delegation zurückbleibt, könnten wir argumentieren, dass sich die Klasse gar nicht mehr trägt:

```
public class Movie {
  public static Euro getCharge(int daysRented) {
    return NewReleasePrice.getCharge(daysRented);
  }
}
```

Erinnern Sie sich jedoch an die Kriterien für Einfachheit: Die Intention auszudrücken ist wichtiger als die Anzahl von Klassen zu minimieren. Die getCharge-Methode kommuniziert das Was getrennt vom Wie. Deshalb können wir den Code nicht integrieren.

Neue Klassen können wie in diesem Beispiel durch Zergliederung der Originalklasse entstehen oder durch Kombination vorhandener Einzelteile zu einem neuen Ganzen. Beide Wege sind sich ähnlich. Während der Zergliederung testen wir den extrahierten Code jedoch zunächst nur indirekt über die Originalklasse und extrahieren erst im Anschluss eine Menge von Tests. Bei der Kombination dagegen gehen wir direkt von neuen Tests aus, hauptsächlich um daraus zu lernen.

5.10 Verschieben von Tests

Sobald sich Code von einer Klasse zu einer anderen Klasse bewegt, müssen die zugehörigen Tests mitwandern. Der Grund dafür ist, dass die Unit Tests möglichst nahe einer Fehlerursache fehlschlagen sollen. Je weiter sich die Tests jedoch vom zu beobachtenden Code entfernen, desto schlechter lassen sich Fehlerquellen einkreisen.

Indirekte Tests

Aus diesem Grund werden indirekte Tests, in denen eine Klasse nur durch die Tests einer anderen Klasse mitgetestet wird, ab einer gewissen Distanz ineffektiv. Oft teilen wir eine Klasse gerade deshalb, damit wir nach der Extraktion die Codeteile testen können, die vorher schlecht testbar waren. Vielleicht müssen wir uns sogar eingestehen, dass der Code ursprünglich eigentlich gar nicht gut getestet war.

Verschieben wir Code vom getesteten Ort zum ungetesteten Ort, müssen wir darauf achten, dass die Tests nicht ihren Fokus verlieren:

- Müssen relevante Tests nachziehen?
- Müssen weitere Tests für die neue Klasse hinzukommen?
- Können zurückgebliebene Tests wegfallen?

Da die bestehenden Tests nicht mehr die Codestruktur widerspiegeln, verschieben wir hier die Tests von der `MovieTest`- in die `NewReleasePriceTest`-Klasse:

***JUnit:** OK*

```
public class NewReleasePriceTest extends TestCase {
  public void testBasePrice() {
    assertEquals(new Euro(2.00), NewReleasePrice.getCharge(1));
    assertEquals(new Euro(2.00), NewReleasePrice.getCharge(2));
  }

  public void testPricePerDay() {
    assertEquals(new Euro(3.75), NewReleasePrice.getCharge(3));
    assertEquals(new Euro(5.50), NewReleasePrice.getCharge(4));
  }
}
```

In der `MovieTest`-Klasse lassen wir sicherheitshalber einen Test zurück, um die Delegation der `Movie`- zur `NewReleasePrice`-Klasse zu testen:

Refactorings:
– ~~Movie: Preis extrahieren~~
– Movie und RegularPrice: Code faktorisieren

```
public class MovieTest extends TestCase {
  public void testUsingNewReleasePrice() {
    assertEquals(new Euro(3.75), Movie.getCharge(3));
  }
}
```

5.11 Abstraktion statt Duplikation

Unser nächstes Refactoring soll die Duplikation zwischen den Klassen `NewReleasePrice` und `RegularPrice` auflösen. Wie in der Mathematik wollen wir die gemeinsamen Faktoren finden und herausfaktorisieren. Zum Teil hat uns das letzte Refactoring bereits den Weg ebnen können und die Preisberechnung von der `Movie`- in die `NewReleasePrice`-Klasse verschoben:

```
public class NewReleasePrice {
  private static final Euro BASE_PRICE = new Euro(2.00);
  private static final Euro PRICE_PER_DAY = new Euro(1.75);
  private static final int DAYS_DISCOUNTED = 2;

  public static Euro getCharge(int daysRented) {
    Euro result = BASE_PRICE;
    if (daysRented > DAYS_DISCOUNTED) {
      int additionalDays = daysRented - DAYS_DISCOUNTED;
      result = result.plus(PRICE_PER_DAY.times(additionalDays));
    }
    return result;
  }
}
```

Zum direkten Vergleich die Preisklasse aus dem vorherigen Kapitel:

```
public class RegularPrice {
  private static final Euro BASEPRICE = new Euro(1.50);
  private static final Euro PRICE_PER_DAY = new Euro(1.50);
  private static final int DAYS_DISCOUNTED = 3;

  public Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;

    int additionalDays = daysRented - DAYS_DISCOUNTED;
    return BASEPRICE.plus(PRICE_PER_DAY.times(additionalDays));
  }
}
```

Wenn wir die beiden Klassen miteinander vergleichen, scheint es eine Menge von Gemeinsamkeiten und Unterschieden zu geben:

- Die Konstanten tragen die gleichen Namen, mit einer Ausnahme.
- Die Werte der Konstanten sind ganz unterschiedlich.
- Die Preisberechnungen ähneln einander, gehen jedoch verschiedene Wege.

Mehrfache Vorkommen gleicher oder sich ähnelnder Codeschnipsel sind stets Indiz dafür, dass uns im Code zur Vereinheitlichung noch eine entscheidende Idee fehlt. So kann uns Codeduplikation auch ihre gute Seite zeigen und uns neue Klassen und Methoden finden lassen.

Zuvor müssen wir jedoch sicherstellen, dass sich die zwei Klassen auch tatsächlich zusammenbringen lassen. Dazu entscheiden wir uns für die besser verständliche der beiden Preisberechnungsmethoden und tauschen die andere gegen sie aus:

Refactoring:
– Substitute Algorithm

```
public class NewReleasePrice {
  private static final Euro BASEPRICE = new Euro(2.00);
  private static final Euro PRICE_PER_DAY = new Euro(1.75);
  private static final int DAYS_DISCOUNTED = 2;

  public static Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;

    int additionalDays = daysRented - DAYS_DISCOUNTED;
    return BASEPRICE.plus(PRICE_PER_DAY.times(additionalDays));
  }
}
```

***JUnit:** OK*

Soweit geht es gut, auch wenn wir dadurch Duplikation par excellence erhalten. Loswerden können wir sie durch Delegation oder Vererbung. Da unsere beiden Preisklassen ohnehin Schnittstellenverwandschaft pflegen, könnten wir einmal die Vererbungsbeziehung probieren.

Mit welcher der zwei Klassen wir beginnen, ist prinzipiell egal. Weil die `NewReleasePrice`-Klasse aber eine statische Methode enthält, wäre `RegularPrice` einfacher. Wir extrahieren also die neue Oberklasse und schieben die Methode und gemeinsamen Felder nach oben:

Refactoring:
– Extract Superclass

```
public class Price {
  private static final Euro BASEPRICE = new Euro(1.50);
  private static final Euro PRICE_PER_DAY = new Euro(1.50);
  private static final int DAYS_DISCOUNTED = 3;

  public Euro getCharge(int daysRented) {
    if (daysRented <= DAYS_DISCOUNTED) return BASEPRICE;

    int additionalDays = daysRented - DAYS_DISCOUNTED;
    return BASEPRICE.plus(PRICE_PER_DAY.times(additionalDays));
  }
}

public class RegularPrice extends Price {
}
```

***JUnit:** OK*

Gerade wenn wir im Team entwickeln, kann uns häufiger das Gefühl befallen, dass wir doppelten Code haben. Oft können wir uns nicht mehr exakt erinnern, wo wir den Code schon einmal gesehen haben. Selbst wenn wir den Code finden, können wir manchmal nicht sagen, ob die Teile wirklich ähnlich sind oder nicht. Ein solcher Moment erfordert die Geduld, die Redundanz im Programmcode aufzulösen. Die stabilsten Abstraktionen finden nicht selten über den Umweg der Duplikation ihren Weg in den Code.

Wenn Sie das Déjà-vu-Erlebnis haben, dass Ihnen eine Zeile Code bekannt erscheint, finden Sie die fehlende Abstraktion, um die beiden Strukturen geeignet zusammenzuführen.

Jede Programmidee nur an einer Stelle

Eine andere Formulierung der Kriterien für ein einfaches Design ist: Jede Idee muss sich im Programmcode einmal und wirklich nur einmal niederschlagen. Das führt typischerweise zu vielen kleinen Klassen. Eine Trennung von Klassen mit unterschiedlichen Änderungsraten führt dann auf natürliche Art und Weise zu Framework-Mechanismen im Code. Flexibilität entsteht, wo sie benötigt wird.

Wenn wir unsere Preisberechnung betrachten, dann gibt es auch hier Elemente, die von Fall zu Fall variieren, und solche, die fix sind. Unsere Konstanten können beispielsweise nicht länger Konstanten bleiben, sondern müssen zu Variationspunkten werden. Im ersten Schritt erklären wir sie deshalb als Variablen:

***JUnit:** OK*

```
public class Price {
  private Euro basePrice = new Euro(1.50);
  private Euro pricePerDay = new Euro(1.50);
  private int daysDiscounted = 3;

  public Euro getCharge(int daysRented) {
    if (daysRented <= daysDiscounted) return basePrice;

    int additionalDays = daysRented - daysDiscounted;
    return basePrice.plus(pricePerDay.times(additionalDays));
  }
}
```

Als Nächstes müssen wir uns überlegen, wie wir unseren Unterklassen den Zugriff auf die Variationspunkte der Oberklasse ermöglichen. Eine Möglichkeit besteht darin, in das *Template Method* Pattern [ga95] zu refaktorisieren, um die Werte in den Unterklassen durch Einschubmethoden redefinieren und somit variieren zu können. Vielleicht geht es aber auch noch einfacher?

Wenn wir die Felder der Oberklasse über ihren Konstruktor setzen könnten, wäre uns ebenso weitergeholfen:

```
public class Price...
  private Euro basePrice;
  private Euro pricePerDay;
  private int daysDiscounted;

  Price(Euro basePrice, Euro pricePerDay, int daysDiscounted) {
    this.basePrice = basePrice;
    this.pricePerDay = pricePerDay;
    this.daysDiscounted = daysDiscounted;
  }
}
```

Die RegularPrice-Klasse wäre so in der Lage, die Werte festzulegen, mit denen die Oberklasse rechnet:

***JUnit:** OK*

```
public class RegularPrice extends Price {
  public RegularPrice() {
    super(new Euro(1.50), new Euro(1.50), 3);
  }
}
```

Die Anpassungen der NewReleasePrice-Klasse laufen entsprechend ab mit dem Unterschied, dass der abhängige Code noch vom statischen Methodenaufruf auf eine Instanz umgestellt werden muss:

```
public class NewReleasePrice extends Price {
  public NewReleasePrice() {
    super(new Euro(2.00), new Euro(1.75), 2);
  }
}
```

***JUnit:** OK*

```
public class Movie {
  public static Euro getCharge(int daysRented) {
    return new NewReleasePrice().getCharge(daysRented);
  }
}
```

Refactorings:
~~– Movie und RegularPrice: Code faktorisieren~~

Viele Restrukturierungen enden so, dass variable Teile in separate Klassen verlagert werden und damit die ursprüngliche Klasse weniger häufig modifiziert werden muss. Unsere Price-Klasse ist jetzt beispielsweise unabhängig von den Werten des angewendeten Preismodells. Dennoch fühlen sich die beiden Unterklassen irgendwie seltsam an, nicht wahr?

5.12 Die letzte Durchsicht

Zum Ende einer Refactoringepisode gehen wir meist noch einmal über alle betreffenden Klassen und stellen sicher, dass das Programm auch als Ganzes kommuniziert, keinerlei Duplikation enthält und so weiter. Auf diese Weise wird das Design in jedem Durchgang weiter verfeinert.

Wenn wir auf unsere Episode zurückschauen, bleiben zwei Fragen:

- Lohnen sich die Konzepte `RegularPrice` und `NewReleasePrice` noch oder tragen sie zu wenig Verantwortung?
- Müssen die zwei Klassen wirklich noch wie gehabt getestet werden oder ist stattdessen die Oberklasse zu testen?

Refactoring:
– Collapse Hierarchy

Rhetorisch wie die Fragen sind, ist die Antwort klar: Die Unterklassen unterscheiden sich kaum von ihrer Oberklasse und bringen keinen zusätzlichen Nutzen. Wir können die Klassen zusammenführen und dadurch unser gesamtes Programm inklusive der Tests vereinfachen:

```
public class Price...
  public final static Price NEWRELEASE =
    new Price(new Euro(2.00), new Euro(1.75), 2);
  public final static Price REGULAR =
    new Price(new Euro(1.50), new Euro(1.50), 3);
}

public class Movie {
  public static Euro getCharge(int daysRented) {
    return Price.NEWRELEASE.getCharge(daysRented);
  }
}
```

***JUnit:** OK*

```
public class PriceTest extends TestCase {
  public void testBasePrice() {
    assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(1));
    assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(2));
  }

  public void testPricePerDay() {
    assertEquals(new Euro(3.75), Price.NEWRELEASE.getCharge(3));
    assertEquals(new Euro(5.50), Price.NEWRELEASE.getCharge(4));
  }
}
```

Aufpassen müssen wir nur, dass unsere Intention und wichtige Namen im Programm erhalten bleiben. Ansonsten können wir durch dieses Refactoring vier Klassen löschen: die Unterklassen und ihre Tests.

5.13 Ist Design tot?

Sie sollten bemerkt haben, wie sich der Prozess in diesem Kapitel von vorigen Kapiteln unterscheidet. Sind wir bisher fast schnurstracks geradeaus gegangen, scheint der Weg hier eher chaotisch zu verlaufen: Wir haben Methoden neu benannt, Klassen umgebaut, Klassen geteilt, Klassen miteinander verschmolzen und mehr. Oft höre ich die Frage, ob wir das Design nicht gleich richtig hinbekommen können?

Es hängt davon ab, wie viel Sie lernen müssen und wie schnell Sie Ihre Missverständnisse korrigieren können.

Es sieht vielleicht danach aus, als müssten Sie nach jeder Änderung stundenlang refaktorisieren. Tatsächlich müssen Sie aber nur so lange refaktorisieren, wie Sie dazulernen.

Es mag sich zusätzlich der Eindruck erhärten, das Design in UML vorausplanen zu können. Tatsächlich ist das möglich, wenn Sie ohne das Feedback fertig werden, das Ihnen der programmierte Code liefert. Die Mehrzahl der Codegerüche drückt sich unterhalb der Auflösungsmöglichkeiten der UML aus, weil vom Code an sich abstrahiert wird. Schlechte Namen und Mikroduplikation sind in Diagrammform fast unaufspürbar. Die Grundlage fürs Refactoring wird damit entzogen. Zwangsläufig kommt dies einer Verzichtserklärung fürs Lernen gleich. Ohne evolutionäre Designstrategie à la Refactoring müssen Sie Ihr Design auf Teufel komm raus richtig hinbekommen.

Anforderungen ändern sich

Die wirkliche Herausforderung ist, dass Software weiterentwickelt werden will: Die Anforderungen ändern sich mit der Zeit. Das ergibt sich schon daraus, dass die Anwender der Software auch dazulernen. Wir können annehmen, dass wir das allgemeine Lernen beschleunigen können, indem wir Software früher und häufiger ausliefern.

Design zieht nach

Mit der Herausforderung, auf späte und unklare Anforderungen einzugehen, kommt die Voraussetzung, das Design auf die aktuellen Anforderungen nachziehen zu können. Wenn also überhaupt eine Designphase existiert, dann ganz zu Beginn der Softwareentwicklung. Anschließend müssen wir den organischen Anpassungsfähigkeiten des Programmcodes vertrauen. Stellen Sie sich etwa vor, dass zwischen den gezeigten Refactorings jeweils ein paar Wochen Zeit vergangen wären. Vielleicht schätzen Sie den Wert der Techniken auf diese Art höher.

Ist Design also tot? Nein! Geplantes Design ist weiterhin wichtig, doch nicht mehr so kritisch. Wir planen immer noch ein wenig Design im Voraus. Wenn sich unsere Ideen hinterher aber als unvollkommen entpuppen, refaktorisieren wir. Die Rolle von Design ändert sich also. In seinem Artikel »Is Design Dead?« [fo$_{00}$] diskutiert Martin Fowler sehr lesenswert das Thema *geplantes kontra evolutionäres Design*.

Durch zerbrochene Fenster dringen Gerüche ein

von Dave Thomas und Andy Hunt, The Pragmatic Programmers

Geruch ist ein erstaunlicher Sinn. Tiere benutzen ihn, um Paarungspartner zu finden und Raubtiere wahrzunehmen, um gutes Futter ausfindig zu machen und schlechtes Futter zu vermeiden. Die olfaktorischen Rezeptoren in unseren Nasen sind über die Hirnrinde mit unserem limbischen System verbunden, dem Teil des Gehirns, der oft als Gefühlszentrum bezeichnet wird. Damit ist es kein Wunder, dass Gerüche so leicht Erinnerungen wachrufen: Der kleinste Hinweis eines bestimmten alten Parfüms oder der Hauch von Kampfer einer Mottenkugel reichen aus, zu einem Nachmittag ins Haus unserer Großmutter in unsere Kindheit zurückzureisen.

Dieses Erwecken von Erinnerungen macht Geruch zu einer so guten Metapher, um problematischen Code zu erkennen. Wir schauen auf eine Seite Code und unser Kopf beginnt, Warnsignale auszusenden: Irgendetwas ist hier faul, es riecht schlecht. Wir graben deshalb weiter, bis wir die Ursache entdeckt haben, und beheben sie.

Obgleich der Duft manchmal auch ganz schwach sein kann. Das entdeckte Problem scheint allzu unbedeutend: eine überflüssige Zuweisung vielleicht oder ein Stück Code, das nicht ausgeführt wird. Ist nicht so schlimm, reden wir uns vielleicht ein. Bei all den anderen Problemen, denen wir gegenüberstehen, lassen wir dieses vielleicht gerade in Ruhe. Wir kommen später darauf zurück.

Natürlich kommt dieses Später nie.

Ein SIGSOFT-Papier [1] zeigt, warum es eine schlechte Idee ist, diese Probleme liegen zu lassen. Die Autoren benutzten einen modifizierten C-Compiler, um nach bestimmten Arten von einfachen Programmierfehlern im Linux-Betriebssystem zu suchen. Ihr Tool schaute nach offensichtlichen Redundanzen, Code, der allem Anschein nach unnütz ist. Dazu gehörten idempotente Operationen, Variablen, denen zwar Werte zugewiesen wurden, die anschließend aber nie gelesen wurden, und Code, der niemals ausgeführt wurde. Sie entdeckten dabei etwas Wertvolles: Dateien mit Code, der diese Redundanzen hatte, zeigte eine Wahrscheinlichkeit von 45 – 100 %, schwerwiegende Fehler zu enthalten: Fehler, die zum Systemabsturz führen können. Sie folgerten, dass diese schlechten Programmierpraktiken, also diese leichten Gerüche, von Entwicklern verursacht worden waren, die verwirrt und weniger kompetent waren, und dass diese Entwickler noch andere schwieriger zu erkennende Fehler einführen würden.

Im Kriminalitätssektor und im städtischen Verfall existiert eine interessante Parallele. In einer bekannten Studie [2] wollten Forscher herausfinden, warum einige Gebäude in Armengegenden in schäbige, von Verbrechen heimgesuchte Ruinen verfielen, während andere, ebenso gefährdete Viertel überlebten. Ihre überraschende Schlussfolgerung ist, dass etwas so Unbedeutendes ausreicht wie eine zerbrochene Fensterscheibe.

Ein Fenster geht zu Bruch, vielleicht nur aus Versehen, aber es bleibt unrepariert. Dann geht vielleicht ein weiteres Fenster zu Bruch. Dieses Mal ist es eventuell kein Unfall mehr. Als Nächstes tauchen Graffitis auf, gefolgt von Abfall. Kleinere Sachbeschädigungen setzen ein, dann beginnen die Mieter zu flüchten. Größere Beschädigungen am Gebäude ereignen sich, bis der Eigentümer es aufgibt, weil die Instandsetzungskosten ihre Investition übersteigt. Nun ziehen die Ganoven ein und alles ist verloren. Sobald diese Eskalation einsetzt, geht der Verfall schnell voran und ist schwer zu stoppen. Wenn Sie aber früher eingreifen können, können Sie das Viertel retten.

Wenn wir die kleinen Probleme beheben, wie sie sich tagtäglich ergeben, haben wir uns mit weniger großen Problemen abzukämpfen. Die Polizei von New York City hat die »No Broken Windows«-Philosophie als Teil von Bürgermeister Giulianis Kampf gegen die Kriminalität übernommen. Indem die kleinen Probleme schnell in Angriff genommen werden, Graffiti, Verschmutzung, Bettler, konnten sie die Kriminalitätsrate deutlich senken: Die Anzahl von Schwerverbrechen, Morden, Raubüberfällen und Einbruchdiebstählen wurde von 1993 bis 1997 halbiert.

Die Lektion ist klar. Nichts ist zu klein, um behoben zu werden, und jedem Geruch, egal wie schwach, muss nachgegangen werden. Trivialprobleme sind offenbar Anzeichen dafür, dass es noch weitere Probleme gibt. Und unbehandelt zurückgelassen gerät jedes kleine Problem zum Nährboden für eine noch ernstere Angelegenheit.

Was sollten Sie also tun, wenn Ihre Nase zu zucken beginnt? Die erste und offensichtliche Sache ist herauszufinden warum. Spüren Sie den schuldigen Code auf und analysieren Sie, was das wirkliche Problem ist. Entscheiden Sie sich dann für eine konstruktive Vorgehensweise. Beheben Sie das Problem, wenn möglich, an Ort und Stelle. Wenn Ihnen die dafür nötige Zeit oder Information fehlt, machen Sie mit dem Code, was man auch mit einem kaputten Fenster machen würde: mit Brettern vernageln. Fügen Sie Kommentare ein, die das Problem beschreiben (und vergessen Sie nicht ein Signalwort zu setzen, nach dem Sie in Zukunft schnell den gesamten Quellcode nach dieser Art von Kommentaren durchsuchen können). Ziehen Sie in Erwä-

gung, Zusicherungen einzufügen, die zur Laufzeit signalisieren, dass der schadhafte Code ausgeführt wird, oder eine Meldung in die Logdatei zu schreiben, die Sie daran erinnert, dass das Problem noch nicht behoben ist.

Nachdem Sie dieses besondere Problem angegangen sind, ist die Arbeit jedoch erst zur Hälfte getan. Sie erinnern sich, dass diese trivialen Probleme oft nur ein Zeichen dafür sind, dass in der Nähe noch größere Gefahren lauern. Nehmen Sie sich die Zeit und gehen Sie die anderen Teile der Quelldatei durch, die den Fehler enthielt. Suchen Sie nach Wiederholungen dieses Problems, beschränken Sie sich jedoch nicht nur darauf. In der gleichen Weise, wie Sie Ihre Sinne schärfen, wenn Sie durch ein übles Viertel laufen, müssen Sie Ausschau halten nach allem, was verdächtig riecht und nicht ganz koscher aussieht. Beheben Sie die Probleme, die Sie finden.

Nachdem Sie das getan haben, sind Sie immer noch nicht ganz fertig. Wenn ein Entwickler diesen besonderen Fehler in dieser Datei einmal gemacht hat, ist die Wahrscheinlichkeit ziemlich hoch, dass derselbe Fehler noch an anderer Stelle im Code der Person auftaucht. Nehmen Sie sich die Zeit, um dieser Spur nachzugehen.

Für uns hat sich diese Technik bezahlt gemacht. Dave schaute einmal durch den Quellcode einer hochgradig nebenläufigen Java-Serveranwendung, als ihm auffiel, dass eine Methode, die eine Liste von Listenern aktualisierte, ohne das Schlüsselwort `synchronized` deklariert war. Unter Last bestand eine marginale Chance, dass sie nicht korrekt funktionierte. Es schien eine Kleinigkeit zu sein, die schnell zu beheben ist. Beim Nachbohren entdeckte er jedoch, dass der Entwickler der fehlerhaften Methode offenbar nicht verstanden hatte, wie man in nebenläufigen Umgebungen programmieren muss: Der Code war voll von Synchronisationsproblemen. Wir fanden und behoben sie, bevor die Anwendung ausgeliefert wurde, ein sehr viel einfacherer Job, als zu versuchen, scheinbar zufällige Fehler beim Endbenutzer zu analysieren.

Gewöhnen Sie sich also an, Ihrer Nase zu vertrauen. Selbst der kleinste Hinweis einer unangenehmen Duftnote kann Sie durch Nachforschen zu einem ganzen Misthaufen faulen Codes führen. Und je früher Sie ihn finden, desto einfacher können Sie ihn beheben.

[1] *Y. Xie and D. Engler:* Using Redundancies to Find Errors. SIGSOFT 2002/FSE-10.

[2] *J. Q. Wilson and G. Kelling:* The police and neighborhood safety. The Atlantic Monthly 249(3). 1982.

5.14 Richtungswechsel ...

Zu guter Letzt möchte ich Ihnen noch ein größeres Beispiel für das organische Codewachstum geben. Dabei werden wir Refactoring mit den Programmiertechniken aus dem vorherigen Kapitel kombinieren. Das Beispiel soll zeigen, wie Sie ein bestehendes System in möglichst kleinen Schritten an neue Anforderungen anpassen können.

Neue Aufgabenstellung

Unsere Aufgabe wird sein, eine Rechnung für die Ausleihvorgänge des DVD-Verleihs zu erstellen: Das Programm erhält die Information, welche Filme ein Kunde wie lange ausgeliehen hat. Die Software soll diese Daten zwischenspeichern und eine Reihe von Berichten drucken. Als ersten Bericht erwartet unser Auftraggeber eine Aufstellung von Einzelposten und Gesamtbetrag pro Kunde. Besonders zu beachten ist, dass die Preise von Filmen häufigen Änderungen unterworfen sind und in den Rechnungen der zuletzt genannte Preis zugrunde gelegt wird.

Puh! Wie und wo fangen wir an?

Nun, bevor wir losprogrammieren können, müssen wir uns erst ein paar Fragen stellen:

- Was ist unsere Aufgabe?
- Wie passen die Anforderungen ins Gesamtbild?
- Welche Systemteile werden wir berühren und anpassen müssen?
- Wo liegen die interessanten Grenzfälle?
- Welche Klassen und Methoden benötigen wir?
- Wie werden die Klassen benutzt?
- Wie können wir das testen?

Die Fragen lenken unsere Aufmerksamkeit zunächst einmal fort von den Entscheidungen über die Implementierungsdetails hin zur Analyse des erforderlichen Programmverhaltens und zum Entwurf geeigneter Klassenschnittstellen.

In der Regel werden Sie nicht alle Klassen im Kopf haben, sondern einen Blick in den Code werfen müssen. Einige Entwickler bevorzugen dafür UML. Andere analysieren das Problem lieber im Kreis mit Ihren Kollegen und entwerfen erste Lösungen an einer Wandtafel.

Grundsätzlich haben wir zwei Möglichkeiten vorzugehen:

- **Top-down** geht von den allgemeinen Klassen aus und arbeitet sich schrittweise zu den spezifischen Klassen vor.
- **Bottom-up** fängt im Gegensatz mit den Teilen an und setzt aus ihnen nach und nach das Ganze zusammen.

Wie orientiert sich Testgetriebene Entwicklung? Eher top-down oder bottom-up? Es hängt davon ab: vom Bekannten zum Unbekannten.

5.15 ... und der wegweisende Test

Unsere Arbeitsweise ist weder rein top-down noch klar bottom-up. Wir starten meist top-down mit den zu erfüllenden Anforderungen, suchen uns einen geeigneten Aufsetzpunkt, drücken im Test unsere Intention aus und identifizieren auf diese Art Klassen und Methoden. Durch den Top-down-Test wissen wir, dass wir nur implementieren, was wir wirklich brauchen.

Nachdem wir die ersten Klassen und Methoden gefunden haben, geht die Entwicklung häufig stellenweise zum Bottom-up-Ansatz über. So verschieben wir unseren Fokus kurzzeitig auf die untergeordneten Klassen und Methoden, die wir zum Aufbau der übergeordneten Strukturen benötigen. Diesen Verfeinerungsprozess wiederholen wir, bis letztlich alle Anforderungen implementiert sind.

Ein größerer Test führt zum Ziel

Für größere Aufgaben schreibe ich mir oft einen größeren Test. Durch das Niederschreiben stelle ich mein Verständnis auf die Probe und die Entwicklung erhält eine Richtung. Dieser erste Test ist mehr Integrationstest als Unit Test. Er ist ein funktionaler Test, bezieht sich also auf das gewünschte Zusammenspiel mehrerer beteiligter Objekte. Was also ist unser Ziel?

```
public class CustomerTest...
  private Movie buffalo66, jungleBook, pulpFiction;

  protected void setUp()...
    buffalo66 = new Movie("Buffalo 66", Price.NEWRELEASE);
    jungleBook = new Movie("Das Dschungelbuch", Price.REGULAR);
    pulpFiction = new Movie("Pulp Fiction", Price.NEWRELEASE);
  }

  public void testPrintingStatement() {
    customer.rentMovie(buffalo66, 4);
    customer.rentMovie(jungleBook, 1);
    customer.rentMovie(pulpFiction, 4);

    buffalo66.setPrice(Price.REGULAR);

    String actual = customer.printStatement();
    String expected = "\tBuffalo 66\t3,00\n"
                    + "\tDas Dschungelbuch\t1,50\n"
                    + "\tPulp Fiction\t5,50\n"
                    + "Gesamt: 10,00\n";
    assertEquals(expected, actual);
  }
}
```

Wie bringt uns dieser Test weiter? Er ist uns ein nützliches Designwerkzeug. Er hilft uns, bestimmte Entwurfsentscheidungen zu treffen:

Entwurfsentscheidungen

- **Umfang begrenzen:** Wir schreiben den Test so, dass er uns verrät, was zu tun ist und was nicht.
- **Startpunkt setzen:** Wir beginnen mit ein oder zwei bekannten oder offensichtlichen Klassen.
- **Kontext beachten:** Wichtig sind nur Probleme, die wir jetzt haben, nicht solche, die wir später haben könnten.
- **Szenarien durchspielen:** Wir starten beim Bekannten und fragen uns durch Was-wäre-wenn-Betrachtungen zum Unbekannten vor.
- **Mitwirkende Objekte finden:** Wir brechen das zu lösende Problem in Teilprobleme.
- **Verantwortlichkeiten verteilen:** Verantwortlichkeiten teilen wir auf vorhandene Klassen auf oder fügen neue Klassen hinzu.
- **Verhalten spezifizieren:** Wir definieren das Protokoll der Klassen, indem wir die Eingabe zur Ausgabe im Test in Beziehung setzen.

Tatsächlich können wir unsere Fragen über das Design beantworten, indem wir ein wenig Testcode schreiben, um die Tragfähigkeit und Vollständigkeit des Designs schnell im Code unter Beweis zu stellen. Der funktionale Test ist demnach auch ein Designtest: Unser Design kristallisiert sich nicht nur aus der Verwendungsperspektive heraus, durch die Betrachtung des Zusammenwirkens verschiedener Klassen werden auch Verantwortlichkeiten ausgeglichen verteilt. Im Beispiel:

***JUnit:** Failure*

```
public class Movie...
  public Movie(String title, Price price) {
  }

  public void setPrice(Price price) {
  }
}

public class Customer...
  public void rentMovie(Movie movie, int daysRented) {
  }

  public String printStatement() {
    return null;
  }
}
```

Der Test bietet so eine Lernumgebung, um im engen Feedbackzylus mit dem Code mit alternativen Designideen zu experimentieren.

5.16 Fake it ('til you make it)

Kent Beck erzählt die Geschichte von einem Buch, das er vor 30 Jahren gelesen hat, in dem der unbekannte Autor vorschlägt, für ein Eingabeband das erwartete Ausgabeband einzutippen und dann solange zu programmieren, bis das wirkliche Ergebnis dem erwarteten Ergebnis entspricht. Eine Programmiertechnik, die Kent dazu sehr passend »Fake it 'til you make it« (zu deutsch sinngemäß »Solange vorgaukeln, bis du's geschafft hast«) [be_{02a}] getauft hat, geht diesen Weg:

***JUnit:** OK*

```
public class Customer...
  public String printStatement() {
    return "\tBuffalo 66\t3,00\n"
         + "\tDas Dschungelbuch\t1,50\n"
         + "\tPulp Fiction\t5,50\n"
         + "Gesamt: 10,00\n";
  }
}
```

Unsere erste Antwort auf den roten Balken ist die erwartete Konstante selbst. Dadurch duplizieren wir zwar die Information aus dem Test, erhalten aber wenigstens unseren grünen Balken zurück und damit sofort auch neuen Auftrieb. Unsere Aufgabe reduziert sich jetzt darauf, die spezielle Lösung schrittweise zur allgemeinen Lösung zu machen.

Als geübter Programmierer werden Sie die »Fake«-Lösung oft nur kurz anschauen müssen und sofort sehen, wie Sie konstante Werte durch wirkliche Objekte ausdrücken können. Die Technik erleichtert auf diese Weise selbst die Programmierung komplizierter Aufgaben.

Viele Programmierer wissen eigentlich nicht, wann sie überhaupt fertig sind, und programmieren asymptotisch der Perfektion entgegen. Wenn wir hier alle Konstanten durch berechnete Werte ersetzt haben, können wir sicher sein, dass wir aufhören können zu programmieren.

Allgemeine Lösung durch weitere Tests motivieren

Während dieser Generalisierung stoßen wir gewöhnlich auf Codestellen, für die wir weitere Unit Tests schreiben müssen. Wir treiben die Entwicklung der verallgemeinerten Lösung also durch neue Tests an, wo es sinnvoll ist. In unserem Fall können wir die Testliste fast eins zu eins aus dem geschriebenen Code ableiten:

- Movie: Filmtitel
- Movie: Berechnung verschiedener Preise
- Movie: Reklassifizierung

Die Preissummierung dagegen sollte wie gehabt funktionieren.

5.17 Vom Bekannten zum Unbekannten

Der beschriebene Weg setzt eine bestimmte Entwicklungsweise voraus: Arbeiten wir uns vom Bekannten zum Unbekannten vor, entwickeln wir verschiedene Klassen quasi parallel. Tatsächlich ändern wir zwar nur einen Test auf einmal, dennoch berühren wir rundum alle Systemteile, die zur gewünschten neuen Programmeigenschaft beitragen.

Ein Problem, das sich uns stellt, besteht darin, die Zeiten kurz zu halten, um den Balken einzugrünen und neuen Code zu stabilisieren. Zur möglichst einfachen Testerfüllung müssen wir »Fake it« deshalb unter Umständen auch rekursiv anwenden. Andernfalls bohren wir uns durch die wachsende Menge von Dingen, die wir implementieren müssen, tiefer und tiefer in die Innereien des Systems. Das Ergebnis dieses Top-down-Abstiegs wäre während der gesamten Bottom-up-Implementierung ungewiss, bis alle unsere Tests wieder laufen.

Wir können versuchen, uns solche Überraschungen zu verkneifen, indem wir etwas mehr Grips in den Testplan stecken und damit hoffen, auf unserer Route über weniger widerborstige Fallstricke zu stolpern. Die Reihenfolge der Testfälle ist entscheidend! Womit fangen wir an?

Den Test `testTitle` können wir uns nach kurzer Überlegung wohl gut und gerne sparen. Bei `get`- und `set`-Methoden oder Konstruktoren, die keine Logik enthalten, passieren uns doch keine Fehler mehr, oder? Der Konstruktor wird anderswo sowieso auf indirekte Art mitgetestet. Ich spare mir hier also einmal mehr, den geschriebenen Code auch abzudrucken.

Tests:
- ~~*Movie: Filmtitel*~~
- *Movie: Berechnung verschiedener Preise*
- *Movie: Reklassifizierung*

Weiter: `Movie`'s `getCharge`-Methode ist immer noch als statische Klassenmethode definiert. Damit sie irgendwann mit den Preisklassen rechnet, stellen wir im Test schon einmal auf eine Instanz um:

JUnit: *OK*

```
public class MovieTest extends TestCase {
  public void testUsingNewReleasePrice() {
    Movie movie = new Movie("Fight Club", Price.NEWRELEASE);
    assertEquals(new Euro(3.75), movie.getCharge(3));
  }
}
```

Im nächsten Schritt müssen wir die Verwender der `getCharge`-Methode umstellen. Davon gibt es einen: `Customer`. Müssten wir mehrere Stellen ändern, könnte die schon beschriebene Schnittstellenevolution helfen.

Hinderlich ist dennoch, dass das `Customer`-Objekt bisher weder ein `Movie`-Objekt zum Argument hat noch den Filmtitel und -preis kennt, um es sich selbst zu erzeugen. Wir haben die `rentMovie`-Methode zwar schon mit der gewünschten Signatur überladen, aber sie ist noch leer.

Damit wir nicht noch eine Baustelle aufmachen, gehen wir den Weg deshalb einfach rückwärts. Das führt uns jedoch zu hässlichem Code:

***JUnit:** OK*

```
public class Customer...
  public void rentMovie(int daysRented) {
    Movie movie = new Movie(null, null);
    totalCharge = totalCharge.plus(movie.getCharge(daysRented));
  }
}
```

Wir haben schon einen Test geschrieben, der uns später daran erinnert, die unsinnige Programmzeile zu entfernen. Dennoch schadet es nicht, sich so kleine Programmiersünden auf einem Stück Papier zu notieren.

Memo:
– Customer: rentMovie

Die Abkürzung bringt uns unserem Ziel gleich ein Stück näher:

***JUnit:** OK*

```
public class Movie...
  public static Euro getCharge(int daysRented) {
    return Price.NEWRELEASE.getCharge(daysRented);
  }
}
```

Danach verbinden wir die beiden `rentMovie`-Methoden miteinander:

***JUnit:** OK*

```
public class Customer...
  public void rentMovie(Movie movie, int daysRented) {
    totalCharge = totalCharge.plus(movie.getCharge(daysRented));
  }

  public void rentMovie(int daysRented) {
    Movie movie = new Movie(null, null);
    rentMovie(movie, daysRented);
  }
}
```

Nachdem wir das bewerkstelligt haben, können wir den `CustomerTest` auf die neue Schnittstelle umklemmen:

***JUnit:** OK*

```
public class CustomerTest...
  public void testRentingOneMovie() {
    customer.rentMovie(pulpFiction, 1);
    assertEquals(new Euro(2.00), customer.getTotalCharge());
  }
}
```

Memo:
– ~~Customer: rentMovie~~

Haben wir das ebenso für den `testRentingThreeMovies`-Testfall hinter uns gebracht, können wir die hässliche `rentMovie`-Methode löschen.

Kurze Verschnaufpause. Wo sind wir? Wohin wollen wir?

Unser `Movie`-Objekt erhält als Zustand `title` und `price`. Allerdings geht das `Price`-Objekt im Konstruktor verloren. Damit unsere Klasse auch mit anderen Preisen als nur Neuerscheinungen zurechtkommt, spendieren wir ihr eine Instanzvariable:

```
public class Movie...
  private Price price = Price.NEWRELEASE;

  public Euro getCharge(int daysRented) {
    return price.getCharge(daysRented);
  }
}
```

JUnit: *OK*

Um uns dieser festkodierten `Price`-Instanz zu entledigen, müssen wir aus dem Test heraus erzwingen, dass die `Movie`-Klasse auch wirklich an das `Price`-Objekt delegiert, mit dem es ursprünglich initialisiert wurde:

```
public class MovieTest...
  public void testUsingRegularPrice() {
    Movie movie = new Movie("Brazil", Price.REGULAR);
    assertEquals(new Euro(1.50), movie.getCharge(3));
  }
}
```

JUnit: *Failure*

Das ist ein Selbstgänger:

```
public class Movie...
  private Price price;

  public Movie(String title, Price price) {
    this.title = title;
    this.price = price;
  }
}
```

JUnit: *OK*

Tests:
- ~~*Movie: Berechnung verschiedener Preise*~~
- *Movie: Reklassifizierung*

Die Berücksichtigung individueller Preise haben wir damit erledigt. Übrig bleibt die Neuklassifizierung. Wir ändern dazu den letzten Test:

```
public class MovieTest...
  public void testSettingNewPrice() {
    Movie movie = new Movie("Brazil", Price.NEWRELEASE);
    movie.setPrice(Price.REGULAR);
    assertEquals(new Euro(1.50), movie.getCharge(3));
  }
}
```

JUnit: *Failure*

Dieser Code ist an und für sich keinen Test wert. Es ist jedoch wichtig, im Test auszudrücken, dass ein Film seine Preisklasse wechseln kann:

***JUnit:** OK*

```
public class Movie...
  public void setPrice(Price price) {
    this.price = price;
  }
}
```

Tests:

– ~~Movie: Reklassifizierung~~

Langsam kommen wir dem Ziel näher. Damit auf der Rechnung der letzte Preis erscheint, müssen wir die Ausleihvorgänge selbst speichern:

```
import java.util.*;

public class Customer...
  private List rentals = new ArrayList();

  public void rentMovie(Movie movie, int daysRented) {
    totalCharge = totalCharge.plus(movie.getCharge(daysRented));
    rentals.add(new Rental(movie, daysRented));
  }
}
```

So kommen wir zu einem *Assoziationsobjekt* zwischen der `Customer`- und `Movie`-Klasse:

***JUnit:** Failure*

```
public class Rental {
  public Rental(Movie movie, int daysRented) {
  }
}
```

Die Verantwortlichkeit der `Rental`-Klasse beschränkt sich also darauf, die Assoziation zu halten, wie lange ein Kunde einen Film geliehen hat:

```
public class RentalTest extends TestCase {
  public void testUsingMovie() {
    Movie movie = new Movie("Blow-Up", Price.NEWRELEASE);
    Rental rental = new Rental(movie, 2);
    assertEquals(new Euro(2.00), rental.getCharge());
  }
}
```

***JUnit:** Failure*

```
public class Rental...
  public Euro getCharge() {
    return null;
  }
}
```

Um für den Ausleihvorgang die aktuelle Gebühr festzustellen, delegiert das `Rental`-Objekt dann an das verknüpfte `Movie`-Objekt:

JUnit: OK

```
public class Rental {
  private Movie movie;
  private int daysRented;

  public Rental(Movie movie, int daysRented) {
    this.movie = movie;
    this.daysRented = daysRented;
  }

  public Euro getCharge() {
    return movie.getCharge(daysRented);
  }
}
```

Schwenken wir die Berechnung des Gesamtbetrags auf die Liste von `Rental`-Objekten um, hat auch die `totalCharge`-Variable ausgedient:

JUnit: OK

```
public class Customer...
  public Euro getTotalCharge() {
    Euro result = new Euro(0);
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      result = result.plus(rental.getCharge());
    }
    return result;
  }
}
```

Unser Ziel ist in greifbare Nähe gerutscht und unsere Ungeduld steigt. Wir wollen endlich mit unserer »Fake«-Lösung weiterkommen und Konstanten durch Variablen ersetzen. Der Ausdruck zur Berechnung des Gesamtbetrags verspricht besonders schnellen Erfolg:

JUnit: Failure

```
public class Customer...
  public String printStatement() {
    return "\tBuffalo 66\t3.00\n"
         + "\tDas Dschungelbuch\t1.50\n"
         + "\tPulp Fiction\t5.50\n"
         + "Gesamt: " + getTotalCharge() + "\n";
  }
}
```

Dieser Schnellschuss erweist sich als Fehler. JUnit sagt uns, wo es hakt: Das Objekt berechnet den Wert zwar bereits, aber das Ausgabeformat ist falsch: `Euro`'s `toString`-Repräsentation liefert uns nämlich `EUR 10.0` und selbst die `getAmount`-Methode liefert nicht die gefragte Menge von Nachkommastellen.

Aller Anfang ist schwer. Ein spezielles Format muss her:

```
public class Customer...
  public String printStatement() {
    return "\tBuffalo 66\t3.00\n"
         + "\tDas Dschungelbuch\t1.50\n"
         + "\tPulp Fiction\t5.50\n"
         + "Gesamt: " + getTotalCharge().format() + "\n";
  }
}
```

Programming by Intention

Wir verwenden die noch zu testende `format`-Funktionalität einfach so, als ob sie schon realisiert wäre. Das ist Rückwärtsprogrammieren, englisch »Programming by Intention« genannt.

Aus dem letzten Kapitel wissen Sie schon, dass wir diese Technik während der Testgetriebenen Programmierung in den Tests anwenden. Tatsächlich ist es jedoch nicht der Testcode, der die neuen Klassen und Methoden erzwingt, sondern der Anwendungscode, der sie benötigt. Deshalb ist es besonders hilfreich, das Rückwärtsarbeiten schon beim Anwendungscode zu beginnen. So finden wir, was wir brauchen:

***JUnit:** Failure*

```
public class Euro...
  public String format() {
    return null;
  }
}
```

... und spezifizieren es schon Sekunden später:

***JUnit:** Failure*

```
public class EuroTest...
  public void testFormatting() {
    assertEquals("2,00", two.format());
  }
}
```

Jetzt haben wir jedoch zwei fehlschlagende Tests!! Wäre ein weiteres »Fake it« vielleicht eine gute Alternative gewesen?

Hier ist das nur halb so schlimm, weil wir ziemlich genau wissen, dass die beiden Fehlschläge die gleiche Ursache haben. Läuft der eine, läuft der andere. Wetten?

JUnit: OK

```
import java.text.NumberFormat;

public class Euro...
  public String format() {
    NumberFormat format = NumberFormat.getInstance();
    format.setMinimumFractionDigits(2);
    return format.format(getAmount());
  }
}
```

Ich muss zugeben, ich musste erst einmal die `NumberFormat`-Klasse nachschlagen. Schneller wäre das vermutlich über die Bühne gegangen, wäre die Klasse auch durch entsprechende Unit Tests dokumentiert.

Ich muss ferner gestehen, dass mein erster Versuch daneben ging. Ich hatte kurzerhand den `setMinimumFractionDigits`-Aufruf vergessen.

Im zweiten Versuch liefen die Tests wieder und die Summierung gilt damit als abgeschlossen. Bald ist es geschafft!

Die `Rental`-Funktionalität können wir nun auch zur Aufstellung der Einzelposten heranziehen. Damit hauchen wir der `printStatement`-Methode endlich wirkliches Leben ein:

```
public class Customer...
  public String printStatement() {
    String result = "";
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      result += "\t" + rental.getMovieTitle()
                + "\t" + rental.getCharge().format() + "\n";
    }
    result += "Gesamt: " + getTotalCharge().format() + "\n";
    return result;
  }
}
```

Inmitten der Implementierung fällt noch auf, dass wir eine zusätzliche Methode zur Ausgabe der Filmtitel benötigen. Voilà!

JUnit: OK

```
public class Rental...
  public String getMovieTitle() {
    return movie.getTitle();
  }
}
```

Geschafft?

5.18 Retrospektive

Zeit zur Reflexion: Was können wir aus der letzten Aufgabe lernen? Werfen wir dazu einen Blick zurück:

metrics for the test-infected programmer
Copyright 2002 Frank Westphal. All rights reserved.

Monday, 30.12.2002

14:17 - 14:47 h

Delta Tests:	+3
Longest Red Bar:	07min 11sec
Longest Break:	04min 01sec
Average:	01min 20sec
Median:	00min 48sec

Abb. 5–1
Die Herztöne der Testgetriebenen Entwicklung

JUnit-EKG

Die Abbildung zeigt ein JUnit-EKG. Ursprünglich einmal für meine Trainingskurse entwickelt, sammelt diese kleine JUnit-Erweiterung Informationen über den Testprozess selbst und generiert daraus einige nützliche Metriken. Das Werkzeug fördert so die Selbstbeobachtung.

Die Zeitlinie zeigt die gesamte Programmierepisode: Ein Segment repräsentiert eine Minute. Die Länge des Testbalkens stellt die Anzahl ausgeführter Testfälle dar: Ein Segment entspricht zehn Tests.

Das EKG lässt erkennen, wie ich einen guten Start hingelegt habe, aber nach zwanzig Minuten und zahlreichen roten Testbalken in Serie vom Königsweg abgekommen bin.

Was verraten die Statistiken?

- **Delta Tests:** Drei neue Unit Tests habe ich geschrieben, neben dem funktionalen Test zum Beginn der Aufgabe.
- **Longest Red Bar:** Sieben Minuten lang habe ich rot sehen müssen. Und das nur, weil ich die `NumberFormat`-Klasse nicht im Kopf hatte.
- **Longest Break:** Vier Minuten vergingen, während ich der Klasse nachgegangen bin.
- **Average:** Im Mittel habe ich die Tests alle 80 Sekunden ausgeführt.
- **Median:** Ohne die Ausreißer habe ich alle 48 Sekunden getestet.

Sind das gute Werte? Es hängt davon ab, was wir in den Zahlen sehen. Die Metriken sind ein Spiegel. Sie dienen der persönlichen Reflexion, nicht zur Bewertung von Arbeitsleistungen.

5.19 Tour de Design évolutionnaire

Werfen wir noch einen Blick darauf, wie sich das Design entfaltet hat. Achten Sie dabei nicht so sehr auf die individuellen Zwischenstände. Wichtiger ist es, die Evolution und Motivation zu erkennen:

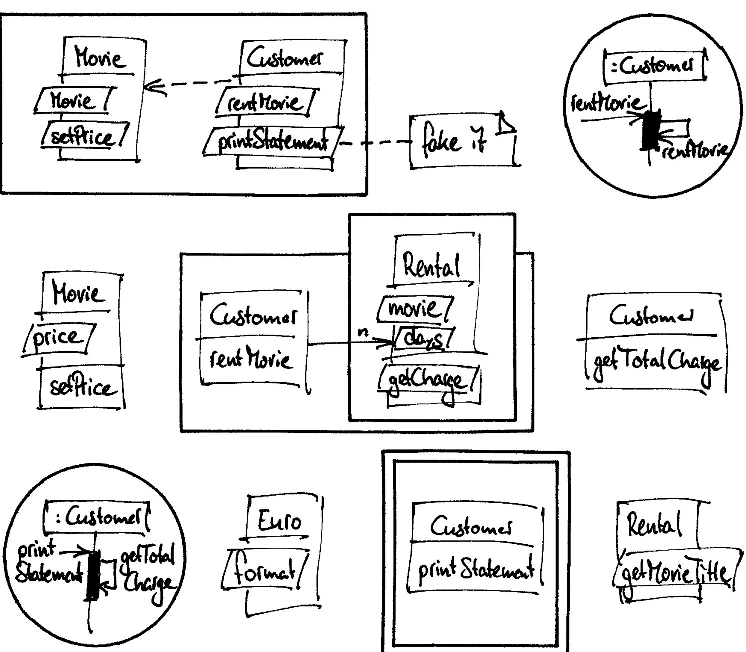


Abb. 5–2
Unsere Entwicklungsroute als Bildergeschichte

Am Anfang stand ein größerer Test. Er gab die etwaige Richtung vor und so kamen der `Movie`-Konstruktor und die insgesamt drei neuen `Movie`- und `Customer`-Methoden hinzu. Die Schlüsseltechnik war hier, die `printStatement`-Methode zunächst nur als »Fake« zur Verfügung zu stellen. Zur bequemen Umstellung wurde dann ein *Bypass* von der alten `rentMovie`-Methode zur neuen gelegt, eine wichtige Hilfstechnik.

Im nächsten Schritt erhielt unsere `Movie`-Klasse eine `Price`-Instanz. Anschließend wurde die `Rental`-Klasse zur Assoziation von `Customer`- mit `Movie`-Objekten eingeführt, um ihre `getCharge`-Anfragen lediglich an das assoziierte `Movie`-Objekt zu delegieren. Die `getTotalCharge`-Methode konnte daraufhin bereits auf `Rental` umgeschwenkt werden.

Jetzt konnte `getTotalCharge` auch in die `printStatement`-Methode eingespannt werden, wobei der `Euro`-Klasse noch die `format`-Methode hinzugefügt werden sollte. Nachdem für die *Bequemlichkeitsmethode* `getMovieTitle` gesorgt war, war die `printStatement`-Methode zu guter Letzt auch in der Lage, korrekte Resultate zu produzieren.

5.20 Durchbrüche erleben

Eine besondere Form von Refactoring musste ich in diesem Kapitel aussparen: Durchbrüche, die erst nach einer gewissen Zeit gelingen und dann zu einer dramatischen Verbesserung des Designs führen.

Zum Großteil kann sich ein gutes Design allein aus einer Reihe lokaler Entwurfsentscheidungen entwickeln. Das Refactoring schlägt schon tagein, tagaus seine Kreise: von einzelnen Klassen über deren Grenzen hinweg zu Nachbarklassen. Diese Strategie birgt allerdings ein gewisses Risiko in sich, mit dem evolutionären Design in einem lokalen Maximum stecken zu bleiben. Um von dort zum globalen Maximum vorzudringen, sind stets erhebliche Umbauten notwendig. Der Weg von einer Bergspitze zur anderen führt nun mal durchs Tal.

Durch evolutionäres Design vom lokalen zum globalen Maximum

Es sind jedoch die kleinen Refactorings, die uns überhaupt den Weg ebnen zu viel tiefer gehenden Erkenntnissen: Ich spreche hier von Einsichten in ein insgesamt einfacheres Design, die so profund sind, dass sie uns erst einmal das Herz in die Hose rutschen lassen und dann eine Schockwelle durch das Projekt jagen. Das Angsteinflößende an diesen Geistesblitzen ist, dass wir uns in dem Moment meist so fühlen, als hätten wir etwas vom grundlegend falschen Ende her aufgezogen. Das Verlockende ist, dass Durchbrüche eine echte Chance bieten, Designs in einer zuvor nicht vorstellbaren Dimension zu vereinfachen. Eine gute Diskussion dazu finden Sie im Buch [ev03] von Eric Evans.

Metaphern

System im System

Manchmal ist es eine *Metapher*, der wir uns bewusst werden und von der wir uns im Design leiten lassen. Manchmal ist es das *System im System*, das uns ein Licht aufgehen lässt. Doch nie kommt dieser Geistesblitz, wenn wir uns den Hintern vor der Tastatur platt sitzen. *Design Satori* kann sich erst einstellen, wenn sich unser Unterbewusstsein einschaltet und der Sache annimmt: zum Beispiel morgens unter der Dusche oder am Freitagabend auf dem Weg ins Wochenende ...

Manchmal kann die transformierende Idee monatelang auf sich warten lassen. Und so mancher zwingende Einfall wird nie gehört. Erleuchtung lässt sich nicht erzwingen, sie lässt sich jedoch einladen: Machen Sie ausreichend Arbeitspausen, treten Sie öfter einmal einen Schritt zurück und tauschen Sie sich untereinander in der Büroküche über Ihre Programmierepisoden und Herausforderungen dabei aus.

Große Refactorings

Viele Durchbrüche führen zu größeren Refactorings und bringen deshalb ein größeres Risiko mit sich. *Große Refactorings* lassen sich stets in kleinere aufbrechen, häufig allerdings nicht mehr in schönen kleinen Schritten erreichen. Die Durchführung großer Refactorings ist jedoch ein eigenes Buch wert. Empfohlen sei hiermit das gleichnamige Buch [ro04] von Stefan Roock und Martin Lippert.

6 Häufige Integration

Anhaltender Entwicklungsfortschritt ist für unsere Arbeit essenziell, besonders bei Teamarbeit. Im Team müssen wir das Vertrauen haben, unsere Codeänderungen in das Gesamtsystem integrieren zu können, ohne die Arbeit unserer Kollegen zu behindern oder zurückzusetzen. Ein Team muss die Arbeit auf der gemeinsamen Codebasis einheitlich koordinieren und die entstehenden Entwicklungszweige regelmäßig synchronisieren.

Als Integration bezeichnen wir das erfolgreiche Zusammenführen der individuellen Entwicklungsergebnisse zu einer lauffähigen und getesteten neuen Systemversion. Mit der Integration geben wir unsere Änderungen und Erweiterungen am Code für alle Teammitglieder zur Verwendung frei. Dieses Kapitel soll zeigen, wie das Projekt als Ganzes in kleinen Schritten vorwärts schreitet: mehrmals täglich ausliefern.

6.1 Die dritte Direktive

Probleme entstehen während der Integration, wenn wir uns gegenseitig in die Quere kommen und zum Beispiel dieselben Codeteile ändern. Da solche Fehler aus den sich überschneidenden Änderungen mehrerer Entwickler entstehen, wird die Behebung von Integrationsproblemen aufwändiger, je länger wir die Integration hinauszögern. Die Häufige Integration lässt uns Probleme erkennen, wenn sie noch klein sind:

3. Direktive: *Integrieren Sie Ihren Code so häufig wie nötig.*

»So häufig wie nötig« bedeutet dabei sehr viel häufiger als gewöhnlich. So hinterlässt die Häufige Integration eine Fußspur des anhaltenden Projektfortschritts:

- Mindestens einmal am Tag integrieren wir unseren Code.
- Der Abschluss einer Aufgabe ist der ideale Integrationszeitpunkt.

6.2 Die Integrationszüge

Ein JUnit-Balken in leuchtendem Grün gibt uns wichtiges Feedback: während der Programmierung, beim Refactoring und ebenso bei der dritten Technik im Bunde, der Häufigen Integration unseres Codes. Erwartungsgemäß wollen wir den grünen Balken bei der Teamarbeit unter allen Umständen aufrechterhalten. Nichts ist wichtiger, als jederzeit ein lauffähiges System demonstrieren zu können, das allen Tests standhält. Ihr Kunde wird es Ihnen danken, wenn Sie die schon fertig gestellte Funktionalität zu jedem Zeitpunkt ausliefern können.

Die erfolgreiche Integration startet und endet beim grünen Balken. Das bedeutet, dass wir Änderungen nur freigeben können, wenn sich der Code danach einwandfrei kompilieren lässt und alle Tests laufen. Durch diese strikte Regelung können wir den Integrationsaufwand gering halten und Fehler schnell einkreisen. Treten dann Probleme auf, können wir sicher sein, dass sie vom gerade eben eingestellten Code verursacht werden. Vor unserem Integrationsversuch lief ja noch alles.

Die Integrationszüge in Kürze:

1. Synchronisieren Sie Ihren lokalen Stand mit dem Repository.
2. Erzeugen Sie einen neuen getesteten Build.
3. Versionieren Sie den aktuellen Stand.

Durch die Häufige Integration können wir im Team mehrmals täglich einen neuen stabilen Stand des Gesamtsystems erstellen. Alle für den Build benötigten Quelldateien erhält jeder Entwickler in der zuletzt freigegebenen Version aus dem gemeinsamen Repository vom Server.

Zur Entwicklung wird lokal ein Abzug des Repositories erstellt, der dann regelmäßig mit den eingestellten Änderungen auf dem Server abgeglichen wird. Bei Häufiger Integration hinken die lokalen Stände einzelner Entwickler selten mehr als eine Version hinterher. Das heißt, wir arbeiten nahezu immer mit der allerneuesten Version von allen Arbeitsdateien und umgehen so die gefürchtete *Bigbang-Integration*.

Bigbang-Integration? Nein danke!

Damit im Team wirklich häufig genug integriert wird, muss der Aufwand dazu minimal sein. Gerade indem wir häufig integrieren, bleibt der Aufwand aber gering. Ein sich selbst verstärkender Effekt. Zusätzlich lässt sich der Aufwand drosseln, da die Integrationszüge nahezu vollständig automatisierbar sind. Die Umsetzung variiert von Projekt zu Projekt, doch als Mindestvoraussetzung für einen effektiven Integrationsprozess zählen:

Mindestvoraussetzungen

- **Versionsverwaltung** zur Historisierung aller Arbeitsdateien
- **Build-Skript** zur vollautomatisierten Erstellung des Gesamtsystems
- **Integrationsserver** zur Referenz mit dem letzten sauberen Stand

6.3 Änderungen mehrmals täglich zusammenführen ...

Schritt 1: Synchronisieren Sie Ihren lokalen Stand mit dem Repository

Der Integrationsprozess findet vorzugsweise in zwei Stufen statt:

1. Zunächst arbeiten wir nur lokal auf unserem Entwicklungsrechner und bringen alle unsere Änderungen mit allen anderen Neuerungen aus dem zentralen Repository zusammen.
2. Ist der lokale Build erfolgreich, wiederholen wir die Prozedur auf dem Integrationsserver.

Der zweistufige Ansatz stellt dabei sicher, dass Integrationsprobleme zunächst lokal behoben werden und dass der integrierte Code nicht nur auf der Plattform eines einzelnen Entwicklers funktioniert.

Code mergen, Konflikte lösen

Das Zusammenführen (engl. *Merging*) der parallel durchgeführten Änderungen kann meist problemlos durch die Versionsverwaltung erfolgen. Wenn zwei Änderungen die gleichen Codeteile berühren, kommt es dabei jedoch zu einem Konflikt. Viele davon können direkt vom Werkzeug aufgelöst werden. Nur wenn die Änderungen nicht miteinander vereinbar sind, müssen wir Konflikte von Hand beheben.

Wie häufig ist häufig?

Das bringt uns noch einmal zur Integrationsschrittweite: Wenn die Integration zu viel Zeit beansprucht, weil Konflikte auftreten, dann integrieren wir zu selten. Die Integration findet häufig genug statt, wenn sie praktisch problemlos läuft. Wer also tagelang nicht integriert, darf sich nicht über Arbeit beschweren. Kleine Schritte auch hier.

Konflikte minimieren

Als weitere Maßnahmen, um Konflikte zu minimieren, existieren:

- **Viele kleine Teile:** Die vier Kriterien für die Einfache Form führen zu vielen kleinen Klassen und Methoden. Durch die Faktorisierung in mehrere und kleinere Teile wiederum reduzieren sich Konflikte.
- **Gemeinsamer Code:** Das natürliche Arbeiten mit Objekten führt uns durch viele Klassen. Schon ein einfaches Refactoring wie die Umbenennung einer Methode macht deutlich, wie sinnvoll es ist, jedem Entwickler die Möglichkeit zu öffnen, jede Klasse im System zu ändern. Das ist nicht nur ein Recht, sondern auch eine Pflicht.
- **Programmierstandards:** Alle Teammitglieder folgen gemeinsamen Konventionen zur Formatierung des Codes und zur Benennung von Bezeichnern. Ohne Richtlinien hagelts reihenweise Konflikte, nur weil die Klammern vielleicht an der falschen Stelle stehen.
- **Teamkoordination:** Eine tägliche Besprechung im Team erlaubt es, die Arbeit im Team zu planen, mögliche Kollisionen zu vermeiden und wichtige Statusinformationen sternförmig zu kommunizieren. Wird dieses Meeting im Stehen gehalten, bleibt es dazu noch kurz.

Locken von Dateien führt u.U. zum Deadlock

Eine weitere Möglichkeit, um Integrationskonflikte zu vermeiden, wäre das Sperren (engl. *Locking*) zur exklusiven Änderung von Dateien. Dieser Mechanismus ist für die Testgetriebene Entwicklung jedoch ungeeignet, da es explizit gewünscht ist, dass sich Refactorings über weite Teile des Codes ausbreiten. Die Wartezeit bis zur Freigabe reservierter Arbeitsdateien würde unsere Entwicklung ausbremsen, weshalb sich ein *optimistisches Locking* bewährt hat.

Konflikte im Paar beheben

Häufig ist zu beobachten, wie sich Entwickler, die sich bei ihren Änderungen gegenseitig über die Füße gelaufen sind, auch gegenseitig bei Integrationsproblemen helfen. Diese Teamdynamik ist so viel mehr wert als jede Maßnahme, um Kollisionen vollständig auszuschalten.

Eine bewährte Idee ist es, nur eine Integration auf einmal zuzulassen. Integrieren nämlich mehrere Entwickler gleichzeitig ihren Code, fliegen ihnen unter Umständen gegenseitig ihre Tests um die Ohren und im Falle von Integrationsproblemen ist es schwerer auszumachen, von woher sie verursacht wurden. In Teams ab einer gewissen Größe bietet es sich darum an, die Integration über ein Token zu serialisieren:

Integration über ein Token koordinieren

Das Token erfüllt die Aufgabe des Staffelstabs: Wer das Token hat, darf integrieren. Stellen Sie sich dazu einfach einen Gegenstand vor, den man sich gegenseitig zuwerfen kann: Das ist ein *Integrationstoken*. In vielen Projekten ist es irgendein Spielzeug.

Versetzen Sie sich in folgendes Szenario: Sie haben soeben mit dem Repository synchronisiert, allen Code von Grund auf neu kompiliert und getestet. Gerade als Sie Ihren aktuellen Stand versionieren wollen, kommen noch Änderungen von Ihrem Teamkollegen herein. Hmmm... Sie synchronisieren also erneut, kompilieren, testen und ... just zu dem Moment, als Sie einchecken wollen, kommt Ihnen wieder jemand mit seinen Änderungen zuvor. Diese Spielchen sind nicht nur ernüchternd, sie lassen sich über ein Integrationstoken einfach vermeiden.

Taxi implements Throwable

von Olaf Kock, abstrakt gmbh

Mein Code ist fertig und muss weg – ins Repository. Also: Arm hoch, »Taxiii« – es kommt sofort. Und zwar geflogen. Gemeint ist unser Integrationstoken aus dem Überraschungsei. Die Uhr läuft, die Integration beginnt: Alle Tests laufen, der Code wird eingecheckt, CruiseControl springt an, übersetzt und testet alles auf einer anderen Maschine. Wir sind am Ziel. Ohne Umweg :-)

Nur: Überraschungsei-Taxis gehen leicht verloren. Wo gibt es Plüschtaxis?

6.4 ... das System von Grund auf neu bauen

Schritt 2: Erzeugen Sie einen neuen getesteten Build

Ein erfolgreicher Build umfasst mindestens die folgenden Schritte:

1. **Build-Erstellung:** Alle Quelldateien werden neu kompiliert.
2. **Build-Verifikation:** Alle Unit Tests laufen fehlerfrei.

Zusätzlich sind projektspezifische Schritte denkbar, wie zum Beispiel:

- Dateien aus Repositories auschecken
- Code und Dokumentation generieren
- Datenbankschema erzeugen
- Konfigurationen herstellen
- Java-Archive (JARs und Konsorten) packen
- Deployment der Archive auf Applikations- und Webserver
- Staging, d.h. Einspielen in Test- und Produktionsumgebung
- E-Mail versenden und Webseite mit Build-Resultaten aktualisieren
- und vieles mehr ...

Allen diesen Schritten ist gemeinsam, dass sie vollautomatisierbar sind. Und da jeder im Team mehrmals täglich durch diesen Build-Prozess stiefelt, schafft ein gemeinsames Build-Skript Abhilfe.

Automatisieren Sie Ihre Arbeit nach der dritten Wiederholung. Besonders die langweiligen, aufwändigen oder fehleranfälligen Routineaufgaben schreien nach Automation.

Die Automation unserer täglichen Arbeit kennt zwar ihre Grenzen, doch oft lässt sie sich noch ein ganzes Stück weiter tragen. Achten Sie einfach mal darauf, welche Arbeiten Sie tagein, tagaus wiederholen.

Build-Skript

Auch das Build-Skript muss einige Anforderungen erfüllen:

- Jeder Entwickler kann plattformunabhängig durch ein einfaches Kommando einen reproduzierbaren Build-Prozess starten.
- Der Build-Prozess läuft ohne manuellen Eingriff und gibt Meldung, ob er erfolgreich war oder welcher der Schritte fehlschlug.

Der Build-Verifikationstestsuite gehören mindestens alle Unit Tests an. Zusätzlich werden häufig noch einige Systemtests gefahren.

Schnelle Build-Läufe

Es sollte klar geworden sein, dass der gesamte Build nur kurze Zeit benötigen darf. Dauert es zu lang, wird nicht häufig genug integriert. Wenn der Build-Prozess mehr Zeit verschlingt als eine Kaffeepause, müssen Sie handeln: Kaufen Sie einen schnelleren Rechner, messen und optimieren Sie die Performanzfresser oder teilen Sie den Build auf.

6.5 ... und ausliefern

Schritt 3: Versionieren Sie den aktuellen Stand

Nach einem erfolgreichen Build darf der Code eingecheckt werden. Die wichtigste Direktive bei der Versionierung ist, dass der aktuelle Stand im Repository zu jeder Zeit sauber sein muss: Stände mit nicht kompilierbarem Code oder mit fehlschlagenden Tests sind deshalb sofort zurückzurollen.

Behüten Sie im Team den grünen Balken. Nichts ist wichtiger, als Build- und Integrationsprobleme sofort zu lösen.

Immergrüner Code

Wir checken nur sauberen Code ein, weil wir ansonsten nie wissen, wie weit wir vom grünen Balken womöglich entfernt sind. Andernfalls drosseln wir den Teamfortschritt oder machen unnötige Rückschritte.

Wenn wir häufig genug integrieren, lassen sich die meisten Fehler auch dadurch eingrenzen, dass wir wissen, welche Codeteile wir seit der letzten Integration angefasst haben: Von dort rührt unser Problem. In manchen Fällen hilft auch ein Vergleich zwischen der Integrations- und Entwicklerplattform. Finden wir die Fehlerursache trotz größter Bemühungen nicht, können wir überlegen, ob wir den missratenen Code nicht besser wegwerfen und in kleinen Schritten neu entwickeln. Fehler werden also entweder sofort behoben oder die Integration wird abgebrochen. Codeteile zum Einchecken auskommentieren zu müssen, kann keine Lösung sein, sondern ein Symptom zu großer Schritte.

Eingecheckt werden alle Arbeitsdateien, die dazu benötigt werden, das gesamte System von Grund auf neu zu bauen und die Systemtests durchzuführen. Dazu gehören neben den Quelldateien für Programm- und Testcode sowie dem Build-Skript zumindest auch alle sonstigen Skripte, erforderlichen Bibliotheken und Konfigurationsdateien. Nur generierte Sachen haben in aller Regel nichts im Repository verloren.

Kein unintegrierter Code

Eine Tugend ist es, wie ein guter Handwerker vor Feierabend das Werkzeug zu putzen und alles an seinem rechten Platz zu verstauen, um den nächsten Arbeitstag auf der sauberen Werkbank zu beginnen: also keinen Code unintegriert auf unseren Rechnern zurückzulassen. Wir können am Abend alle unsere Änderungen einchecken und am Morgen synchronisieren oder frisch aus dem Repository auschecken. Auf diese Weise stellen wir außerdem noch sicher, dass alle unsere Änderungen stets auf der aktuellen Version basierend beginnen.

Halten Sie Ihre Änderungen nie länger als einen Tag zurück. Versuchen Sie, zum Tagesabschluss auch einen Arbeitsabschluss zu erreichen.

Wenigstens einmal täglich stellen wir unsere eigenen Änderungen ins Versionskontrollsystem zurück. Mehrmals täglich jedoch holen wir uns die Änderungen unserer Kollegen zur Synchronisation unseres lokalen Entwicklungsstandes herein. Wann wir mit der Hauptversion aufholen, hängt ganz stark davon ab, wann wir für Codeänderungen anderer empfänglich sind. Denn jeder von uns benötigt seine lokale Umgebung, in der er abgeschottet von Änderungen aus dem Team und ohne Auswirkung auf das Team experimentieren kann.

Auch zwischendurch häufiger aufholen

Einen großen Unterschied macht dabei unsere Arbeitsumgebung. In einem eng zusammensitzenden Team können wir auch viel enger zusammenarbeiten und uns mit jeder Integration regelrecht neuen Code zuspielen: Wir können direkt mitzählen, wie viele Versionen wir lokal hinterherhinken, und synchronisieren und integrieren schon allein aufgrund dieser Information viel häufiger.

Die Häufige Integration wirkt sich positiv auf die Teamarbeit aus, weil wir regelmäßig ins Team kommunizieren, welche Klassen von allen verwendet werden können, und sich das Team gemeinsam dafür verantwortlich fühlt, den Fortschritt am Laufen zu halten. Den Build zu brechen ist dann eine ziemlich unangenehme Angelegenheit, die im Extremfall auch mit einem T-Shirt »I broke the Build!« belohnt wird. Und dieses T-Shirt wäscht niemand.

Ständige Lieferfähigkeit schafft Vertrauen

Im Vordergrund steht, unsere Lieferfähigkeit ab Tag eins unter Beweis zu stellen und unseren Fortschritt sichtbar und damit ebenso messbar zu machen. Denn das Vertrauen der Kunden wächst mit der Zuverlässigkeit, mit der wir funktionierende Software liefern können. Haben wir im Team ebenso viel Vertrauen in die produzierte Qualität, ist jede Integration ein Release-Kandidat.

Häufiges Ausliefern übt auch. Bis zu dem Tag, an dem das System in Produktion gehen soll, haben wir schon so viele Male ausgeliefert, dass die Arbeit zur täglichen Routine geworden ist. Nicht jedes interne Release ist dabei gleichzeitig auch ein externes Release: Je nach Projekt wird häufiger oder seltener zum Kunden ausgeliefert. Auf dem Web können wir zum Beispiel stündlich neue Funktionalität installieren, während wir im Produktsektor vielleicht eher quartalsweise ausliefern. Liefern wir die automatisierten Tests mit aus, ist es auch noch möglich, die Softwareinstallation vor Ort auf Herz und Nieren zu prüfen.

Branchen

Noch ein Tipp für die parallele Arbeit an verschiedenen Versionen: Bei der Häufigen Integration laufen alle Codeänderungen auf einer Entwicklungslinie ein. Sollen mehrere Versionszweige (engl. *Branches*) unterstützt werden, reichen die hier beschriebenen Schritte nicht aus. Bewährte Strategien dafür und wie Sie das Branching unter Umständen auch ganz vermeiden, geben Steve Berczuk und Brad Appleton [ber02].

6.6 Versionsverwaltung (mit CVS oder Subversion)

Noch einige Worte zur Versionsverwaltung: State of the Art ist zurzeit Subversion [svn]. Von größter Relevanz für testgetriebene Entwickler dürfte sein, dass Subversion im Gegensatz zu CVS [cvs] in der Lage ist, die Historie über Dateiumbenennungen hinweg zu erhalten.

Wichtig ist nur, dass Ihre Versionsverwaltung Ihnen nicht Stöcke zwischen die Beine wirft. Einige Hersteller kommerzieller Werkzeuge setzen wirklich alle Schalthebel in Bewegung, um Änderungen nur ja so aufwändig wie nur irgend möglich zu machen. Die Kultur Ihrer Werkzeuge muss also zur Kultur Ihres Entwicklungsprozesses passen.

6.7 Build-Skript mit Ant

De facto ist Ant [ant] das Build-Management-Tool für Java. Ursprünglich im Apache Jakarta-Projekt entwickelt, kommt keine integrierte Entwicklungsumgebung heute mehr ohne die fleißige Ameise aus. Dass sich Ant größter Beliebtheit erfreut, liegt sicher daran, dass nahezu alle wiederkehrenden Aufgaben zur Herstellung eines zuverlässigen Builds automatisiert werden können und dazu noch plattformunabhängig. Alles, was wir dazu tun müssen, besteht darin, die für einen Build-Lauf notwendigen Arbeitsschritte in einer Ant-Build-Datei zu beschreiben, zusammen mit den Abhängigkeiten zwischen den einzelnen Aufgaben. Gehen wir einmal eine solche typische `build.xml`-Datei durch:

```
<?xml version="1.0"?>
```

Der Build-Prozess wird deklarativ in XML beschrieben. Jedes Build-Skript definiert dabei genau ein Projekt:

```
<project name="videostore example" basedir="." default="all">
```

Ein Projekt besitzt drei Attribute: einen Namen, das Wurzelverzeichnis dieses Projekts und ein Ziel (engl. *Target*) dafür, wenn der Build ohne Angabe eines bestimmten Ziels angestoßen wird (Default-Target `all`). Doch mehr dazu nach einem Blick auf die Konfigurationskonstanten:

```
  <property name="src.dir" location="src"/>
  <property name="build.dir" location="build"/>
  <property name="report.dir" location="junitreport"/>
```

Hier belegen wir Properties mit der Arbeitsstruktur fürs Projekt vor. Im Weiteren können wir dann auf alle Dateipfade relativ zum Wurzelverzeichnis über ihre Namen zugreifen und müssen nicht hart kodierte Pfade über das gesamte Build-Skript verstreuen.

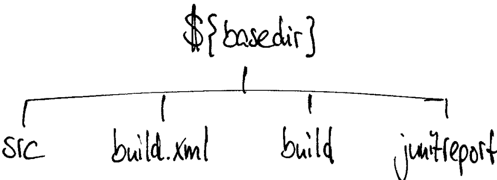


Abb. 6–1
Verzeichnisbaum unseres kleinen Ant-Projekts

Das Aufsetzen einer geeigneten Projektstruktur ist häufig schon eine interessante Herausforderung für sich. Unser kleines Projektlayout ist jedoch sehr einfach: Quelldateien werden im `src`-Verzeichnis gesucht, übersetzte Dateien im `build`-Verzeichnis abgelegt und Ergebnisse des JUnit-Testlaufs im `junitreport`-Verzeichnis.

Der Zugriff auf unsere definierten Property-Werte erfolgt über ein $-Zeichen und den Property-Namen, eingerahmt in geschwungenen Klammern:

```
<target name="prepare">
  <mkdir dir="${build.dir}"/>
  <mkdir dir="${report.dir}"/>
</target>
```

Ein Projekt enthält mehrere Targets: `prepare` ist ein *internes* Target, das sich um die nötige Verzeichnisstruktur der Ausgabedateien sorgt. Ein Target wiederum setzt sich aus Aufgaben (engl. *Tasks*) zusammen, von denen Ant mehr als 70 Standard-Tasks im Repertoire hat, gefolgt von fast ebenso vielen optionalen Tasks für weitere Spezialtätigkeiten. `mkdir` ist eine Task zum Anlegen von Verzeichnissen.

```
<target name="build" depends="prepare"
        description="compiles everything">
  <javac srcdir="${src.dir}" destdir="${build.dir}"/>
</target>
```

Ein jedes Target kann seinerseits von anderen Targets abhängig sein: Das `build`-Target hängt beispielsweise davon ab, dass der beschriebene Verzeichnisbaum vorhanden ist. Ant löst die deklarierte Abhängigkeit auf und führt hier zum Beispiel das `prepare`- vor dem `build`-Target aus. `build` ist ein *externes* Target, zu erkennen an dem Beschreibungstext. Jedem Ant-Lauf können wir gezielt mitteilen, welches dieser externen Targets angestoßen werden soll.

Die `javac`-Task wirft den Compiler an, übersetzt den Java-Code aus dem `src`-Verzeichnis und schreibt die Binärdateien wie vereinbart ins `build`-Verzeichnis.

```
<target name="test" depends="build"
        description="runs the unit tests">
  <junit errorproperty="test.failed"
         failureproperty="test.failed">
    <classpath path="${build.dir}"/>
    <formatter type="brief" usefile="no"/>
    <formatter type="xml"/>
    <batchtest todir="${report.dir}">
      <fileset dir="${build.dir}" includes="**/*Test.class"/>
    </batchtest>
  </junit>
```

Unseren Tests spendieren wir ein weiteres Target: `test`. Hier begegnen wir nun der `junit`-Task und ihren Subelementen. Das `fileset` sammelt dazu einfach alle Klassen ein, die auf `Test` enden, und `batchtest` führt sie aus, während zwei `formatter` den ganzen Testlauf protokollieren: Der erste gibt `brief`, das heißt kurz und knapp, auf der Konsole aus, der zweite erstellt für jede Testklasse eine XML-Datei mit Resultaten im `junitreport`-Verzeichnis.

```
  <junitreport todir="${report.dir}">
    <fileset dir="${report.dir}" includes="TEST-*.xml"/>
    <report todir="${report.dir}" format="noframes"/>
  </junitreport>
```

Die `junitreport`-Task generiert aus dem XML über XSLT schließlich einen hübschen Testreport in Form eines bunten HTML-Dokuments.

```
  <fail message="tests failed" if="test.failed"/>
</target>
```

Schlägt ein Test fehl, setzt die `junit`-Task das `test.failed`-Property. Bei gesetztem Property lassen wir den Build zuletzt fehlschlagen.

```
<target name="clean"
        description="deletes the output directories">
  <delete dir="${build.dir}"/>
  <delete dir="${report.dir}"/>
</target>
```

Damit die `junitreport`-Task nicht die Artefakte früherer Testläufe mit berücksichtigt, löschen wir alte Verzeichnisse im `clean`-Target.

```
  <target name="all" depends="clean, test"
   description="cleans up, compiles, and tests all classes"/>
</project>
```

Um einen Build zu erstellen, führen wir dann einfach das Kommando `ant` auf der Kommandozeile aus: Ant öffnet unser `build.xml`-Skript und stößt das Default-Target `all` an. An den Ausgaben erkennen Sie, wie die Targets in Reihenfolge der Abhängigkeiten ausgeführt werden. Natürlich muss Ant dazu auf Ihrem System installiert sein. Außerdem muss das `junit.jar` im Klassenpfad stehen.

```
$ ant
Buildfile: build.xml

clean:
   [delete] Deleting directory /Users/frank/K6/build
   [delete] Deleting directory /Users/frank/K6/junitreport

prepare:
    [mkdir] Created dir: /Users/frank/K6/build
    [mkdir] Created dir: /Users/frank/K6/junitreport

build:
    [javac] Compiling 11 source files to /Users/frank/K6/build

test:
    [junit] Testsuite: CustomerTest
    [junit] Tests run: 4, Failures: 0, Errors: 0,
            Time elapsed: 0.177 sec

    [junit] Testsuite: EuroTest
    [junit] Tests run: 7, Failures: 0, Errors: 0,
            Time elapsed: 0.015 sec

    [junit] Testsuite: MovieTest
    [junit] Tests run: 1, Failures: 0, Errors: 0,
            Time elapsed: 0.025 sec

    [junit] Testsuite: PriceTest
    [junit] Tests run: 2, Failures: 0, Errors: 0,
            Time elapsed: 0.021 sec

    [junit] Testsuite: RentalTest
    [junit] Tests run: 1, Failures: 0, Errors: 0,
            Time elapsed: 0.036 sec

[junitreport] Transform time: 1961ms

all:

BUILD SUCCESSFUL
Total time: 7 seconds
```

Abb. 6–2
Testreport

Unit Test Results

Designed for use with JUnit and Ant.

Summary

Tests	Failures	Errors	Success rate	Time
15	0	0	100.00%	0.274

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Note: package statistics are not computed recursively, they only sum up all of its testsuites numbers.

Name	Tests	Errors	Failures	Time(s)
	15	0	0	0.274

Package

Name	Tests	Errors	Failures	Time(s)
CustomerTest	4	0	0	0.177
EuroTest	7	0	0	0.015
MovieTest	1	0	0	0.025
PriceTest	2	0	0	0.021
RentalTest	1	0	0	0.036

Back to top

TestCase CustomerTest

Name	Status	Type	Time(s)
testRentingNoMovie	Success		0.013
testRentingOneMovie	Success		0.001
testRentingThreeMovies	Success		0.000
testPrintingStatement	Success		0.010

Properties »

Back to top

TestCase EuroTest

Name	Status	Type	Time(s)
testAmount	Success		0.002
testRounding	Success		0.001
testAdding	Success		0.000
testMultiplying	Success		0.000
testFormatting	Success		0.001
testEquality	Success		0.000
testNegativeAmount	Success		0.000

Properties »

Back to top

TestCase MovieTest

Name	Status	Type	Time(s)
testSettingNewPrice	Success		0.005

Properties »

Back to top

TestCase PriceTest

Name	Status	Type	Time(s)
testBasePrice	Success		0.004
testPricePerDay	Success		0.000

Properties »

Back to top

TestCase RentalTest

Name	Status	Type	Time(s)
testUsingMovie	Success		0.011

Properties »

Back to top

Den Testreport können Sie in zwei Formen haben: mit HTML-Frames oder wie hier als ein langes Dokument. Letztere Variante erleichtert es, den Report auch per E-Mail zu verschicken, ins Intranet zu stellen oder auf dem Drucker auszugeben.

6.8 Build-Prozess-Tuning

Mit zunehmender Projektkomplexität ist oft zu beobachten, wie die Build-Zeiten langsam über ein nicht mehr erträgliches Maß anwachsen und im Team aufgrund dessen weniger häufig integriert wird. Wo die Toleranzgrenze liegt, das muss jedes einzelne Team selbst entscheiden. Nehmen wir jedoch an, das Bauen und Testen allein verschlänge schon 15 Minuten. Wer wollte in dem Fall bitte noch stündlich integrieren? Es wäre zu 25 % Verschwendung!

Kurze Build-Zeiten

Wer möchte, dass in seinem Team häufig genug integriert wird, muss die Build-Zeiten drücken. Ohne regelmäßiges Geschwindigkeitstuning nehmen sie sonst immer weiter zu und fressen sich schließlich in den Stundenbereich. Einer der größten Übeltäter dafür sind Unit Tests, die diesen Namen zu Unrecht tragen, weil sie zu viel Zeit rauben. Strikte Unit Tests sind extrem schnell. Sie testen orthogonale Aspekte kleiner isolierter Einheiten. Binnen Sekunden können tausende solcher Tests laufen und somit schnelleres Feedback liefern.

Schnelle Tests

Nehmen Sie die Testlaufzeiten ernst. Sie zeigen Ihnen nicht nur auf, wo die Langläufer unter Ihren Tests sitzen. Sie können Ihnen auch als Hinweisquelle für ein Performanz-Tuning der Anwendung dienen. Gehen Sie Ihre Testreports deshalb regelmäßig durch und machen Sie die Tests mit den schlimmsten Zeitwerten in der Time-Spalte zu einem Refactoringkandidaten. Kapitel 7, *Testfälle schreiben von A bis Z*, und Kapitel 8, *Isoliertes Testen durch Stubs und Mocks*, zeigen Ihnen, wie Sie schnelle Unit Tests schreiben und erhalten.

6.9 Integrationsserver mit CruiseControl

Um automatisierte Tests im gesamten Entwicklungsteam zu etablieren, ist ein Integrationsserver überaus hilfreich: *CruiseControl* [cc], als Open Source auf SourceForge erhältlich, automatisiert die Integration, indem neu eingecheckter Code sofort einen Build- und Testzyklus des Gesamtsystems auf einer dedizierten Integrationsmaschine anstößt. Dieser *Integrations-Build* soll unabhängig vom Entwicklungsrechner sicherstellen, dass der aktuelle Repository-Stand wirklich funktioniert: Wer dann einmal vergisst, eine referenzierte Datei mit einzuchecken, kann sich nicht mehr mit »Bei mir läuft's aber!« aus der Sache reden. Darüber hinaus können wir dem Integrationsserver auch die Build- und Testschritte aufbürden, die so zeitaufwändig sind, dass wir sie beim besten Willen nicht vor jedem Einchecken ausführen könnten: beispielsweise die Durchführung umfassender Codeanalysen oder der Akzeptanztests.

Build überwachen

CruiseControl überwacht die Aktivitäten im zentralen Repository: Eine *Build-Schleife* wacht dazu in Intervallen oder zu gewissen Zeiten auf und ermittelt, ob seit dem letzten Lauf neu eingecheckt wurde. Existieren keine Änderungen, legt sich dieser Prozess wieder schlafen. Bei Änderungen wird automatisch unser Ant-Build-Skript gestartet; dieses sollte in der Regel den letzten Build löschen, das Projekt auf dem Integrationsserver neu auschecken, bauen und testen. Verläuft der Build-Versuch erfolgreich, ist es außerdem ratsam, den ausgecheckten Stand im Repository mit der aktuellen Build-Nummer zu stempeln. Danach beginnt die Build-Schleife von vorn.

Build-Status publizieren

Der Build-Status wird von CruiseControl protokolliert und kann über eine Vielzahl von Medien publiziert werden: So können etwa alle Entwickler, die Code für den Integrations-Build beigesteuert haben, per E-Mail benachrichtigt werden. In den allermeisten Teams gilt es in dem Fall als schlechtes Betragen, in den Feierabend zu verschwinden, bevor nicht diese E-Mail eingetrudelt ist. Eine weitere Möglichkeit, den *aktuellen* Build-Status zu publizieren, läuft über einen RSS-Kanal *(Really Simple Syndication)*. Als äußerst hilfreich erweist sich auch die *Build-Results-Webseite*, dargestellt auf der gegenüberliegenden Seite. Über die verschiedenen Reiter lassen sich weitere Build-Informationen aggregieren, so dass die Seite als *Projekt-Dashboard* ein umfassendes Bild über den aktuellen Gesundheitszustand des Projektes liefern kann. Die psychologische Wirkung, die von dieser einfachen Seite ausgeht, ist enorm: Oft kann man beobachten, wie sich nicht nur Entwickler, sondern auch Kunden und Manager mehrmals täglich über den Status informieren und vom Projektfortschritt überzeugen.

Build-Results-Webseite

... als Projekt-Dashboard

Die Build-Results-Seite hält das gesamte Entwicklungsteam auf Spur: In der linken Spalte sehen Sie die einzelnen Build-Läufe und -Resultate. Der letzte Versuch schlug fehl, zu erkennen an der nicht vergebenen Build-Nummer. In der rechten Spalte sehen Sie, woran es scheiterte. Klicken Sie links auf einen der früheren Builds, erhalten Sie die Details rechts ausgetauscht. Dargestellt wird: Wer was wann eingecheckt hat, ob es Compiler-Fehler gab und ob alle Tests durchliefen. Das HTML dieser Seite würde auch mit der *Build-Verifikations-E-Mail* verschickt.

Abb. 6–3
CruiseControls Build-Results-Seite mit Build-Fehler: Offenbar läuft hier auf der Integrationsmaschine ein anderes Locale als auf der Entwicklungsplattform

6.10 Aufbau einer Staging-Umgebung

Meist ist es sinnvoll, den Integrations-Build stufenweise auszuführen. Ein solches *Staging* ist sinnvoll, um den Prozess für das wertvollste Feedback zu beschleunigen. So kann eine erste Stufe zum Beispiel alle Unit Tests enthalten, eine zweite Stufe die System- und Akzeptanztests und eine dritte Stufe das Bauen von Installationspaketen.

Vorrangiges Feedback beschleunigen, nachrangiges nachliefern

Die einzelnen Stufen können von ein und derselben CruiseControl-Instanz direkt nacheinander ausgeführt werden oder auch zu unterschiedlichen Zeiten laufen: nach dem Einchecken, stündlich, täglich. Oft ist es auch hilfreich, sich über mehrere CruiseControl-Instanzen von der Entwicklungsplattform über eine *Test- und Integrationsumgebung* der letztendlichen Produktionsplattform schrittweise zu nähern. Jede dieser Stufen nimmt, soweit ihre Akzeptanzkriterien erfüllt sind, automatisch ein Deployment auf der darauf folgenden Stufe vor und schließt somit jeweils einen wichtigen Feedbackzyklus. Und wichtig: Am Ende der Kette steht ein getesteter Build zum Ausliefern bereit!

Sicherheitsnetze auf unterschiedlichen Ebenen

6.11 Teamübergreifende Integration

Je größer und verteilter ein Projekt, desto stärker wird es typischerweise in mehr oder weniger selbstständige Teilprojekte zergliedert und desto mächtiger werden damit die Risiken von Integrationsfehlern. Eine Herausforderung großer Projekte ist also, die Vorteile einer Modularisierung des Gesamtsystems zu erreichen, ohne die Nachteile einer Bigbang-Integration der Einzelteile in Kauf nehmen zu müssen. Eine Organisationsform, mit der ich gute Erfahrungen gemacht habe, sind kleine *Feature-Teams* [pa_{02}] mit jeweils vier bis acht Entwicklern, die eben entlang von Features strukturiert sind, weil die vertikalen Architekturbelange in der Regel unabhängiger voneinander sind als die horizontalen Architekturschichten. Bei Eric Evans [ev_{03}] lesen Sie, wie Sie Domänen modularisieren und die Gesamtintegrität bewahren.

Feature-Teams bilden

Empfehlenswert ist, die Abhängigkeiten unter den Teilteams auf eine kleine Menge *publizierter Schnittstellen* und *Datentransferobjekte* zu reduzieren. So kann jedes Team seine eigene Codebasis entwickeln, selbst einen Integrationsserver haben und ist dadurch so weit wie nötig von der Arbeit in parallelen Teams entkoppelt. Die Arbeitsergebnisse aller Parteien können durch einen übergeordneten Integrationsserver eine regelmäßige Systemintegration anstoßen. Der *System-Build* kann (und sollte meistens) jedoch auf Binärebene stattfinden, also in schon übersetzter Form, um hier die Build-Zeit niedrig zu halten. Ein Zugriff auf die Quellen ist dann allerdings zur Fehlerdiagnose essenziell.

Integration im Kleinen und im Großen

6.12 Gesund bleiben

In einem gesunden Projekt wird die Integration zum wahrnehmbaren Pulsschlag des Entwicklungsteams. Eine gute Idee ist es zu verfolgen, wie häufig integriert wird, wie häufig der Build gebrochen wird und was die häufigsten Gründe dafür sind, und den Integrationsprozess dementsprechend zu tunen. Sind die Schritte zu groß oder ohne Fokus? Das ist oft sehr gut an den Eincheckkommentaren auszumachen.

Es ist kein Desaster, ab und zu den Build zu brechen. Im Gegenteil, denn sonst bräuchten wir ja keinen Integrationsserver. Ab und zu den Build zu brechen, beweist uns, dass im Team nicht der totale Overkill zur Vermeidung von Build-Brüchen betrieben wird.

Der größte Vorteil der Häufigen Integration ist wohl menschlich: Die Integration gibt eine prima Gelegenheit, sich kurz vom Sitzplan zu erheben, einmal zu strecken und mit frischen Getränken zu versorgen. So schenkt uns die Integration eine willkommene und verdiente Pause, um den Kopf wieder frei zu bekommen ... und gesund zu bleiben.

Eine Geschichte über die Häufige Integration

von Lasse Koskela, Reaktor Innovations

Ein Freund von mir stieß zu einem kleinen Softwareentwicklungsteam hinzu, welches schon eine relativ lange Geschichte hinter sich hatte. Das Team produzierte halbjährlich eine neue Version einer Inhouse-Projektmanagementsoftware. Ihre Codebasis war ein großes Durcheinander, die Entwickler waren unerfahren mit einigen der zerbrechlichsten Teile, der Build-Prozess dauerte Ewigkeiten und das Testteam stand überaus häufig nur mit einem gebrochenen Build da, der über den ersten Bildschirm hinaus nicht getestet werden konnte.

Zum Glück hatte das Team auch einen Konfigurationsmanager bekommen, einen anderen Freund von mir, mit ausreichend viel Energie und Willen, die Dinge zu verbessern. Im Team entschieden sie, etwas Neues auszuprobieren. Und das war Häufige Integration.

Der Konfigurationsmanager setzte einen automatischen Build-Prozess auf, der periodisch, einmal stündlich, den letzten Stand aller Dateien aus dem Versionskontrollsystem holte und das Build-Skript startete, das effektiv versuchte, den gesamten Code zu kompilieren und darauf alle vorhandenen Tests auszuführen. Das Build-System sendete dann eine E-Mail an alle Mitglieder des Entwicklungsteams, wann immer der automatische Build fehlschlug.

Offensichtlicherweise verschickte das Build-System am laufenden Band diese »Build failed«-E-Mails an das komplette Team, da die Entwickler wiederholt vergaßen, alle ihre Änderungen einzuchecken, es unterließen, den Quellcode nach kleinen Änderungen lokal zu kompilieren, und vernachlässigten, die automatisierten Tests vor dem Einchecken auszuführen.

Was war die Reaktion des Teams auf die nun einsetzende Welle unangenehmer E-Mails, die über jeden kleinsten Fehler Beschwerde einreichten? Richtig geraten! Sie baten den Konfigurationsmanager, den automatischen Build weniger häufig laufen zu lassen, mit der Begründung, ihre Briefkästen wären so verstopft mit »Build failed«-Nachrichten, dass sie unentwegt diese Spam-Mails löschen müssten und gar nicht mehr zur wirklichen Arbeit kämen. Immerhin wüssten sie ja bereits aus der allerersten E-Mail, dass der Build zerbrochen sei. Es wäre also unnötig, daraufhin weitere Nachrichten zu verschicken.

Diese Reaktion war natürlich keine Überraschung, bedenkt man, wie lästig es ist, Stunde für Stunde die gleiche Jammer-E-Mail zu bekommen. Wie reagierte der Konfigurationsmanager also auf diesen Unzufriedenheitsausbruch? Er ließ die automatischen Builds doppelt so häufig laufen.

Mein Freund (nicht der Konfigurationsmanager, der andere) erzählte mir, dies wäre für ihn der Moment gewesen, als er begriff, was die Häufige Integration tatsächlich bewirken konnte: Den Build *grün* zu halten. Die Offenbarung war für ihn demnach, dass die Absicht eben nicht allein war, die Entwickler über einen gebrochenen Build zu informieren, sondern zu verhindern, dass der Build für längere Zeiträume gebrochen bleibt.

Schließlich, nach ein oder zwei Tagen des Nörgelns, hörte es plötzlich auf. Keine zwei »Build failed«-Nachrichten mehr pro Stunde, keine Nörgelei über die ungebetenen E-Mails, rein gar nichts. Der Build hatte begonnen, *grün* zu bleiben. Natürlich brach er hin und wieder noch mal, doch es war mehr die Ausnahme als die Regel. Die Entwickler sahen die »Build failed«-Nachrichten jetzt als wertvolles Feedback und nicht mehr als wertlose Spam-Mail. Entwickler, die zunächst Mail-Filter dafür eingerichtet hatten, um jede E-Mail von der Build-Maschine automatisch in den Mülleimer zu befördern, waren nun süchtig nach dem unbezahlbaren Feedback, das ein System für die Häufige Integration liefern kann.

Für mich klingt diese Geschichte nach einem überzeugenden Beweis, dass die Häufige Integration wirklich wert ist, ausprobiert zu werden.

7 Testfälle schreiben von A bis Z

Das Design der Testfälle verlangt ebenso viel Sorgfalt wie das Design der Anwendung. Um JUnit möglichst effektiv in Ihrer täglichen Arbeit einzusetzen, müssen Sie wissen, wie man effektive Testfälle schreibt. Sie müssen die Prinzipien verstehen, was guten Testcode ausmacht, und noch wichtiger: Sie müssen sich die Codegerüche einprägen für weniger guten Testcode. Sie müssen das alles wissen, weil Sie durch die Testautomatisierung in eine beträchtliche Menge Testcode investieren.

Ich habe in diesem Kapitel das Wissen zusammengetragen, das ich mir als Grundlage gewünscht hätte, als ich meine ersten JUnit-Tests zu schreiben begann. Die diskutierten Punkte gehen dabei von allgemein anerkannten Testmustern bis hin zu JUnit-Idiomen, die bedingt sind durch die Architektur des Test-Frameworks. Das Schreiben dieses Kapitels hat mir ganz besonderen Spaß gemacht, weil es viele wichtige Themen anspricht, die teils zusammenspielen und teils für sich stehen. Ich hoffe, Ihnen geht es ähnlich beim Lesen der Tipps und Tricks!

7.1 Aufbau von Testfällen

Beziehung von Testfallklassen zu Anwendungsklassen

Fangen wir damit an, wie JUnit-Testfälle aufgebaut werden. Was oft zur Verwirrung führt: Testfallklassen sind immer um ihre Test-Fixture orientiert. Wenn wir anfangs auch meist für jede Anwendungsklasse eine Testfallklasse aufbauen, ist ein 1:1-Verhältnis nicht zwingend. Anwendungsklassen verlangen manchmal mehrere Testfallklassen und Testfallklassen testen manchmal auch mehrere Anwendungsklassen in Interaktion. Wie viele Testfallklassen wir erhalten, ist allein eine Frage, wie wir unsere Test-Fixture aufbauen, wie viele verschiedene Fixtures wir testen und wie wir den gemeinsamen Setup-Code faktorisieren. Testfallmethoden stehen nur zusammen in einer Testfallklasse, weil sie sich gemeinsame Fixture-Objekte teilen. Die Klasse `TestCase` ist ein Mechanismus, um Testcode effektiv wiederverwenden zu können.

Wiederverwendung von Testcode

Robert Wenner zeichnete im Review zu diesem Buch folgendes Bild: Meistens ergibt sich auch ein logischer Zusammenhang. Zum Beispiel testet eine Testklasse alles zur Interaktion von Äpfeln mit Obstkörben und hat in ihrer Fixture ein Apfel- und ein Obstkorb-Objekt. Eine andere Fixture testet Äpfel und Birnen, beinhaltet also ein Apfel- und Birne-Objekt, aber kein Korb-Objekt.

Was ist die Unit?

Die Fixture ist demnach die *Unit im Unit Test*. Aus diesem Grund steht am Anfang immer die Frage: Was ist die isoliert zu testende Unit? Nur wenn diese Frage geklärt ist, können wir eine passende Fixture aufbauen.

Der strukturelle Aufbau von Testfällen ist immer gleich:

1. Fixture-Objekte erzeugen und in den Ausgangszustand bringen
2. Methoden der Objekte exerzieren
3. Erwartete und tatsächliche Resultate vergleichen

Häufig können wir für neue Testfallmethoden direkt eine existierende Fixture wiederverwenden. Passt keine der bestehenden Test-Fixtures, müssen wir zweifellos eine neue Testfallklasse aufbauen: Wenn sich die Testfallmethoden einer Testfallklasse nicht einhellig ihr Setup teilen können, zeugt dies von einem Codegeruch, der uns eine neue Fixture extrahieren lässt. Mit der Zeit kristallisieren sich auf diese Weise neue weiterverwendbare Fixtures heraus.

Oft gibt uns auch das Präfix oder Suffix der Namen im Programm einen Hinweis zum Refactoring: Würden wir in einer Testfallklasse `TopTenTest` zum Beispiel die drei Testfälle `testSortingWithNoRentals`, `testTrimmingWithNoRentals` und `testListingWithNoRentals` schreiben, stehen die Sterne eher für eine neue Testklasse `EmptyTopTenTest` und drei Testfälle mit Namen `testSorting`, `testTrimming` und `testListing`.

Extract Fixture, für mehr Übersicht

Das Herausfaktorisieren von Testfällen mit gemeinsamem Fixture-Code soll so für verbesserte Übersicht sorgen. Das erscheint zunächst nicht intuitiv: Warum nicht den Testcode, der eine Anwendungsklasse betrifft, in einer korrespondierenden Testklasse anhäufen? Testfälle sollen schließlich möglichst einfach der getesteten Klasse zuordenbar sein, oder nicht? Die Frage ist beantwortet, indem Sie sich vorstellen, `TestCase` in `TestFixture` umzubenennen, was eigentlich korrekt wäre.

Probleme entstehen bei erzwungener 1:1-Beziehung mit langem, unübersichtlichem, teils unnützem oder auch zu duplizierendem Setup-Code. Übersicht in einer 1:n-Beziehung lässt sich dagegen über Namen erreichen: Benennen Sie Ihre Testfallklassen immer nach der Klasse, die Sie vorrangig testen. Benutzen Sie dann zur Navigation im Code die Möglichkeiten moderner Klassenbrowser, mit denen man leicht die Deklarationen, Referenzen u.Ä. zu jeder Klasse aufspüren kann.

7.2 Benennung von Testfällen

Testfallmethoden benennen die getestete Funktionalität. Ist dies eine einzelne Methode, haben wir es einfach: Wir bezeichnen den Testfall entsprechend. Vielfach testen wir jedoch eher entlang von Features: *Vom Bekannten zum Unbekannten* fokussiert unsere Anstrengungen so besser auf das verlangte Programmverhalten. Welche Klassen und Methoden wir dazu heranziehen, entdecken wir oft erst auf dem Weg. Der Testfallname drückt in solchen Fällen also unsere Intention aus: Was wir entwickeln, nicht welche Methoden vielleicht getestet werden.

Manchmal schreiben wir für eine Methode oder ein Feature auch mehrere Testfälle, um jeweils eine besondere Bedingung unter die Lupe zu nehmen: Von besonderem Interesse ist das »Good Day«-Szenario, wenn alles gut geht. Aber auch »Bad Day«-Szenarien, wenn erwartete oder unvorhergesehene Fehler auftreten, verdienen ihre eigenen Tests. In jedem Fall müssen unsere Testfälle im Namen in geeigneter Weise ausdrücken, welches Szenario sie genauer betrachten: Gang und gäbe sind Namenskonventionen wie `test...With...` und `test...For...` zur Beschreibung des Testfalls.

Je genauer der Kontext für einen Namen definiert ist, desto kürzer kann der Name häufig auch ausfallen. Wenn eine Testfallklasse also schon `SortTest` heißt, können die Testfälle auch ganz knapp nur mit `testOneElement`, `testTwoInSortedOrder`, `testTwoInUnsortedOrder` usw. bezeichnet werden.

7.3 Buchführung auf dem Notizblock

Als Orientierungshilfe während der Entwicklung hilft es, alle Ideen zu sammeln, die sonst verloren gehen könnten. Dazu gehören:

- zu schreibende Testfälle
- ausstehende Refactorings
- andere offene Enden

Zentrales Instrument ist der klassische Notizblock neben der Tastatur: Mister Schreibblock passt auf, dass wir uns nicht verzetteln.

Elektronische Todo-Listen kommen der Arbeitsweise aus meiner Sicht in die Quere, aber da scheiden sich auch die Geister.

Persönlich verwende ich Karteikarten: Mir gefällt daran, dass ich stets alle Punkte vor mir habe. Zum Feierabend will ich meine Liste abgearbeitet haben. Wenn Punkte offen bleiben, übertrage ich sie mir für den nächsten Tag auf eine nigelnagelneue Karte.

7.4 Der erste Testfall

Ein Testfall ist dann sinnvoll, wenn wir durch ihn etwas dazulernen: Jeder Testfall, den wir schreiben, muss seinen Wert dadurch beweisen, dass er die Entwicklung vorantreibt.

Kapitel 5.15, *... und der wegweisende Test*, ging beispielsweise von einem funktionalen Test aus, um uns im größeren Kontext erkennen zu lassen, wie sich das neue Feature ins bestehende System einfügt:

```
public void testPrintingStatement() {
  customer.rentMovie(buffalo66, 4);
  customer.rentMovie(jungleBook, 1);
  customer.rentMovie(pulpFiction, 4);

  buffalo66.setPrice(Price.REGULAR);

  String actual = customer.printStatement();
  String expected = "\tBuffalo 66\t3,00\n"
                  + "\tDas Dschungelbuch\t1,50\n"
                  + "\tPulp Fiction\t5,50\n"
                  + "Gesamt: 10,00\n";
  assertEquals(expected, actual);
}
```

Wenn wir eine Aufgabe mit kleinen fokussierten Unit Tests beginnen, besteht eine gewisse Gefahr, das tatsächliche Ziel aus dem Blickfeld zu verlieren. Ich fange aus diesem Grund nur ungern mit Unit Tests an, bevor ich nicht einen Akzeptanztest oder einen ähnlichen funktionalen Test habe, der mir sagt, an welcher Geschichte ich eigentlich arbeite.

Der erste Test stürzt sich meist auf tief hängende Früchte. Schneller Erfolg ist wichtig! Anschließendes Refactoring presst den Saft aus dem ersten Test. Die nachfolgenden Tests sind dann mehr risikoorientiert.

7.5 Der nächste Testfall

Mit jedem Testfall wollen wir einen möglichst kleinen Schritt gehen. Die Reihenfolge, in der wir uns die Testfälle aus unserer Todo-Liste vornehmen, ist dabei entscheidend: Der natürliche Arbeitsfluss der Testgetriebenen Entwicklung entsteht, wenn ein neuer Testfall dort aufsetzt, wo der vorherige in seinen Anforderungen Halt gemacht hat. Oder wenn wir erkennen, dass wir für den Code, den wir schreiben, weitere Testfälle benötigen. Generell zu vermeiden ist die Situation, mit einem Test einen großen Satz zu machen und dadurch gleichzeitig andere Tests mitzuerledigen. Kleine Schritte, nicht Bockspringen!

7.6 Erinnerungstests

Mitunter lassen wir Code zurück, der eigentlich noch nicht fertig ist. Wir wissen um unsere Nachlässigkeit und hinterlassen aus Gewissensbissen noch einen erinnerungsträchtigen `FIXME`- oder `TODO`-Kommentar. Noch nie gemacht? Bestimmt aber schon mal gesehen, oder?

Zurückgelassen werden diese kleinen Denkmäler im Code jedoch im Sinne des Wortes, denn sind wir mal ehrlich: Wie viele von diesen Baustellen bleiben tatsächlich offen? Und wer hat schon Zeit und Lust, viel später noch einmal die Schippe in die Hand zu nehmen?

Wir können nicht verhindern, dass unfertiger Code hin und wieder mit eingecheckt wird. Wir können aber verhindern, dass wir später unverhofft in die offene Baugrube plumpsen. Erfahrung macht gute Gewohnheit: Bauen wir uns doch einfach einen Test, der fehlschlägt, wenn das Problem nicht innerhalb kürzester Zeit behoben ist:

Vorsicht, Baustelle!

```
public void test...For...() {
  Calendar today = Calendar.getInstance();
  Calendar dueDate
    = new GregorianCalendar(2006, Calendar.FEBRUARY, 28);
  assertFalse("fix the $#!% class", today.after(dueDate));
}
```

Sie erinnern sich ja: Mit rotem Balken darf nicht eingecheckt werden. Und das Auskommentieren dieses Tests gehört verboten!

Wehe dem jedoch, der sich dafür eine Abstraktion bauen muss, von wegen Codeduplikation oder so.

7.7 Ergebnisse im Test festschreiben, nicht berechnen

Erwartete Werte werden im Testfall als Literal kodiert. In Kapitel 2.6, *Natürlicher Abschluss einer Programmierepisode*, hatten wir den Fall:

```
assertEquals(13.25, customer.getTotalCharge(), 0.001);
```

Stattdessen hätten wir den Betrag `13.25` auch als Summe der einzelnen Leihgebühren schreiben können: `2.00 + 2.00 + 3.75 + 5.50`

Der Test wäre so sicher lesbarer gewesen, doch die Rundungsfehler vom `double` würden dadurch auf Test- wie Anwendungsseite zu Buche schlagen und dadurch maskieren, was eigentlich einen Fehler darstellt.

Wenn wir Anwendungslogik im Test reproduzieren, reproduzieren wir auch ihre Fehler. Berechnungen sind deshalb im Test zu vermeiden: Die Idee ist ja gerade, zwei voneinander unabhängige Quellen für die gleiche Information zu haben. Unsere Tests nehmen nur Stichproben.

Tests sind Stichproben

7.8 Erst die Zusicherung schreiben

Schreibblockade ist beim Testschreiben genauso frustrierend wie beim Bücherschreiben. Eine Technik, die ich während der *XP Immersion II* von Jim Newkirk abgeguckt habe, rollt den Test von seinem Ende auf: Wir starten mit dem Ziel im Kopf und programmieren uns rückwärts zur Startlinie vor.

Tests rückwärts entwickeln

Für die ersten Tests gegen eine neue Klasse ist es meist erheblich einfacher, unsere Absicht in ungewohnter Reihenfolge auszudrücken. Angenommen wir schreiben den Jungferntest für eine Top-10-Liste und wollen zählen, wie viele Tage unsere DVDs insgesamt ausgeliehen wurden:

1. Welches Ergebnis erwarten wir?
   ```
   assertEquals(7, ...);
   ```
2. Welches Objekt kann uns diese Antwort liefern?
   ```
   assertEquals(7, topTen ...);
   ```
3. Wie bringen wir das Objekt dazu?
   ```
   assertEquals(7, topTen.daysRented(...));
   ```
4. Welche anderen Objekte benötigen wir noch?
   ```
   Movie ilPostino = new Movie(...);
   assertEquals(7, topTen.daysRented(ilPostino));
   ```
5. Wie arbeiten unsere Objekte zusammen?
   ```
   Movie ilPostino = new Movie(...);
   topTen.addRental(new Rental(ilPostino, ...));
   assertEquals(7, topTen.daysRented(ilPostino));
   ```
6. Und wie erzeugen wir den nötigen Ausgangszustand?
   ```
   Movie easyRider = new Movie("Easy Rider", Price.REGULAR);
   Movie ilPostino = new Movie("Il Postino", Price.REGULAR);
   TopTen topTen = new TopTen();
   topTen.addRental(new Rental(ilPostino, 3));
   topTen.addRental(new Rental(ilPostino, 4));
   assertEquals(0, topTen.daysRented(easyRider));
   assertEquals(7, topTen.daysRented(ilPostino));
   ```

Nachdem das Zusammenspiel der Klassen klar geworden ist, können wir weitere Tests hinzufügen, um zum Beispiel auch einen Ladenhüter geprüft zu haben.

7.9 Features testen, nicht Methoden

In meinen ersten JUnit-Tests habe ich versucht, für jede Methode einen Testfall zu schreiben. Diese Idee erschien mir damals zutiefst intuitiv, entpuppte sich jedoch kurze Zeit später als ungeeignet.

Es kann keine 1:1-Entsprechung von Testfallmethode zu getesteter Methode geben, weil wir zu dem Zeitpunkt, in dem wir den Test schreiben, noch überhaupt nicht über nötige Methoden nachdenken. Vielmehr konzentrieren wir uns auf ein zu implementierendes Feature. Erst indem wir den Test für dieses Feature schreiben, entwerfen wir uns im Testcode eine mögliche Klassenschnittstelle.

Der Test soll eine Geschichte darüber erzählen, wie wir gedenken, den Code im Programm zu benutzen. Es kann deshalb sein, dass wir für ein neues Feature gleich mehrere neue Methoden benötigen. Genauso kann es vorkommen, dass ein zu implementierendes Feature überhaupt keiner neuen Methode bedarf.

Viele Methoden lassen sich auch gar nicht sinnvoll einzeln testen. Wie sollten wir auch `public void`-Methoden testen, wo sie doch keinen Rückgabewert liefern? Wie testen wir zum Beispiel diese Methode:

```
public class TopTen {
  public void addRental(Rental rental) {
    Movie movie = rental.getMovie();
    int total = rental.getDaysRented() + daysRented(movie);
    rentals.put(movie.getTitle(), new Integer(total));
  }

  private Map rentals = new HashMap();
}
```

Methoden ohne Rückgabewert lassen sich nicht ohne weiteres testen. Sie besitzen jedoch immer eine erwartete Auswirkung: entweder auf das Objekt selbst oder auf ein anderes Objekt in der Testumgebung. Dieser Seiteneffekt lässt sich wiederum testen. In unserem Beispiel über eine andere Methode, ohne die das Feature nicht vollständig wäre:

```
public class TopTen...
  public int daysRented(Movie movie) {
    String movieTitle = movie.getTitle();
    return rentals.containsKey(movieTitle)
      ? ((Integer) rentals.get(movieTitle)).intValue() : 0;
  }
}
```

7.10 Finden von Testfällen

Die Suche nach Testfällen erfolgt explorativ: Zunächst einmal vom Bekannten zum Unbekannten. Anschließend vom Groben zum Feinen: von der Unit des Unit Tests über die Methoden und ihre möglichen Eingabewerte zu den erwarteten Ausgabewerten.

Sobald wir meinen, verstanden zu haben, was das Problem ist, machen wir uns ans Werk: Je früher wir die Ideen im Code beweisen, desto schneller erhalten wir Gewissheit über die unbekannten Aspekte. Dass wir in unseren Versuchen manchmal auch Schiffbruch erleiden, lässt sich nicht vermeiden. Wenn wir aber schon baden gehen müssen, um zu lernen, dann bitte möglichst fix gleich zu Beginn.

Während wir unsere Testliste abarbeiten und den Code ausbrüten, fallen uns vielfach weitere Bedingungen auf, für die wir neue Testfälle benötigen. Um den Überblick zu bewahren, ist es sehr ratsam, diese Testfälle zu notieren, insbesondere wenn wir bisher nur ein »Fake it« implementiert haben. Der geschriebene Code schlägt so nach und nach neue Testfälle für die Testliste vor.

Wann immer wir einen weiteren Testfall in die Liste aufnehmen, sollten wir uns fragen: Welche anderen, vielleicht ähnlichen Testfälle könnten wir auch noch schreiben?

Oder anders herum: Können wir dem existierenden Code ansehen, welche geplanten Testfälle vielleicht überflüssig sind?

7.11 Generierung von Testdaten

Erzeugung gebrauchsfertiger Geschäftsobjekte

Tests benötigen manchmal eine umfangreiche Menge von Testdaten. Besonders der Aufbau von Netzwerken von Geschäftsobjekten führt auf diese Weise vielfach zu einem hohen Grad an Redundanz im Setup. Wenn wir für den DVD-Verleih noch ein paar weitere Testfallklassen schreiben, würde uns auffallen, dass wir stets ähnliche Objektgeflechte erzeugen: Ein `Customer` kann mehrere `Rentals` besitzen, diese erhalten jeweils ein `Movie` und diese wiederum einen `Price`.

Testhelferklassen

Um die komplette Struktur verknüpfter Objekte nur an einem Ort aufzubauen, bieten sich kleine *Testhelferklassen* an. Diesen Klassen können wir aus den Tests heraus Anweisung geben, welche Testobjekte wir mit welchen Werten benötigen. Durch aggressives Refactoring des sich wiederholenden Setup-Codes kristallisieren sich so mit der Zeit eine Reihe von *Builder*- und *Factory*-Objekten [ga95] heraus. Es folgt ein kleines Beispiel. Eine nähere Diskussion dieses Testmusters finden Sie im Papier von Peter Schuh [sc01].

Kleine Testhelferklassen zahlen sich dann aus, wenn wir mit immer weniger Code immer umfangreichere Test-Fixtures erzeugen können. Eine einfache Fabrikmethode kann zum Beispiel helfen, für den Test auf die Schnelle einen Kunden mit ein paar Ausleihen zu erzeugen:

```
public class FixtureFactory {
  public static Customer newCustomerWithThreeRentals() {
    return new FixtureBuilder()
      .newCustomer()
        .addRental("Buffalo 66", Price.NEWRELEASE, 4)
        .addRental("Das Dschungelbuch", Price.REGULAR, 1)
        .addRental("Pulp Fiction", Price.NEWRELEASE, 4)
      .getCustomer();
  }
}
```

Unsere Fabrikmethode verwendet ihrerseits einen Builder, dem wir für den Bau der benötigten Objektstruktur die Konstruktionsanweisungen mitgeben. Liefert die Instanz sich selbst als Methodenresultat zurück, lässt sich der Konstruktionscode wie oben strukturgemäß einrücken:

```
public class FixtureBuilder {
  private List customers = new ArrayList();
  private Customer customer;

  public FixtureBuilder newCustomer() {
    customer = new Customer();
    customers.add(customer);
    return this;
  }

  public FixtureBuilder addRental(String title, Price price,
                                  int daysRented) {
    customer.rentMovie(new Movie(title, price), daysRented);
    return this;
  }

  public Customer getCustomer() {
    return customer;
  }

  public Iterator getCustomers() {
    return customers.iterator();
  }
}
```

Der Vorteil dieser Testhelferklassen liegt darin, dass wir mit wenigen Methodenaufrufen ein komplexes Objektgeflecht aufbauen können, was gleichzeitig auch ihr Nachteil ist: Zwar vermeiden wir unnötige Duplikation im Setup mehrerer Testfallklassen, können einer Fixture jedoch auch nicht mehr ohne weiteres ihren Gesamtaufbau ansehen.

Übergang zum funktionalen Test

Eine Schlusswarnung: Die Erzeugung komplexer Testdaten steht an der Schwelle zum funktionalen Testen. Im strikten Unit Test sind derartig umfangreiche Fixtures ein ernst zu nehmender Codegeruch. Die Generierung von Testdaten ist deshalb wirklich nur dann sinnvoll, wenn die Objekte bereits eingehend mit allem Pipapo getestet sind. Sind sie dagegen nicht vertrauenswürdig, sind die Abhängigkeiten des zu testenden Codes zu kappen und das Geschäftsobjektnetzwerk im Test geeignet zu simulieren. Sonst gerät die Unit im Test zu groß und eventuelle Fehler lassen sich kaum noch orten.

7.12 Implementierungsunabhängige Tests

Weder Black Box noch White Box

Tests und Code haben eine interessante Relation in der Testgetriebenen Entwickung. Unsere Tests sind an sich Black-Box-Tests: Wir schreiben den Test, ohne die Implementierungsdetails des zu testenden Codes zu kennen. Schließlich wollen wir den Code ja hinterher erst entwickeln. Unsere Tests sind aber auch White-Box-Tests: Immerhin begutachten wir anschließend den existierenden Code, um zu entscheiden, welche Tests wir vielleicht noch schreiben müssen.

Tests laufen gegen das öffentliche API

Beiden Testarten ist bei Testgetriebener Entwicklung gemeinsam, dass die Tests gegen die *öffentliche Klassenschnittstelle* gerichtet sind: Wie jede Klasse darf auch ein Test einer anderen Klasse nicht in den Bauch greifen. Die Trennung von Schnittstelle und Implementierung soll uns ja gerade erlauben, sie unabhängig voneinander zu ändern. Aus diesem Grund sind Tests, die auf den *privaten Innereien* einer Klasse basieren, oft derart fragil, dass sie gleich besser von vornherein zu vermeiden sind.

Privat: Zutritt verboten!

Sich Zugriff auf Variablen oder private Methoden zu wünschen zeigt, dass dem Code eine feinere Modularisierung und damit noch eine entscheidende Designidee fehlt. Dieses Prinzip gilt nicht allein für lesende Zugriffe, sondern ebenso für schreibende Eingriffe: Manchmal befällt uns im Test zum Beispiel das Verlangen, in einem Testobjekt einen bestimmten Zustand setzen zu wollen, weil sich sonst vielleicht eine bestimmte Grenzbedingung gar nicht testen lässt. Auch wenn es uns noch so in den Fingern juckt, was uns oft wirklich fehlt, ist eine zusätzliche Indirektion: ein neuer Parameter, eine weitere Methode oder eine noch zu extrahierende Klasse.

7.13 Kostspielige Setups

Für bestimmte Testressourcen wird es zu teuer, sie für jeden Testfall neu aufzubauen und anschließend wieder freizugeben. Zu teuer heißt dabei vor allem, dass sie den Lauf unserer Testsuite verlangsamen.

Geschickte Abhilfe bietet der `TestSetup`-*Decorator* [ga95] aus dem `junit.extensions`-Package. Davon abgeleitete Klassen implementieren die bekannten Methoden `setUp` und `tearDown` mit dem Unterschied, dass sie für eine ganze Suite von Tests laufen, nicht für einzelne Tests:

```
public class CommonSetup extends junit.extensions.TestSetup {
  public CommonSetup(Test suite) {
    super(suite);
  }

  protected void setUp() { /* one-time setUp wrapper */ }
  protected void tearDown() { /* one-time tearDown wrapper */ }
}
```

Für welche Testsuite das einmalige `setUp` und `tearDown` durchgeführt werden soll, teilen wir unserem Decorator im Konstruktor mit:

```
public class AllTestsWithCommonSetup {
  public static Test suite() {
    TestSuite suite = new TestSuite("database-related tests");
    suite.addTest(database.AllTests.suite());
    suite.addTest(dbconfig.AllTests.suite());
    return new CommonSetup(suite);
  }
}
```

Typische Kandidaten wären der Aufbau von Datenbank- oder anderen Netzwerkverbindungen sowie das Einlesen von Dateien. Unsere Tests laufen von nun an jedoch nicht mehr nur auf der hermetischen Fixture, sondern können über die miteinander geteilten Testressourcen unter Umständen auch sehr subtile Seiteneffekte erzeugen.

Die Frage ist deshalb: Warum benötigen überhaupt so viele Tests Zugriff auf diese Ressource? Zeugt dies nicht eher von einem Problem im Design? Oder schreiben wir hier Integrationstests statt Unit Tests?

Teure Testressourcen vermeiden

Die meisten Tests sollten ohne teure Testressourcen laufen können: Indem wir unser Design entwirren, können wir den Objekten vielleicht nur die Daten reichen, die sie sonst aus der Datenbank geholt hätten. Ob die Datenbankanbindung funktioniert, kann unabhängig davon *ein* anderer Test beweisen. Dieser Test bleibt weiterhin kostspielig. Deshalb zwingen wir uns ja, ihn möglichst nur einmal zu machen.

7.14 Lange Assert-Ketten oder mehrere Testfälle?

Interessant ist die Frage: Wie lange können wir weitere Zusicherungen einfach in eine der schon existierenden Testfallmethoden stecken und ab wann müssen wir für sie mit einer brandneuen Methode beginnen?

Der Testlauf eines JUnit-Testfalls wird beendet, sobald eine seiner Zusicherungen fehlschlägt: Der Testfall wird aus dem einfachen Grund nicht zu Ende geführt, da jeder weitere Fehler nur eine Rückwirkung des aufgetretenen Fehlerzustandes sein könnte. Das Test-Framework erspart uns somit also ein Bombardement mit möglichen Fehlalarmen.

Zur Fehlerlokalisierung ist es trotzdem extrem hilfreich zu wissen, ob nur die eine Zusicherung gescheitert ist oder ob es mehrere sind. Wenn JUnit die Testfallausführung aber beim ersten Fehlschlag stoppt, können wir nicht erkennen, welche der anschließenden Zusicherungen vielleicht noch erfüllt gewesen wären. Damit unsere Zusicherungen in JUnit demnach wirklich unabhängig voneinander verifiziert werden, müssen wir sie auch in getrennten Testfallmethoden unterbringen.

Orthogonale Aspekte

Unabhängig voneinander sind Zusicherungen jedoch nur dann, wenn sie sich auf orthogonale Aspekte des getesteten Codes beziehen. Dieses Kriterium ist erfüllt, wenn alle Zusicherungen in einer Testfallmethode für einen gegebenen Programmierfehler sehr wahrscheinlich entweder gemeinsam erfüllt sind oder gemeinsam fehlschlagen.

Keine Seiteneffekte

Ein zweites Kriterium ist, dass Zusicherungen keine Seiteneffekte besitzen, etwa durch verändernde Methodenaufrufe. Die Reihenfolge, in der Zusicherungen ausgewertet werden, muss praktisch egal sein.

Ein Beispiel aus Kapitel 5.12, *Die letzte Durchsicht*:

```
public void testBasePrice() {
  assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(1));
  assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(2));
}

public void testPricePerDay() {
  assertEquals(new Euro(3.75), Price.NEWRELEASE.getCharge(3));
  assertEquals(new Euro(5.50), Price.NEWRELEASE.getCharge(4));
}
```

Hätten wir hier alle vier Asserts in einer längeren Testfallmethode geschrieben, könnten wir die zwei Aspekte nicht getrennt voneinander beobachten. Brechen wir die beiden Methoden dagegen weiter auf, würden ihre Teile ohnehin immer den gleichen Ausschlag liefern.

Vorsicht ist jedoch geboten bei Testfallmethoden, die ein wenig auf den Testobjekten arbeiten, dann ein wenig testen, ein wenig arbeiten, ein wenig testen ... Solche Tests sind oft zu fleißig für *einen* Testfall.

7.15 Lerntests

Hin und wieder ist uns die Verwendungsweise einer benötigten Klasse nicht geläufig. Um unser Verständnis zu verbessern, können wir

1. den Testcode der Klasse lesen,
2. ihren Quellcode lesen,
3. ein Experiment machen oder
4. die Dokumentation lesen.

Diese Liste repräsentiert meine persönliche Präferenz, geordnet nach der Zuverlässigkeit der Informationen. Geschriebene Dokumentation stimmt aus meiner Erfahrung am seltensten mit dem Code überein, deshalb schaue ich dort oft zuletzt nach. Ausnahmen gibts natürlich.

Liegt der Code aber ohne Tests oder unter Umständen auch nur in übersetzter Form vor, weil es sich um eine Bibliothek ö.Ä. handelt, kommt als erste Möglichkeit ein Programmierexperiment in Frage. Dies kann in dem Code geschehen, den wir ohnehin gerade schreiben, mithilfe des aktuell zu erfüllenden Tests. Oder als Alternative dazu: mithilfe eines Lerntests.

In Kapitel 5.17, *Vom Bekannten zum Unbekannten*, hätte ich zum Beispiel so einen Lerntest für die `NumberFormat`-Klasse schreiben sollen:

```
public void testNumberFormatting() {
  NumberFormat format = NumberFormat.getInstance();
  format.setMinimumFractionDigits(2);
  assertEquals("2,00", format.format(2));
}
```

Läuft dieser Test, können wir den Code direkt in die zu entwickelnde Klasse kopieren. Der Lerntest bleibt zur zusätzlichen Fixierung stehen.

In der Testklasse entwickeln

Eine weitere Idee ist, lose gekoppelte Methoden in der Testklasse zu entwickeln und anschließend in ihre neue Heimat zu verfrachten:

```
public class EuroTest...
  public void testNumberFormatting() {
    assertEquals("2,00", format(2));
  }

  public String format(double number) {
    NumberFormat format = NumberFormat.getInstance();
    format.setMinimumFractionDigits(2);
    return format.format(number);
  }
}
```

7.16 Minimale Fixture!

Steigen wir sofort mit einem Beispiel ein: Acht Seiten zurück in Kapitel 7.8, *Erst die Zusicherung schreiben*, haben wir den Test von seinem Ende aufgerollt. Mit folgendem Ergebnis:

```
public void testRentals() {
  Movie easyRider = new Movie("Easy Rider", Price.REGULAR);
  Movie ilPostino = new Movie("Il Postino", Price.REGULAR);
  TopTen topTen = new TopTen();
  topTen.addRental(new Rental(ilPostino, 3));
  topTen.addRental(new Rental(ilPostino, 4));
  assertEquals(0, topTen.daysRented(easyRider));
  assertEquals(7, topTen.daysRented(ilPostino));
}
```

Ich weiß nicht, wie Sie über diesen Test denken, aber mir scheinen das eine ganze Menge Objekte zu sein, nur um ein so einfaches Feature zu testen:

```
public class TopTen {
  private Map rentals = new HashMap();

  public void addRental(Rental rental) {
    Movie movie = rental.getMovie();
    int total = rental.getDaysRented() + daysRented(movie);
    rentals.put(movie.getTitle(), new Integer(total));
  }

  public int daysRented(Movie movie) {
    String movieTitle = movie.getTitle();
    return rentals.containsKey(movieTitle)
      ? ((Integer) rentals.get(movieTitle)).intValue() : 0;
  }
}
```

Der Blick auf die getestete Klasse enthüllt, dass sie sich gar nicht für *ganze* Objekte interessiert, sondern nur für *vereinzelte* Informationen. Die Abhängigkeiten auf `Rental`, `Movie` und `Price` haben völlig unnötig zu einer viel zu starken Kopplung zwischen den Klassen geführt.

Faulheit siegt!

Lassen Sie beim Testschreiben also ruhig den kleinen Faulpelz in Ihnen heraus und entkoppeln Sie dadurch gleichzeitig Ihre Klassen: Reichen Sie Konstruktoren und Methoden nur das, was minimal nötig ist! Nehmen Sie zu aufwändig erscheinende Tests als Zeichen einer Designschwäche! Finden Sie auf kreativfaule Art ein einfaches Design!

Zum Vergleich: Nach der Vereinfachung müssen wir für den Test erheblich weniger tippen:

```
public void testRentals() {
  TopTen topTen = new TopTen();
  topTen.addRental("Il Postino", 3);
  topTen.addRental("Il Postino", 4);
  assertEquals(0, topTen.daysRented("Easy Rider"));
  assertEquals(7, topTen.daysRented("Il Postino"));
}
```

Und unser Code nach der Kur:

```
public class TopTen {
  private Map rentals = new HashMap();

  public void addRental(String movieTitle, int daysRented) {
    int total = daysRented + daysRented(movieTitle);
    rentals.put(movieTitle, new Integer(total));
  }

  public int daysRented(String movieTitle) {
    return rentals.containsKey(movieTitle)
      ? ((Integer) rentals.get(movieTitle)).intValue() : 0;
  }
}
```

Einfache Tests – einfaches Design

von Dierk König, Canoo Engineering AG

Wenn ich per Test-First-Design ein Design mache, versuche ich, es danach zu beurteilen:

- wie einfach sich der Test liest
- wie einfach der Test zu schreiben ist
- wie einfach ich meine Aufgabe damit fertig stellen kann

Ein Design, das mich zum iterativen Abstieg zwingt, Stubs/Mocks o.Ä. benötigt oder mich sonst lange davon abhält, etwas fertig zu bekommen, versuche ich zu verbessern. Meist durch Vereinfachung. Mein Testcode gibt mir dieses Feedback.

Lange Zeit hat es mir gereicht, den Test zuerst zu schreiben. Ich meinte, ich sei dann fertig. Es hat mich doch einige Zeit gekostet zu erkennen, dass auch mein Test-First-Design einiger Zyklen bedarf, bis es gut genug ist.

7.17 Negativtests

Neben den gewöhnlichen Tests für das positive Programmverhalten wollen wir in einigen Fällen auch negative Konstellationen durchtesten und dadurch das möglichst robuste Verhalten im Fehlerfall absichern.

Negativtests konzentrieren sich auf zwei Fragen: Fehler erkannt? Gefahr gebannt?

- Wird der Fehler bemerkt und die erwartete Exception geworfen?
- Wird die Exception abgefangen und wie erwartet behandelt?

Häufig richten sich die Fragen an unterschiedliche Klassen, und zwar wenn eine Klasse eine Exception wirft und die entsprechende Fehlerbehandlung die delegierende Klasse trifft. Festzulegen ist dabei der Ort, wo der aufgetretene Fehler vielleicht noch behoben werden könnte.

Zu unterscheiden haben wir dann zwischen zwei *Fehlerkategorien*:

- unvorhergesehene, meist unbehebbare Fehler
- erwartete und damit zu behandelnde Fehler

Unvorhergesehene Fehler

Unvorhergesehene Fehler resultieren entweder aus Programmierfehlern oder aus einer fehlerhaften Systemumgebung. Im Betrieb müssen wir verhindern, dass solche Fehler noch größeren Schaden anrichten. In schwerwiegenden Fällen wird die weitere Programmausführung deshalb kontrolliert und möglichst graziös abgebrochen.

Erwartete Fehler

Erwartete Fehler entstehen aus Fehlerszenarien, mit denen wir im Betrieb mehr oder weniger rechnen müssen. Typische Beispiele sind temporär nicht erreichbare Server, fehlerhafte Daten und dergleichen. Unser Fokus liegt dabei nur auf den Problemfällen, die wahrscheinlich auftreten werden, nicht auf spekulativen Absurditäten. Eine kleine Analyse hilft, auf dem Teppich zu bleiben, vor allem bei Einbeziehung des Kunden: Die Fälle, die gar nicht auftreten können, müssen wir dann gar nicht weiter testen. Stattdessen können wir uns mehr um die Fälle kümmern, die unter Umständen wirklich zu Problemen führen.

In Kapitel 3.14, *Testen von Exceptions*, haben wir zum Beispiel getestet, negative Eurobeträge als Fehlerbedingung zu erkennen:

```
public void testNegativeAmount() {
  try {
    final double NEGATIVE_AMOUNT = -2.00;
    new Euro(NEGATIVE_AMOUNT);
    fail("amount must not be negative");
  } catch (IllegalArgumentException expected) {
  }
}
```

Johannes Link machte den Vorschlag, den negativen Wert durch eine Variable zu erklären. So dokumentieren wir, dass der Konstruktor nicht nur auf den gewählten Repräsentanten `-2.00` hin die erwartete Exception werfen soll, sondern für negative Werte im Allgemeinen.

Das Testen von Negativbeispielen führt nebenbei gesagt auch zum Design besserer Exception-Objekte. Neben der erwarteten Exception interessiert sich der Test vielleicht nämlich auch für Objektreferenzen, die wir zur weiteren Fehlerdiagnose ins Exception-Objekt einhängen.

Wie Sie Ausnahmesituationen im System gezielt testen können, erfahren Sie im Kapitel 8, *Isoliertes Testen durch Stubs und Mocks*.

7.18 Organisation von Testfällen

Der Aufbau unserer Testsuiten wird vorrangig geleitet durch die Frage: Wann führen wir welche Testfälle aus?

Die Drei-Sekunden-Suite

Allgemein gilt das Prinzip: Je häufiger wir eine Testsuite ausführen, desto schneller muss sie laufen. Die Suite, die uns im Testgetriebenen Entwicklungszyklus minütlich Feedback gibt, muss aus meiner Sicht in weniger als drei Sekunden laufen. Dauert es erheblich länger, fühle ich mich in meinem Arbeitsfluss beeinträchtigt und teste weniger häufig. Drei Sekunden reichen jedoch locker für ein paar hundert Unit Tests.

Langsame Tests

Die Tests, die an die Systemgrenzen stoßen und zum Beispiel die Datenbank mittesten, benötigen oft mehr als hundert Millisekunden. Diese Tests gehören nicht in unsere Unit Testsuite, sondern in eigene, langsam laufende Testsuiten ausgelagert. Fahren Sie diese Suite nur, wenn Sie wirklich gerade an der Datenbankschnittstelle zugange sind. Oder bevor Sie zur Mittagspause oder ins Meeting verschwinden. Oder lassen Sie sie einfach jede Nacht laufen, zusammen mit der langsamen Akzeptanztestsuite.

Da alle Unit Tests im System eines Tages länger benötigen werden, als wir nach jedem Kompilieren bereit sind zu warten, spendieren wir jedem Package seine eigene `AllTests`-Klasse. Technisch ist es möglich, Testfallklassen im Klassenpfad automatisch zur Testsuite zu bündeln. Was dabei verloren geht, ist die einfache Möglichkeit, durch Hin- und Herblättern zwischen verschiedenen spezifischen Testsuiten genau zu regulieren, wie viel Feedback wir wirklich gerade benötigen. Aus dem Grund bin ich selbst für die manuelle Pflege der Testsuitehierarchien.

Zu guter Letzt stellt sich die Frage: Wohin bloß mit den Testklassen? Packen wir sie ins gleiche Package wie die getestete Klasse oder in ein eigenes? Trennen wir Produktions- und Testcode in verschiedenen Verzeichnisbäumen? Ich bin da ganz pragmatisch: gleiches Package, gleiches Verzeichnis. Was spricht dagegen, die Tests mitauszuliefern?

7.19 Orthogonale Testfälle

Manchmal kommen wir in die Verlegenheit, *zu viele* Tests anpassen zu müssen, nur um *eine* Codeänderung machen zu können. Wenn das nötig wird, heißt es für uns innehalten: Mühe und Schmerzen sind eine Warnung, dass etwas nicht stimmt:

- Die betreffenden Tests decken offensichtlich zu viel auf einmal ab.
- Oder der betreffende Code wird von zu vielen Tests abgedeckt.

Da Tests und Code stark miteinander verzahnt sind, können wir aus beiden heraus die Erkenntnis in ein einfacheres Design finden:

- Wie können wir die Tests ändern, um den Code zu vereinfachen?
- Wie können wir den Code ändern, um die Tests zu vereinfachen?

Jede der beiden Seiten ist in der Lage, die Designschwächen auf der anderen Seite zu spiegeln. Sind unsere Tests zu *änderungsanfällig*, bedeutet das in der Regel, dass die Granularität der Tests zu grob ist, vermutlich weil die Modularität des Codes schon nicht fein genug ist. Es sollte also nicht überraschen, dass der Weg zu *feingranularen* Tests über *viele kleine Codeteile* führt.

Indem wir eine größere Klasse in mehrere kleinere zerteilen oder eine lange Methode in mehrere kürzere, erhalten wir die Möglichkeit, sie unabhängig voneinander zu ändern. *Orthogonalität* ist erreicht, wenn ein Teil des Codes von den Auswirkungen der Änderungen an einem anderen Teil isoliert ist. Unsere Tests sind demnach orthogonal, wenn wir die zu testenden Aspekte des Prüflings in unterschiedlichen, sich nicht überschneidenden Testfällen abdecken.

Orthogonalität

Ein Beispiel: In Kapitel 5.17, *Vom Bekannten zum Unbekannten*, haben wir diesen Testfall erfüllt:

```
public void testPrintingStatement() {
  customer.rentMovie(buffalo66, 4);
  customer.rentMovie(jungleBook, 1);
  customer.rentMovie(pulpFiction, 4);

  buffalo66.setPrice(Price.REGULAR);

  String actual = customer.printStatement();
  String expected = "\tBuffalo 66\t3,00\n"
                  + "\tDas Dschungelbuch\t1,50\n"
                  + "\tPulp Fiction\t5,50\n"
                  + "Gesamt: 10,00\n";
  assertEquals(expected, actual);
}
```

Der Test ging gezielt auf das funktionale Zusammenspiel der Klassen ein und diente uns damals vorrangig als zusätzliches Designwerkzeug. Diesen Zweck hat er auch bestens erfüllt, als Regressionstest taugt er jedoch nur bedingt, wirkt sich doch jederlei Preisänderung auf ihn aus.

Schauen wir doch einmal, was dieser Test alles gleichzeitig testet:

- Erfassung der Ausleihvorgänge eines Kunden
- Preisliche Neuklassifizierung von Filmen
- Ausgabe der Rechnungsposten in einem definierten Format
- Preisberechnung von Filmen verschiedener Preisklassen
- Summierung zum Gesamtbetrag

Die Aspekte Neuklassifizierung, Preisberechnung und Summierung (Punkt 2, 4 und 5) haben wir bereits anderenorts getestet. Ihren Test indirekt zu wiederholen ist nicht nur unnötig, sondern kostet uns bei der Instandhaltung der Tests später zusätzliche Mühe. Deshalb breche ich größere Tests, nachdem sie ihren Dienst getan haben, häufig in mehrere orthogonale Unit Tests klein. Dabei besteht jedoch die Gefahr, dass die Tests an Scharfsinn verlieren und am Ende zu wenig testen. Beim Refactoring von Testcode ist umsichtiges Vorgehen angesagt!

Interessant wird das möglichst orthogonale Testen, wenn wir mit Bestürzung feststellen müssen, dass unser Code gar nicht orthogonal testbar ist. Wie sollen wir beispielsweise die Erfassung von Ausleihvorgängen losgelöst testen? Eine Möglichkeit, diesen Vorgang zu prüfen, ist ja gerade, seine Auswirkung auf die Betragssummierung unter die Lupe zu nehmen. Diese basiert jedoch wiederum auf dem Aspekt der Einzelpreisberechnung. Um überhaupt orthogonale Tests schreiben zu können, muss unser Programm auch nach orthogonalen Aspekten strukturiert sein, was unser Code offensichtlich nicht ist:

```
public class Customer...
  public String printStatement() {
    String result = "";
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      result += "\t" + rental.getMovieTitle()
              + "\t" + rental.getCharge().format() + "\n";
    }
    result += "Gesamt: " + getTotalCharge().format() + "\n";
    return result;
  }
}
```

Wie kommen wir also dahin, die bisher nicht orthogonal testbaren Aspekte unabhängig voneinander zu testen?

- Die mit `rentMovie` erfassten Ausleihen isoliert von den restlichen Aspekten zu testen, verspricht wenig Besserung. Zwar könnten wir unserer `Customer`-Klasse eine Methode zur Abfrage der Anzahl der geliehenen Filme spendieren, doch andere Aspekte beruhen im Test ebenfalls auf der Filmausleihe, um darüber einen wohl definierten Anfangszustand im `Customer`-Objekt aufzubauen.
- Die von `printStatement` formatierte Ausgabe so zu testen, dass wir die Einhaltung des Ausgabeformats leicht von der Einbettung der variablen Ausgabewerte trennen können, ist ohne ein paar kleine Extrakniffe nicht möglich. Wie das geht, schauen wir uns jedoch im Kapitel 8, *Isoliertes Testen durch Stubs und Mocks*, an.

Das hat uns irgendwie nicht wirklich weitergebracht, aber Hopfen und Malz ist noch nicht verloren: Wenn wir die `printStatement`-Methode betrachten, können wir zumindest zwei Aspekte orthogonal anordnen. Dazu extrahieren wir die Ausgabe der Einzelposten und die Ausgabe des Gesamtbetrags jeweils in eigene Methoden:

```
public class Customer...
  public String printStatement() {
    return printStatementDetail()
         + printStatementFooter();
  }

  public String printStatementDetail() {
    String result = "";
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      result += "\t" + rental.getMovieTitle()
              + "\t" + rental.getCharge().format() + "\n";
    }
    return result;
  }

  public String printStatementFooter() {
    return "Gesamt: " + getTotalCharge().format() + "\n";
  }
}
```

Wenn wir das gemacht haben, können wir uns überlegen, ob und wie wir unsere Tests für die neuen Methoden umstrukturieren müssen.

Die printStatementFooter-Methode benötigt meiner Ansicht nach keinen zusätzlichen Testfall. Dort kann praktisch nichts schief gehen, was wir nicht ohnehin schon abgetestet hätten. Es ist gewiss richtig, dass wir keinen Unit Test für das erzeugte Ausgabeformat besitzen, aber dieser Gesichtspunkt wird hoffentlich durch einen Akzeptanztest abgedeckt. Wenn wir es uns aber genau überlegen, können wir auch einen der drei bestehenden Testfälle um die Formatprüfung erweitern. Nehmen wir doch zum Beispiel testRentingOneMovie und benennen die Testfallmethode in testPrintingStatementFooterWithOneMovie um:

```
public void testPrintingStatementFooterWithOneMovie() {
  customer.rentMovie(pulpFiction, 1);

  String actual = customer.printStatementFooter();
  String expected = "Gesamt: 2,00\n";
  assertEquals(expected, actual);
}
```

Zum Test der printStatementDetail-Methode können wir unseren vorhandenen Testfall testPrintingStatement hernehmen und daraus einfach alle nicht mehr zu testenden Aspekte streichen:

```
public void testPrintingStatementDetail() {
  customer.rentMovie(buffalo66, 4);
  customer.rentMovie(jungleBook, 1);
  customer.rentMovie(pulpFiction, 4);

  buffalo66.setPrice(Price.REGULAR);

  String actual = customer.printStatementDetail();
  String expected = "\tBuffalo 66\t5,50\n"
                  + "\tDas Dschungelbuch\t1,50\n"
                  + "\tPulp Fiction\t5,50\n";
  assertEquals(expected, actual);
}
```

Es wird nie möglich sein, ein Programm allein mit objektorientierten Sprachmitteln in vollkommen orthogonale Aspekte zu modularisieren. Dennoch sind wir unserem Ziel ein kleines Stück näher gekommen: Die Tests sind weniger fragil und der Code ist mobiler.

Orthogonale Tests werden wie in diesem Fall oft überhaupt erst durch den begrenzten Fokus kleinerer Klassen und Methoden möglich. Doch auch kleine Codeteile decken häufig unterschiedliche Aspekte ab und lassen sich deshalb auch wieder gut aus mehreren unabhängigen Richtungen testen.

Ein gutes Beispiel ist der Test von Kontrollstrukturen:

```
public class Customer...
  public String printStatementDetail() {
    String result = "";
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      result += "\t" + rental.getMovieTitle()
              + "\t" + rental.getCharge().format() + "\n";
    }
    return result;
  }
}
```

Schleifenkopf und -rumpf sind hier ganz vorzüglich getrennt testbar:

- Ist die Anzahl von Ausleihen auf dem Kundenbeleg korrekt?
- Sind die Informationen der ausgeliehenen DVDs korrekt und im erwarteten Ausgabeformat?

Im `testPrintingStatementDetail`-Test haben wir die beiden Aspekte noch leicht miteinander kombiniert testen können. Bei komplexeren Aspekten erfordert dieses Vorgehen jedoch zum einen unnötig viele Tests und hat zum anderen störende Seiteneffekte zwischen den Tests von an sich beziehungslosen Aspekten zur Folge: Wenn wir für den einen Aspekt fünf Tests schreiben müssten und für den anderen drei, dann müssten wir in den Tests insgesamt 15 Kombinationen abdecken. Außerdem würde sich dann jede Änderung im Aspekt mit den drei Varianten auf die fünf Tests auswirken, in denen die Aspektvariante mitgetestet wird. Und anders herum natürlich genauso. Schrecklich!

Kombinatorische Explosion von Testfällen

Wenn wir bemerken, dass die Menge von Testfällen durch die *kombinatorische Explosion* an Möglichkeiten hochgetrieben wird, müssen wir uns faul stellen: Wir wollen nicht $5 \times 3 = 15$ Tests schreiben. Wir wollen ein einfacheres Design finden, für das wir dann lediglich $5 + 3 = 8$ Tests schreiben müssen.

Das Gleiche gilt, wenn die Tests änderungsanfällig sind gegenüber Aspekten, bei denen sie sich eigentlich gleichgültig verhalten könnten: In dem Fall wollen wir einfachere Tests schreiben, so dass sich jeder Aspekt möglichst nur noch in einem Test niederschlägt.

Zum Beispiel können wir einen Test schreiben, der die Details der Ausgabezeile kennt, mehr aber auch nicht. Ferner können wir einen zweiten Test schreiben, der die Zeilen auf dem Kundenbeleg zählt, aber keine Details über den Zeilenaufbau hat.

Der eine Test prüft also, ob der Schleifenrumpf, *einmal ausgeführt*, die erwartete Positionszeile liefert:

```
public void testStatementDetailForRentalDetails() {
  customer.rentMovie(buffalo66, 1);

  String statement = customer.printStatementDetail();
  assertEquals("\tBuffalo 66\t2,00\n", statement);
}
```

Der andere Test prüft, ob der Schleifenkopf, *mehrmals ausgewertet*, die erwartete Anzahl von Positionszeilen liefert:

```
public void testStatementDetailForRentalLines() {
  customer.rentMovie(buffalo66, 4);
  customer.rentMovie(jungleBook, 1);
  customer.rentMovie(pulpFiction, 4);

  String statement = customer.printStatementDetail();
  String exactlyThreeLines = "(.*\n){3}";
  assertTrue(statement.matches(exactlyThreeLines));
}
```

Zur Formulierung des zweiten Tests habe ich mich dazu entschieden, mir das Zeilenzählen möglichst einfach zu machen, und deshalb zu einem *regulären Ausdruck* gegriffen. Reguläre Ausdrücke beschreiben ein Zeichenmuster, welches wir in einer Zeichenkette erkennen wollen. Im Beispiel wollen wir im Kundenbeleg drei Ausgabezeilen vorfinden. Der konkrete Zeileninhalt kann uns egal sein und deshalb im regulären Ausdruck durch Platzhalter ersetzt werden. Zur Wiederholung von Teilausdrücken besteht dann die Möglichkeit der Klammerung und Qualifizierung.

Gehen wir den regulären Ausdruck von innen nach außen durch:

- ein beliebiges Zeichen: `.`
- beliebig häufig wiederholt: `.*`
- und durch einen Zeilenvorschub abgeschlossen: `.*\n`
- und von diesem Teilausdruck genau drei Vorkommen: `(.*\n){3}`

Unsere beiden Tests sind nun weitgehend orthogonal. Eine bleibende Überschneidung ist das `\n`-Zeichen. Dieses wird von dem oberen Test sichergestellt und von dem unteren vorausgesetzt: **Häufig lässt sich ein Test extrem vereinfachen, indem er Annahmen macht, die ein anderer Test bereits verifiziert hat.** Den Programmcode so zu strukturieren, dass wir kleine fokussierte Tests für kleine unabhängige Codestücke schreiben können, das ist die kleine Kunst des orthogonalen Testens.

7.20 Parameterisierbare Testfälle

Eine Parameterisierung von Testfällen ist nützlich, wenn die gleichen Testfallmethoden immer wieder, jedoch mit wechselnden Testdaten ausgeführt werden müssen. *Datengetrieben* sind zum Beispiel Tests für Methoden, die im Wesentlichen mathematische Funktionen abbilden. Auch funktionale Tests prüfen häufig mehrfach das gleiche Szenario, jedoch mit unterschiedlichen Datentupeln.

Eine Parameterisierung ermöglicht uns, die Testdaten dynamisch aus externer Quelle zu laden. Sinnvoll ist das beispielsweise, wenn die Testdaten aus einem anderen System kommen, aus einer Simulation, dem abzulösenden Altsystem, einer Datei mit Referenzergebnissen oder in einer Tabellenkalkulation spezifiziert werden. Immer dann, wenn Testcode und Testdaten unabhängig voneinander änderbar sein müssen, ist ihre örtliche Trennung notwendig.

Testcode und Testdaten unabhängig änderbar

In reinen Unit Tests ist eine solche Trennung relativ selten nötig, weshalb JUnit von Haus aus keine parameterisierbaren Testfälle bietet. Maßgeschneidert für datengetriebene Tests ist das FIT-Framework. FIT-Tests werden tabellarisch geführt und laufen deshalb nicht unter der JUnit-Oberfläche. Wollen wir datengetriebene Tests auch mit JUnit ausführen, müssen wir uns einen parameterisierbaren Testfall bauen. Das heißt, eigentlich möchten wir bei einem gewöhnlichen JUnit-Test beginnen und ihn in einen parameterisierbaren Testfall refaktorisieren. Wenn wir diesen weiter verallgemeinern, kämen wir irgendwann beim FIT-Mechanismus heraus. Soweit werden wir jedoch nicht gehen.

Leider verschluckt selbst ein kleines Beispiel schon mehrere Seiten am Stück. Aus diesem Grund verheimliche ich Ihnen die zurückgelegte Refactoringroute und zeige hier nur meinen Start- und Endpunkt.

Als Beispiel verwende ich die `PriceTest`-Klasse. Unser Ziel ist es, die Testeingaben und erwarteten Ergebnisse aller vier Zusicherungen über eine Textdatei variieren zu können:

```
public class PriceTest extends TestCase {
  public void testBasePrice() {
    assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(1));
    assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(2));
  }

  public void testPricePerDay() {
    assertEquals(new Euro(3.75), Price.NEWRELEASE.getCharge(3));
    assertEquals(new Euro(5.50), Price.NEWRELEASE.getCharge(4));
  }
}
```

Textdatei für Testdaten

Der Aufbau der Textdatei ist einfach: Jede Zeile enthält die konkreten Testdaten für jeweils einen Testfall: den Beschreibungstext, die Anzahl Ausleihtage und die erwartete Gebühr. Die einzelnen Parameter sind durch Komma getrennt und der Beschreibungstext steht nicht in Anführungszeichen. Jede Tabellenkalkulation kann eine solche Datei im *CSV-Format (Comma-Separated Values)* schreiben und lesen:

```
1-day rental,1,2.00
2-days rental,2,2.00
3-days rental,3,3.75
4-days rental,4,5.50
```

Unsere erste Aufgabe wird sein, diese Textdatei zeilenweise einzulesen und für jede Zeile einen Testfall zu erzeugen. Schreiben wir für diesen Code auch Tests? Nur wenn wir Angst haben, dass er vielleicht nicht funktionieren könnte! Da wir den Code für den Test selbst schreiben, werden wir aber ohnehin bald sehen, ob er funktioniert.

Die Testdaten habe ich unter dem Namen `prices.csv` gespeichert. Eingelesen wird diese Datei in der `suite`-Methode unserer Testklasse. Dass Java von uns ganze acht Zeilen Code verlangt, um eine simple Textdatei zu lesen, lässt sich nicht übersehen:

```
public class PriceTest extends TestCase {
  public static Test suite() throws IOException {
    final String DATAFILE = "prices.csv";
    TestSuite suite = new TestSuite();
    BufferedReader reader = new BufferedReader(
                            new FileReader(DATAFILE));
    try {
      String line;
      while ((line = reader.readLine()) != null) {
        suite.addTest(new PriceTest("testPrice", line));
      }
    } finally {
      reader.close();
    }
    return suite;
  }
}
```

Wirklich zu verstehen ist hier nur der Code, der für jede gelesene Zeile einen neuen Testfall erzeugt und diesen einer Testsuite hinzufügt:

```
suite.addTest(new PriceTest("testPrice", line));
```

Der PriceTest-Konstruktor bekommt den Namen der auszuführenden Testfallmethode testPrice und seine Testdaten nun also als Parameter:

```
public class PriceTest...
  public PriceTest(String name, String line) {
    super(name);

    StringTokenizer tokenizer = new StringTokenizer(line, ",");
    message = tokenizer.nextToken();
    days = Integer.parseInt(tokenizer.nextToken());
    charge = Double.parseDouble(tokenizer.nextToken());
  }

  private String message;
  private int days;
  private double charge;
}
```

Der Konstruktor reicht zunächst den Namen der Testfallinstanz an die Oberklasse weiter. Hier kommt der alte TestCase-Konstruktor noch einmal zum Vorschein, der mit JUnit-Version 3.8 hinfällig wurde.

Testdaten parsen

Anschließend zerlegen wir die Testdatenzeile in ihre Bestandteile und speichern die Parameter in Instanzvariablen der Testfallklasse. Das Parsen der Testdaten ist mitnichten robust programmiert. Sind die Daten also nicht korrekt als CSV-Datei hinterlegt, kracht es kräftig.

Unsere Testfallmethode kann so auf die gesetzten Instanzvariablen zurückgreifen und die gewünschten Zusicherungen machen:

```
public class PriceTest...
  public void testPrice() {
    assertEquals(message, new Euro(charge),
                 Price.NEWRELEASE.getCharge(days));
  }

  public String getName() {
    return super.getName() + ":" + message;
  }
}
```

Zum Ende überschreiben wir noch die getName-Methode, um im Falle eines Fehlschlags neben dem Namen der ausgeführten Testfallmethode auch die Beschreibung der Testdaten zu protokollieren. Unser Testfall wird jetzt schließlich vier Mal mit unterschiedlichen Daten ausgeführt.

Zusätzliche Testfallmethoden ließen sich ausführen, indem wir sie wie auf der vorherigen Seite gezeigt manuell der Testsuite hinzufügen.

7.21 Qualität der Testsuite

Bevor wir über Qualität nachdenken, müssen wir definieren, was wir darunter verstehen wollen: Unsere Testsuite soll uns Sicherheit geben. Ein grüner Balken schenkt uns Vertrauen in die funktionale Qualität unseres Programms. Gleichzeitig gibt uns der grüne Balken den Mut, auch die strukturelle Qualität der Software hochzuhalten.

Trügerisches Grün

Ein grüner Balken kann uns jedoch in falscher Sicherheit wiegen: Dann nämlich, wenn die Testsuite gar nicht das Sicherheitsnetz bietet, von dem wir angenommen haben, dass wir es vielleicht hätten. Sich auf löchrige Testfälle zu stützen, kann so zu übermütigen Schritten führen und diese unter Umständen zum Lapsus.

Ich denke, jeden Entwickler beschleicht gelegentlich ein Gefühl, das sich am besten mit *Grünblindheit* umschreiben lässt: eine gewisse Unsicherheit, die dadurch entsteht, dass wir all unser Urteilsvermögen, ob der Code einwandfrei ist, einer stummen Zusicherung anvertrauen. Früher haben wir vielleicht Debug-Ausgaben in den Code eingestreut. Heute gibt es nur noch einen grünen JUnit-Balken. Das verunsichert natürlich hin und wieder und deshalb gibt es einfach diese Momente, in denen wir vorsätzliche Fehler in den Code einbauen, nur um zu sehen, ob unsere Testsuite sie auch wirklich zu fassen bekommt.

Das Vertrauen in unseren Code ist direkt abhängig von unserem Vertrauen in die Testsuite. Das bringt uns unwillkürlich zu der Frage: Wer testet eigentlich unsere Tests?

Wer testet unsere Tests?

Zu einem gewissen Grad testet der Code unsere Tests: Da wir zwei voneinander unabhängige Repräsentationen für die gleiche Information haben, kann ein logischer Fehler auf der einen Seite von der jeweils anderen Seite aufgedeckt werden, vorausgesetzt wir haben ihn nicht auf beiden Seiten identisch gemacht. Wir nutzen diese Chance zur Fehlerreduzierung zum Beispiel, indem wir testen, ob ein neuer Testfall auch tatsächlich fehlschlägt.

Den roten Balken zu testen, reduziert zum einen die Möglichkeit, dass unser Testfall selbst fehlerhaft ist. Zum anderen demonstriert uns der vorübergehende Fehlschlag, dass wir in unseren vorigen Schritten keinen ungetesteten Code produziert haben. Lassen wir diesen kleinen Zwischenschritt dagegen aus, arbeiten wir immer von Grün auf Grün. Dadurch ginge uns der Zusammenhang zwischen der getroffenen Entscheidung und ihrem Feedback verloren. Wer könnte dann noch sagen, ob unsere Testfälle wirklich testen, was wir denken, was sie testen, wenn wir nie einen roten Testbalken gesehen haben?

Wie finden wir also heraus, wie viel Sicherheit uns unsere Tests wirklich bieten?

Analyse der Codeabdeckung

Eine Möglichkeit, um die Güte der Tests zu bestimmen, besteht darin, ihre *Codeabdeckung* (engl. *Code Coverage*) zu ermitteln. Dieses Maß gibt Auskunft darüber, welche Programmteile durch unsere Testsuite eigentlich wirklich ausgeführt werden.

Dogmatisch betrieben gestattet die Testgetriebene Entwicklung, keinen Code zu tippen, der nicht vorher durch einen fehlschlagenden Testfall motiviert wurde. Die strikte Einhaltung der ersten Direktive würde implizieren, dass es keinen ungetesteten Code im Projekt gibt. Wenn dem so wäre, hätten wir mit dem Test-First-Vorgehen erreicht, was die Test-Last-Literatur als Idealmaß ansieht.

Da 100-prozentige Abdeckung in Test-Last-Code nur sehr schwer erreichbar ist, definiert man dort gewöhnlich ein Testendekriterium: Man versucht, eine gewisse Codeabdeckung nicht zu unterschreiten.

Für Test-First-Code dreht sich das Bild um: Wir könnten locker nur Tests schreiben, die unseren Code zu 100% abdecken. Aus wirtschaftlichen Gesichtspunkten würde dadurch aber ein Haufen Geld aus dem Fenster geworfen werden. Bei 100-prozentiger Abdeckung würden wir immer *zu viele* Tests schreiben. Wir wollen im Gegenteil herausfinden, wie wenige Tests wir schreiben müssen, um trotzdem die 100-prozentige Gewissheit zu haben, dass unser Code funktioniert.

Um diese Einstellung zu veranschaulichen, habe ich einfach einmal einen *Coverage-Analyzer* über den Code aus Kapitel 5, *Refactoring*, laufen lassen. Das Ergebnis ist wenig überraschend:

Abb. 7–1 *Codeabdeckung von Bedingungen, Anweisungen und Methoden*

Clover coverage report - clover demo
Coverage timestamp: So Mrz 23 2003 18:06:51 GMT+01:00
Overview **Package** File

package stats: **LOC:** 289 **Methods:** 40 **NCLOC:** 229 **Classes:** 11 **Files:** 11

Package	Conditionals	Statements	Methods	TOTAL
default-pkg	100%	97,1%	95%	96,8%

Classes	Conditionals	Statements	Methods	TOTAL
Euro	100%	87,5%	77,8%	86,2%
EuroTest	-	95,2%	100%	96,6%
AllTests	-	100%	100%	100%
Customer	100%	100%	100%	100%
CustomerTest	-	100%	100%	100%
Movie	-	100%	100%	100%
MovieTest	-	100%	100%	100%
Price	100%	100%	100%	100%
PriceTest	-	100%	100%	100%
Rental	-	100%	100%	100%
RentalTest	-	100%	100%	100%

Report generated by Clover v1.1 So Mrz 23 2003 18:06:52 GMT+01:00. | *30 day Evaluation License. You have 21 days before your Evaluation License expires.*

Natürlich ist dies nur ein kleines Buchbeispiel. Die in einem größeren Projekt anzutreffenden Entscheidungen sind jedoch nicht anders.

Coverage-Tool Clover

Das eingesetzte Werkzeug, *Clover* [cl], instrumentiert den Code, baut und testet das System und generiert eine Reihe von Berichten, unter anderem in HTML und PDF, und alles mittels Ant-Integration. Zu den Berichtsformen gehört eine Gesamtübersicht aller Packages, Klassen pro Package und Codeabdeckung pro Klasse. Das Tool war über längere Zeit lang frei, wird jetzt jedoch kommerziell vertrieben mit einem guten Preis-Leistungs-Verhältnis, wie ich meine.

Clover kann die Codeüberdeckung von Methoden, Anweisungen und Bedingungen messen. Unsere Gesamtabdeckung beträgt 96,8%. Die meisten Klassen erreichen sogar eine 100-prozentige Abdeckung. Nur die beiden Klassen `EuroTest` und `Euro` tanzen hier aus der Reihe. Ist das schlimm? Es kommt darauf an!

Vollständige Abdeckung gar nicht ideal

Lassen Sie sich keinesfalls einreden, eine vollständige Abdeckung wäre ideal. Sie selbst kennen Ihren Code am besten. Es gibt Klassen, die es verdienen, vollständig getestet zu werden. Und es gibt Klassen, die weniger intensiv oder gar nicht getestet werden müssen.

Der ganze Sinn, ein Coverage-Tool einzusetzen, besteht aus meiner Sicht darin, eine informierte Entscheidung darüber treffen zu können, wie gut unsere Testsuite eigentlich ist. Wenn wir wissen, welchen Code unsere Tests wirklich abdecken, können wir uns verantwortungsvoll verhalten. Wir können wie erwachsene Menschen entscheiden, wo wir noch weitere Tests hinzufügen müssen und wo wir vielleicht zu viele Tests geschrieben haben.

Sehen wir uns einmal an, was Clover für uns gefunden hat:

- Die `fail`-Anweisung im `testNegativeAmount`-Testfall,
- die `toString`-Methode der `Euro`-Klasse und
- die `hashCode`-Methode wurden kein einziges Mal ausgeführt.

Blinder Alarm also: Die `fail`-Anweisung soll nur ausgeführt werden, wenn der Testfall fehlschlägt. Ebenso die `toString`-Methode, die wir nur zu Testzwecken erstellt haben. Ferner sind `toString` und `hashCode` so einfache Methoden, dass ich mich über jede Klasse wundern sollte, in der sie getestet werden müssen.

Vollständige Codeabdeckung kann nicht unser Ziel sein. Das wäre unproduktiv und langweilig. Stellen Sie sich dazu einen Kletterer vor, der alle zwanzig Zentimeter einen Sicherheitshaken in die Felswand einschlägt. Zu viele Haken zu setzen produziert zu große Aufwände, die ab einem bestimmten Punkt durch keinen Mehrwert an Vertrauen mehr eingelöst werden können.

Für besonders wertvoll halte ich das Clover-Feature, die Codeüberdeckungsmetriken direkt neben dem betreffenden Programmcode anzubieten. Für jede Quelldatei wird dann eine HTML-Seite generiert, in der in jeder Programmzeile vermerkt ist, wie häufig eine Methode, Anweisung oder Bedingung während des Testlaufs exerziert wurde:

Abb. 7–2
Codeabdeckung direkt pro Codezeile

Clover coverage report - clover demo
Coverage timestamp: So Mrz 23 2003 18:06:51 GMT+01:00
Overview Package **File**

file stats: **LOC:** 50 **Methods:** 9
NCLOC: 37 **Classes:** 1

Source file	Conditionals	Statements	Methods	TOTAL
Euro.java	100%	87,5%	77,8%	86,2%

```
 1      import java.text.NumberFormat;
 2
 3      public class Euro {
 4          private long cents;
 5
 6  37      public Euro(double euro) {
 7   1          if (euro < 0) throw new IllegalArgumentException("negative amount");
 8
 9  36          cents = Math.round(euro * 100.0);
10          }
11
12  40      private Euro(long cents) {
13  40          this.cents = cents;
14          }
15
16   6      public double getAmount() {
17   6          return cents / 100.0;
18          }
19
20  27      public Euro plus(Euro other) {
21  27          return new Euro(this.cents + other.cents);
22          }
23
24  13      public Euro times(int factor) {
25  13          return new Euro(cents * factor);
26          }
27
28   5      public String format() {
29   5          NumberFormat format = NumberFormat.getInstance();
30   5          format.setMinimumFractionDigits(2);
31   5          return format.format(getAmount());
32          }
33
34   0      public String toString() {
35   0          return "EUR " + getAmount();
36          }
37
38  20      public boolean equals(Object o) {
39  20          if (o == null || !o.getClass().equals(this.getClass())) {
40   2              return false;
41              }
42
43  18          Euro other = (Euro) o;
44  18          return this.cents == other.cents;
45          }
46
47   0      public int hashCode() {
48   0          return (int) cents;
49          }
50      }
```

Report generated by Clover v1.1 So Mrz 23 2003 18:06:52 GMT+01:00. | *30 day Evaluation License. You have 21 days before your Evaluation License expires.*

Mutation Testing

Die Ermittlung der Codeüberdeckung ist an sich noch unzureichend, da sie lediglich wiedergeben kann, ob unsere Testsuite auch all unseren Code ausführt. Offen bleibt dagegen, ob durch die Suite auch alle erwarteten Resultate getestet werden. Der Code könnte zu 100% abgedeckt sein, ohne dass seine Arbeitsweise auch nur ein einziges Mal wirklich überprüft worden wäre: Testcode ohne ein einziges `assert`(!)

Unsere Tests müssen so wasserdicht wie nötig sein, sonst täuscht selbst der grünste JUnit-Balken über unsere löchrige Testsuite hinweg. Unser Ziel ist, den Programmcode durch orthogonale Tests zu fixieren. Idealerweise schlägt für jeden möglichen Defekt genau ein Test fehl. Wie dicht unsere Suite an dieses Ideal herankommt, können wir durch *Mutation Testing* messen. Die Idee dahinter ist, den Programmcode durch gezielte Änderungen zu sabotieren, um nachprüfen zu können, ob der eingestreute Defekt auch tatsächlich zu einem Fehlschlag führt. Mutation Testing stellt eine gute Ergänzung zur Coverage-Analyse dar, weil wir jene Codeteile ausfindig machen können, die zwar ausgeführt, aber nicht getestet werden.

Pro möglichem Defekt ein fehlschlagender Test

Das Einstreuen von Defektstellen kann sowohl manuell als auch durch ein Werkzeug automatisiert passieren. Frei erhältlich ist zum Beispiel der Testtester *Jester* [je] von Ivan Moore.

Testtester Jester

Jester sabotiert eine Klasse nach der anderen nach einstellbaren Mutationsstrategien, rekompiliert den Code und startet die Tests an. Alle Mutationen, die Jester vornehmen kann, ohne dass die Testsuite Alarm zu schlagen beginnt, werden protokolliert und mithilfe eines mitgelieferten Python-Skripts auf einer HTML-Seite pro Klasse direkt im Programmcode vermerkt.

Zu Jesters Sabotagerepertoire gehören so fiese Änderungen wie modifizierte Literale oder bedingte Ausdrücke, die entweder immer zu `true` oder immer zu `false` ausgewertet werden. Eingestreut werden die Änderungen eine nach der anderen und unabhängig voneinander nur eine Mutation auf einmal. Sabotiert wird nicht nur der Anwendungscode, sondern der Testcode wird auf die gleiche Weise behandelt.

Nachteilig am Mutation Testing ist der erforderliche zeitliche Aufwand: Um Jester auf den DVD-Verleih loszulassen, verstreichen zum Beispiel schnell drei Minuten, da Jester 83 Mal den Java-Compiler und ebenso häufig unsere Testsuite anwerfen muss. Etwa die gleiche Zeit fließt anschließend in die Durchsicht und Interpretation der Jester-Ergebnisse. In unserem Fall findet Jester sieben potenzielle Probleme:

```
7 mutations survived out of 83 changes. Score = 92
took 3 minutes
```

Grundsätzlich lassen sich Jesters Ergebnisse wie folgt interpretieren:

- Lässt sich Anwendungscode mutieren, ohne dass JUnit aufmuckt, ist der betreffende Aspekt entweder ungetestet oder redundant.
- Lässt sich Testcode selbst mutieren, ist die betreffende Zusicherung entweder zu schwach formuliert oder der mutierte Wert irrelevant.

Jester-Ergebnisse

Von den Erfolgsmeldungen sind in mühsamer Kleinarbeit all jene Fehlalarme zu subtrahieren, die nur eine *semantikerhaltende Mutation* darstellen. Alle sieben von Jester berichteten Mutationen fallen in diese Kategorie. Ernste Probleme bietet unser Code nicht, deshalb muss die Diskussion der Jester-Ergebnisse recht spröde ausfallen. Schauen wir uns aber trotzdem an, was Jester zu berichten hat:

```
public class CustomerTest...
  public void testRentingOneMovie() {
    customer.rentMovie(pulpFiction, 12);
    assertEquals(new Euro(2.00), customer.getTotalCharge());
  }
}
```

Hier hat Jester die Ausleihdauer von einem auf zwei Tage heraufgesetzt und irrtümlich erwartet, dass sich dadurch das Resultat ändert. Der Preis beträgt für zwei Tage jedoch ebenso zwei Euro. Den gleichen Fehlalarm bekommen wir für den Testfall `testPrintingStatement` gemeldet und noch einmal im `testBasePrice` der `PriceTest`-Klasse, was klar und deutlich macht, dass wir mehr als einen Testfall auf das gleiche Ziel gerichtet haben. Das indirekte Mittesten der `Price`-Klasse beginnt sich hier störend bemerkbar zu machen!

Einen anderen Fall will Jester in `testEquality` der `EuroTest`-Klasse entdeckt haben. Der Vergleich auf Ungleichheit lässt sich jedoch nicht anders ausdrücken. Die ursprüngliche Absicht blieb trotz Mutation erhalten:

```
assertFalse(two.equals(new Euro(78.00)));
assertFalse(two.equals(new Euro(7.010)));
```

Identisch die Situation im Negativtest `testNegativeAmount` der Klasse:

```
final double NEGATIVE_AMOUNT = -23.00;
final double NEGATIVE_AMOUNT = -2.010;
```

In der Praxis verschlingt das Mutation Testing für mein Empfinden zu viel Zeit in der Auswertung der Testergebnisse. Aus diesem Grund kann ich Jester nur für die kritischen Programmteile empfehlen sowie als Spiegel, um über die eigene Arbeitsweise zu reflektieren.

Metriken als Spiegel der Gewohnheit

Bedeutet es nun, dass Sie Coverage-Analyzer und Mutation Testing einsetzen müssen? Nein! Ich bin vier Jahre lang ohne ausgekommen und habe erst im vergangenen Jahr entdeckt, wie mir diese Tools bei der Reflexion über meine Testgewohnheiten unter die Arme greifen. Beide Werkzeuge visualisieren, wie ich meinen Code zu testen pflege. Herausfinden konnte ich beispielsweise, dass mich Jester eher berät, wo ich versäumt habe, Tests zu schreiben, während Clover mir verrät, wo ich eher zu tüchtig war. Wichtig finde ich vor allem, dass die Tools die Frage beantworten können, ob mir meine Refactorings auch keine Löcher ins Sicherheitsnetz der Testsuite gerissen haben.

Als natürlicher Zeitpunkt, um einen Blick auf die Testmetriken zu werfen, ist das Ende unserer Programmierepisoden wie geschaffen. Eine Integration mit dem Build-Prozess ist zumindest so lange sinnvoll, wie die Schritte Coverage-Analyse und Mutation Testing optional sind oder besser: via CruiseControl auf der Integrationsmaschine laufen. Das Gute daran ist, dass geografisch gesehen ein einziger Ort existiert, um sich ein jederzeit aktuelles Bild über den Zustand der Testsuite und den Stand des Projektes zu machen.

Metriken im Build-Prozess

Es sollte deutlich geworden sein, dass nur ein Bruchteil aller Tests, die wir schreiben könnten, wirklich wertvoll sind. Wie viele Tests wir schreiben müssen, um vollstes Vertrauen in das Programm zu haben, ist eine persönliche Entscheidung, abhängig vom Erfahrungsschatz, Programmiertalent, Selbstvertrauen, von der Tagesform und vielen anderen Einflussfaktoren. Die Regel lautet, nur den Code zu testen, wo wir Angst haben, dass er unter Umständen nicht funktionieren könnte. Anders ausgedrückt: Fügen Sie so lange Tests hinzu, bis Ihre Angst in Langeweile umschlägt.

Testtesten von Hand

Ein prima Consulting-Trick, um die Qualität von Unit Tests in einem Projekt einzuschätzen, geht wie der alte Zaubertrick, um die fehlende Karte in einem Deck zu finden: Ich lösche eine x-beliebige Zeile Code, die mir gerade interessant genug erscheint, und schiebe die Tastatur zu einem Projektmitglied rüber. Wer auch immer gerade neben mir sitzt, er oder sie möge bitte den Defekt lokalisieren. Eine gute Testsuite gibt allein über die Namen der jetzt fehlschlagenden Testfälle die ungefähre Fehlerquelle bekannt. Ein guter Test für gute Tests ist demnach, ob uns die Suite mit einem Debugging-Problem im Dunkeln stehen lässt oder ob sie, im Gegenzug, Licht auf das betreffende Stückchen Code wirft. Diesen Test können Sie selbst ganz einfach ausprobieren ...

7.22 Refactoring von Testcode

Für die Tests gilt der gleiche Qualitätsanspruch wie für den übrigen Code: selbstdokumentierend und möglichst wenig Codeduplikation. Deshalb ist es nicht überraschend, dass natürlich auch der Testcode regelmäßiges wie sorgfältiges Refactoring erlebt.

Ideal ist ein 1:1-Verhältnis von Testcode zu Anwendungscode. Lange Zeit war ich überzeugt, dass es durchaus üblich ist, in einigen Fällen mit zwei- bis viermal so viel Testcode wie Anwendungscode rauszukommen. Eines Tages habe ich gelernt, wie sich mein Testcode mit wachsender Anwendungsgröße immer stärker schrumpfen ließ. Heute bin ich der Ansicht, dass ein Ungleichgewicht von 2:1 und mehr eine Chance zum Testcode-Refactoring bietet.

Grundsätzlich refaktorisieren wir unseren Testcode nicht anders, als wir auch unseren sonstigen Code refaktorisieren. Ein Unterschied existiert jedoch: Wir haben kein direktes Sicherheitsnetz für die Arbeit am Testcode und müssen deshalb ein bisschen behutsamer vorgehen, um weder die Semantik unserer geschätzten Testfälle zu verfälschen noch ihre Codeabdeckung zu verschlechtern.

Kein Sicherheitsnetz bei Refactoring von Testcode

Da die Tests in gewisser Weise auch durch den Code selbst getestet werden, schlägt der rote Balken in der Regel sofort Alarm, sollte die Refactoringabsicht einmal in die Hose gehen. Vorausgesetzt natürlich, dass wir Test- und Anwendungscode nicht gleichzeitig refaktorisieren. Zudem ist der Testcode erheblich einfacher als der Anwendungscode. Deshalb sind die meisten Testcode-Refactorings entweder superleicht von Hand zu machen oder können gar per Knopfdruck von den heute erhältlichen Refactoringwerkzeugen erledigt werden.

Ob wir durch die Umstrukturierung der Testsuite nicht eventuell Einbußen auf Seiten der Codeabdeckung erlitten haben, lässt sich am besten mit einem Coverage-Analyzer oder Mutation-Testing-Tool in Erfahrung bringen. Entfernt besteht noch die Chance, dass ein Defekt, der uns in den Unit Tests durch die Lappen geht, von Akzeptanztests gefangen wird, so dass wir eventuelle Löcher in unserer Unit Testsuite trotzdem finden und stopfen können.

Da automatisierte Tests eine Investition in unsere Software sind, erhalten wir ihren Wert durch fürsorgliches Refactoring aufrecht. Kommt dem Testcode nicht genügend Aufmerksamkeit zu, wird seine Anpassung an das sich ständig weiterentwickelnde Programm früher oder später zur Qual. Unser Testcode ist in dieser Beziehung nicht anders als Anwendungscode: einfach zu ändern und klar verständlich, solange er in Form bleibt.

Die Codegerüche, die sich im Testcode zeigen können, besitzen nur eine kleine Schnittmenge mit ihren im Refactoringbuch [fo99] beschriebenen Artverwandten. Am häufigsten sind sicherlich duplizierter Testcode, lange Testfallmethoden und große Setups. Ansonsten findet man im Testcode einige Codegerüche, die spezifisch für die Art und Weise sind, wie Testfälle in JUnit geschrieben und ausgeführt werden.

Test Smells

Einige typische »Test Smells« sind externe Testressourcen, unnütze Testumgebungen und Testlaufinterferenzen [deu01], doch auch zu lang laufende oder leicht zerbrechliche Tests [be02a]. Wie schon erläutert, hilft die Verzahnung von Tests und Code, mit einem Smell auf der einen Seite ein potenzielles Refactoring auf der anderen Seite ausfindig zu machen. Oft riecht nicht nur der Testcode. Manchmal ist der Anwendungscode genauso problematisch und häufig ist er sogar die Ursache des Smells. Meist können wir deshalb durch ein Refactoring auf der Testseite den Code auf der Anwendungsseite mit vereinfachen.

Testcode-Refactorings

Zu den mit Abstand häufigsten Testcode-Refactorings zählen jene, die mit der Organisation des Testcodes an sich verbunden sind:

- *Extract Field*, um Testobjekte in die Fixture zu ziehen
- *Extract Method*, um duplizierten Code aufzulösen
- *Extract Class*, um eine neue Test-Fixture abzuspalten

Jedes Refactoring ist jedoch durch ein *inverses Refactoring* umkehrbar. Das heißt, ebenso häufig sind *Inline Field/Method/Class*-Refactorings für Konzepte, wo die Indirektion nicht länger ihren Zweck erfüllt.

Da sich der Großteil an Code in den Testklassen darum bemüht, die komplette Testumgebung aufzubauen und in einen bestimmten Anfangszustand zu bringen, ist dieser Setup-Code auch am anfälligsten für Wiederholungen. Um der Codeduplikation ein Ende zu bereiten, bieten sich neben den schon erwähnten Organisationsstrukturen wie Feld, Methode und Klasse noch folgende Möglichkeiten an:

- **kleine Testhelferklassen** zur Generierung von Testdaten
- **eine einfache Testsprache** für die Problemdomäne

Allein indem wir nur Ausschau nach doppelten Codestückchen halten, kristallisieren sich mit der Zeit die nötigen Testcodestrukturen heraus. Gerade die Testhelferklassen und eine domänenspezifische Testsprache können gar nicht anders entstehen als auf organischem Weg.

Manchmal müssen wir beim Refactoring erkennen, dass vielleicht ein Test fehlt oder sich mehrere Tests überdecken. Ob wir fehlende Tests noch nachholen, ist eine Frage des Gewissens. Redundante Tests können aber gewissenlos gelöscht werden: Der Test, der die größte Dokumentationswirkung besitzt, darf stehen bleiben.

7.23 Reihenfolgeunabhängigkeit der Tests

In JUnit dürfen einzelne Testfälle weder voneinander abhängig sein noch Seiteneffekte aufeinander oder gar auf spätere Testläufe ausüben, selbst wenn während der Testausführung ein Fehlschlag oder eine Exception notiert wird. Warum? Ohne diese *Testisolation* könnten wir unsere Testfälle nicht mehr unabhängig voneinander schreiben oder ändern oder ausführen.

Es existieren zwei Arten von Seiteneffekten zwischen Testfällen: erwartete und nicht erwartete. Beide sind ähnlich widerwärtig!

Erwartete Seiteneffekte

Erwartete Seiteneffekte entstehen aus eng miteinander verzahnten Testfällen, wenn ein Testfall zum Beispiel eine Datei zum Lesen öffnet und ein anderer Testfall die bereits geöffnete Datei explizit voraussetzt. Derartig stark verkoppelte Testfälle entpuppen sich als extrem instabil: Schlägt nämlich das Öffnen der Datei aus irgendeinem Grund fehl, reißt der erste Testfall natürlich den nachfolgenden Testfall mit runter. Das Testergebnis wird auf diese Weise verfälscht, denn vielleicht wäre der zweite Testfall ansonsten korrekt gelaufen: ein »False Positive«. Ebenso gut könnte der zweite Testfall durch den zurückgelassenen Fehlerzustand auch plötzlich durchlaufen, obwohl der Code eigentlich einen Defekt besitzt: ein »False Negative«.

Nicht erwartete Seiteneffekte

Nicht erwartete Seiteneffekte resultieren aus unbeabsichtigten Fernwirkungen zwischen einzelnen Testfällen, wenn mehrere Testfälle zum Beispiel globale Ressourcen wie etwa *Singleton*-Objekte [ga95] manipulieren oder sich externe Ressourcen wie eine Datenbank teilen. Für sich genommen würde vielleicht jeder Testfall ohne Makel laufen. Sobald jedoch mehrere Testfälle nacheinander ausgeführt werden, kommen sie sich gegenseitig in die Quere und legen ein unerklärliches, weil nicht deterministisches Verhalten an den Tag.

Welche Optionen bieten sich uns, um diese Effekte zu vermeiden? Verzahnte Testfälle lassen sich in so gut wie allen Fällen vereinfachen:

Design verbessern durch orthogonale Testfälle

Stattdessen kann das Design meist in der Weise verbessert werden, dass sich Testfälle orthogonal schreiben, ändern und ausführen lassen. Johannes Link beschreibt in seinem Buch [li05] ein schönes Beispiel, wo die direkte Abhängigkeit auf eine externe Arbeitsdatei durch einen entkoppelnden `InputStream` aufgehoben wird. Die Testfälle, die unter normalen Umständen mit physikalischen Dateien hantieren würden, lassen sich auf diese Weise auf einen simplen `StringBufferInputStream` reduzieren. Lässt sich die unschöne Abhängigkeit dagegen tatsächlich nicht brechen, zum Beispiel weil die verschiedenen Tests nicht unabhängig voneinander durchführbar sind, müssen wir sie halt in einen gemeinsamen Testfall packen.

Temporale Kopplung bei funktionalen Tests

Unvermeidbar ist die *temporale Kopplung* von Testfällen beim funktionalen Testen. Akzeptanztests wären beispielsweise so ein Fall. Sie testen das Gesamtsystem von oben bis unten und leiten sich aus konkreten Anwendungsszenarien ab. Eine Akzeptanztestsuite könnte zum Beispiel so aussehen, dass zunächst ein Test des Logins läuft, abgelöst durch einen Test der Eingabemaske, abgeschlossen durch einen Test der Bestandsübersicht.

In JUnit lässt sich die Reihenfolge von Tests explizit beeinflussen, indem wir unsere Suiten manuell zusammenbauen, so wie wir es von unseren `AllTests`-Klassen gewohnt sind. Für gewöhnliche Unit Tests sollte dies jedoch nicht nötig sein. Man könnte auf die Idee kommen, die Testsuite so zu ordnen, dass alle unabhängigen Low-Level-Klassen vor den von ihnen abhängigen High-Level-Klassen getestet werden. Die Hoffnung wäre, dass zum Beispiel ein Fehler in der `Euro`-Klasse zuoberst im Testergebnis auftauchen würde, vor all den Fehlschlägen der Klassen, die intern `Euro` verwenden. In der Praxis stellt sich diese Problematik jedoch überhaupt nicht, da wir unsere Suite nach jeder winzigen Änderung ausführen und die Fehlerursache somit ohnehin unmittelbar diagnostizierbar ist.

Vermeidung von Nebeneffekten

Welche Möglichkeiten haben wir gegen ungewollte Nebeneffekte? Nun, Designs sind ohne Singletons und andere globale Variablen eh besser verständlich und einfacher änderbar, also reduzieren oder besser vermeiden wir sie von vornherein. Was externe Testressourcen angeht, bieten sich diverse Optionen an: Konflikte zwischen unterschiedlichen Testfällen lassen sich zum Beispiel durch markierte oder nicht sensitive Testdaten verhindern. Rückstände aus gelaufenen Testfällen können beseitigt werden, indem unsere Tests die Testressourcen zurücksetzen. Datenbestände können initialisiert werden, bevor unsere Tests laufen. Testlaufinterferenzen mit anderen Personen im Projekt können durch exklusive Ressourcenreservierung und -freigabe mittels Semaphore vermieden werden.

Ein häufig gemachter Anfängerfehler ist, die Testumgebung im Konstruktor oder Deklarationsteil der Testfallklasse zu initialisieren anstatt in der `setUp`-Methode. Da das Test-Framework aber erst alle Testfallobjekte in der Suite erzeugt und danach seinen Testlauf startet, in dessen Verlauf dann erst `setUp` für jeden Testfall ausgeführt wird, können dadurch ganz widerspenstige Effekte zwischen den Testfällen auftreten.

Ein guter Test für eine Testsuite wäre in der Tat, die Reihenfolge der Testfälle einmal dem Zufall zu überlassen. Ein Problem wäre nur, dass unsere Testläufe nicht mehr wiederholbar und ebenso wenig reproduzierbar wären.

7.24 Selbsterklärende Testfälle

Johannes Link kommentierte das Kapitel 5.12, *Die letzte Durchsicht*, in seinem Review damit, dass die PriceTest-Klasse nicht unabhängig von der Konstante NEWRELEASE und der Price-Klasse verständlich wäre:

```
public class PriceTest extends TestCase {
  public void testBasePrice() {
    assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(1));
    assertEquals(new Euro(2.00), Price.NEWRELEASE.getCharge(2));
  }

  public void testPricePerDay() {
    assertEquals(new Euro(3.75), Price.NEWRELEASE.getCharge(3));
    assertEquals(new Euro(5.50), Price.NEWRELEASE.getCharge(4));
  }
}
```

Natürlich hat er Recht. In seinem Buch schreibt er schließlich auch: Testdaten und erwartete Ergebnisse sollten nahe beieinander stehen. Befolgen wir also seinen Rat:

```
public class PriceTest extends TestCase {
  private Price price;

  protected void setUp() {
    price = new Price(new Euro(2.00), new Euro(1.75), 2);
  }

  public void testBasePrice() {
    assertEquals(new Euro(2.00), price.getCharge(1));
    assertEquals(new Euro(2.00), price.getCharge(2));
  }

  public void testPricePerDay() {
    assertEquals(new Euro(3.75), price.getCharge(3));
    assertEquals(new Euro(5.50), price.getCharge(4));
  }
}
```

Der Gewinn dieses Refactorings ist, dass sowohl der Konstruktor in Aktion gezeigt wird als auch die Geschäftsregeln deutlicher werden, da man nun die Parameter sieht, von denen die Preisberechnung abhängt.

Man könnte meinen, dass die Konstruktorparameter immer noch unklar wären. Speziell dafür bieten sich jedoch besser die Tooltipps im Entwicklungswerkzeug an: basePrice, pricePerDay, daysDiscounted.

Generell ist es ein gutes Ziel, jede Testfallmethode so zu schreiben, dass sie für sich genommen verständlich ist und eine kleine Geschichte über unser Programm erzählt. Durchkreuzt wird dieses Ziel manchmal jedoch dadurch, dass wir Gemeinsamkeiten ins Setup herausziehen. Unsere Testfälle können dann nicht mehr einfach von oben nach unten gelesen werden. Oft helfen zwar Namen zur Erklärung der Umstände. Wird aber trotz guter Namen verschleiert, was der Test genau testet, müssen wir die Fixture-Variablen wieder zurück in den Testfall ziehen. Unsere Absicht über den Testcode zu kommunizieren, ist schließlich wichtiger, als der Duplikation von Setup-Code zu entrinnen.

Äußere Abhängigkeiten vermeiden

Ähnlich sieht es für externe Testressourcen aus, weil wir uns zwei Stellen anzusehen haben, um einen Testfall zu verstehen. Eine bessere Lösung ist deswegen, in den Unit Tests von externen Datenquellen und -senken unabhängig zu sein. Ein oder zwei Integrationstests können den Durchgriff auf externe Ressourcen durchtesten. Der zu testende Code dagegen benötigt nur die Daten, die normalerweise aus externer Quelle stammen bzw. ins externe Ziel geschrieben werden. Auf diese Weise lassen sich äußere Abhängigkeiten sehr einfach kappen und alle Informationen für einen Testfall an Ort und Stelle lokalisieren.

7.25 String-Parameter von Zusicherungen

Was gibt es zum Begleittext der Zusicherungsmethoden zu sagen? Einige Entwickler schwören darauf. Ich denke, sie sind ein Test Smell. Zur Illustration diene unser Code aus Kapitel 3.13, *Zwei Methoden, die das Testen vereinfachen*:

```
public void testRounding() {
  Euro rounded = new Euro(1.995);
  assertEquals("amount not rounded", two, rounded);
}
```

Hier ist der Begleittext durch die Verwendung von `assertEquals` und den Testfallnamen `testRounding` eigentlich vollkommen überflüssig.

Deshalb: Alle Fälle, wo mir ein Erklärungstext über den Weg gelaufen ist, hatten ein oder mehrere der folgenden Probleme:

- schlecht benannte Testfallmethoden
- zu viele Zusicherungen pro Testfall
- fehlende spezialisierte `assert...`-Methoden

Aber, wie gesagt, einige Entwickler schwören darauf. Besser Sie fällen, wie für alles, was Sie lesen und hören, Ihr eigenes Urteil!

7.26 Szenarientests

Eine Art von Testfällen, auf die mich Johannes Link häufiger hinweist, weil ich sie (meist auf der Suche nach möglichst orthogonalen Tests) gerne vergesse, ist an typischen Verwendungsszenarien orientiert. Diese Testfälle sollen jene Probleme aufdecken, die sich erst nach ein paar Zustandsänderungen eines Objekts überhaupt beobachten lassen.

Charakteristisch ist, dass sie eine Reihe von Methodenaufrufen absetzen, wie sie für den Lebenszyklus des Objekts in der Anwendung typisch und sinnvoll sind. Je mehr Zustände und Zustandsübergänge ein Objekt kennt, desto komplexer fallen die Szenarien natürlich aus. Sinnvoll sind Positivtests für alle erlaubten Statusübergänge, Negativtests für nicht gültige Übergänge und vielleicht auch Recovery-Tests, die überprüfen, ob das Objekt durch ein Fehlerszenario nicht völlig korrumpiert wird.

7.27 Testexemplare

Bleiben wir noch bei dem Test fürs Runden von Eurobeträgen stehen. Warum ist dieser Test eigentlich ziemlich schlecht?

```
public void testRounding() {
  Euro rounded = new Euro(1.995);
  assertEquals("amount not rounded", two, rounded);
}
```

Er testet gar nicht, was er zu testen vorgibt. Was wirklich getestet wird, wird ganz deutlich, wenn wir die Euro-Klasse geeignet sabotieren:

```
public class Euro...
  private final long cents;

  public Euro(double euro) {
    if (euro < 0)
      throw new IllegalArgumentException("negative amount");

    cents = Math.round(euro * 100.0);
  }

  public double getAmount() {
    return cents / 100.0;
  }
}
```

Man mag seinen Augen ja kaum trauen, aber wir können tatsächlich die Euro-Cents-Umwandlung löschen, ohne dass auch nur ein einziger Test der Klasse fehlschlägt. Ein gutes Beispiel dafür, dass vollständige Codeabdeckung nur eine Metrik unter vielen ist für die Testqualität. Der Code an sich war auch vollkommen korrekt. Es ist der Test selbst, der hier unzureichend ist.

Wie kann das angehen? Der Test täuscht vor, eine Fließkommazahl zu testen. Tatsächlich ist der Testrepräsentant `1.995` allerdings so unglücklich gewählt, dass er durch das Runden zur Ganzzahl `2` wird, was den Test schließlich zufrieden stellt. Um wirklich die Genauigkeit von zwei Nachkommastellen zu garantieren, müssen wir auch Proben für signifikante Fließkommazahlen nehmen:

```
assertEquals(new Euro(0.49), new Euro(0.494));
assertEquals(new Euro(0.50), new Euro(0.495));
```

Da wir stets nur exemplarisch testen können, ist die Güte gewählter Testrepräsentanten das A und O fürs effektive Testen.

Was können wir aus diesem Fehler lernen? Die erhoffte Wirkung, ob auf- oder abzurunden ist, war im Test gar nicht wirklich erfasst. Um dies zu tun, müssen wir entlang interessanter Grenzbedingungen testen. Das sind jene Bedingungen, anhand derer unser Code seine Wirkungsweise entscheidend ändert: entweder indem der Kontrollfluss einen gewissen Pfad einschlägt oder indem der Definitionsbereich erlaubter Eingabewerte verlassen wird und unser Programm gänzlich aufhört zu funktionieren.

Grenzfälle testen

Testgetriebene Testfälle besitzen die interessante Eigenschaft, *natürlicher Attraktor* dafür zu sein, gerade die interessanten Grenzfälle als Testexemplare auszuwählen. Das bedeutet nicht, dass uns deshalb keine Fehler unterlaufen. Die Betrachtung der Grenzbedingungen führt jedoch zu neuen Testfällen und diese erst zur Motivation der nötigen Programmlogik und Entscheidungspunkte.

Verfahren zur Testfallermittlung

Die Literatur zum Softwaretesten ist gespickt mit Verfahren zur Testfallermittlung: Äquivalenzklassenbildung, Grenzwerteanalyse und Ursache-Wirkungs-Graphen sind häufig anzutreffende Testmethoden. Ob und welche Rolle die klassischen Methoden in der Test-First-Welt spielen könnten, ist mir bis dato unklar. Was jedoch damit zu tun haben kann, dass ich mir nicht bewusst darüber bin, wie viel ich davon intuitiv anwende. Einzelne Aspekte sind auch in der testgetriebenen Arbeitsweise wiederzuerkennen. Wenn ich mir allerdings die in den traditionellen Testbüchern diskutierten Beispiele anschaue, handelt es sich vielfach um dermaßen widerspenstige Programmstrukturen, dass sie durch Test-First gar nicht erst hätten entstehen können.

7.28 Testsprachen

Wächst das Verhältnis von Test- zu Anwendungscode, ist das mehr als häufig ein Hinweis auf fehlende Abstraktionen im Testcode. Schauen wir uns ein wenig in unseren Tests um, dann fällt schnell auf, dass sich hie und da Duplikation eingeschlichen hat:

```
public void testBasePrice() {
  assertEquals(new Euro(2.00), price.getCharge(1));
  assertEquals(new Euro(2.00), price.getCharge(2));
}

public void testPricePerDay() {
  assertEquals(new Euro(3.75), price.getCharge(3));
  assertEquals(new Euro(5.50), price.getCharge(4));
}
```

Ob wir diese leichte Duplikation schon auflösen, ist davon abhängig, wie gut verständlich der Test mit einer extrahierten Methode noch ist:

```
public class PriceTest...
  public void testBasePrice() {
    assertCharge(2.00, 1);
    assertCharge(2.00, 2);
  }

  public void testPricePerDay() {
    assertCharge(3.75, 3);
    assertCharge(5.50, 4);
  }

  public void assertCharge(double charge, int days) {
    assertEquals(new Euro(charge), price.getCharge(days));
  }
}
```

Das Refactoring bringt hier den Gewinn, dass wir nur noch eine Stelle in den Tests berühren, sollten wir die Preisberechnung ändern müssen. Mittelfristig zeichnen sich auf diesem Weg neue `assert...`-Methoden ab, später einige Testhelferklassen, und langfristig bildet sich eine eigene domänenspezifische Testsprache heraus, mit deren Hilfe wir mit abnehmendem Aufwand zunehmende Mengen Code testen können.

Was ich hier nicht gezeigt habe, weil es sich nicht gelohnt hätte, was jedoch für einige spezialisierte `assert...`-Methoden nützlich ist: Spendieren Sie Ihren Zusicherungen eine ausdruckskräftige Meldung: Warum und vor allem mit welchen Werten bin ich fehlgeschlagen?

7.29 Umgang mit Defekten

Testgetriebene Entwicklung, das zeigen die Erfahrungen ihrer Pioniere, versetzt uns in die Lage, Software mit deutlich geringeren Defektraten zu entwickeln. Trotzdem finden wir natürlich ab und zu auch Fehler. Das müssen gar keine Programmierfehler sein. Manchmal handelt es sich einfach um eine Anforderung, die wir missverstanden haben und deshalb im Code richtig stellen müssen.

Doch was tun, wenn ein Fehler entdeckt wird? Natürlich schreiben wir erst einen neuen Test, der den Fehler entlarvt. Oder passen einen alten Test an oder erweitern ihn. Erst danach wird der Code korrigiert! Und was tun wir mit dem Feedback, das uns ein durchgeschlüpfter Fehler bietet? Das ist viel, viel wichtiger!

Ein einfaches Beispiel: Leider ist unsere Multiplikation von Eurobeträgen fehlerhaft:

```
public class Euro...
  public Euro times(int factor) {
    return new Euro(cents * factor);
  }
}
```

In Kapitel 3.14, *Testen von Exceptions*, wurde einfach so entschieden, negative Beträge allgemein zu versagen. Ein kurzer Lerntest kann jedoch aufdecken, dass wir noch problemlos `Euro`-Exemplare mit negativem Wert erzeugen können:

```
public void testNegativeAmount() {
  try {
    final double NEGATIVE_AMOUNT = -2.00;
    new Euro(NEGATIVE_AMOUNT);
    fail("amount must not be negative");
  } catch (IllegalArgumentException expected) {
  }

  try {
    two.times(-1);
    fail("multiplying with a negative number not allowed");
  } catch (IllegalArgumentException expected) {
  }
}
```

Die meisten Fehler sind wie dieser: unvorhergesehene Konsequenzen von Änderungen und Erweiterungen am bestehenden Code, die häufig ganz offensichtlich scheinen, nachdem sie erst einmal gefunden sind.

Fehler reproduzieren

Damit einmal entdeckte Fehler später nicht noch einmal auftreten, fügen wir zunächst einen neuen Test hinzu. Der Test sollte laufen, sobald das Problem behoben ist. Der rote Balken verifiziert uns dabei, ob wir die Fehlerursache begriffen haben und reproduzieren können. Ist dies getan, gehts ans eigentliche Problem.

In unserem Fall heißt es, die Multiplikation mit einem negativen Skalar ganz und gar zu verbieten. Recht einfach geht das, indem wir der `times`-Methode eine passende Guard Clause verpassen. Aber das scheint nicht der wahre Kern des Problems zu sein. Fehler häufen sich oftmals dort, wo schon andere gefunden wurden. Vielfach gibt dies Aufschlüsse über die wahre Natur unseres Codes.

Das wirkliche Problem ist vielmehr eine kleine Codeduplikation im öffentlichen und privaten Konstruktor, die wir geschickt auflösen, indem wir den einen auf den anderen zurückführen, was sowieso eine gute Idee ist. In diesem Zuge können wir die bestehende Guard Clause in den privaten Konstruktor verschieben und haben damit nur noch eine einzige Codestelle, wo die `cents`-Instanzvariable verändert wird:

```
public class Euro...
  public Euro(double euro) {
    this(Math.round(euro * 100.0));
  }

  private Euro(long cents) {
    if (cents < 0)
      throw new IllegalArgumentException("negative amount");

    this.cents = cents;
  }
}
```

Vom Defekt im Code zum Defekt im Entwicklungsprozess

Wenn ein Fehler durchrutscht, müssen wir nicht nur ausfindig machen, wo, sondern vor allem auch wie er durchrutschen konnte. In vielen Fällen zeigt ein Defekt im Code einen Defekt im Entwicklungsprozess an. Indem wir der wahren Natur unserer Fehler auf den Grund gehen, können wir unseren Prozess schrittweise verbessern. Das ist ein kurzer Prozess-Check in Form einer Mea-Culpa-Session (lat. meine Schuld):

- Welchen Test hätten wir schreiben müssen, um den Fehler ganz und gar zu vermeiden?
- Welche zusätzlichen Tests suggeriert der Fehler an anderer Stelle?
- Woran erkennen wir zukünftig, dass wir den Fehler wiederholen?
- Wie können die Kundenanforderungen so kommuniziert werden, dass wir Missverständnisse schneller aufdecken?

Eine von vielen guten Ideen des *Toyota Produktionssystems* [po_{03}] ist, sich über *fünf W-Fragen* zur wirklichen Fehlerursache durchzufragen:

Fünf W-Fragen

1. Es lassen sich negative Eurobeträge erzeugen! Doch warum?
2. Die `times`-Methode lässt einen negativen Skalar zu! Warum?
3. Keine Guard Clause, die es verhindern würde! Warum nicht?
4. Die Guard Clause ist nur im anderen Konstruktor! Warum?
5. Weil der eine nicht auf den anderen zurückgeführt wird! Aha!

Ein schöner Effekt ist, wie unsere Testsuite als *Bug-Datenbank* dient. Führen wir einen Fehler später noch einmal ein, weil wir beispielsweise eine unbedachte Änderung machen, erhalten wir eine Erinnerung von unserem Test. Einmal beseitigte Fehler bleiben damit auch beseitigt.

Die Testsuite als Datenbank aller gefundenen und behobenen Bugs

Angefangen beim JUnit-Projekt etabliert sich in der Open Source Community seither die Tugend, als *Bug-Report* oder *Feature-Request* immer auch einen Test zur Erklärung des Problems mitzuliefern.

Eine Frage der (Test-)Kultur

von Christian Junghans, arvato mobile & Olaf Kock, abstrakt gmbh

Während eines unserer Projekte arbeiteten zwei Entwicklungsteams, die auch räumlich getrennt waren, an einer Applikation. Unser Team, welches das Projekt begonnen hatte, entwickelte testgetrieben; das zweite Team schrieb keine Unit Tests und führte auch die bestehenden Unit Tests nicht aus.

Nach dem Auschecken aus dem gemeinsamen CVS-Repository ließen wir jeweils die Tests laufen. Dabei fand sich eines Tages die Fehlermeldung:

```
AssertionFailedError: testSendMails: expected 4 but was 3
```

Ein kurzer Blick in den vom anderen Team geänderten Code zeigte einen klassischen »off by one«-Fehler. Bei einer Iteration über die Datensätze blieb jeweils die letzte E-Mail immer unverschickt.

Der Entwickler aus unserem Team, der den Test ausgeführt hatte, informierte also den verantwortlichen Entwickler aus dem anderen Team über den Fehler. Er beschrieb den Fehler (mit Code-Snippet) und bat darum, doch bitte die Tests auszuführen, um keine implementierten und getesteten Features zu zerstören. Das Resultat war niederschmetternd: Beim nächsten Checkout war der fehlerhafte Code unverändert. Dafür war die fehlgeschlagene Testmethode auskommentiert, was aber trotzdem zu einem roten Balken führte. Da nämlich alle Testmethoden in der Testklasse auskommentiert waren, bemängelte JUnit freundlicherweise die testmethodenlose Testklasse.

7.30 Umgang mit externem Code

Eigentlich wollen wir externen Code nicht viel anders behandeln als unseren eigenen Code, doch meist stehen uns dabei die gesetzten Codegrenzen im Weg.

Im eigenen Entwicklungsteam können wir durch die Öffnung der Codegrenzen erreichen, dass der Code wirklich dem Team gehört. Unsere Tests ermöglichen diese *gemeinsame Codeverantwortlichkeit*, da sie uns das Vertrauen geben, auch fremden Code ändern zu können, selbst wenn wir ihn vielleicht nicht vollständig verstanden haben.

Publizierte Schnittstellen

Ganz anders sieht die Situation aus, sobald wir mit *publizierten Schnittstellen* zu tun bekommen, d.h., wenn wir nur eine der beiden Seiten eines Schnittstellensaums kontrollieren können, wir als Anbieter einer Schnittstelle also keinen Zugriff auf die Kundenseite haben oder als Kunde keine Macht über den Anbieter. Bekannte Probleme treten zum Beispiel an der Schnittstelle zu Bibliotheken, Altsystemen oder auch anderen Entwicklungsteams auf.

Zumindest noch so lange, wie wir extern produzierten Code nicht immer mit einer Suite automatisierter Selbsttests geliefert bekommen, müssen wir ab und an in den sauren Apfel beißen und das Versäumnis in unserer eigenen Testsuite nachholen: Besonders nützlich sind Tests, die unsere Annahmen über fremden Code sowohl bestätigen wie auch dokumentieren. Eine Art des Lerntests. Wertvoll sind solche Tests, um die im externen Code befindlichen Fehler per Brandzeichen zu kennzeichnen: Wird später eine neue Version der Software ausgeliefert, können wir einfach unsere Tests starten und sofort sehen, welche bekannten Fehler der Anbieter behoben hat und welche neuen Fehler er eingebaut hat.

Fehler im externen Code durch Tests kennzeichnen

Schnittstellen zwischen Teams

Um am Schnittstellensaum zwischen zwei Teams zu Schnittstellenentwürfen zu kommen, die sich sowohl an den Anforderungen der Kundenseite orientieren als auch an den Möglichkeiten des Anbieters, ist ein *Dialog* zwischen den beiden Parteien notwendig. Kent Beck stellt sich diesen Austausch so vor, dass Mitglieder aus beiden Teams paarweise die gemeinsame Schnittstelle erarbeiten [be_{02b}]. Die Schnittstellensemantik, auf die sie sich einigen, quasi ihren Vertrag, gießen sie in eine Suite von Tests. Um ihre Arbeit voneinander zu entkoppeln, programmiert die Kundenseite nur gegen einen primitiven Platzhalter, auch *Stub* genannt, der die Funktionalität der Anbieterseite geeignet simuliert. Die entscheidende Idee liegt im *sozialen Netzwerk* zwischen Kunden- und Anbieterseite. Wichtig ist, an der Schnittstelle der zwei Teams mit kurzen Evolutionszyklen ein für beide Seiten ausgewogenes Design zu erreichen.

7.31 Was wird getestet? Was nicht?

Die knappste und meiner Meinung nach immer noch beste Antwort auf die oben gestellte Frage stammt von Dierk König: »unsere Logik«. Mit Betonung auf beiden Wörtern! »Unsere« bedeutet, dass wir nicht testen, was uns nicht gehört. Und »Logik« heißt, dass wir nicht testen, was nicht schief gehen kann.

Testen privater Methoden

Eine häufige Frage ist: Wie testen wir nicht öffentliche Methoden? Überhaupt nicht! Methoden, die `private` oder `protected` sind, gehen bei Testgetriebener Entwicklung durch Extraktion duplizierten Codes aus öffentlichen Methoden hervor, werden von diesen verwendet und damit indirekt mitgetestet. Wenn private Methoden einzeln getestet werden müssen, ist dies ein Indiz, dass wir mehr testen wollen als nur ein Implementierungsdetail. In Wirklichkeit geht die Klasse schwanger, denn die Methode will Teil einer öffentlichen Klassenschnittstelle sein. Es ist unsere Aufgabe, einen Kandidaten dafür zu finden.

Eine wichtige Grundregel zum Abschluss: Tests sollen keine eigene Logik enthalten, sonst müssten sie eigentlich wieder getestet werden. Tammo Freese verlangt, dass Testcode auf der Grundlinie verläuft: keine Logik = keine syntaktische Einrückung.

7.32 Zufälle und Zeitabhängigkeiten

Code zu testen, der von einem Zufallsgenerator oder der Systemzeit abhängt, kann schwierig sein. Der Kniff dabei ist, den Produzenten und Konsumenten der nicht deterministischen Daten in zwei Objekte aufzutrennen. So können wir Ersteren simulieren, um Letzteren mit deterministischen Werten gefüttert zu testen. Wie das im Detail geht, erklärt Lasse Koskela; wie es in der Praxis läuft, das nächste Kapitel.

Der zeitlose Weg des Testens

von Lasse Koskela, Reaktor Innovations

Zeit ist eine merkwürdige Sache. Selten dauert etwas für genau die richtige Länge an: Manchmal braucht etwas zu lange, manchmal ist etwas zu schnell vorüber, fast nie jedoch ist etwas *genau richtig*. Wenn Sie jemals versucht haben, zeitabhängigen Code nachträglich mit Tests auszustatten, wissen Sie wahrscheinlich, dass es nahezu unmöglich ist, gute verlässliche Tests zu schreiben, die schnell sind und über mehrere Testläufe hinweg auch auf konsistente Weise durchlaufen oder fehlschlagen.

Die Zeit zu kontrollieren, ist offensichtlich möglich. Nach meiner Erfahrung probieren es die meisten nur gar nicht erst, was dann zu langsamen Testsuiten führt oder dem kompletten Mangel an Tests in dem Bereich. Glücklicherweise ist es erheblich einfacher, wenn man Testgetriebener Entwicklung folgt und von der ersten Zeile Code an über Testbarkeit nachdenkt.

Lassen Sie mich eine kleine Geschichte erzählen.

Ein Kollege und ich trafen uns eines Morgens im Büro, um eine Codebasis im Paar zu programmieren, die wir später in einem internen Training verwenden wollten. Zu diesem Zeitpunkt hatten wir noch keine Zeile Code und starteten mit einer schnellen Skizze der Benutzerschnittstelle für die Benutzergeschichten, die wir für unsere Software ins Auge gefasst hatten: eine persönliche Zeiterfassung.

Teil unserer Benutzerschnittstelle war eine Tabellenspalte, in der eine Stoppuhr mitzählen sollte, wie viel Zeit bisher mit einer bestimmten Aufgabe verbracht worden war. Die Uhr sollte nur für die gerade aktiv gesetzte Aufgabe mitlaufen, während alle anderen Zähler bleiben sollten, wie sie waren.

Nun, offensichtlich musste unsere Anzeige aktualisiert werden. Wir mussten den Zähler dazu bringen, seinen Wert wenigstens einmal pro Sekunde neu darzustellen. Wie testet man so was? Eine Reihe unserer Tabelle zu selektieren, war ein Kinderspiel. Es war jedoch nicht unmittelbar ersichtlich, wie wir überprüfen sollten, dass die Stoppuhr richtig arbeitete und die Anzeige mit der korrekten Anzahl von Sekunden aktualisierte. Wir hätten einfach den Thread für eine Sekunde blockieren und nachsehen können, ob der Wert währenddessen von 0:00:00 auf 0:00:01 gesprungen ist. Aber das ist hässlich, unzuverlässig und langsam. Und so was wollen wir doch nicht, oder?

Die Lösung, die ich zuvor bereits verwendet hatte und von der ich wusste, dass sie funktionierte, war: das Konzept der Systemzeit in einer eigens dafür geschaffenen `Time`-Klasse zu abstrahieren und die Kontrolle über das Timing umzudrehen: von der zeitabhängigen Komponente zur `Time`-Klasse. Mein Kollege war der Idee gegenüber etwas misstrauisch und fand, dass wir die Komplexität unnötig in die Höhe treiben würden und so weiter. Die anfängliche Skepsis wich jedoch, als ich ihm zeigte, wie ich mir die für die Zeitabstraktion nötige Schnittstelle vorstellte, wie unser Test aussähe und wie wir nicht mehr von der wirklichen Zeit abhängig wären.

Zeit von Anfang an unter Kontrolle zu halten, ist nicht so schwer und zahlt sich wirklich aus, wenn man mit zeitbezogenen Funktionen wie Schaltuhren, Stoppuhren und dergleichen zu tun hat.

Haben Sie Ihre Zeit bereits unter Kontrolle?

8 Isoliertes Testen durch Stubs und Mocks

Unit Tests sollen jeweils eine Funktionseinheit in Isolation von anderen Einheiten testen. Sie beziehen sich auf kleinere Programmeinheiten, wie etwa eine einzelne Klasse oder ein Team von wenigen, stark zusammenspielenden Klassen. Doch schon einfache Abhängigkeiten im Programmcode können zu aufwändigeren Testumgebungen führen. Zahlreiche Klassen lassen sich gar nicht einzeln testen, sondern eben nur in einer eng zusammenhängenden Gruppe. Das ist ein effektiver Mikrointegrationstest und nicht ungewöhnlich für objektorientierten Code, da sich nahezu alle Klassen auf anderen Klassen abstützen.

Übersteigen diese Abhängigkeiten nur ein gewisses Maß, nimmt der Testaufwand überproportional zu. Zum einen muss dann für den geringsten Test bereits ein erheblicher Anwendungskontext aufgebaut und verwaltet werden. Zum anderen wächst mit dem Umfang der Testumgebung auch die Anzahl zu schreibender Tests dramatisch an. Während wir testgetrieben entwickeln, ist es leicht, diese Symptome frühzeitig zu erkennen und zu vermeiden: Wir müssen uns dazu nur möglichst wenig Arbeit machen wollen mit dem Schreiben neuer Tests. Anders sieht die Situation hingegen für zu pflegende Altsysteme aus: Hier bekommen wir es oft mit Hinterlassenschaften zu tun, die ohne große Teile des Gesamtsystems nicht lauffähig sind und damit kaum über Unit Tests getestet werden können.

Wenn Sie dieses Kapitel gelesen haben, sollten Sie in der Lage sein, auch dunkelste Ecken im Code durch fokussierte Tests auszuleuchten. Dazu werden wir unser Repertoire um weitere Techniken ausbauen, mit denen selbst hartnäckige Testprobleme geknackt werden können. Diese Testtechniken helfen einerseits, von vornherein zu verhindern, dass Ihre Unit Tests zu aufwändig werden. Andererseits kann Ihre Arbeit von den diskutierten Techniken profitieren, wenn Sie mit schlecht testbarem Code zu tun haben und nicht länger damit leben können.

8.1 Verflixte Abhängigkeiten!

Im effektiven Unit Test müssen sich Softwarestrukturen unabhängig voneinander testen lassen. Dabei stellen uns aber Objekte, die noch von anderen Objekten abhängig sind, vor neue Herausforderungen: Die Ursache von Fehlern lässt sich oft nur effizient eingrenzen, solange wir eine Programmeigenschaft für sich allein unter die Lupe nehmen. Fokussierte Unit Tests wiederum sind erst möglich, sofern wir alle Abhängigkeiten zu entfernten Teilen, die wir im betreffenden Testfall nicht indirekt mittesten wollen, minimiert haben.

In nahezu allen Fällen wollen uns Probleme beim Testschreiben auf Problemzonen im Softwareentwurf aufmerksam machen. Eine Frage, die wir uns stellen müssen, wann immer sich der Code gegen den Test verschworen hat, lautet daher: Wieso kann das Stückchen Code nicht ohne die Existenz der anderen Codeteile getestet werden? Der Testcode ist letztendlich nur ein weiterer Verwender der getesteten Einheit. Er spiegelt die typischen Benutzungsszenarien wider, die auch spätere Verwender im Anwendungscode reproduzieren werden. Die Tests erlauben uns somit, wichtige Rückschlüsse auf die Modularisierung des Designs zu ziehen. Wenn der Test einer Klasse zum Beispiel noch ein Heer von anderen Klassen erfordert, zeugt das von missratenem Codeabhängigkeitsmanagement. Ein Review der Tests gibt uns die Gelegenheit, das Design neu zu durchdenken, potenzielle Refactorings zu lokalisieren und dadurch sogleich die Verwendungsmöglichkeiten unserer Klassen zu vereinfachen.

Tests als Spiegel

So sehr uns Abhängigkeiten zwischen einzelnen Codeteilen das Testen erschweren, minimale Abhängigkeiten sind naturgegeben nicht vermeidbar. Klassen und Objekte arbeiten nun mal immer in Teams. Beim Testschreiben ergeben sich daraus für uns jedoch trotzdem noch ein paar kleine Herausforderungen:

Probleme beim Testschreiben

- **Der Aufbau der Testumgebung ist aufwändig,** weil durch manche Abhängigkeiten ein ganzer Rattenschwanz mitarbeitender Objekte erzeugt und konfiguriert werden will.
- **Die Tests werden beim Überschreiten der Systemgrenzen langsam,** weil die Interaktion mit Datenbank und Middleware meist relativ viel Laufzeit frisst.
- **Das Testen von Ausnahmesituationen gestaltet sich als schwierig,** weil es verhältnismäßig aufwändig ist, Fehler im System oder in der Peripherie gezielt zu (re-)produzieren.

Kurzum: Wir müssen dafür Sorge tragen, dass unsere Tests nicht zu *komplex*, nicht zu *langsam* und nicht zu *löchrig* werden.

8.2 Was ist die Unit im Unit Test?

Eine Frage, die wir uns immer stellen müssen, bevor wir einen Test schreiben, heißt: Was genau ist denn die Unit, die wir testen wollen? Um eine Antwort darauf zu finden, müssen wir das System in kleine, unabhängig voneinander testbare Einheiten isolieren. Wir müssen uns ein klares Bild von unserem Programmcode machen und ihn nach Zusammenhängen teilen. Diese Entkopplung führt zu Schnittstellen in unserem Code und diese wiederum ermöglichen uns den Test eines Softwarebausteins in Isolation von anderen Bausteinen.

Isolation durch Schnittstellen

Betrachten wir zum Beispiel einen Systemausschnitt, der aus drei Programmeinheiten besteht: Die erste Einheit soll fünf Eigenschaften besitzen, die zweite drei und die dritte zwei. Es könnte sich bei den drei Bausteinen um gewöhnliche Klassen wie auch um Schichten in einer Schichtenarchitektur handeln. Oder stellen Sie sich vor, wir möchten drei Funktionen unter fünf Browsern und zwei Datenbanken testen:

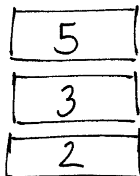


Abb. 8–1
Unsere drei Testgegenstände ...

Die Frage nach der Unit im Unit Test kennt schon in diesem einfachen Fall mehrere Antworten:

- Wir könnten die drei Einheiten im engen Zusammenspiel einem Mikrointegrationstest unterziehen (links in Abb. 8–2).
- Wir könnten jede Einheit vollständig von ihrer Umgebung isolieren und einem strikten Unit Test unterziehen (rechts in Abb. 8–2).

Im häufigsten Fall ist die Antwort keiner der beiden Extrempunkte, sondern vielmehr ein »Sweet Spot«, der zwischen diesen zwei Polen liegt und uns mit einer angemessenen Anzahl von Tests Sicherheit gibt. Insgesamt existieren hier vier Möglichkeiten, das Skalpell anzusetzen:

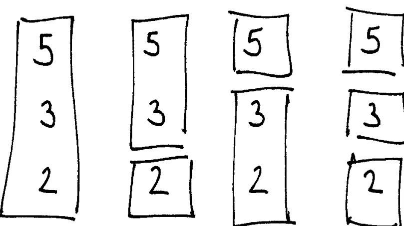


Abb. 8–2
... und ihre Testkombinatorik

8.3 Mikrointegrationstest versus strikter Unit Test

Wie viele Tests wir für ein System schreiben, hängt davon ab, wie wir die Teilsysteme entwerfen und welche Einheiten wir im Test zugrunde legen. Werfen wir zunächst einen Blick auf den Mikrointegrationstest: Wenn wir das Zusammenspiel der beschriebenen drei Einheiten als zu testende Einheit auffassen, schreiben wir bis zu 30 Testfälle:

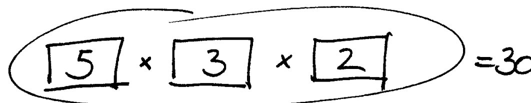


Abb. 8–3
Mikrointegrationstest = viele Kombinationen zu testen

Integrationstests erfordern unter Umständen, das Produkt(!) aller möglichen Kombinationen in der Interaktion beteiligter Programmkomponenten abzutesten. Die Gegenüberstellung mit dem Unit Test: Wenn wir jede Einheit in strikter Isolation testen, schreiben wir gerade zehn Testfälle:

5 + 3 + 2 =10

Abb. 8–4
Strikte Unit Tests = keine Kombination

Die Ersparnis strikter Unit Tests erscheint auf den ersten Blick riesig, doch vollständig isoliertes Testen ist leider mit Folgekosten verbunden, die uns zum Abwägen zwingen: Unit Tests erfordern unter Umständen, die Verwendungsbeziehung zu mitwirkenden Programmkomponenten während des Testlaufs gegen simulierte Komponenten auszutauschen. Diesen Aufwand müssen wir in unsere Abschätzung mit einbeziehen. Wann beginnen strikte Unit Tests also ineffektiv zu werden?

- Wenn wir viele verwendete Klassen im Test ersetzen müssen

Wann werden Mikrointegrationstests ineffektiv?

- Wenn wir das Produkt an Testfällen schreiben müssen
- Wenn wir nicht alle interessanten Grenzfälle testen können

Diese einfachen Heuristiken können uns bei der Entscheidung helfen, ob wir die betreffenden Bausteine nun besser miteinander integriert oder voneinander isoliert testen. Was ich nicht suggerieren möchte, ist, dass man die Anzahl der Tests einfach berechnen kann. Denn trotz umfassender Unit Tests für alle Einzelteile sind trotzdem noch Tests für ihr korrektes Zusammenspiel sinnvoll und notwendig!

8.4 Vertrauenswürdige Komponenten

Wie selbstverständlich hantieren wir in unserem Code mit anderen Klassen, die zum Teil gut getestet und zum Teil gar ungetestet sind. Hier stellt sich eine weitere Frage: Für welche der Systemkomponenten können wir im Test implizit voraussetzen, dass sie korrekt arbeiten? Wir müssen vorab entscheiden, welche Komponenten wir als stabil betrachten und gar nicht erst mittesten müssen. Zu der Gruppe von Komponenten, denen wir im Allgemeinen vertrauen, gehören unter anderen die folgenden:

Komponenten, die nicht getestet werden müssen

- **Die Java-Standardbibliothek.** Natürlich ist selbst dieser Code nicht ganz frei von Fehlern, doch wir verwenden diese Klassen meist so, als wären sie tadellos.
- **Eingesetzte Drittanbieter-Bibliotheken.** Die Qualität von Code aus dritter Hand ist stark schwankend. Grundsätzlich gelten hier die gleichen Gesetze wie für die Standardbibliotheken. Ein bereits vorgestellter Trick besteht darin, unser Verständnis von fremdem Code durch einige Lerntests zu verifizieren und zu dokumentieren.
- **Eigene getestete Klassen.** Ein Ziel ist, möglichst orthogonale Tests zu schreiben. Das heißt, wenn eine Klasse einmal getestet wurde, können wir sie anderenorts vertrauensvoll verwenden, ohne uns auf ungeliebte Überraschungen gefasst machen zu müssen.
- **Klassen, die wir als zu einfach erachten, um sie zu testen.** Tja, und einige Klassen sind so dumm, dass jeder Test rausgeworfenes Geld wäre. Einfache Datenhalterklassen, die keinerlei Logik enthalten, fallen in diese Kategorie.

Immer dann, wenn ein Test fehlschlägt, wollen wir annehmen können, dass sich der fehlerhafte Code in der getesteten Einheit selbst befindet. Werden in dieser Einheit aber andere Elemente implizit mit verwendet, verfälschen sie das Testergebnis möglicherweise. Aus diesem Grund müssen alle von einer Unit verwendeten Klassen

- **entweder vertrauenswürdig** sein
- **oder im Test ersetzt** werden.

Würden wir zum Beispiel die Einheit »5« testen wollen und dabei der Meinung sein, dass wir den Kumpanen »3« und »2« nicht trauen, müssten wir unserer »5« eine Einheit zur Verfügung stellen, die unsere wirkliche Verbundeinheit aus »3« und »2« zu Testzwecken adäquat ersetzen kann. Mit möglichst cleveren Mogelpackungen dieser Art wird sich der Rest des Kapitels beschäftigen.

Vertrauen – und Tests

von Bastiaan Harmsen, Software Team GmbH

Vertrauen ist eine Beziehung, die in unvorstellbar vielen Ausprägungen existiert. Wir kommen schnell von den einfachen Vertrauensbeziehungen zu den komplexen. Das Vertrauen in physikalische Eigenschaften von Gegenständen erscheint trivial – der Stuhl, auf dem Sie gerade dieses Buch lesen, wird mit hoher Wahrscheinlichkeit nicht zusammenbrechen. Auch ist nicht zu erwarten, dass Sie es mit einer spontanen Selbstentzündung des Buches zu tun bekommen oder beim Umblättern zur nächsten Seite ein Kobold aus dem Buch springt. Vertraute Welt halt.

Komplizierter ist das bei Sachen, die wir nicht richtig durchblicken. Das können komplizierte technische Konstruktionen wie zum Beispiel Software sein – oder andere Menschen. Entscheidende Fragen sind hier: Welches Vertrauen ist angemessen und was sind gute Kriterien für Vertrauen oder Misstrauen?

Vertrauen entsteht (unter anderem), wenn die Dinge so sind, wie man sie erwartet. Für wirklich lebenswichtige Dinge haben Menschen Verfahren entwickelt, mit denen Sie sich gegen eine zu große Differenz zwischen Erwartetem und Erlebtem absichern. Gute Beispiele sind hier Bauvorschriften oder Sicherheitsvorschriften für elektrische Geräte.

Gegenstände aus der physikalischen Alltagswelt sind oft dazu geeignet, stichprobenartig geprüft zu werden (statistische Qualitätssicherung) – weil ihre Qualität eine gewisse Stetigkeit besitzt. Wenn die Armierungsstähle für Betonbauten einer Fertigungsreihe entstammen, dann muss nicht jeder vor dem Einbau geprüft werden. Außerdem wird mit Toleranzen und Redundanzen gearbeitet.

Wie ist das jetzt mit Software? Aus der häufigen, (einigermaßen) erfolgreichen Verwendung von Software-Stückchen wie Editoren, Java-Compilern oder Betriebssystemteilen entstehen Vertrauensbeziehungen. Mehr oder minder tiefe – wer hat nicht schon mal am Compiler gezweifelt und sich bei der Aussage der erfahrenen Kollegen »es liegt nie am Compiler« unwohl gefühlt. Hat man sich so getäuscht? Manchmal liegt es ja doch am Compiler ... aber halt sehr, sehr selten.

Wenn Sie genau darüber nachdenken – wie vielen Millionen oder Milliarden Zeilen Code vertrauen Sie? Code, der in der Hardware versteckt ist, in Betriebssystemen, in der Anzeige-Hardware ... und wie halten Sie es mit der selbst entwickelten Software?

Als Entwickler haben wir oft unsere eigenen Vorstellungen über unsere starken und unsere schwachen Punkte. Wir ahnen, dass manche Codestellen nicht so sauber sind, womöglich nicht ganz verstanden wurden, und von anderen Stellen sind wir 100% überzeugt.

Was wissen wir wirklich? Können wir unserer Software vertrauen? Wer vertraut unserer Software? Wem vertraut unsere Software, weil sie unvermeidlich auf den unüberschaubar vielen Abstraktionsschichten aufsetzt?

Tests helfen uns, sinnvolle Vertrauensbeziehungen zu entwickeln.

Dieses Vertrauen steht in direktem Zusammenhang mit der Qualität der Tests. Wenn die Tests wiederholbar sind und nachweisen, dass die angenommenen Eigenschaften auch nach Veränderungen noch vorhanden sind, steigert das unser Vertrauen – und das Vertrauen der Nutzer in unsere Software.

Von Brad Cox (Vater von Objective-C) stammt die Idee, einen Reifegrad *(Maturity Index)* für Softwarebibliotheken zu definieren. Dieser Reifegrad beschreibt, wie intensiv ein Stückchen Software getestet wurde und wie vertrauenswürdig es ist. Die Grundidee geht davon aus, dass eine Bibliothek umso vertrauenswürdiger ist, je mehr Nutzer sie hat. Mit einer Testsuite gibt es mindestens einen Benutzer für unsere Software – das erhöht hoffentlich das Vertrauen anderer Nutzer.

Wenn es um die Prüfung der Vertrauenswürdigkeit geht, ist Software übrigens wesentlich geduldiger als Menschen. Man darf ohne Murren immer wieder prüfen, ob die Software noch das kann, was man erwartet (und was in den Tests formuliert ist). Die Feststellung, dass die Tests noch nicht genügend Vertrauen geben, ist auch kein Problem – man formuliert einfach noch weitere. So betrachtet ist die Coverage-Analyse ein Verfahren, mit dem sich Aussagen über die Vertrauenswürdigkeit von Tests machen lassen.

Zwei Seelen in einer Brust – irgendwie ist das eine interessante Umschreibung für die Testgetriebene Entwicklung. Beide Seelen brauchen, schaffen und ringen um Vertrauen. Die Entwicklerseele muss das Vertrauen haben, dass ihr das Richtige einfällt, um die Anforderungen (in Form von Tests) zu realisieren. Weiterhin braucht sie das Vertrauen, entstandene Smells zu erkennen und immer wieder Refactorings durchzuführen, mit denen die Software sich verbessert. Die Testerseele formuliert ihre Erwartungen als Tests und prüft ständig, ob ihr Vertrauen noch gerechtfertigt ist.

Das mit den zwei Seelen klingt ein bisschen unheimlich – aber wem vertrauen wir mehr? Den Entwicklern mit der unerschütterlichen Überzeugung »das läuft schon, Tests kosten nur Zeit ...« oder den Entwicklern mit den zwei Seelen?

8.5 Austauschbarkeit von Objekten

Übertragen wir die Ideen der vorangangenen Seiten auf ein Beispiel: Wir erhalten die Anforderung, die Kundenbelege auf einem Drucker auszugeben. Werfen wir zur Erinnerung einen Blick auf einen unserer betreffenden Tests:

```
public class CustomerTest...
  protected void setUp()...
    buffalo66 = new Movie("Buffalo 66", Price.NEWRELEASE);
  }

  public void testStatementDetailForRentalDetails() {
    customer.rentMovie(buffalo66, 1);

    String statement = customer.printStatementDetail();
    assertEquals("\tBuffalo 66\t2,00\n", statement);
  }
}
```

Ein Review der zentralen Methode:

```
public class Customer...
  public String printStatementDetail() {
    String result = "";
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      result += "\t" + rental.getMovieTitle()
              + "\t" + rental.getCharge().format() + "\n";
    }
    return result;
  }
}
```

Bevor wir die Fähigkeit zum Drucken aber in unser System integrieren, bringen wir den Code in eine Form, die uns ein unabhängigeres Testen der interessanten Funktionalität erlauben wird. Wir haben im letzten Kapitel schon diskutiert, dass Änderungen der `Price`-Konstanten die Tests abhängiger Klassen äußerst fragil machen. Aus diesem Grund werden wir unser `Price`-Objekt für den Test austauschen. Hier ist das einfach möglich, da die `Movie`-Klasse damit bereits parameterisiert ist. Später werden wir uns dann hartnäckigere Fälle vorknöpfen.

8.6 Stub-Objekte

Mitwirkende Objekte, die selbst nicht direkt getestet werden sollen und von denen wir unter Umständen auch nicht wissen, ob sie wirklich funktionieren, werden zur Isolation im Test durch Imitationen ersetzt. Eine sehr einfache solche Attrappe ist ein Stub-Objekt, vom Kollegen Johannes Link auch auf den Namen »Dummy« getauft.

... auch Dummy genannt

Stubs liefern für alle Funktionen, die während des Tests simuliert werden sollen, zweckmäßige Resultate zurück. Ihre Implementierung ist überaus einfach: Sie enthält keine Logik und macht keine Arbeit. Häufig antworten Stubs ganz primitiv nur mit konstanten Werten, manchmal lassen sie sich jedoch auch minimal konfigurieren.

In unserem Fall ersetzen wir das `Price`-Objekt im Test durch ein `StubPrice`-Objekt. Wir können dem Original hier durchaus vertrauen, die Klasse ist ausgiebig getestet, doch wir wollen nicht länger die Tests der abhängigen Klassen ändern, wann immer sich ein Preis ändert. Durch den Stub wird unser Test an dieser Stelle unabhängig:

```
public class CustomerTest...
  protected void setUp()...
    buffalo66 = new Movie("Buffalo 66", new StubPrice());
  }
}
```

Wie Sie sehen, mogeln wir dem `Movie`-Objekt den `StubPrice` unter. Damit diese Schummelei nicht auffliegt, muss unser Stub die Schnittstelle der zu ersetzenden Klasse implementieren. Damit das passt, extrahieren wir also ein Interface für den Typ `IPrice` und leiten dann im Code alle Verwendungs- und Implementierungsstellen darauf um. Ihre Entwicklungsumgebung sollte dieses Refactoring beherrschen. Halten Sie Ausschau nach dem Menüpunkt »Extract Interface«:

Interface extrahieren

```
public interface IPrice {
  public Euro getCharge(int daysRented);
}
```

Unser `Movie`-Objekt merkt überhaupt nicht, welches Objekt es hinter diesem Interface konkret verwendet, solange das `StubPrice`-Objekt nur plausible Ergebnisse liefert:

```
class StubPrice implements IPrice {
  public Euro getCharge(int daysRented) {
    return new Euro(2.00);
  }
}
```

8.7 Größere Unabhängigkeit

Ein Stub-Objekt entkoppelt die getestete Implementierung von einem mitwirkenden Objekt und ermöglicht so einen unabhängigeren Test. Mithilfe von Stubs sind wir also in der Lage, die Unit im Unit Test exakt einzugrenzen und die umgebenden Objekte frei zu kontrollieren. Ihr Anwendungsgebiet liegt deshalb vor allem an den Systemgrenzen und Schnittstellen innerhalb des Systems.

Wenn es viel Zeit, viele Nerven oder viel Geld kostet, die reale Umgebung für den Test zur Verfügung zu stellen, ersetzen Sie sie für Testzwecke mit einem Stub.

Durch Stubs zu ersetzen

Wann sollten Sie den Einsatz von Stubs erwägen? Stubs ersetzen

- **Komponenten mit Seiteneffekten**, wenn die Wiederholbarkeit und damit die Automatisierung unserer Tests auf dem Spiel steht;
- **Objekte in einem speziellen Zustand**, wenn die wirklichen Objekte nicht in den notwendigen Zustand gebracht werden können;
- **Objekte im Fehler- und Ausnahmezustand**, wenn das Fehlverhalten in den mitarbeitenden Objekten nur schwer zu generieren ist;
- **trödelnde Dienstobjekte**, wenn diese den Test unnötig verzögern;
- **Klassen mit vielen eigenen Beziehungen**, wenn bei einem einzelnen Test die Masse von Abhängigkeiten außer Kontrolle gerät;
- **Komponenten mit hohen Änderungsraten**, wenn unzählige von indirekten Tests von nervtötenden Anpassungen befallen werden;
- **noch nicht vorhandene Klassen**, wenn die Entwicklung ansonsten nicht fortschreiten kann.

Keine Logik in Stubs

Stub-Objekte sollten sehr einfach zu schreiben oder zu generieren sein. Wir wollen auf keinen Fall die Originalklasse auch nur in Ansätzen nachprogrammieren. Stubs dürfen keinerlei Logik enthalten, sonst müssen wir sie wiederum testen. Ihre Implementierung besteht deshalb häufig nur aus einem hart kodierten Rückgabewert. Manchmal ist das Ergebnis jedoch von den aktuellen Argumenten abhängig, so dass Stubs auch schon mal intelligenter ausfallen können.

Konkrete Beispiele zum Einsatz von Stubs sind Zugriffsschichten zur Datenbank oder Schnittstellen zur Hardware. In beiden Fällen fällt es äußerst schwer, eine wirkliche Testumgebung aufzubauen. Selbst in jenen Fällen, wo es trotzdem gelingt, werden Ausnahmezustände im Test oft stiefmütterlich behandelt. Kein Wunder also, dass Fehler in Fehlerbehandlungsroutinen manchmal über Jahre hinweg unentdeckt bleiben können.

8.8 Testen durch Indirektion

Schauen wir uns das Prinzip des isolierten Testens anhand der Stub-Verwendung noch einmal im Detail an. Wir reisen dazu zurück zu einem Stand aus Kapitel 5.12, *Die letzte Durchsicht*: dem Zeitpunkt, als unsere Movie-Klasse noch direkt an eine bestimmte Price-Instanz gebunden und das IPrice-Interface noch nicht eingezogen war:

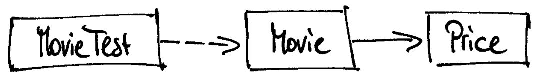


Abb. 8–5
Noch hart verdrahtet

Wir erkennen, dass unser Test offenbar keine Kontrolle darüber hatte, mit welchem konkreten Preis die Movie-Klasse gerechnet hat:

```
public class MovieTest extends TestCase {
  public void testUsingNewReleasePrice() {
    assertEquals(new Euro(3.75), Movie.getCharge(3));
  }
}
```

Denn im Bauch des Movie-Objekts wurde damals noch unumstößlich, weil fest verdrahtet, die Price.NEWRELEASE-Instanz herangezogen:

```
public class Movie {
  public static Euro getCharge(int daysRented) {
    return Price.NEWRELEASE.getCharge(daysRented);
  }
}
```

Dass Objekte sich wie hier selbst ihre benötigten Objekte erzeugen, resultiert leider überaus häufig in totaler Untestbarkeit: Wie können wir die Movie-Klasse noch unabhängig von der Price-Klasse testen? Ausflüchte wie »Dieser Code kann *wirklich* nicht getestet werden« maskieren nur ein Designproblem.

Was dagegen immer hilft, ist eine weitere Indirektionsstufe:

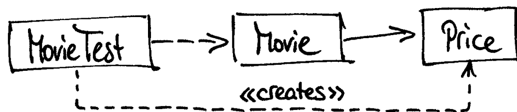
1. Öffnen Sie die zu testende Klasse.
2. Ziehen Sie eine Schnittstelle ein.
3. Ersetzen Sie die mitarbeitende Klasse im Test.

Das Prinzip des isolierten Testens:
Jedes Testproblem kann durch eine zusätzliche Indirektion gelöst werden.

Schritt 1: Öffnen Sie die zu testende Klasse

Der Mechanismus, um direkte Abhängigkeiten zu brechen, ist immer der gleiche. Es beginnt damit, dass wir verschlossene Klassen öffnen. Anstatt dass ein Objekt die anderen Objekte, von denen es abhängt, selbst erzeugt, werden ihm diese Objekte von außen hineingereicht:

Abb. 8–6
Gelockert



Für unser Beispiel hat das Öffnen der Schnittstelle bedeutet, dass wir unserer `Movie`-Klasse einen Konstruktor spendiert und diesen mit der zu verwendenden `Price`-Instanz ausstaffiert haben:

```
public class MovieTest...
  public void testUsingNewReleasePrice() {
    Movie movie = new Movie(Price.NEWRELEASE);
    assertEquals(new Euro(3.75), movie.getCharge(3));
  }
}

public class Movie...
  private Price price;

  public Movie(Price price) {
    this.price = price;
  }

  public static Euro getCharge(int daysRented) {
    return price.getCharge(daysRented);
  }
}
```

Dieses Refactoring trägt die Entscheidung nach außen, an welches Objekt hinter den Kulissen delegiert wird. Das hat den interessanten Nebeneffekt, dass unsere Designs viel stärker durch Komposition erweiterbar sind als durch den Vererbungsmechanismus allein.

Reichen Sie Ihre Objekte in den Code hinein, der sie benötigt, damit Ihre Objekte via Komposition und Delegation arbeiten.

Schritt 2: Ziehen Sie eine Schnittstelle ein

Als Nächstes eliminieren wir die direkte Abhängigkeit zwischen den Klassen. Wir führen dazu eine Schnittstelle als Indirektionsstufe ein. Anstatt dass eine Klasse die konkreten Klassen, mit denen sie arbeitet, direkt kennt, wird sie nur noch an deren Typ gebunden:

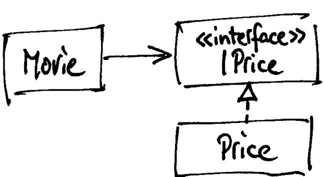


Abb. 8–7
Entkoppelt

Implementierungsdetails hinter einfachen Schnittstellen zu verbergen und alle unsere Abhängigkeiten auf diese Schnittstellen zu verlagern, folgt der trefflichen Idee des *Dependency-Inversion*-Prinzips [ma_{02}].

```
public interface IPrice {
  public Euro getCharge(int daysRented);
}
```

Das `IPrice`-Interface definiert so ein Protokoll, das viele verschiedene Klassen implementieren können, unter anderen unsere `Price`-Klasse:

```
public Price implements IPrice...
```

Unsere `Movie`-Klasse muss somit nicht mehr von einer konkreten Klasse abhängen. Sie bezieht sich jetzt nur noch auf den Typ: `IPrice`.

```
public class Movie...
  private IPrice price;

  public Movie(IPrice price) {
    this.price = price;
  }
}
```

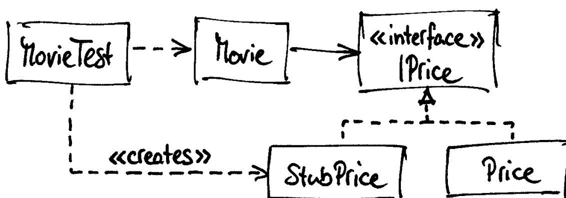
Da jetzt sowohl `Movie` als auch `Price` nur noch von der gewonnenen `IPrice`-Schnittstelle abhängen, existiert keinerlei Abhängigkeit mehr zwischen ihnen.

Abhängigkeiten durch Schnittstellen explizit zu machen und von außen zu konfigurieren, welche Instanzen konkret zugewiesen werden, entspricht dem *Dependency-Injection*-Muster [fo_{04}].

Schritt 3: Ersetzen Sie die mitarbeitende Klasse im Test

Zum Schluss testen wir die Klasse in Isolation. Dazu stellen wir eine Stub-Implementierung für die vorher extrahierte Schnittstelle bereit. Anstatt dass eine Klasse mit den Klassen getestet wird, mit denen sie gewöhnlich kooperiert, wird sie von diesen unabhängig getestet:

Abb. 8–8
Isoliert



Durch die zusätzliche Indirektion haben wir erreicht, dass unsere `Movie`-Klasse die konkreten `Price`-Klassen durch ein schmales Interface polymorph behandelt. Dadurch sind wir schließlich in der Lage, die `Price`-Klasse im Test gegen eine `StubPrice`-Attrappe auszutauschen:

```
public class MovieTest...
  public void testUsingStubPrice() {
    Movie movie = new Movie(new StubPrice());
    assertEquals(new Euro(2.00), movie.getCharge(3));
  }
}
```

Unser Test gewinnt auf diese Weise die Kontrolle darüber, mit welchem Preis gerechnet wird. Der Stub muss dazu lediglich einen normalerweise im `Price`-Objekt berechneten Preis gegen eine Konstante aus der Konserve austauschen:

```
class StubPrice implements Price {
  public Euro getCharge(int daysRented) {
    return new Euro(2.00);
  }
}
```

Eine alternative Stub-Implementierung hätte hier so aussehen können, dass im Konstruktor der gewünschte Resultatwert für den Stub gesetzt wird, doch diese Variabilität wird jetzt nicht benötigt.

8.9 Stub-Variationen

Stubs von konkreten Klassen

Es gibt Fälle, in denen es uns nicht möglich ist, eine Schnittstelle zur zusätzlichen Indirektion einzuführen. Denken wir nur an Programme, deren Quellcode wir nicht ändern können, weil wir ihn nicht haben, oder nicht ändern dürfen, weil die Schnittstellen an viele Verwender publiziert wurden. In diesen Fällen können wir dennoch einen Stub implementieren, indem wir die betreffende konkrete Klasse erweitern:

```
class StubPrice extends Price {
  public Euro getCharge(int daysRented) {
    return new Euro(2.00);
  }
}
```

Eine große Gefahr dabei ist, dass irgendwo im Test irrtümlich doch das Verhalten der Oberklasse verwendet wird. Das kann schon passieren, wenn die Oberklasse ohne Kenntnis der Unterklassen geändert wird, und führt hinterher zu irreleitenden Fehlern. Deshalb hier der Tipp, alle Methoden der Originalklasse in der Stub-Klasse zu überschreiben und in allen nicht ausgestopften Methoden stets eine `RuntimeException` zu werfen, damit solche Trugschlüsse leichter aufgedeckt werden.

Anonyme und innere Stubs

Aus meiner Sicht sollten Klassen nicht in anderen versteckt werden, sondern den großen Namensraum beanspruchen, weil sie so einfacher von Entwicklern gefunden werden. Der Vollständigkeit halber möchte ich jedoch erwähnen, dass unter Verdunklungsgefahr die Möglichkeit besteht, Stubs innerhalb der Testfallklasse oder -methode selbst zu definieren, abhängig davon, welche Sichtbarkeit erforderlich ist:

```
public class MovieTest...
  public void testUsingStubPrice() {
    IPrice stubPrice = new IPrice() {
      public Euro getCharge(int daysRented) {
        return new Euro(2.00);
      }
    };
    Movie movie = new Movie(stubPrice);
    assertEquals(new Euro(2.00), movie.getCharge(3));
  }
}
```

8.10 Testen von Mittelsmännern

Setzen wir unseren Weg fort, um den Ausdruck der Kundenbelege zu realisieren. Wir haben gesehen, dass unsere Klassen zum isolierten Test offen sein müssen. Springen wir also einmal durch die drei vorher beschriebenen Schritte zum Testen durch Indirektion:

```
public class CustomerTest...
  private IPrinter stubPrinter;

  protected void setUp()...
    buffalo66 = new Movie("Buffalo 66", new StubPrice());
    stubPrinter = new StubPrinter();
  }

  public void testStatementDetailForRentalDetails() {
    customer.rentMovie(buffalo66, 1);

    customer.printStatementDetail(stubPrinter);
    assertEquals("\tBuffalo 66\t2,00\n", ??? );
  }
}
```

Was haben wir unternommen?

- Die `printStatementDetail`-Methode parameterisiert
- Zur Druckersteuerung das `IPrinter`-Interface definiert
- Den Drucker durch die `StubPrinter`-Attrappe ersetzt

Der Drucker erfüllt gleich mehrere unserer Kriterien aus Kapitel 8.7, *Größere Unabhängigkeit*, um einen idealen Kandidaten für einen Stub abzugeben: Da wir weder eine Klasse zur Ansteuerung der wirklichen Hardware haben noch mit dem wirklichen Drucker testen könnten, sind wir auf dem richtigen Weg.

Dennoch stellt uns der neue Test vor ein Problem: Dadurch, dass die `printStatementDetail`-Methode keinen Resultatwert mehr liefert, sondern seine Ausgabe stattdessen direkt auf den Drucker umleitet, auf den wir keinen Zugriff haben, geht uns die Möglichkeit verloren, Zusicherungen für unsere Mittelsmannklasse zu schreiben.

Loggen statt Mocken

»Programming by Intention« könnte suggerieren, unserem Stub eine Methode zu verpassen, um nachzufragen, was er gedruckt hat. Die Idee hätte durchaus etwas für sich, hält jedoch den Nachteil bereit, erst relativ spät fehlzuschlagen, wie wir noch sehen werden:

```
assertEquals("\tBuffalo 66\t2,00\n", stubPrinter.getOutput());
```

8.11 Self-Shunt

Hin und wieder kommen wir in die Situation, Code testen zu müssen, der selbst keinerlei Resultate liefert. Stattdessen hat dieser Code meist Seiteneffekte, die ausdrücklich erwünscht sind und durch die der Code wiederum testbar wird. Beispiele dafür sind unter anderem:

- Methoden, die lediglich den Zustand des Zielobjekts ändern
- Methoden, die nur auf anderen Objekten arbeiten
- Methoden, die schreibend auf periphere Systeme zugreifen

Das beschriebene Problem tritt auf, sobald das getestete Objekt nicht dem Testfall antwortet, sondern stattdessen mit anderen Objekten spricht. Da sich die Schallwellen also quasi in die entgegengesetzte Richtung von uns wegbewegen, benötigt unser Test ein Ohr am Ende der Leitung: um geschickt mithorchen zu können, was das getestete Objekt zu den mitarbeitenden Objekten sagt:

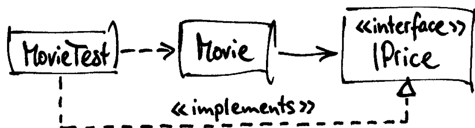


Abb. 8–9
Rückkopplung zum Test

Eine mustergültige Struktur ist als *Self-Shunt* [fe_{01}] bekannt. Erfinder Michael Feathers beschreibt es auf folgende Art: »Stellen Sie sich vor, Sie wären ein Testfall. Was Sie machen können, ist sich selbst in die Objekte hineinreichen, die Sie testen, um an mehr Informationen zu kommen.«

Testfallklasse implementiert benötigte Schnittstelle zur Rückkopplung

Beim Self-Shunt implementiert die Testfallklasse also persönlich die von der zu testenden Klasse benötigte Schnittstelle und nimmt damit den Platz der ersetzten Klasse ein, wie zuvor der Stub. Dadurch entsteht über die Schnittstelle eine gewollte Rückkopplung zu unserer Testfallklasse:

```
public class CustomerTest implements IPrinter...
```

Der Test des Aufrufs erfolgt in der Implementierung der Schnittstelle. Dazu müssen wir jedoch erst einmal ein einfaches Interface definieren, mit dessen Hilfe unser Code mit der Druckerhardware sprechen kann. Wir belassen es dazu beim einfachsten:

```
public interface IPrinter {
  public void print(String line);
}
```

Wir sind jetzt in der Lage, mit geringfügigen Änderungen den Test zu schreiben, den wir schon vor drei Minuten zu schreiben vorhatten:

```
public class CustomerTest implements IPrinter...
  private String printerOutput;

  protected void setUp()...
    buffalo66 = new Movie("Buffalo 66", new StubPrice());
    printerOutput = "";
  }

  public void print(String line) {
    printerOutput += line;
  }

  public void testStatementDetailForRentalDetails() {
    customer.rentMovie(buffalo66, 1);

    customer.printStatementDetail(this);
    assertEquals("\tBuffalo 66\t2,00\n", printerOutput);
  }
}
```

Wir haben entschieden, das `IPrinter`-Interface so zu implementieren, dass die Testfallklasse für die zu druckenden Zeilen ein Journal schreibt. So weit wären wir zwar auch noch mit unserer `StubPrinter`-Klasse mit `getOutput`-Funktion gekommen, doch damit ist gleich Schluss.

Der Trick ist, dass die `CustomerTest`-Klasse das `IPrinter`-Interface zum Callback implementiert. Während der Test die `Customer`-Klasse exerziert, ruft diese also kurzzeitig den Test auf der `print`-Funktion zurück und sendet ihm seine Resultate, die wir anschließend testen.

Memo:
– printStatementDetail(): Tests auf neue Signatur umstellen

Ich dupliziere hier einmal zur Abkürzung des Buchbeispiels unsere Methode, jedoch mit dem zusätzlichen Parameter. Der nötige Umbau ist damit ein Klacks, weil wir nicht sofort alle schon bestehenden Tests anpassen müssen:

```
public class Customer...
  public void printStatementDetail(IPrinter printer) {
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      printer.print("\t" + rental.getMovieTitle()
                + "\t" + rental.getCharge().format() + "\n");
    }
  }
}
```

8.12 Testen von innen

Bislang konnten wir nur das nach außen sichtbare Verhalten prüfen, und zwar unmittelbar nachdem der zu testende Code exerziert wurde. Zuweilen passiert es jedoch, dass die wirkliche Fehlerursache dabei verloren geht. Unser Test schlägt dann zwar fehl, doch an die Objekte, die uns preisgeben könnten, wo der Fehler tatsächlich unterlaufen ist, kommen wir anschließend von außen nicht mehr heran.

Dadurch, dass wir unser Testfallobjekt beim Self-Shunt ohnehin in das Zielobjekt einschleusen, erhalten wir die Möglichkeit, von innen zu testen, ob unser `Movie`-Objekt das `IPrinter`-Interface auch wirklich mit den erwarteten Werten verwendet. Das Testen von innen verändert dabei den Aufbau des Tests, da wir ihm unsere Erwartungen an den zu testenden Code schon mit auf die Reise ins Zielobjekt geben müssen:

```
import com.mockobjects.ExpectationList;

public class CustomerTest implements IPrinter...
  private ExpectationList printerOutput;

  protected void setUp()...
    printerOutput = new ExpectationList("printer output");
  }

  public void print(String line) {
    printerOutput.addActual(line);
  }

  public void testStatementDetailForRentalLines() {
    printerOutput.addExpected("\tBuffalo 66\t2,00\n");
    printerOutput.addExpected("\tDas Dschungelbuch\t1,50\n");

    customer.rentMovie(buffalo66, 4);
    customer.rentMovie(jungleBook, 1);

    customer.printStatementDetail(this);
    printerOutput.verify();
  }
}
```

Um unsere Erwartungen zu spezifizieren, verwenden wir die Klasse `ExpectationList` aus dem *mockobjects*-Projekt [mo]:

- `addExpected(Object expected)` setzt ein Objekt auf die Gästeliste.
- `addActual(Object actual)` prüft, ob das Objekt erwartet wurde.
- `verify` prüft, ob wirklich alle erwarteten Objekte eingetroffen sind.

8.13 Möglichst frühzeitiger Fehlschlag

Die Besonderheit des Testens von innen besteht darin, Fehler im Code schon erkennen zu können, während sich der Kontrollfluss noch im getesteten Objekt befindet. Das heißt, unser Test kann auf der Stelle fehlschlagen, sobald eine unserer gesetzten Erwartungen verletzt wird, und uns dadurch erheblich scharfsinnigere Fehlermeldungen anbieten.

Die zwei für uns interessanten Fälle sind:

- **Ein nicht erwartetes Ereignis trifft ein.**
- **Ein erwartetes Ereignis trifft nicht ein.**

Wenn unser Code zum Beispiel den Fehler enthält, dass keine Daten an die Druckerschnittstelle gesendet werden, erhalten wir diese Meldung:

```
junit.framework.AssertionFailedError:
  printer output did not receive the expected collection items
Expected: [\tBuffalo 66\t2,00\n, \tDas Dschungelbuch\t1,50\n]
Received: []
at CustomerTest.testPrintingStatementDetailWithOneMovie
  (CustomerTest.java:41)
```

Werden ihm die falschen Daten gesendet, beispielsweise in verdrehter Reihenfolge, ändert sich die Meldung zu:

```
junit.framework.AssertionFailedError:
  printer output added item does not match
Expected: [\tBuffalo 66\t2,00\n]
Received: [\tDas Dschungelbuch\t1,50\n]
at CustomerTest.testPrintingStatementDetailWithOneMovie
  (CustomerTest.java:33)
```

Und werden dem Drucker mehr Datensätze gesendet, als er erwartet:

```
junit.framework.AssertionFailedError:
  printer output had different sizes
Expected size: 2
Received size: 3 when adding: \tMemento\t2,00\n
at CustomerTest.testPrintingStatementDetailWithOneMovie
  (CustomerTest.java:33)
```

Sie können diese Fehlerszenarien leicht durchspielen, indem Sie den Code einfach probehalber an verschiedenen Stellen sabotieren. Achten Sie dabei einmal auf die unterschiedlichen Zeilennummern, die im Stacktrace anzeigen, wo der Test fehlschlägt: Der erste Fehler kann erst bei Aufruf der `verify`-Methode entdeckt werden, der zweite und dritte Fehler dagegen schon früher bei Aufruf der `addActual`-Methode.

8.14 Erwartungen entwickeln

Klassen, die Erwartungen kapseln und verifizieren, sind so nützlich, dass Sie zur Übung einmal eine Klasse für einfache Erwartungswerte entwickeln können. Es ist wichtig, dass Sie den Mechanismus der Erwartungen verinnerlichen. Wenn Sie gerade im Bus oder am Strand sitzen, studieren Sie die Tests bitte umso genauer und überlegen Sie, warum der Test der Fehlschläge anders gelöst ist als in Kapitel 3.14, *Testen von Exceptions*:

```
public class SimpleExpectationValueTest extends TestCase {
  private SimpleExpectationValue value;

  protected void setUp() {
    value = new SimpleExpectationValue("expectation value");
  }

  public void testExpectedValue() {
    value.setExpected("Night on Earth");
    value.setActual("Night on Earth");
    value.verify();
  }

  public void testWrongValue() {
    value.setExpected("Stranger than Paradise");
    try {
      value.setActual("Down by Law");
    } catch (AssertionFailedError expected) {
      return;
    }
    fail("Should have detected a failed expectation");
  }

  public void testNoValue() {
    value.setExpected("Dead Man");
    try {
      value.verify();
    } catch (AssertionFailedError expected) {
      return;
    }
    fail("Should have detected a missed expectation");
  }
}
```

Eine mögliche Realisierung sehen Sie, wenn Sie geschwind umblättern.

Den ersten Test erfüllen wir in einem größeren Schritt:

```
public class SimpleExpectationValue {
  private String name;
  private Object expected;
  private Object actual;

  public SimpleExpectationValue(String name) {
    this.name = name;
  }

  public void setExpected(Object expected) {
    this.expected = expected;
  }

  public void setActual(Object actual) {
  }

  public void verify() {
  }
}
```

Für den zweiten Test prüfen wir den tatsächlich aufgetretenen Wert:

```
public class SimpleExpectationValue...
  public void setActual(Object actual) {
    junit.framework.Assert.assertEquals(
      name + " did not receive the expected value\n",
      expected, actual);
  }
}
```

Der dritte Test prüft, ob der erwartete Wert wirklich aufgetreten ist:

```
public class SimpleExpectationValue...
  public void setActual(Object actual) {
    this.actual = actual;
    verify();
  }

  public void verify() {
    junit.framework.Assert.assertEquals(
      name + " did not receive the expected value\n",
      expected, actual);
  }
}
```

8.15 Gebrauchsfertige Erwartungsklassen

Eigentlich entwickeln wir Erwartungsklassen nicht ad hoc, sondern kommen auf natürlicherem Weg zu guten Erwartungen, indem wir sie als *refaktorisierte Zusicherungen* aus unserem Testcode extrahieren. Tim Mackinnon und Kollegen sind so vorgegangen und so war die Idee zur *Erwartungsklasse* geboren. Ihre Bibliothek von Erwartungen ist mit dem bereits erwähnten *mockobjects*-Projekt als Open Source verfügbar:

mockobjects-Projekt

```
http://www.mockobjects.com
```

Die Erwartungsklassen finden Sie dort im `com.mockobjects`-Package. Für einzelne Werte können daraus folgende Klassen helfen:

Erwartungen für einzelne Werte

- `ExpectationValue` verifiziert, dass ein Wert auftritt. Unterstützt werden die Typen `long`, `int`, `double`, `boolean` und `Object`.
- `ExpectationDoubleValue` verifiziert eine Fließkommazahl innerhalb eines Toleranzbereichs.
- `ExpectationSegment` verifiziert, dass eine Teilzeichenkette auftritt.
- `ExpectationCounter` verifiziert, dass eine bestimmte Anzahl von Aufrufen stattfindet, um zum Beispiel Methodenaufrufe zu zählen.

Zur Verifikation von Collections stehen diese Klassen zur Verfügung:

Für Collections

- `ExpectationList`: reihenfolgeabhängig, Duplikate sind möglich
- `ExpectationSet`: reihenfolgeunabhängig, Duplikate nicht möglich
- `ExpectationMap`: reihenfolgeunabhängig, Duplikate nicht möglich

An Stelle von Zusicherungen à la `assertEquals(expected, actual)` verwendet man das bereits beschriebene Muster:

```
public void setExpected(...) {...}
public void setActual(...) {...}
public void verify() {...}
```

Für Collections heißt es dementsprechend `addExpected` und `addActual`, wobei die Methoden in ihren beiden Varianten `addExpectedMany` und `addActualMany` auch Collections akzeptieren.

Im Fall, dass keine Erwartungen zu haben die richtige Antwort ist, kann jede der Klassen mittels `setExpectNothing` so geschaltet werden, dass ein Zugriff auf die `setActual`/`addActual`-Methoden fehlschlägt.

Und falls ein Fehlschlag nicht unmittelbar und frühestmöglich, sondern erst in der `verify`-Phase erfolgen soll oder auch erfolgen darf, kann der Schalter `setFailOnVerify` betätigt werden.

8.16 Testen von Protokollen

Zurück zu unserer Druckerschnittstelle: Eigentlich haben die Formatanweisungen \t und \n überhaupt nichts in der Ausgabezeile zu suchen. Vielmehr sollten sie Teil des Druckerprotokolls sein:

```
public interface IPrinter {
  public void print(String line) throws OutOfPaperException;
  public void tab() throws OutOfPaperException;
  public void crlf() throws OutOfPaperException;
  public void cutPaper() throws OutOfPaperException;
}
```

Dazu gehört dann auch die Möglichkeit, den gedruckten Kassenbeleg von der Papierrolle zu schneiden. Ferner können alle vier Operationen auch eine OutOfPaperException werfen, sollte uns das Papier ausgehen. Die Herausforderung ist jetzt, die korrekte Interaktion der Customer-Klasse mit unserer IPrinter-Schnittstelle, also das *Kommunikationsprotokoll* zweier Klassen, sicherzustellen. Zu testen wäre dabei, ob die erwarteten Nachrichten tatsächlich in der erwarteten Reihenfolge mit den erwarteten Daten eintreffen. Wie testet man so etwas?

Kleine Schritte: Zunächst einmal bringen wir unserer Klasse das Druckerprotokoll bei. Ich gehe hier vom Code anstatt vom Test aus, damit Sie besser erkennen, wohin wir mit dem Test wollen. Dadurch, dass wir das Protokoll zweier Klassen testen, lässt sich ohnehin nicht vermeiden, auch einen Teil der Implementierung preiszugeben. Mit der Realisierung im Kopf ist der Test dann hoffentlich auch besser zu verstehen. Vorher jedoch hier die Anpassung unserer Anwendungslogik:

```
public class Customer...
  public void printStatementDetail(IPrinter printer)
  throws OutOfPaperException {
    for (Iterator i = rentals.iterator(); i.hasNext(); ) {
      Rental rental = (Rental) i.next();
      printer.tab();
      printer.print(rental.getMovieTitle());
      printer.tab();
      printer.print(rental.getCharge().format());
      printer.crlf();
    }
    printer.cutPaper();
  }
}
```

```
public class CustomerTest implements IPrinter...
  private ExpectationList printerOutput;
  private ExpectationCounter tabCalls;
  private ExpectationCounter crlfCalls;
  private ExpectationCounter cutPaperCalls;

  protected void setUp()...
    printerOutput = new ExpectationList("printer output");
    tabCalls = new ExpectationCounter("tab() calls");
    crlfCalls = new ExpectationCounter("crlf() calls");
    cutPaperCalls = new ExpectationCounter("cutPaper() calls");
  }

  public void testStatementDetailForRentalLines()
  throws Exception {
    printerOutput.addExpected("Buffalo 66");
    printerOutput.addExpected("2,00");
    printerOutput.addExpected("Das Dschungelbuch");
    printerOutput.addExpected("1,50");

    tabCalls.setExpected(4);
    crlfCalls.setExpected(2);
    cutPaperCalls.setExpected(1);

    customer.rentMovie(buffalo66, 4);
    customer.rentMovie(jungleBook, 1);

    customer.printStatementDetail(this);
    Verifier.verifyObject(this);
  }

  public void print(String output) throws OutOfPaperException {
    printerOutput.addActual(output);
  }

  public void tab() throws OutOfPaperException {
    tabCalls.inc();
  }

  public void crlf() throws OutOfPaperException {
    crlfCalls.inc();
  }

  public void cutPaper() throws OutOfPaperException {
    cutPaperCalls.inc();
  }
}
```

Zugegeben, die Testklasse ist nicht mehr ganz so einfach zu verstehen. Außerdem macht der Test viel zu viel, schließlich passt er ja kaum auf eine Buchseite. Vielleicht ein gutes Kriterium, um die Klasse zu teilen. Zunächst will ich aber erklären, was der Test eigentlich testet.

Als Erstes stechen die drei `ExpectationCounter` ins Auge. Mit ihrer Hilfe zählen wir die Aufrufe der `tab`-, `crlf`- und `cutPaper`-Methoden. Für die Ausgabe von zwei Ausleihen erwarten wir vier Tabulatoren, zwei Zeilenvorschübe und einen Schneidevorgang. In den betreffenden Methodenrümpfen der Druckerschnittstelle machen wir nichts weiter, als die jeweils zuständigen Zähler von den stattgefundenen Methodenaufrufen in Kenntnis zu setzen.

Weiter fällt auf, dass wir für die `print`-Methode vier Ausgaben erwarten: unsere zwei DVD-Titel und deren Beträge. Die Reihenfolge, in der die Ausgabewerte tatsächlich an den Drucker gesendet werden, muss wegen Verwendung einer `ExpectationList` exakt der erwarteten Reihe entsprechen, sonst kommt es zu einem Fehlschlag.

Noch etwas rätselhaft mag die letzte Zeile des Tests erscheinen:

```
Verifier.verifyObject(this);
```

ersetzt die Tipperei von:

```
printerOutput.verify();
tabCalls.verify();
crlfCalls.verify();
cutPaperCalls.verify();
```

Der `Verifier` aus dem `util`-Paket der mockobjects-Bibliothek nimmt unser Objekt per Reflection unter die Lupe und verifiziert automatisch alle unsere Erwartungen. Auf diese Weise vergisst man vielleicht nicht so leicht, die `verify`-Methoden wirklich aller `Expectation`-Objekte zu rufen.

Doch wie weit wird das Ausgabeprotokoll nun wirklich getestet? Wir sichern eine bestimmte Reihenfolge für unsere vier Ausgaben zu. Wir sichern keine explizite Reihenfolge für unsere Format- und Steuerbefehle zu, sondern zählen nur ihre Vorkommen. Generell gilt hier, dass unsere Tests umso empfindlicher gegenüber Änderungen an der Implementierung werden, je strikter wir die Erwartungen formulieren. Deshalb stellt das Testen von innen immer eine Gratwanderung dar, bei der wir gerade so viele Geheimnisse preisgeben wollen wie nötig. Zunächst tun wir aber etwas dagegen, dass unser Self-Shunt aus allen Nähten platzt, bevor wir den kompletten Reihenfolgecheck angehen.

8.17 Mock-Objekte

Endlich kommen wir zu den legendären Mock-Objekten [ma00]: Mocks sind Stubs mit eingebetteter Testfunktionalität. Mock-Objekte kapseln die Erwartungen, die wir im Self-Shunt noch in der Testklasse definiert haben, in einer eigenen Klasse. Diese Mock-Implementierung wird in eine zu testende Klasse eingeschleust, um deren Verhalten von innen zu testen. Mocks haben also genau wie Shunts die Eigenschaft, bei unerwarteter Verwendung frühzeitig fehlzuschlagen. Über die für Mocks typische `verify`-Methode lässt sich dann sicherstellen, dass alle Erwartungen erfüllt wurden.

Mocks besitzen zwei Gesichter: Erwartungen und Stub-Verhalten. Zunächst ein Blick auf die Erwartungen; in unserem Fall sind sie vollständig aus unserer vorhandenen Testklasse zu extrahieren:

```
class MockPrinter extends MockObject implements IPrinter...
  private ExpectationList printerOutput
    = new ExpectationList("printer output");
  private ExpectationCounter tabCalls
    = new ExpectationCounter("tab() calls");
  private ExpectationCounter crlfCalls
    = new ExpectationCounter("crlf() calls");
  private ExpectationCounter cutPaperCalls
    = new ExpectationCounter("cutPaper() calls");

  void addExpectedOutput(String output) {
    printerOutput.addExpected(output);
  }

  void setExpectedTabCalls(int numberOfCalls) {
    tabCalls.setExpected(numberOfCalls);
  }

  void setExpectedCrlfCalls(int numberOfCalls) {
    crlfCalls.setExpected(numberOfCalls);
  }

  void setExpectedCutPaperCalls(int numberOfCalls) {
    cutPaperCalls.setExpected(numberOfCalls);
  }

  public void verify() {
    Verifier.verifyObject(this);
  }
}
```

Auf der anderen Seite implementiert ein Mock-Objekt die Schnittstelle der zu imitierenden Implementierung. In den Stub-Methoden dieser Schnittstelle erfolgt der Test der vorher gesetzten Erwartungen:

```
class MockPrinter extends MockObject implements IPrinter...
  public void print(String output) throws OutOfPaperException {
    printerOutput.addActual(output);
  }

  public void tab() throws OutOfPaperException {
    tabCalls.inc();
  }

  public void crlf() throws OutOfPaperException {
    crlfCalls.inc();
  }

  public void cutPaper() throws OutOfPaperException {
    cutPaperCalls.inc();
  }
}
```

Die Statik beim Testen mit Mocks unterscheidet sich von der Verwendung von Stubs nicht (siehe Abb. 8–8). In der Dynamik ergeben sich bei der Verwendung von Mocks drei grundlegende Schritte:

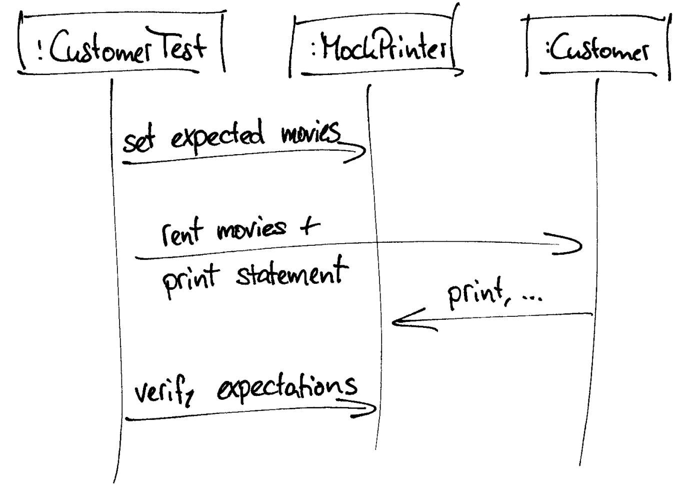


Abb. 8–10
Wie Mock-Objekte die Verhältnisse zwischen Testgegenstand und ersetzter Komponente bespitzeln

1. Spezifizieren Sie Verhalten und Erwartungen des Mocks.
2. Testen Sie den Prüfling mit dem Mock.
3. Verifizieren Sie die erwartete Verwendung.

Schritt 1: Spezifizieren Sie Verhalten und Erwartungen des Mocks

Gehen wir einmal in den drei Schritten durch unseren Test:

```
public class CustomerTest...
  private MockPrinter mockPrinter;

  protected void setUp()...
    mockPrinter = new MockPrinter();
  }

  public void testStatementDetailForRentalLines()
  throws Exception {
    mockPrinter.addExpectedOutput("Buffalo 66");
    mockPrinter.addExpectedOutput("2,00");
    mockPrinter.addExpectedOutput("Das Dschungelbuch");
    mockPrinter.addExpectedOutput("1,50");

    mockPrinter.setExpectedTabCalls(4);
    mockPrinter.setExpectedCrlfCalls(2);
    mockPrinter.setExpectedCutPaperCalls(1);
```

Wo es ursprünglich vor `Expectation`-Objekten nur so wimmelte, findet sich jetzt unser `MockPrinter`-Objekt wieder. Das Objekt kapselt seine Erwartungen hinter dem dafür typischen Bündel unterschiedlichster `setExpected`- und `addExpected`-Methoden, derer wir uns im Test auf die gewohnte Art und Weise bedienen.

Müssen wir im Mock noch das Verhalten des ersetzten Objekts emulieren, weil dieses Resultate an den Prüfling zurückliefern möchte, finden sich hier zusätzlich `setup`-Methoden für das Mock-Verhalten. Da unsere `IPrinter`-Schnittstelle nur `public void`-Methoden anbietet, schauen wir uns dies jedoch erst im nächsten Beispiel an.

Schritt 2: Testen Sie den Prüfling mit dem Mock

Haben wir alle Erwartungen gesetzt, können wir unseren Prüfling dem anvisierten Test unterziehen. Dazu müssen wir ihm das Mock-Objekt auf dem einen oder anderen Weg untermogeln und die zu testenden Methoden exerzieren. Unser Mock verhält sich nun im Zielobjekt angekommen genau so, wie vorher aus unserem Test befohlen:

```
    customer.rentMovie(buffalo66, 4);
    customer.rentMovie(jungleBook, 1);
    customer.printStatementDetail(mockPrinter);
```

Während des Tests kann das Mock-Objekt dann geschickt überprüfen, ob es vom Prüfling tatsächlich mit den von uns erwarteten Parametern verwendet wird. Ist das der Fall, gibt es dem Prüfling die befohlenen Antworten. Ergibt sich in der Interaktion stattdessen eine Diskrepanz, weil beispielsweise der `cutPaperCalls ExpectationCounter` öfter als erwartet inkrementiert wird, schlägt der Test auf der Stelle fehl und meldet die Abweichung.

Schritt 3: Verifizieren Sie die erwartete Verwendung

Ob tatsächlich alle Erwartungen erfüllt wurden, können wir erst zum Abschluss des Tests feststellen. Wird der `cutPaperCalls`-Zähler zum Beispiel überhaupt nicht inkrementiert, können wir den Umstand hier noch bemängeln und protokollieren:

```
        mockPrinter.verify();
    }
}
```

Zwei Bemerkungen zur `verify`-Methode: Nicht vergessen sollten Sie, das `verify` vor Ende des Tests auch wirklich auf allen Mocks zu rufen. Getrost vergessen können Sie dagegen, die `verify`-Methode auf allen Ihren Mocks zu definieren. Erweitert die Mock-Klasse nämlich die `MockObject`-Basisklasse, erbt sie dort eine Defaultimplementierung für das `verify`, die schon genau das erledigt, was wir noch mühselig in unsere eigene Methode getippt haben. Kann unsere Mock-Klasse aber nicht aus der `MockObject`-Klasse erben, weil wir keine Schnittstelle, sondern eine konkrete Klasse mocken wollen, müssen wir das `verify` in unserer Unterklasse dagegen selbst definieren.

8.18 Wann verwende ich welches Testmuster?

Die Antwort auf diese Frage sollte eigentlich keine Überraschung sein: Einfach starten und mit den Anforderungen wachsen:

- **Stubs** bieten die minimalste Fake-Implementierung einer Klasse oder Schnittstelle. Ich verwende sie immer, wenn weder der Aufruf an sich, noch seine Aktualparameter für den Test interessant sind.
- **Self-Shunt** ist sehr elegant, solange die implementierte Schnittstelle schmal bleibt. Bei mir kommen sie zum Einsatz, wenn der Aufruf, das heißt eine Interaktion zwischen Objekten, geprüft werden soll.
- **Mocks** sind die unbestreitbaren Könige unter den Hochstaplern. Ich benötige sie, wenn sie duplizierten Testcode vermeiden helfen oder etwas mehr als nur triviales Verhalten vorzugaukeln ist.

Wo Mock-Objekte herkommen

von Tim Mackinnon & Ivan Moore, ThoughtWorks
& Steve Freeman, M3P

Die Idee zu Mock-Objekten kann zurückverfolgt werden zu einem Treffen der London Extreme Group im September 1999. Bei diesem Treffen diskutierten Tim Mackinnon, Peter Marks, Ivan Moore und John Nolan übers Testen. Alle hatten wir Software mit den XP- und Test-First-Techniken entwickelt und versuchten, die Qualität unserer geschriebenen Tests zu verbessern. In diesen frühen Tagen machte uns Sorge, dass viele der Tests Extramethoden erforderlich machten, also die Kapselung brachen, damit wir die Objekte testen konnten, oder dass unsere Tests manchmal schwer zu schreiben waren wegen des Zwiebelproblems (bei dem man sich, um zum Kern vorzudringen, durch Schicht um Schicht kämpfen muss). Wir suchten damals nach Techniken, um spezielle »only-for-testing«-Methoden zu vermeiden und zum Testen in Isolation.

Anfangs fragten wir uns: »Ist es so schlimm, Methoden hinzuzufügen, damit etwas einfach getestet werden kann?« Wir stellten fest, dass wir Datenkapselung gegen Bequemlichkeit tauschten und dass wir den Blick auf interne Zustände freilegten. Ausgehend von dieser Diskussion erforschten wir den entgegengesetzten Ansatz »Isolierung durch Schnittstellen und Dummy-Implementierungen«. Was wäre, wenn wir »for-testing«-Methoden vermieden und in einem mehr kompositorischen Stil programmierten? Diese Idee begeisterte uns.

Am nächsten Tag setzen wir (Tim, Peter und John) uns im Büro (bei Connextra) zusammen und schrieben Teile unseres Codes so um, dass wir keine »just-for-testing«-Methoden mehr benötigten und stattdessen Dummy-Implementierungen herumreichten. Wir nannten diese Dummies »Mock-Objekte«, um sie per Namenskonvention zu erkennen.

Wichtig zu verstehen ist, dass das Mock-Konzept inkrementell entstanden ist: angefangen bei einfachen Stubs und weiterentwickelt zu dem Mock-Objekt-Ansatz, wie er heute dokumentiert ist. Mit der Zeit bemerkten wir ein wiederkehrendes Muster in unseren Tests: Zum Ende unserer Tests hatten wir eine Reihe von Zusicherungen, die die relevanten Werte holten und prüften, ob sie korrekt waren. Häufig waren diese Zusicherungen von Test zu Test dupliziert und so entschieden wir, sie an einem Ort zusammenzufassen. Obwohl es möglich war, sie zu einer Hilfsmethode der Testklasse zu machen, fanden wir es klarer, sie ins Mock-Objekt selbst zu stecken: Unsere `verify`-Methode war geboren. An diesem Punkt begannen unsere Mocks, sich von traditionellen Stubs zu unterscheiden.

Durch fortgesetztes Testen entdeckten wir weitere Muster: Tests, die Interaktionen zwischen Objekten zählten oder Reihen von als Methodenparameter erwarteten Werten erzeugten. In beiden Fällen war unsere Erstimplementierung einfach gehalten. Um Interaktionen zu zählen, setzten wir einen Zähler auf die Zahl erwarteter Aufrufe, dekrementierten ihn mit jedem Aufruf und warfen schließlich eine `AssertionFailedException`, sobald unser Zähler ins Negative lief. Ziemlich schnell fiel uns dabei jedoch auf, dass die Qualität dieser Fehlermeldungen nicht ideal war, da es sehr hilfreich war, zu wissen, was denn die ursprüngliche Erwartung gewesen war: Die Tatsache, fünf Aufrufe erwartet und einen sechsten erhalten zu haben, gab uns nützliche Information, um den Test schneller in Ordnung zu bringen (statt einem Fehler, der nur sagt: »Hab zu viele Aufrufe erhalten«). Wir refaktorisierten unseren Zähler zu einem Objekt mit dem Namen `ExpectationCounter` und erhielten auf ähnliche Weise eine `ExpectationList` und ein `ExpectationSet` (mit der Fähigkeit, die Liste erwarteter und tatsächlich erhaltener Elemente anzuzeigen). Auf diesem Weg kamen wir also zu einheitlichen und nützlichen Fehlermeldungen. (Später entdeckten wir dann auch die Notwendigkeit für einen `ExpectationValue`.)

Während sich diese Konzepte in der Codebasis von Connextra entwickelten, berichteten wir der XPDeveloper-Community in unseren wöchentlichen Treffen des Extreme Tuesday Clubs häufig davon, was wir herausgefunden hatten. Dort überzeugte uns Steve Freeman, dass unser Entwicklungsstil mehr war als eine triviale Lösung unseres Testproblems. In dieser Hinsicht ermunterte und half er uns, unsere Arbeit zu formalisieren und so konsistent zu machen, um sie auf der XP2000-Konferenz auf Sardinien vorzustellen. Anschließend an die Konferenz nahmen wir viele Ideen aus dem Connextra-Code und machten daraus das Open-Source-Projekt *mockobjects*.

Mock-Objekte sind immer noch in Bewegung. Mit zunehmender Erfahrung wächst unser Verständnis, wie Erwartungen zu definieren sind. Wir haben gelernt, unsere Absicht klarer darzulegen, indem wir unsere Zusicherungen abschwächen durch den Gebrauch von Wildcards anstatt einfacher Gleichheitsprüfung. Wir haben auch unsere Werkzeugkiste erweitert: Zum einen haben wir MockMaker, der die Erzeugung von Mock-Implementierungen durch die Generierung von Klassen erleichtert. Zum anderen haben wir JMock entwickelt, eine Bibliothek zur dynamischen Implementierung von Mock-Objekten. JMock verbessert den Prozess der Testgetriebenen Programmierung, indem es verdeutlicht, wie Objekte miteinander interagieren.

8.19 Crashtest-Dummies

Simulation von Ausnahmen und Fehlern

Ausnahme- und Fehlerverhalten ist häufig überaus schwer testbar. Ein Grund dafür ist, dass wir zum Test Ausnahme- und Fehlerzustände simulieren müssen, die wir in der tatsächlichen Soft- oder Hardware nur mit sehr viel Aufwand oder gar unmöglich produzieren können. Mit Mocks stellt das jedoch weniger ein Problem dar, müssen wir doch nur auf Zuruf das gewünschte Verhalten generieren können:

```
public class CustomerTest...
  public void testStatementDetailThrowingPrinterException() {
    mockPrinter.setupThrowException(new OutOfPaperException());

    try {
      customer.rentMovie(pulpFiction, 4);
      customer.printStatementDetail(mockPrinter);
      fail("should have raised a PrinterException");
    } catch (PrinterException expected) {
    }
  }
}
```

Hier sehen wir einmal, wie wir das Mock-Verhalten näher steuern können: Wir geben dem `MockPrinter` eine `OutOfPaperException` mit auf die Reise ins `Customer`-Objekt und lassen dort die Bombe hochgehen. Wird nämlich die `print`- oder eine andere Methode der `IPrinter`-Schnittstelle beansprucht, wird diese Exception zurückgeworfen:

```
class MockPrinter...
  private OutOfPaperException exception;

  public void setupThrowException
              (OutOfPaperException exception) {
    this.exception = exception;
  }

  public void print(String output) throws OutOfPaperException {
    if (exception != null) throw exception;
    printerOutput.addActual(output);
  }
}
```

Auf diesem Weg lässt sich testen, ob Exceptions wie hier in Exceptions eines anderen Typs übersetzt werden, Ressourcen auch im Fehlerfall noch korrekt freigegeben werden oder was eigentlich passiert, wenn uns die Datenbankverbindung abreißen sollte.

8.20 Dynamische Mocks mit EasyMock

Nun machen uns Mocks und Konsorten aber auch zusätzliche Arbeit, wollen sie ja fürs Erste programmiert und am Ende gepflegt werden. Schon getan ist diese Arbeit für die Teile der Java-Standardbibliothek, die relativ häufig gemockt werden müssen: Im mockobjects-Projekt sind für solche Schnittstellenpakete wie `java.io`, `javax.servlet.http` und `java.sql` weitestgehend vollständige Mock-Implementierungen zusammengetragen worden. Für unsere eigenen Klassen kann uns die Arbeit jedoch niemand abnehmen. Außer vielleicht ein Werkzeug ... und genau hier kommt eine weitere Bibliothek ins Spiel:

```
http://www.easymock.org
```

EasyMock

Mit EasyMock hat Tammo Freese die clevere Idee umgesetzt, Mocks direkt im Testcode zu spezifizieren und hinter den Kulissen mithilfe dynamischer Proxies zu generieren. Dieser seit JDK 1.3.1 verfügbare Mechanismus ermöglicht eine Implementierung von Interfaces noch zur Laufzeit. EasyMock bedient sich dabei geschickt der Möglichkeit, die erwartete Interaktion mit dem Mock-Objekt programmatisch aufzuzeichnen wie ein Videorecorder. Dies funktioniert, indem wir das EasyMock-Objekt im Test bereits so verwenden, wie wir es später von unserem Prüfling verwendet sehen wollen. Ist dies getan, schalten wir das Mock-Objekt scharf, EasyMock geht in die Wiedergabe über und spielt das vorher aufgezeichnete Verhalten ab und stellt dabei unsere Erwartungen sicher. Führen wir den Test des `MockPrinter`-Protokolls noch einmal mit EasyMock 1.2 (Version 2.0 benötigt Java 1.5) durch:

Recorder-Metapher: Aufzeichnung und Wiedergabe

```
import org.easymock.MockControl;

public class CustomerTest...
  private MockControl control;
  private IPrinter mockPrinter;

  protected void setUp()...
    control = MockControl.createStrictControl(IPrinter.class);
    mockPrinter = (IPrinter) control.getMock();
  }
```

Dynamische Proxies können das zu implementierende Interface nicht um zusätzliche Methoden erweitern. Deshalb zerfällt ein EasyMock-Objekt stets in zwei Teile: zum einen in das zu mockende Interface, `IPrinter`, zum anderen in eine Fernsteuerung für das Mock-Objekt namens `MockControl`, die alle mockspezifischen Methoden aufnimmt und uns unter anderem das EasyMock-Objekt in die Hand gibt.

Ein Glanzpunkt von EasyMock ist, wie Mocks mit Erwartungen gefüttert werden: Die erwarteten Methodenaufrufe und ihre Resultate, das gesamte Protokoll, zeichnen wir einfach auf, indem wir das Interface direkt ansprechen, so als wäre es die Implementierung in persona:

```
public void testStatementDetailForRentalLines()
throws Exception {
  mockPrinter.tab();
  mockPrinter.print("Buffalo 66");
  mockPrinter.tab();
  mockPrinter.print("2,00");
  mockPrinter.crlf();
  mockPrinter.tab();
  mockPrinter.print("Das Dschungelbuch");
  mockPrinter.tab();
  mockPrinter.print("1,50");
  mockPrinter.crlf();
  mockPrinter.cutPaper();
```

Anschließend schalten wir unser Mock-Objekt vom Aufnahme- in den Wiedergabebetrieb um. Ab dann verhält es sich so wie programmiert:

```
  control.replay();
```

Zum Abschluss exerzieren wir unseren Prüfling wie gewohnt und überprüfen, ob alle erwarteten Methodenaufrufe getätigt wurden:

```
  customer.rentMovie(buffalo66, 4);
  customer.rentMovie(jungleBook, 1);
  customer.printStatementDetail(mockPrinter);

  control.verify();
 }
}
```

Die Ersparnisse mit EasyMock sind riesig: Die Programmierarbeit für Mock-Objekte entfällt gänzlich. Die Kosten dafür: Der Testcode ist für Nichteingeweihte schlechter verständlich, da er wie Anwendungscode ausschaut.

Durch den Gebrauch dynamischer Proxies können nur Interfaces direkt gemockt werden. Wer auch Klassen mit Mocks ersetzen will, kann auf die *EasyMock Class Extension* zurückgreifen. Dass der Wunsch, Klassen direkt zu mocken, eine Designschwäche offenbart, muss ich nicht extra betonen; für jene Fälle, die wir sonst ungetestet lassen müssten, ist es trotzdem ein letzter Ausweg.

8.21 Stubs via Record/Replay

Nun ist es in vielen Fällen gar nicht notwendig und oft sogar höchst unerwünscht, die genaue Einhaltung eines Schnittstellenprotokolls im Test zu garantieren. Je strikter wir unsere Erwartungen formulieren, desto empfindlicher sind die Tests natürlich gegenüber Änderungen, ein Problem, dem wir uns gleich anschließend noch zuwenden werden. Deshalb bietet EasyMock über `MockControl` verschieden scharfsinnige Mocks an, für jeden Geschmack etwas:

Verschiedene Kontrollstufen

- `createStrictControl` prüft die exakte Reihenfolge von Aufrufen.
- `createControl` ist die Reihenfolge von Methodenaufrufen egal.
- `createNiceControl` meckert nicht einmal unerwartete Aufrufe an.

Insbesondere die beiden letztgenannten Stufen eignen sich prima dazu, um mit EasyMock auf die Schnelle einen Stub zu programmieren. Unser `StubPrice` aus Kapitel 8.6, *Stub-Objekte*, sähe dann beispielsweise so aus:

```
MockControl control = MockControl.createControl(IPrice.class);
IPrice stubPrice = (IPrice) control.getMock();
stubPrice.getCharge(3);
control.setReturnValue(new Euro(2.00));
```

Wer es ganz knapp mag, darf die letzten zwei Zeilen auch abkürzen:

```
control.expectAndReturn(stubPrice.getCharge(3), new Euro(2.00));
```

Soll der Aufruf zwischen zwei- und viermal stattfinden, tippen wir:

```
control.setReturnValue(new Euro(2.00), 2, 4);
```

Und ist es uns völlig egal, wie häufig die Methode gerufen wird:

```
control.setReturnValue(new Euro(2.00),
                       MockControl.ZERO_OR_MORE);
```

Auch unseren *Crashtest-Dummy* aus Kapitel 8.19 können wir im Handumdrehen dynamisch erzeugen lassen:

```
mockPrinter.cutPaper();
control.setThrowable(new OutOfPaperException());
```

EasyMock bietet wirklich extrem viele Kombinationsmöglichkeiten, so dass ich nur einen relativ kleinen Rahmen davon vorstellen kann. Weitere Feinheiten werden wir uns jedoch im nächsten Kapitel zu Gemüte führen, in dem uns EasyMock vom Anfang der Geschichte an begleiten wird.

8.22 Überspezifizierte Tests

An die Eigenheit von EasyMock, dass der Testcode wie Anwendungscode aussieht, muss man sich zugegebenermaßen zunächst gewöhnen. Die Vorteile liegen jedoch auf der Hand: So werden beispielsweise das Refactoring der Schnittstelle und die Code-Completion weiterhin von der Entwicklungsumgebung unterstützt. Ein Nachteil ist, dass der Testcode nicht nur dem Anwendungscode ähnelt, nein, dass er ihn manchmal sogar 1:1 spiegelt. Diese Art von Tests neigt dann dazu, wegen der starken Implementierungsabhängigkeit sehr fragil zu sein. Das kann explizit gewünscht sein oder auch nicht.

Generell kann man sagen, dass uns Mocks stärker dazu bewegen, die Beziehungen und Protokolle der beteiligten Objekte zu betrachten. Dass dabei dann auch mehr Implementierungsdetails enthüllt werden, sollte uns nicht überraschen. Andererseits kann die Gefahr einer Überspezifikation vermieden werden, wenn wir diese Details nur auf der richtigen Ebene exponieren. Häufig ist dieses Niveau genau dort erreicht, wo wir das absolute Minimum spezifizieren, das für den Test überhaupt von Relevanz ist. Eine gute Richtlinie ist deshalb, den Effekt zu testen, den der Code hat, nicht seinen Algorithmus. Je kleiner die getestete Einheit aber ist, desto schwieriger wird dies leider zu oft.

8.23 Überstrapazierte Mocks

Kurz vor Ende des Kapitels möchte ich dann aber doch noch eine kleine Warnung aussprechen, habe ich schon zu viele Projekte gesehen, die jedes ihrer Testprobleme mit einem Mock-Objekt beworfen haben. Für jemanden mit einem Hammer sieht alles aus wie ein Nagel? Nein! Damit Sie Ihre Mocks nicht überanspruchen, hier einige Hinweise, wann bei Ihnen die Alarmglocken läuten sollten.

Zu viel des Guten

Wir wissen, dass wir es mit den Mocks übertrieben haben,

- wenn wir vertrauenswürdige Klassen durch Mocks ersetzen,
- wenn der Test einer Klasse mehr als drei Mocks verlangt,
- wenn Mocks wieder Gebrauch von anderen Mocks machen,
- wenn für die Mehrzahl der Entitäten im System stets ein Interface, ein Mock und die tatsächliche Anwendungsklasse existiert,
- wenn Mocks ihrerseits so kompliziert sind, dass sie eigentlich schon wieder eigene Tests benötigen.

Ich kann Sie beruhigen, ein paar Ausreißer sind ganz natürlich. Meist weist ein Mock-Problem aber auf eine eigentlich vorhandene Designschwäche hin, die, auf Neudeutsch gesagt, *refactort* werden will.

8.24 Systemgrenzen im Test

Während dieses Buch entstand, habe ich von mehreren Reviewern den Kommentar erhalten, dass die vorgestellten Tests ja toll klappen, solange man schön abgetrennte Klassen hat, dass es in der Praxis aber Probleme gäbe mit Objekten, die Daten aus der Datenbank ziehen etc. Das ist alles richtig. Die Tests werden um ein Vielfaches interessanter, wenn unser Code komplizierte Abhängigkeiten auf andere Systemteile besitzt. Die Testgetriebene Entwicklung ermöglicht uns jedoch gerade, nur noch schön abgetrennte Klassen zu schreiben. Wir wollen über unsere Tests ja eben ein Design finden, mit dem wir alle Programmteile auch außerhalb ihrer späteren Umgebung testen können.

Showstopper sind oft die Systemgrenzen zur GUI, zur Datenbank und zum Application Server. Um die Tests bis an die Grenze zu führen, verwenden wir die gleichen Testmuster, wie wir sie schon zur Isolation innerhalb des Systems verwendet haben: Stubs, Shunts und Mocks. Das gleiche alte Prinzip des isolierten Tests gilt auch hier:

Prinzip des isolierten Tests

1. Ziehen Sie zur Indirektion einen *Adapter* [ga95] oder eine *Fassade* [ga95] ein.
2. Testen Sie vom System zum Adapter.
3. Testen Sie vom Adapter zur Systemgrenze.
4. Testen Sie das Zusammenspiel durch Integrationstests.

Beispiel: persistente Objekte

Schauen wir uns als Beispiel ein Problem aus dem J2EE-Umfeld an: persistente Objekte.

Schritt 1: Ziehen Sie zur Indirektion einen Adapter oder eine Fassade ein

Persistenzmechanismus über DAOs

Ein möglicher Persistenzmechanismus besteht darin, den Zugriff auf persistente Geschäftsobjekte rein über *Data Access Objects (DAOs)* abzuwickeln. Ihre Schnittstelle sollte möglichst schlank sein, da wir sie für darüber liegende Schichten mocken wollen. Gewöhnlich bedarf es wenigstens der vier CRUD-Methoden: create, read, update und delete.

Aufs OO/Relational-Mapping aufsetzend ist dann häufig noch eine Schnittstelle ganz hilfreich, die uns nicht mit Entitäten bestückt, wie sie uns die Datenbank an die Hand gibt, sondern wie sie für unsere Anwendung sinnvoll wären. Typischerweise zieht man aus diesem Grund über den DAOs noch eine Business-Logic-(BL)-Schicht ein.

Um die Persistenz als eigenständige Funktionseinheit zu testen, müssen wir unsere Implementierung an ihrer Schnittstelle isolieren. Spendieren wir unserer Datenzugriffsschicht also noch ein Interface zur Entkopplung. Das resultierende Design könnte dann zum Beispiel wie folgt aussehen:

Abb. 8–11
Testing at the Edge

Jetzt können wir die zwei Units separat testen:

- die `BL`-Klasse in Abhängigkeit vom `DAO`-Interface und dem `MockDAO`,
- die `DAO`-Implementierung an sich.

Schritt 2: Testen Sie vom System zum Adapter

1. Unit Test bis DAO-Schnittstelle

Mit unserem ersten Unit Test gehen wir bis an die DAO-Schnittstelle. Dieser Test ist vollkommen unabhängig von der konkret verwendeten Datenbank, da wir dem getesteten System einen Mock unterschieben. Das bedeutet, dass unser Test nicht fehlschlägt, sollten wir die Implementierung der Datenbankschicht ändern. Außerdem müssen wir sicherstellen, dass sich Mock und echte Datenbankschnittstelle für eine endliche Menge von Fällen semantisch decken.

Schritt 3: Testen Sie vom Adapter zur Systemgrenze

2. Unit Test: Durchgriff auf Datenbank

Der zweite Unit Test erst geht auf die wirkliche Implementierung ein. Zu testen haben wir dabei den Durchgriff auf die Datenbanktabellen, nicht das Datenbanksystem selbst. Diese Art von Tests sollte relativ einfach von der Hand gehen, besitzen sie doch nur eine Abhängigkeit auf das existierende Datenbankschema und seine Constraints.

Schritt 4: Testen Sie das Zusammenspiel durch Integrationstests

Integrationstests

Natürlich müssen wir auch die miteinander integrierten Komponenten noch einmal auf den Prüfstand stellen. Unit Tests sind ja nur geeignet, um Teile des Systems für sich zu testen. Probleme, die sich erst bei der Integration der Einzelteile ergeben können, finden sie jedoch nicht. Deshalb bietet sich eine weitere Abwehrlinie von Integrationstests an, die aufs korrekte Zusammenwirken der schon getesteten Units abzielt. Die integrierten Einheiten bahnen sich häufig sogar als Abstraktion ihren Weg in den Code. Allgemein gilt, dass diese Schicht so dünn wie möglich sein sollte. Im besten Fall ist sie reine Fassade. Vor allem EJBs sollten sich daran ein Beispiel nehmen. Wenn sie ihre wirkliche Arbeit nur noch an *POJOs (Plain old Java Object)* delegieren, benötigen sie meist nicht einmal mehr eigene Unit Tests, nur noch Integrationstests.

Mock-Objekte machen glücklich

von Moritz Petersen, Consileon GmbH

»Ist wirklich sichergestellt, dass die Komponente bei Fehlerfällen genau so reagiert, wie es erforderlich ist?« – Die Komponente, von der unser Kunde sprach, war eine der wichtigsten und zugleich umfangreichsten unseres Projektes: Ihre Aufgabe war es, die Daten aus vielen externen Datenquellen in unsere Anwendung zu importieren. Und bei den angesprochenen Fehlerfällen handelte es sich um die typischen Probleme, die bei der Anbindung an Fremdsysteme passieren können: unvollständige Daten, falsche Formatierungen usw.

Die richtige Funktionsweise einzelner Klassen hatten wir durch Unit Tests bereits sichergestellt. Doch um unseren Kunden in dieser Frage zufrieden zu stellen, war ein End-to-End-Test notwendig. Natürlich hätten wir eine Reihe von Testdaten präparieren können, doch der umständliche Import von echten Datensätzen wäre enorm zeitaufwändig gewesen. Auch wäre dieser weder automatisierbar noch nachvollziehbar gewesen für nicht direkt beteiligte Entwickler.

Wir verwendeten also Mock-Objekte und konnten daher alle Nachteile der Tests mit Echtdaten vergessen. Ein weiterer Vorteil von Mock-Objekten war, dass alle nur denkbaren Szenarien gleichzeitig von verschiedenen Entwicklern getestet werden konnten, ohne dass wir uns Sorgen um den Zustand der Datenbank machen mussten: Das Mock-Objekt hatte unseren Test von dieser komplett entkoppelt. So haben Mock-Objekte nicht nur unseren Kunden, sondern auch unsere Entwickler glücklich gemacht.

9 Entwicklung mit Mock-Objekten

Striktes Unit-Testen mit Mocks beeinflusst den Programmierstil auf so bemerkenswerte Weise, dass dieser Aspekt eigens ein Kapitel verdient. Wir werden einen Neuanfang machen und den Querschnitt unseres DVD-Verleihs mit Mock-Objekten neu entwickeln. Der Einsatz der Mocks wird dabei extrem sein, ja auf den absoluten Gipfel getrieben. Der Grund dafür ist, dass wir versuchen werden, *alle* Abhängigkeiten der zu testenden Klasse durch Mock-Implementierungen zu ersetzen. Beachten Sie beim Lesen, welchen Einfluss dies auf das Design und den Programmierstil hat.

9.1 Tell, don't ask

Dass Mock-Objekte mehr sind als ein wiederkehrendes Testmuster, wurde mir erst während der XP2002-Konferenz wirklich klar. In einer kleinen Programmiersession mit Steve Freeman und Michael Feathers führte Steve uns vor Augen, was die Entwicklung mit Mocks eigentlich wirklich ausmacht: den Designstil.

Einen oder zwei Tage zuvor hatten wir im Testing-Workshop die Diskussion geführt, wie weit man mit Mock-Objekten gehen sollte. Während einige von uns, einschließlich mir selbst, gebrannte Kinder waren und sich gegen ihre Überanspruchung aussprachen, suchten die Geistesväter der Mock-Technik, Steve Freeman und Tim Mackinnon, gewollt das Feuer und verwendeten Mocks in fast verschwenderischer Art und Weise.

Was machten wir also anders als die beiden? Gelöst wurde dieses Rätsel in besagter Programmiersession, in der wir von Steve lernten, wie die Entwicklung von Unit Tests im Mock-Objekt-Stil das Design auf interessante Weise umzudrehen vermag, und zwar hin zu einem Designstil, den man sogar als den wahren OO-Stil bezeichnen könnte: *Tell, don't ask*.

Objekte zeichnen sich unter anderem dadurch aus, dass sie im Gegensatz zu prozeduralem Code ihr Verhalten exponieren, nicht ihre Daten oder ihren Zustand. Nichtsdestotrotz schleicht ein schlechter Programmierstil durch die OO-Gemeinde, wo Programmierer den einen Teil ihrer Objekte mit `get`- und `set`-Methoden übersäen und in dem anderen Teil das wirkliche Verhalten implementieren, indem sich die intelligenten Arbeiterklassen die nötigen Daten aus den dummen Datencontainerklassen besorgen.

Besonders in Java, C++ oder C# kann man diesem Irrtum leicht verfallen, da die Methodenaufrufe stark dem Prozeduraufruf ähneln. Anders dagegen etwa in Ruby oder Smalltalk, wo Objekte miteinander Nachrichten austauschen. Der bessere Weg wäre sicher, das Objekt mit den Daten zuständig zu machen für alle Operationen auf diesen Daten. Den Blick auf den eigentlich privaten Zustand des Objekts müssten wir somit auch nicht öffentlich freigeben.

»Tell, don't ask« rät genau dazu: den Objekten Befehle zu geben, anstatt ihnen Löcher in den Bauch zu fragen. Wie ein Objekt seine Aufgabe erledigt, unterliegt dann vollkommen seiner Verantwortung. Wobei wir die Verantwortlichkeiten so im System verteilen wollen, dass jede Klasse nur für eine Sache zuständig ist und nicht mehr.

9.2 Von außen nach innen

Die Überschrift trifft den Kern der Sache auf zweierlei Art und Weise: Erstens starten wir vom äußeren Ende des Problems. Vom Bekannten zum Unbekannten. Zweitens reichen wir dem zu testenden Subjekt *alle* Objekte, die es für seinen Dienst benötigt, als Mock-Objekte von außen herein. Wir machen damit in der Klassenschnittstelle explizit, was unsere Klasse *bietet* und was sie dazu von ihren Klienten *verlangt*.

Felix: *Wo fangen wir an?*

Ulrich: *Im Vordergrund steht wohl, Geld zu verdienen. Schlage vor, wir fangen mit den Rechnungen an.*

Felix: *Dazu müssten wir dann wissen, an wen und für was wir unsere Rechnung stellen.*

Ulrich: *Vielleicht so? Kunde leiht DVDs und erhält Beleg gedruckt?*

```
Customer customer = new Customer("Zing Zang Zong");
customer.addRental(mockRental);
mockStatement.printStatement(customer);
```

9.3 Wer verifiziert wen?

Als Nächstes müssen wir uns überlegen, wie unsere Klasse mit ihren Mock-Objekten zusammenspielen soll: Wer tut was, welche Methoden werden wie mit welchen Werten aufgerufen et cetera. Wir müssen uns die Frage stellen, was wir zum Ende des Tests denn verifizieren wollen. Oder in anderen Worten: Welches Mock-Objekt könnte uns verraten, ob unsere Klasse ihren Job getan hat?

Felix: *Stopp mal! Wir erwarten doch, dass unser `Customer` seinen `Rentals` befiehlt, sich selbst ins `Statement` einzutragen, oder?*

Ulrich: *Korrekt!*

Felix: *Kann ich ja einfach mal hinschreiben ... Doch das wird nicht funktionieren.*

```
mockRental.printOn(mockStatement);

Customer customer = new Customer("Zing Zang Zong");
customer.addRental(mockRental);
mockStatement.printStatement(customer);
```

Ulrich: *Wieso denn nicht?*

Felix: *Weil unser `Customer` doch gar kein `Statement` hat!?*

Ulrich: *Hmm ... Wir könnten doch schreiben ...*

```
public class Statement {
  public void printStatement(Customer customer) {
    customer.printOn(this);
  }
}
```

Felix: *Ändert aber ja nix an der Tatsache, dass wirs verdreht haben ... Ich glaube, wir verzetteln uns gerade.*

Ulrich: *Du hast Recht. Soll der `Customer` sein `Statement` füttern und nicht das `Statement` seinen `Customer` ausfragen, müssen wirs umdrehen.*

```
customer.printOn(mockStatement);
```

Ulrich: *So weit das.*

Felix: *Jetzt kann er sich auch selbst im `Statement` eintragen.*

```
mockStatement.printCustomerName("Zing Zang Zong");
```

Ulrich: *Gut, wie viel haben wir jetzt?*

```
public class StatementTest extends TestCase {
  public void testCustomer() {
    MockControl controlStatement =
      MockControl.createControl(IStatement.class);
    IStatement mockStatement =
      (IStatement) controlStatement.getMock();
    mockStatement.printCustomerName("Zing Zang Zong");

    MockControl controlRental =
      MockControl.createControl(IRental.class);
    IRental mockRental = (IRental) controlRental.getMock();
    mockRental.printOn(mockStatement);

    controlStatement.replay();
    controlRental.replay();

    Customer customer = new Customer("Zing Zang Zong");
    customer.addRental(mockRental);
    customer.printOn(mockStatement);

    controlStatement.verify();
    controlRental.verify();
  }
}
```

9.4 Schnittstellen finden auf natürlichem Weg

Indem wir unsere Klasse mit Mock-Implementierungen für die sie umgebenden Objekte testen, faktorisieren wir für Letztere explizit eine gut verwendbare Schnittstelle heraus. Unsere Mock-Objekte helfen so, gegen die geplante Schnittstelle zu programmieren und sie zu fixieren, bevor wir ihre konkrete Implementierung angehen. Zum Zeitpunkt, wenn unsere Tests laufen, hat sich die Schnittstelle so weit stabilisiert, dass wir genug gelernt haben, um die wirkliche Klasse zu schreiben. Refactoringwerkzeuge flachen die Kostenkurve ab, um Schnittstellen zu extrahieren und zu implementieren.

Folgende zwei Schnittstellen konnten wir bisher ausmachen:

```
public interface IStatement {
  void printCustomerName(String name);
}

public interface IRental {
  void printOn(IStatement statement);
}
```

Ulrich: *Das `IRental`-Interface benötigt nur eine einzelne Methode?*

Felix: *Eigentlich müssen sich die `Rentals` doch nur im `Statement` verewigen können?!*

Ulrich: *Gut, schließen wir erst mal diese Klasse ab ...*

```
public class Customer {
  private List rentals = new ArrayList();
  private String name;

  public Customer(String name) {
    this.name = name;
  }

  public void addRental(IRental rental) {
    rentals.add(rental);
  }

  public void printOn(IStatement statement) {
    statement.printCustomerName(name);
    for (Iterator i = rentals.iterator(); i.hasNext();) {
      IRental rental = (IRental) i.next();
      rental.printOn(statement);
    }
  }
}
```

Ulrich: *Wir haben noch was vergessen.*

Felix: *Was haben wir noch vergessen?*

Ulrich: *Soll nicht am Fuß der Rechnung der Gesamtbetrag stehen?*

Felix: *Selbstverständlich.*

Ulrich: *Dazu fehlt uns dann aber noch eine Erwartung ...*

```
    mockStatement.printTotalCharge();
```

Ulrich: ... *und die entsprechende Methode im Interface.*

```
public interface IStatement {
  void printCustomerName(String name);
  void printTotalCharge();
}
```

Felix: *Ich mach mal weiter ... hab da nämlich 'ne Idee.*

(schnappt sich die Tastatur)

9.5 Komponierte Methoden

Methodenrümpfe sollten sich auf einem Abstraktionsniveau befinden. Dieses Prinzip steht zwar nicht im direkten Zusammenhang mit Mocks, ist jedoch so nützlich, dass ich es unbedingt erwähnen muss. Formulieren wir unsere Methoden nämlich in dem für sie richtigen Abstraktionsgrad, lässt sich Code lesen wie sonst nur Kommentare.

```
public class Customer...
  public void printOn(IStatement statement) {
    statement.printCustomerName(name);
    for (Iterator i = rentals.iterator(); i.hasNext();) {
      IRental rental = (IRental) i.next();
      rental.printOn(statement);
    }
    statement.printTotalCharge();
  }
}
```

Felix: *Unsere Methode verletzt hier das Composed-Method-Pattern [be96]. Deshalb würde ich ...*

Ulrich: *Bitte, was? Du sprichst in Literaturhinweisen! (grinst)*

Felix: *(lacht) Ich meinte, dass das erste und letzte Statement auf höherem Niveau sind und die Schleife mehr low level.*

Ulrich: *Was willst du machen? Methode extrahieren?*

Felix: *Bingo!*

```
public class Customer...
  public void printOn(IStatement statement) {
    statement.printCustomerName(name);
    printRentalsOn(statement);
    statement.printTotalCharge();
  }

  private void printRentalsOn(IStatement statement) {
    for (Iterator i = rentals.iterator(); i.hasNext();) {
      IRental rental = (IRental) i.next();
      rental.printOn(statement);
    }
  }
}
```

9.6 Vom Mock lernen für die Implementierung

Mock-Objekte treiben die Entwicklung auf zweierlei Wegen voran: Zum einen können wir die in Arbeit befindliche Klasse mit ihrer Hilfe vollständig implementieren, obwohl die Klassen, auf die sie aufsetzt, noch nicht existieren. Da die Softwareentwicklung dabei explorativ vom Bekannten zum Unbekannten voranschreitet, ist es im Gegensatz zur Bottom-up-Implementierung wahrscheinlich, dass die produzierte Klasse sich so auch als brauchbar bewährt. Zum anderen können wir unseren Mock-Objekten unmittelbar entnehmen, welche Schnittstellen die fertig gestellte Klasse von ihren umgebenden Objekten erwartet.

Um etwas Platz zu sparen, schenke ich mir ab sofort die Definition der EasyMocks und die filigrankleinen Schritte.

Ulrich: *Vom `Rental` erwarten wir nur, dass es die `printOn`-Methode implementiert.*

Felix: *... und dass es an `Price` und `Movie` weiterdelegiert.*

```
public class StatementTest...
  public void testRental()...
    mockMovie.printOn(mockStatement);
    mockPrice.printOn(mockStatement, 1);

    controlStatement.replay();
    controlMovie.replay();
    controlPrice.replay();

    new Rental(mockMovie, mockPrice, 1).printOn(mockStatement);

    controlStatement.verify();
    controlMovie.verify();
    controlPrice.verify();
  }
}
```

Ulrich: *Warum muss `Rental` denn überhaupt an `Price` delegieren?*

Felix: *Wie sonst?*

Ulrich: *Kann das nicht die `Movie`-Klasse übernehmen?*

Felix: *Dann wäre der Preis an den Film gebunden, nicht wahr?*

Ulrich: *Ist er das nicht?*

Felix: *Das habe ich anders im Kopf ... Aber lass uns mal besser Uta dazuholen.*

(Uta ist die Domänenspezialistin – ein anderes Wort für Kundin.)

Ulrich: *(winkt) Uta, kannst du uns bitte mal helfen?*

Uta: *Wo drückt euch der Schuh?*

Ulrich: *Wir müssen von dir wissen, ob die Leihgebühr an den Film oder die Ausleihe geknüpft ist.*

Uta: *Verstehe eure Frage nicht.*

Felix: *Wenn ich eine DVD heute als Neuerscheinung leihe und Uli mietet sie morgen mit normalem Preis, weil der Film genau an diesem Tag seinen Neu-Status verliert, bezahlen wir wie viel?*

Uta: *In dem Fall bezahlt der Uli natürlich den normalen Preis ... und du die Neuerscheinung. War das nicht klar?*

Felix: *Noch nicht ganz ... aber jetzt.*

Ulrich: *Macht ja anders auch kaum Sinn. Danke, Uta!*

Uta: *Aber gerne.*

Felix: *So so, macht ja anders keinen Sinn. (grinst)*

Ulrich: *Da dem so ist, könnten wir uns fast überlegen, ob sich unsere `Movie`-Klasse überhaupt noch lohnt.*

Felix: *Gute Idee. Bleibt nicht viel übrig außer einem `String`.*

Ulrich: *Ich würde sie integrieren. Was meinst du?*

Felix: *Nur zu, Maestro!*

```
public class StatementTest...
  public void testRental()...
    mockStatement.printMovieTitle("Amélie");
    mockPrice.printOn(mockStatement, 1);

    controlStatement.replay();
    controlMovie.replay();
    controlPrice.replay();

    new Rental("Amélie", mockPrice, 1).printOn(mockStatement);

    controlStatement.verify();
    controlMovie.verify();
    controlPrice.verify();
  }
}
```

Ulrich: *Sehr interessant, ein DVD-Verleihsystem zu programmieren, in dem sich ein so zentrales Konzept nicht in einer eigenen Klasse niederschlägt. Es gibt doch noch Überraschungen ...*

Felix: *Unsere `Rental`-Klasse wird dadurch auch gleich ein bisschen einfacher ...*

```
public class Rental implements IRental {
  private String movieTitle;
  private IPrice price;
  private int daysRented;

  public Rental(String movieTitle, IPrice price, int days) {
    this.movieTitle = movieTitle;
    this.price = price;
    this.daysRented = days;
  }

  public void printOn(IStatement statement) {
    statement.printMovieTitle(movieTitle);
    price.printOn(statement, daysRented);
  }
}
```

9.7 Viele schmale Schnittstellen

Unverkennbar resultiert der propagierte Designstil in einer größeren Menge von Schnittstellen, als es die meisten Entwickler gewöhnt sind. Auf der Plusseite wird unser System dadurch um Größenordnungen modularer und leichter erweiterbar. Mit jeder Schnittstelle erhalten wir letztendlich einen potenziellen Plug-in-Punkt, durch den sich unsere Klassen komponentenartig zusammenstecken und kombinieren lassen. Auf der Negativseite schleppen wir eben mehr Schnittstellen mit und unsere Klassen mehrere Parameter.

```
public interface IStatement {
  void printCustomerName(String name);
  void printMovieTitle(String title);
  void printTotalCharge();
}

public interface IPrice {
  void printOn(IStatement statement, int daysRented);
}
```

Felix: *Schade, dass unser Preis die Anzahl Tage mitbekommt ...*

Ulrich: *Wäre jetzt auch ein bisschen zu schön gewesen, wenn sich alle unsere Klassen ein Interface teilen könnten.*

9.8 Kleine fokussierte Klassen

In gleicher Weise, wie wir bei vielen schmalen Schnittstellen landen, erhalten wir mehr kleine und lose gekoppelte Klassen. Allein, weil es schwer ist, Klassen zu testen, die zu viel Verantwortung übernehmen, sind diese Klassen stärker auf Teilaspekte fokussiert als gewöhnlich. Eine Besonderheit ist, dass sowohl die Schnittstellenimplementierung als auch die Komposition von Klassen ein stärkeres Gewicht bekommt gegenüber der Implementierungsvererbung.

Da wir unsere Tests auf die Interaktion von Klassen ausrichten, sind unsere Designs also automatisch mehr schnittstellenorientiert und mehr verhaltengetrieben. Das mockgetriebene, strikte Unit-Testen sorgt so unter anderem dafür, dass jede Klasse ihre Abhängigkeiten explizit in ihrer Schnittstelle reflektiert und wir diese Abhängigkeiten dadurch früher erkennen und ihnen gezielter entgegentreten können. Die Folge ist auch, dass alle Klassen in ihrer Schnittstelle ausdrücken, welche Objekte von welchem Typ sie als enge Mitarbeiter benötigen. Das Zusammenspiel der Klassen wird so deutlich im Code sichtbar.

Ulrich: *Das Interface `IPrice` fühlt sich schon irgendwie komisch an, findest du nicht auch?*

Felix: *Das ist nur die Umgewöhnung, dass unsere Objekte langsam erwachsen werden ... und mehr Verantwortung akzeptieren.*

```
public class StatementTest...
  public void testBasePrice() {
    verifyPrice(1.50, 1);
    verifyPrice(1.50, 3);
  }

  public void testIncrementalPrice() {
    verifyPrice(3.00, 4);
    verifyPrice(7.50, 7);
  }

  private void verifyPrice(double charge, int daysRented) {
    mockStatement.printMovieCharge(new Euro(charge));
    controlStatement.replay();
    new Price().printOn(mockStatement, daysRented);
    controlStatement.verify();
    controlStatement.reset();
  }
}
```

9.9 Tell und Ask unterscheiden

Eine gute Idee ist, die Methoden einer Klasse nach zwei Aspekten zu trennen: in Befehls- und Abfragemethoden: Tell und Ask [th98]. Auf diese Weise erreichen wir eine wünschenswerte Orthogonalität im Methodenentwurf: Wir unterteilen nämlich den Stamm von Methoden in solche, die einen gewünschten Seiteneffekt auf das betreffende oder die mitarbeitenden Objekte haben, und solche, die den beobachtbaren Zustand der Objekte unverändert lassen. Diese Trennung ist nützlich, weil sie unserem Schema aus Kapitel 7.1, *Aufbau von Testfällen*, folgt:

Befehls- und Abfragemethoden

1. Fixture-Objekte erzeugen und in den Ausgangszustand bringen
2. **Tell:** Methoden der Objekte exerzieren
3. **Ask:** Erwartete und tatsächliche Resultate vergleichen

Ferner drücken wir im Code aus, welche Methoden wir seiteneffektfrei aufrufen können und welche mit Seiteneffekten behaftet sind. Trotz »Tell, don't ask« müssen dennoch nicht alle Methoden Befehle sein, auch wenn dieser Wunsch für `IStatement` in Erfüllung gegangen ist:

```
public interface IStatement {
  void printCustomerName(String name);
  void printMovieTitle(String title);
  void printMovieCharge(Euro charge);
  void printTotalCharge();
}
```

Die folgende Methode könnten wir auf beiderlei Arten entwickeln, auch wenn sie sich als Befehl doch besser ins Design einfügt:

```
public class Price implements IPrice {
  public void printOn(IStatement statement, int daysRented) {
    Euro flat = new Euro(1.50);
    Euro incremental = new Euro(1.50).times(daysRented - 2);
    statement.printMovieCharge(Euro.max(flat, incremental));
  }
}
```

Und hier haben wir ohne Frage eine klassische Abfrage:

```
public class Euro...
  public static Euro max(Euro a, Euro b) {
    return a.cents > b.cents ? a : b;
  }
}
```

9.10 nereimmargorP sträwkcüR

Trotz der Gefahr, dass ich mich zu wiederholen beginne, will ich erneut die Technik des Rückwärtsarbeitens von Jim Newkirk aufgreifen: In Kapitel 7.8, *Erst die Zusicherung schreiben*, haben wir unseren Test vorausgedacht, indem wir von einer Zusicherung ausgegangen sind. Besonders hilfreich finde ich diese Arbeitsweise für die Entwicklung mit Mock-Objekten, da sie uns auf natürlichem Weg mit einer Vision unseres Zielpunkts beginnen lässt.

Felix: *Mit welchem Objekt können wir das* `Statement` *verifizieren?*

Ulrich: *Sicher ist wohl, dass das* `Statement` *mit dem* `Printer` *spricht.*

```
Statement statement = new Statement(mockPrinter);
controlPrinter.verify();
```

Felix: *Dann lass uns noch mal das geforderte Interface anschauen.*

```
public interface IStatement {
  void printCustomerName(String name);
  void printMovieTitle(String title);
  void printMovieCharge(Euro charge);
  void printTotalCharge();
}
```

Ulrich: *Versuchen wir mal, darüber einen Beleg zu drucken ...*

```
Statement statement = new Statement(mockPrinter);
statement.printCustomerName("Zing Zang Zong");
statement.printMovieTitle("Bladerunner");
statement.printMovieCharge(new Euro(1.50));
statement.printTotalCharge();

controlPrinter.verify();
```

Felix: *... aber dann bitte gleich mit zwei DVDs, um auch wirklich die Betragssumme mitzutesten.*

```
Statement statement = new Statement(mockPrinter);
statement.printCustomerName("Zing Zang Zong");
statement.printMovieTitle("Bladerunner");
statement.printMovieCharge(new Euro(1.50));
statement.printMovieTitle("Taxi Driver");
statement.printMovieCharge(new Euro(4.50));
statement.printTotalCharge();

controlPrinter.verify();
```

Ulrich: *So weit das. Was erwarten wir denn genau für diesen Fall?*

```
public class StatementTest...
  public void testStatement() {
    MockControl controlPrinter =
      MockControl.createStrictControl(IPrinter.class);
    IPrinter mockPrinter = (IPrinter) controlPrinter.getMock();
    mockPrinter.print("Rental records for Zing Zang Zong");
    mockPrinter.crlf();
    mockPrinter.tab();
    mockPrinter.print("Bladerunner");
    mockPrinter.tab();
    mockPrinter.print("1,50");
    mockPrinter.crlf();
    mockPrinter.tab();
    mockPrinter.print("Taxi Driver");
    mockPrinter.tab();
    mockPrinter.print("4,50");
    mockPrinter.crlf();
    mockPrinter.print("Amount owed is 6,00");
    mockPrinter.crlf();
    mockPrinter.cutPaper();

    controlPrinter.replay();

    Statement statement = new Statement(mockPrinter);
    statement.printCustomerName("Zing Zang Zong");
    statement.printMovieTitle("Bladerunner");
    statement.printMovieCharge(new Euro(1.50));
    statement.printMovieTitle("Taxi Driver");
    statement.printMovieCharge(new Euro(4.50));
    statement.printTotalCharge();

    controlPrinter.verify();
  }
}
```

Ulrich: *Somit hätten wir dann auch gleich die Ausgabeschnittstelle ...*

```
public interface IPrinter {
  void tab();
  void print(String output);
  void crlf();
  void cutPaper();
}
```

9.11 Schüchterner Code und das Gesetz von Demeter

Sprich nicht mit Fremden!

Mockgetrieben entwickelter Code besitzt die interessante Eigenschaft, dass Objekte generell nur noch Methoden lokaler Objekte verwenden. Gemäß dem Demeter-Gesetz sollte ein Objekt nur Methoden von

- sich selbst,
- als Parameter übergebenen Objekten,
- selbst erzeugten Objekten oder
- direkt aggregierten Objekten aufrufen [hu99].

Vermieden werden also Methodenaufrufe auf Objekten, die ihrerseits schon Resultat eines Methodenaufrufs waren, wodurch sich die Abhängigkeiten im Code drastisch reduzieren lassen:

```
public class Statement implements IStatement {
  private IPrinter printer;
  private Euro total = new Euro(0);

  public Statement(IPrinter printer) {
    this.printer = printer;
  }

  public void printCustomerName(String name) {
    printer.print("Rental records for " + name);
    printer.crlf();
  }

  public void printMovieTitle(String title) {
    printer.tab();
    printer.print(title);
  }

  public void printMovieCharge(Euro charge) {
    printer.tab();
    printer.print(charge.format());
    printer.crlf();
    total = total.plus(charge);
  }

  public void printTotalCharge() {
    printer.print("Amount owed is " + total.format());
    printer.crlf();
    printer.cutPaper();
  }
}
```

9.12 Fassaden und Mediatoren als Abstraktionsebene

Um die durch den kompositorischen Programmierstil hochgetriebene Menge kleinerer Klassen und Schnittstellenparameter zu gruppieren, empfiehlt es sich, für eng zusammenhängende Teile eine Fassade oder einen *Mediator* [ga95] einzuführen.

Felix: *Zum Abschluss würde ich gerne noch ein Zwischenobjekt haben, das uns die Arbeit mit all diesen Klassen abnimmt.*

Ulrich: *Ein Objekt, mit dem man den gesamten DVD-Verleih in der Hand hat? Blendende Idee!*

```
public class Videostore {
  private Customer customer;

  public void rentMovie(String movieTitle, int daysRented) {
    IPrice price = new Price();
    IRental rental = new Rental(movieTitle, price, daysRented);
    customer.addRental(rental);
  }

  public void printStatement() {
    IPrinter printer = new Printer();
    IStatement statement = new Statement(printer);
    customer.printOn(statement);
  }
```

Ulrich: *Wir müssten der Fassade dann nur noch erklären können, welchen Kunden wir aktuell in den Händen halten wollen ...*

```
  private static Map customers = new HashMap();

  public static Videostore bindCustomer(String name) {
    if (!customers.containsKey(name)) {
      customers.put(name, new Customer(name));
    }
    Customer customer = (Customer) customers.get(name);
    return new Videostore(customer);
  }

  private Videostore(Customer customer) {
    this.customer = customer;
  }
}
```

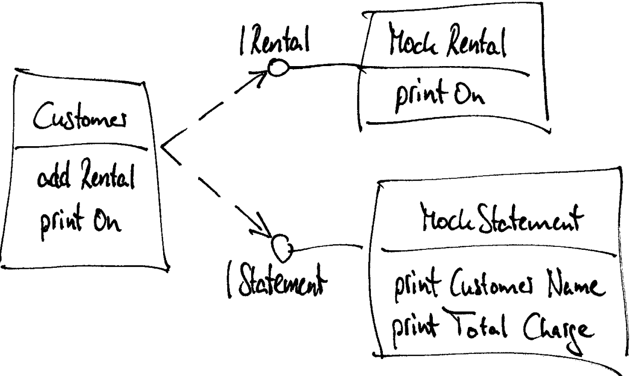
Felix: *Schulterklopf! Genug für heute, checken wir ein ...*

9.13 Rekonstruktion

Die in diesem Kapitel beschriebenen Techniken sind mit Sicherheit die fortgeschrittensten im ganzen Buch. Wenn ich Sie also mit dem Stoff abgehängt habe, ist das nicht so schlimm (außer dass ich offenbar mein Ziel verfehlt habe, eine komplexe Geschichte mit einfachen Worten wiederzugeben). Ich habe auch lange überlegt, ob ich dieses Kapitel überhaupt mit ins Buch aufnehmen soll, mich schließlich jedoch dafür entschieden, weil es für viele Leser eine interessante Alternative zum angewöhnten Programmierstil darstellen dürfte. Deshalb möchte ich die Geschehnisse mit einer kurzen Bilderstrecke zusammenfassen:

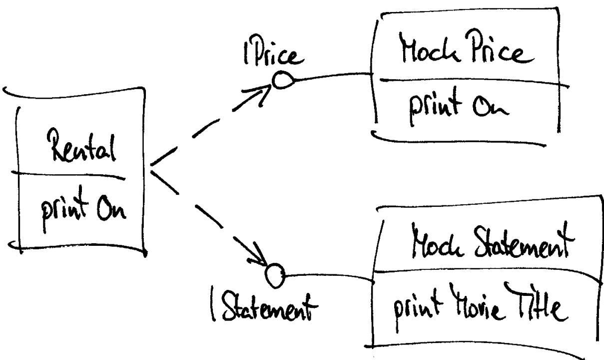
Zunächst haben wir die `Customer`-Klasse ins Leben getestet. Damit sie Ausleihen verwalten und Belege erstellen kann, haben wir mit `IRental` und `IStatement` zwei Schnittstellen identifiziert und gemockt:

Abb. 9–1
Vgl. Seite 216-220: Customer



Danach haben wir uns die `Rental`-Klasse vorgenommen. Die gemockte Schnittstelle diente uns als Vorlage. Um ihre Zuständigkeit zu erfüllen, wurde die `IPrice`-Schnittstelle eingeführt und `IStatement` erweitert:

Abb. 9–2
Vgl. Seite 221-223: Rental



Gleiches gilt für die `Price`-Klasse: Die Mock-Implementierung trieb die Implementierung der wirklichen Klasse an. Ihre Verantwortlichkeit erforderte, dass wir die `IStatement`-Schnittstelle weiter aufbohren:

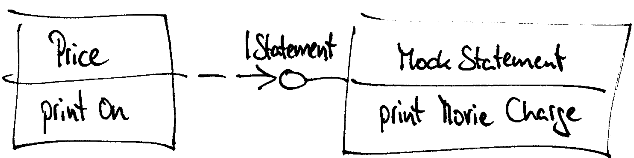


Abb. 9–3
Vgl. Seite 224-225: Price

Dann kam `Statement` an die Reihe: Aus den vorausgegangenen Tests hatten wir genug über die geforderte Schnittstelle gelernt. Die Klasse benötigte für ihre Services wiederum einige Services, die wir mit der `IPrinter`-Schnittstelle in Form gießen konnten:

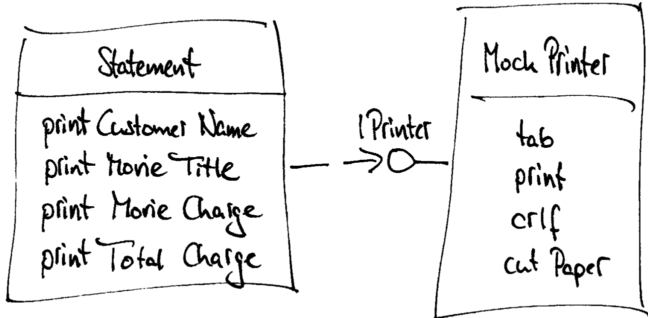


Abb. 9–4
Vgl. Seite 226-228: Statement

Zu guter Letzt haben wir die vielen kleinen, lose gekoppelten Teile mit einer Fassade abgedeckt. Wer einen *Inversion-of-Control*-Container wie das *Spring-Framework* verwendet, wird die Konfiguration seiner Komponentenstruktur entsprechend in XML hinterlegen:

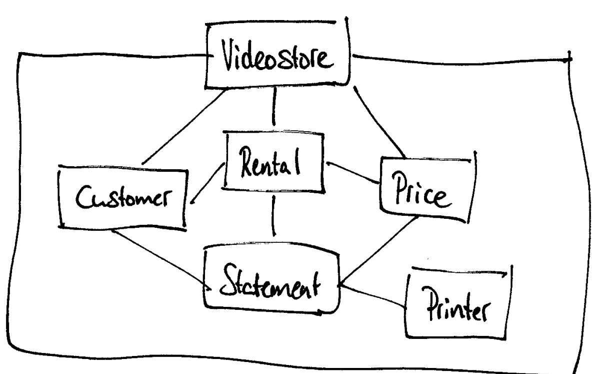


Abb. 9–5
Vgl. Seite 229: Videostore

In Typen und Rollen denken, nicht in Klassen

Was ist nun das Bemerkenswerte an diesem Programmierstil? Nun, zum einen orientiert sich unser Design viel mehr an den Typen als an den Klassen. Die Mocks helfen, die Typen zu identifizieren, basierend auf den Rollen, die ein Objekt im System spielt [fr04]. Die Klassen, die den Typ oder die Rolle hinterher einnehmen, sind austauschbar.

Responsibility-Driven Design

Zum anderen drehen die Mocks das Design um. Sie sorgen ganz automatisch für die Einhaltung des Gesetzes von Demeter und bringen Strukturen zum Vorschein, die stärker auf das Verhalten als auf die Daten ausgerichtet sind. Wer anderes gewohnt ist, für den fühlen sich die resultierenden Entwürfe ungewohnt an. Nicht selten gelangen wir wie von selbst zu fortgeschrittenen Entwurfsmustern wie *Visitor* [ga95] oder *Collecting Parameter* [be96]. Aus objektorientierter Sicht sind die Entwürfe geradezu ideal, weil die Abhängigkeiten explizit gemacht werden (Dependency-Injection-Muster) und nur noch zu abstrakten Schnittstellen existieren statt zu konkreten Klassen (Dependency-Inversion-Prinzip). Beachten Sie auch einmal, wie einzelne orthogonale Verantwortlichkeitsaspekte fast wie von Geisterhand in verschiedenen Klassen gelandet sind.

Vom Erfinden zum Entdecken

Anstatt möglichst gut zu erraten, was die zu entwickelnde Klasse wohl an Funktionalitäten zu bieten haben sollte, können wir nun die Entwürfe unmittelbar an den Anforderungen orientieren. Wir können den Schnittstellen und Diensten, die die bereits entwickelten Klassen für ihre Funktionsweise voraussetzen, direkt entnehmen, was wir noch bereitstellen müssen. Der Designprozess bekommt dadurch eine neue Gestalt und bessere Richtung.

Meister, ...

von Dierk König, Canoo Engineering AG

Der Lehrling fragt den Meister:

»Meister, wie soll ich Entscheidungen treffen?«
»Welche Möglichkeiten siehst du denn?«, fragt der Meister zurück.

Der Lehrling denkt kurz nach und antwortet:

»Ich könnte scharf nachdenken – oder ausprobieren.«
»Und was würdest du bevorzugen?«

Der Lehrling überlegt und zögert:

»Scharf nachzudenken – denke ich.«
»Gut«, sagt der Meister. »Probier das aus.«

10 Akzeptanztests mit FIT

Unsere zweite Verteidigungslinie neben Unit Tests sind Akzeptanztests. Unterstützen uns die Unit Tests dabei, stetigen Fortschritt zu erzielen, geben uns die Akzeptanztests das eigentliche Ziel der Entwicklung vor: Sie *spezifizieren* zum einen, was der geforderte Funktionsumfang ist, und *verifizieren* zum anderen, ob diese Anforderungen erfüllt werden. Sie sind *Black-Box-Tests* und testen das System als Ganzes.

Vorgegeben werden Akzeptanztests von den Ansprechpartnern auf Kundenseite mit der nötigen Fachkenntnis über die Problemdomäne. Damit diese Tests das Vertrauen in den produzierten Code stärken und unter Umständen gar als Abnahmekriterium für die Vertragserfüllung dienen können, müssen sie auch die *Sprache des Kunden* sprechen. Deshalb scheidet JUnit für den Akzeptanztest aus und wir müssen uns mit einem kundenfreundlicheren Testansatz vertraut machen.

FIT-Framework zur Automatisierung von Akzeptanztests

FIT steht für »Framework for Integrated Test«, ein Framework von Ward Cunningham für die Automatisierung von Akzeptanztests. FIT-Tests sind datengetrieben: Testdaten werden tabellarisch erstellt: in HTML, mit Microsoft Word, Excel oder in Wiki (dazu noch später). FIT liest diese Tests als Dokument ein, führt unser Programm mit den Tabelleneinträgen aus, erstellt dabei eine Kopie des Dokuments und schreibt seinen Befund an Ort und Stelle in dieses Ergebnisdokument. So sind die Akzeptanztests für Menschen sehr viel besser verständlich. Eine konkrete Testspezifikation sieht beispielsweise so aus:

Pricing		
daysRented	regularPrice()	newRelease()
1	1.50	2.00
2	1.50	2.00
3	1.50	3.75
4	3.00	5.50
5	4.50	7.25

Abb. 10–1
FIT-Tabelle mit spezifizierten Testdaten

10.1 Von einer ausführbaren Spezifikation ...

Bei Testausführung arbeitet FIT unsere HTML-Tabelle zellenweise ab, interpretiert die Testdaten und reicht sie an unseren Testcode weiter. Dieser exerziert damit unser Programm und liefert Antworten zurück. FIT vergleicht darauf die in der Tabelle spezifizierten erwarteten Werte mit der tatsächlich berechneten Antwort und tunkt die verarbeitete Tabellenzelle je nach Ergebnis in pastelliges Grün, Rot oder auch Gelb. That's it:

Abb. 10–2
FIT-Tabelle nach dem Testlauf in beruhigendes Grün getaucht

Pricing		
daysRented	regularPrice()	newRelease()
1	1.50	2.00
2	1.50	2.00
3	1.50	3.75
4	3.00	5.50
5	4.50	7.25

In einem Dokument können wir nun mehrere solche Tabellen einfach hintereinander hängen und so ein umfassendes Testszenario aufbauen. Den Raum um die Tabellen herum können wir zur Dokumentation von Testfall und Anforderungen verwenden. FIT ignoriert diese Texte, die außerhalb der Tabellen stehen. Auf diesem Weg erstellen wir im Projektverlauf ein *ausführbares Anforderungsdokument*, das zunächst alle fachlichen Anforderungen anhand konkreter Akzeptanzkriterien spezifiziert und hinterher per einfachem Knopfdruck verifizieren kann, wie *fit* unser System wirklich ist.

10.2 Download Now

fit-java-1.0.zip

FIT ist unter *GNU General Public License (GPL)* gestellt und bereits für eine Vielzahl der gängigen Programmiersprachen portiert. Im Buch verwende ich die Version 1.0. Aktuell steht der Versionszähler auf 1.1. Zu den Neuerungen gehören kundenfreundlichere Fehlermeldungen und Namen. Die aktuelle Version finden Sie im *FIT-Wiki*:

```
http://fit.c2.com
```

Zurzeit wird die Distribution als ZIP-Archiv ausgeliefert, das neben dem Framework (in `fit.jar`) die Quellen, FIT-Akzeptanztests (FAT) und eine Fülle von Beispielen enthält. Installieren Sie bitte das `fit.jar` in Ihrem `CLASSPATH` und das Akzeptanztesten kann losgehen.

10.3 Schritt für Schritt für Schritt

Mein Akzeptanztest an dieses Kapitel ist, dass Sie Ihren ersten FIT-Test zum Laufen bringen und verstehen, wie das FIT-Framework arbeitet. Dazu werden wir zunächst für die schon bestehende Funktionalität entsprechende Akzeptanztests nachschieben. Natürlich fallen die Tests nicht irgendwo vom Himmel. Vielmehr sind sie unmittelbar an die fachlichen Anforderungen des Kunden geknüpft und dokumentieren als solches seine Erwartungen, wann die von uns produzierte Software für ihn *akzeptabel* ist.

Anforderungen ohne Akzeptanztests sind keine Anforderungen

Eine Benutzergeschichte

Wie Sie Akzeptanztests formulieren, das erfahren Sie im Kapitel 10.42, *Akzeptanztesten aus Projektsicht*. Hier geht es allein ums Rüstzeug. Als Beispiel dafür soll uns die Preiskategorie aus dem Kapitel 4 dienen. Auf Seite 55 hieß es zum Preismodell lapidar:

> *Filme kosten regulär für die ersten drei Tage insgesamt 1,50 Euro. Für jeden weiteren Ausleihtag kommen 1,50 Euro dazu.*

Spezifikation am Beispiel

Starten wir also Word und legen eine kleine Tabelle mit Testdaten an, die, wenn erfolgreich durchgelaufen, beweisen, dass unser Programm mit den illustrierten Beispielen richtig rechnet:

Abstrakte Anforderungen in konkrete Beispiele überführen

days rented	regular price
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

Abb. 10–3
Klare Anforderungen: links die Ausleihdauer, rechts der erwartete Preis

Erster Testlauf ...

Dann speichern wir die Tabelle als HTML-Dokument `alltests.html` und starten FITs `FileRunner` mit diesen zwei Programmargumenten: Eingabe- und Ausgabedateipfad für den FIT-Lauf:

Save as HTML Document

```
java fit.FileRunner alltests.html alltests-results.html
```

Wenn nun alles gut geht, teilt uns FIT das Ergebnis des Testlaufs mit:

```
0 right, 0 wrong, 0 ignored, 1 exceptions
```

... und der erste FIT-Report

Unser Stapellauf war erfolgreich, obwohl FIT uns hier eine Exception vorwirft. Schauen wir uns also zunächst an, was denn genau anliegt: Das Test-Framework hat uns während des Testlaufs eine Kopie unseres Dokuments mit Namen `alltests-results.html` ins Arbeitsverzeichnis gestellt und darin direkt seinen Befund vermerkt:

Abb. 10–4
Die Testresultate werden auf den nächsten Seiten wie folgt in der Marginalie dargestellt:

0 right,
0 wrong,
0 ignored,
1 exceptions

days rented java.lang.ClassNotFoundException: days rented at java.lang.Class.forName0(Native Method) at java.lang.Class.forName(Class.java:140) at fit.Fixture.doTables(Fixture.java:69) at fit.FileRunner.process(FileRunner.java:29) at fit.FileRunner.run(FileRunner.java:22) at fit.FileRunner.main(FileRunner.java:17)	regular price
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

Im Störungsfall

In der ersten Tabellenzelle, in Gelb hinterlegt, tritt eine Exception auf: Es existiert keine Klasse namens `days rented`. Beheben wir gleich ... Erhalten Sie dagegen eine andere Meldung, liegt es wohl daran, dass

- die `fit.FileRunner`-Klasse, d.h. `fit.jar`, nicht im `CLASSPATH` steht,
- die Datei `alltests.html` oder `alltests-results.html` nicht gelesen bzw. geschrieben werden kann (Arbeitsverzeichnis okay?) oder
- Ihr HTML-Export den FIT-Parser irgendwie durcheinander bringt. In diesem Fall tippen Sie das HTML probeweise von Hand ein:

```
<html>
  <head> <title> FIT-Tests </title> </head>
  <body>
    <table border cellspacing="0" cellpadding="3">
      <tr> <td> days rented </td> <td> regular price </td> </tr>
      <tr> <td> 1 </td> <td> 1.50 </td> </tr>
      <tr> <td> 2 </td> <td> 1.50 </td> </tr>
      <tr> <td> 3 </td> <td> 1.50 </td> </tr>
      <tr> <td> 4 </td> <td> 3.00 </td> </tr>
      <tr> <td> 5 </td> <td> 4.50 </td> </tr>
    </table>
  </body>
</html>
```

Interpretation der Testdaten

Weiter mit unserer `ClassNotFoundException`: Wir müssen FIT erklären, wie die Zelleninhalte unserer HTML-Tabelle denn zu verarbeiten sind. Die Verbindung zum Testcode wird dabei über die Namen in der Tabelle hergestellt, wobei die erste Tabellenzelle besondere Bedeutung besitzt: Sie sagt dem Framework genau, welche Klasse zuständig ist, das Format und den Inhalt der jeweiligen Tabelle zu interpretieren. Diese zentrale Klasse ist die *Fixture*.

Test-Fixture

In unserem Beispiel wollen wir das Preismodell testen und nennen unsere Fixture-Klasse deshalb einfacherweise `Pricing`. Die Klasse wird wie alle Beispiele im Buch im Default-Package landen, deshalb entfällt hier die Paketangabe. Ansonsten muss natürlich der vollqualifizierte Klassenname inklusive Klassenpaket im Tabellenkopf spezifiziert sein, denn sonst kann die Klasse nicht geladen werden.

Fügen wir den Klassennamen unserer Wahl als Tabellenkopf ein, speichern das HTML-Dokument und lassen FIT erneut darüberlaufen, so zeigt sich ein neues Bild:

Pricing

```
java.lang.ClassNotFoundException: Pricing
        at java.net.URLClassLoader$1.run(URLClassLoader.java:198)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(URLClassLoader.java:186)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:299)
        at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:265)
        at java.lang.ClassLoader.loadClass(ClassLoader.java:255)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:315)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Class.java:140)
        at fit.Fixture.doTables(Fixture.java:69)
        at fit.FileRunner.process(FileRunner.java:29)
        at fit.FileRunner.run(FileRunner.java:22)
        at fit.FileRunner.main(FileRunner.java:17)
```

days rented	regular price
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

Abb. 10–5
Angabe des Tabelleninterpreters:

0 right,
0 wrong,
0 ignored,
1 exceptions

Selbstverständlich, die Klasse gibts noch nicht. Das alte Spiel:

```
public class Pricing {
}
```

In FIT werden alle Fixture-Objekte dynamisch via Reflection erzeugt, weshalb sie einen öffentlichen Defaultkonstruktor besitzen müssen. Das ist hier jedoch kein Problem, da wir die Klasse `Object` beerben. Existiert ein solcher nicht, gibt es eine entsprechende Exception.

Die Test-Fixture

Das nächste Feedback auf unserem Weg ist eine `ClassCastException`. Und obwohl es so aussehen mag, als machten wir keinerlei Fortschritt gegenüber der vorausgehenden `ClassNotFoundException`, bringt uns dieser Schritt ein ordentliches Stück weiter:

Abb. 10–6 *Unsere Klasse wird wenigstens schon einmal erfolgreich geladen:*

0 right, 0 wrong, 0 ignored, 1 exceptions

Pricing java.lang.ClassCastException at fit.Fixture.doTables(Fixture.java:69) at fit.FileRunner.process(FileRunner.java:29) at fit.FileRunner.run(FileRunner.java:22) at fit.FileRunner.main(FileRunner.java:17)	
days rented	regular price
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

FIT findet unsere `Pricing`-Klasse, instanziiert sie und versucht darauf vergeblich einen Downcast auf die Framework-Basisklasse `Fixture`, die ihrerseits das Protokoll zur Traversierung von Tabellen festlegt. Wollen wir unsere Test-Fixture erfolgreich ins Framework einklinken, müssen wir sie von `Fixture` oder einer ihrer Unterklassen ableiten. Eine solche FIT-Fixture ist `ColumnFixture`:

```
public class Pricing extends fit.ColumnFixture...
```

Ein wiederholter Probelauf bringt nun folgendes Ergebnis:

Abb. 10–7 *Je weiter wir uns zu den Rights und Wrongs vorkämpfen, desto besser:*

0 right, 0 wrong, 10 ignored, 2 exceptions

Pricing	
days rented java.lang.NoSuchFieldException: days rented at java.lang.Class.getField0(Class.java:1735) at java.lang.Class.getField(Class.java:900) at fit.ColumnFixture.bindField(ColumnFixture.java:99) at fit.ColumnFixture.bind(ColumnFixture.java:84) at fit.ColumnFixture.doRows(ColumnFixture.java:16) at fit.Fixture.doTable(Fixture.java:82) at fit.Fixture.doTables(Fixture.java:72) at fit.FileRunner.process(FileRunner.java:29) at fit.FileRunner.run(FileRunner.java:22) at fit.FileRunner.main(FileRunner.java:17)	regular price java.lang.NoSuchFieldException: regular price at java.lang.Class.getField0(Class.java:1735) at java.lang.Class.getField(Class.java:900) at fit.ColumnFixture.bindField(ColumnFixture.java:99) at fit.ColumnFixture.bind(ColumnFixture.java:84) at fit.ColumnFixture.doRows(ColumnFixture.java:16) at fit.Fixture.doTable(Fixture.java:82) at fit.Fixture.doTables(Fixture.java:72) at fit.FileRunner.process(FileRunner.java:29) at fit.FileRunner.run(FileRunner.java:22) at fit.FileRunner.main(FileRunner.java:17)
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

Testeingaben

Unsere erste Tabellenzeile haben wir soweit unter Dach und Fach. Dass sie weiß bleibt und keinen Stacktrace auswirft, ist Zeichen dafür, dass dort zumindest mal nichts schief gegangen ist. Doch werden nun unsere Tabellenwerte ignoriert und deshalb allesamt in Grau getrübt. Die beiden `NoSuchFieldExceptions` in der zweiten Zeile geben uns dazu einen Hinweis: Über die Namen in dieser Zeile werden die Daten aus der HTML-Tabelle an gleichnamige Elemente in unserem Java-Code gebunden.

Über die Namen im HTML erfolgt die Verbindung zu unserem Fixture-Code

Nehmen wir die erste Spalte, die unsere Eingabedaten darstellt: Die `ColumnFixture`-Klasse bildet von Haus aus die einzelnen Tabellenspalten auf die Felder und Methoden ihrer Klasse ab. Bevor FIT unsere Fixture jedoch mit den Werten der Tabelle parameterisieren kann, muss unsere Klasse dafür ein korrespondierendes *öffentliches* Feld von passendem Datentyp anbieten:

```
public class Pricing extends fit.ColumnFixture {
  public int daysRented;
}
```

Das Framework sucht sich nun das Feld, das den Spaltennamen trägt, und füllt es beim Lesen einer Tabellenzeile mit der gelesenen Eingabe. Bevor wir das aber weiter ausprobieren, passen wir den Spaltennamen im HTML noch an unsere Namenskonvention aus der Java-Welt an:

Pricing	
daysRented	regular price java.lang.NoSuchFieldException: regular price at java.lang.Class.getField0(Class.java:1735) at java.lang.Class.getField(Class.java:900) at fit.ColumnFixture.bindField(ColumnFixture.java:99) at fit.ColumnFixture.bind(ColumnFixture.java:84) at fit.ColumnFixture.doRows(ColumnFixture.java:16) at fit.Fixture.doTable(Fixture.java:82) at fit.Fixture.doTables(Fixture.java:72) at fit.FileRunner.process(FileRunner.java:29) at fit.FileRunner.run(FileRunner.java:22) at fit.FileRunner.main(FileRunner.java:17)
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

Abb. 10–8
Umbenennung der ersten Tabellenspalte:

0 right,
0 wrong,
5 ignored,
1 exceptions

Wieder sind wir ein Stückchen fortgeschritten: Statt zehn ignorierter Tabellenzellen, bleiben uns noch fünf. Bisher stellt der Test mit den Eingaben auch überhaupt nichts an. Machen wir uns also ans Werk!

Erwartete und tatsächliche Resultate

Unsere zweite Tabellenspalte stellt die Ergebnisse der Preisfunktion in Abhängigkeit von der Ausleihdauer dar. Um die Werte dieser Spalte mit den Werten, wie sie unser Programm berechnet, zu vergleichen, benötigt unsere Test-Fixture noch eine Abfragemethode passenden Rückgabetyps:

```
public class Pricing...
  public double regularPrice() {
    return 0;
  }
}
```

Zusätzlich müssen wir die Tabellenspalte so nennen, wie auch unsere Methode heißt. Das bedeutet: a) Namenskonvention von Methodennamen einhalten und b) Methodennamen mit Klammern abschließen.

Erstmalig ist unser Akzeptanztest nun so weit gediehen, dass er uns das FIT-Äquivalent des roten JUnit-Balkens zum Ausdruck bringt:

Abb. 10–9 *Umbenennung der zweiten Tabellenspalte:*

0 right, 5 wrong, 0 ignored, 0 exceptions

Pricing	
daysRented	regularPrice()
1	1.50 expected / 0.0 actual
2	1.50 expected / 0.0 actual
3	1.50 expected / 0.0 actual
4	3.00 expected / 0.0 actual
5	4.50 expected / 0.0 actual

Was geschieht nun genau hinter den Kulissen?

- Zeile 1 erzeugt ein Fixture-Objekt der Klasse `Pricing`.
- Zeile 2 bindet die Spalte 1 an die Member-Variable `daysRented` und die Spalte 2 an die Member-Funktion `regularPrice`.
- Zeile 3 befüllt die `daysRented`-Variable mit dem Tabellenwert `1`, ruft die Funktion `regularPrice` und vergleicht ihren Rückgabewert mit dem Tabellenwert `1.50`.
- Zeile 4 entsprechend mit den Werten `2` und `1.50` usw.

Was zum Erfolg noch zu tun bleibt, ist Fixture und Code zu verbinden:

Unser Fixture-Code stellt die Verbindung zum Prüfling her

```
public class Pricing...
  public double regularPrice() {
    return Price.REGULAR.getCharge(daysRented).getAmount();
  }
}
```

... und abzuwarten, ob uns das Glück in Grün heute hold ist:

Pricing	
daysRented	regularPrice()
1	1.50
2	1.50
3	1.50
4	3.00
5	4.50

Abb. 10–10
Am Ziel:

`5 right,`
`0 wrong,`
`0 ignored,`
`0 exceptions`

Damit ist der erste Test abgehakt und wir können uns fragen, was wir unter Umständen vergessen haben zu testen. Ich will jedoch gar nicht so sehr auf weitere Tabellenzeilen hinaus als auf eine dritte Spalte:

Pricing		
daysRented	regularPrice()	newRelease() java.lang.NoSuchMethodException: Pricing.newRelease() at java.lang.Class.getMethod0(Class.java:1756) at java.lang.Class.getMethod(Class.java:963) at fit.ColumnFixture.bindMethod(ColumnFixture.java:95) at fit.ColumnFixture.bind(ColumnFixture.java:82) at fit.ColumnFixture.doRows(ColumnFixture.java:16) at fit.Fixture.doTable(Fixture.java:82) at fit.Fixture.doTables(Fixture.java:72) at fit.FileRunner.process(FileRunner.java:29) at fit.FileRunner.run(FileRunner.java:22) at fit.FileRunner.main(FileRunner.java:17)
1	1.50	2.00
2	1.50	2.00
3	1.50	3.75
4	3.00	5.50
5	4.50	7.25

Abb. 10–11
Weitere Tabellenspalte:

`5 right,`
`0 wrong,`
`5 ignored,`
`1 exceptions`

... und den entsprechenden Fixture-Code dafür:

```
public class Pricing...
  public double newRelease() {
    return Price.NEWRELEASE.getCharge(daysRented).getAmount();
  }
}
```

10.4 ... zum ausführbaren Anforderungsdokument

Haben Sie's eigentlich bemerkt? Das war ein siebenseitiger Lerntest! Die Testläufe waren uns zur Exploration des FIT-Frameworks nützlich und ihre Feedbacks haben uns jeweils zum nächsten Schritt geführt.

Akzeptanztests dokumentieren Anforderungen

Unsere Akzeptanztests sollten jetzt alle laufen, so dass ich erneut auf den Dokumentencharakter der FIT-Tests zurückkommen möchte. Zahlen und Worte ergänzen sich: Meist reichen ein bis zwei Sätze aus, um die Tests verständlicher zu machen. Ebenso werden die oft recht abstrakten Anforderungen am Zahlenbeispiel sehr viel greifbarer:

Abb. 10–12 *FIT-Tests als ausführbares Anforderungsdokument:*

`10 right, 0 wrong, 0 ignored, 0 exceptions`

Verleihpreise berechnen

Die Verleihkosten pro Film staffeln sich danach, wie lange ein Film geliehen wird. Dabei sind zwei Preiskategorien zu unterscheiden:

1. Filme kosten regulär für die ersten drei Tage insgesamt 1,50 Euro. Für jeden weiteren Ausleihtag kommen 1,50 Euro dazu.
2. Neuerscheinungen kosten für zwei Tage 2 Euro. Ab dem dritten Ausleihtag pro Tag zusätzliche 1,75 Euro.

Hier zeigen wir die Preisfunktionen mit Beispieldaten:

Pricing		
daysRented	regularPrice()	newRelease()
1	1.50	2.00
2	1.50	2.00
3	1.50	3.75
4	3.00	5.50
5	4.50	7.25

Lücke von der Anforderungsanalyse zum Akzeptanztest schließen

Das Testmodell von FIT, die Tests als lesbares Dokument zu verfassen und dieses Dokument im Testverlauf mit dem Teststatus zu annotieren, verbindet so effektiv die Anforderungsanalyse mit dem Akzeptanztest:

- **Kommunikation.** Das Dokument unterstützt die Zusammenarbeit zwischen den Domänenexperten, die die Fachlichkeit kennen, und den Entwicklern, die sie realisieren. FIT bringt sie faktisch *auf einer Seite* zusammen und integriert ihre unterschiedlichen Blickwinkel.
- **Exploration.** Eine Idee von FIT ist, dass die Testdaten unabhängig vom Testcode änderbar sein müssen. Dadurch versetzen wir die Domänenexperten in die Lage, selbst neue Tests zu schreiben und auch direkt auszuführen: Vorausgehende Akzeptanztests steuern gezielt die Testgetriebene Entwicklung. Nachgelagerte Tests helfen, die Fähigkeiten der schon entwickelten Software zu erforschen und unmittelbare Rückschlüsse auf ihre Qualität zu ziehen.

10.5 Die drei Basis-Fixtures

Die vorausgehenden Abschnitte sollten Ihnen einen ersten Eindruck gegeben haben, was FIT als Framework leistet und wie es funktioniert. Später werden wir noch tiefer in die Interna eintauchen. Jetzt nehmen wir den Fokus jedoch erst einmal wieder zurück und widmen uns den drei grundlegenden Arten von Fixtures:

Logiktest

- **`ColumnFixture`** ist uns ja bereits über den Weg gelaufen. Sie prüft Eingabedaten gegen erwartete Ausgabedaten und eignet sich daher besonders zum Test der Geschäftslogik.

Mengentest

- **`RowFixture`** stellt eine Ergebnismenge dar, wie wir sie zum Beispiel in einer Collection o.Ä. erwarten. Sie sammelt die Ergebnisse ein und vergleicht sie dann auf Vollständigkeit und Korrektheit.

Interaktionstest

- **`ActionFixture`** orientiert sich näher an der Benutzerschnittstelle und ist szenariobasiert. Sie führt eine Reihe von Kommandos aus und wird deshalb häufig für Oberflächentests verwendet.

Diese Fixtures gehören zum Kern und Lieferumfang des Frameworks. Je nachdem, wie unsere Tabellenformate interpretiert werden sollen, leiten wir unsere anwendungsspezifischen Fixtures von diesen drei Basisklassen ab. In Kombination reichen `ColumnFixture`, `RowFixture` und `ActionFixture` aus, um die Mehrzahl aller Testfälle abzudecken. Ansonsten ist FIT aber auch sehr einfach um neue Fixture-Typen erweiterbar, ein wahrliches Meisterstück von Framework-Entwurf, das sich lohnt, eingehend studiert zu werden.

10.6 »ActionFixture«

Wie spielen wir verschiedene Benutzerszenarien durch?

Interaktionstest

Die `ActionFixture` führt pro Tabellenzeile ein Kommando oder eine Benutzeraktion aus. Um ein Programm mit Testszenarien anzutreiben, um so beispielsweise eine Interaktion mit der grafischen Oberfläche zu emulieren, bietet sie vier Aktionen:

Vier Aktionen

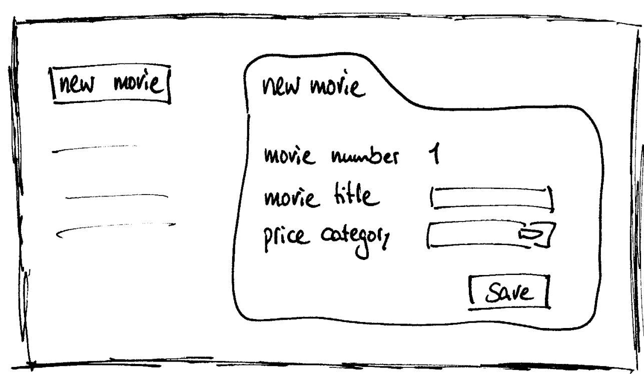
- **`press`** simuliert die Betätigung einer Schaltfläche, zu Neudeutsch: das Klicken auf einen Button.
- **`enter`** entspricht der Tastatureingabe in einem Eingabefeld.
- **`check`** vergleicht den Inhalt eines Feldes mit einem erwarteten Wert.

Außerdem müssen wir der `ActionFixture` mitteilen, welches Fenster oder Formular wir aktiv ansprechen wollen:

- **`start`** dient der Navigation zu einer bestimmten Bildschirmmaske.

Was wird ein Benutzer mit unserem Verleihsystem anstellen wollen? Da man mit einer leeren Videothek nicht viel Geld verdienen kann, will unser Kunde zu Anfang zunächst ein paar Filme eingeben können. Das entsprechende Formular stellt er sich in etwa so vor:

Abb. 10–13 Eingabeformular für die Filmverwaltung



Unser Testszenario läuft dann wie folgt: Wir starten auf dem Formular für die Filmverwaltung und lösen dort die Aktion zur Neuanlage aus. Daraufhin erwarten wir, dass eine neue Filmnummer vergeben wird. Wir geben den DVD-Titel und die Preiskategorie ein und fertig!

Abb. 10–14 Eingaben in ActionFixture-Form

Inventar

Zur Neuanlage benötigen wir den Filmtitel der DVD und ihre Preiskategorie. Die Filmnummern werden aufsteigend vom System vergeben.

fit.ActionFixture		
start	MovieAdministration	
press	new movie	
check	movie number	1
enter	movie title	Pulp Fiction
enter	price category	regular price
press	save	

HTML-gesteuerte Skripting-Möglichkeiten

In der Tabellenspalte eins versteht die `ActionFixture` die Befehle `start`, `press`, `check` und `enter`. Spalte zwei und drei werden befehlsabhängig interpretiert. Wie sieht nun unsere dazugehörige Fixture-Klasse aus? Sie muss Methoden anbieten entsprechend den Namen der zweiten Tabellenspalte, mit dem Unterschied, dass die Tabelleneinträge zu erlaubten Java-Namen gewandelt werden. Das bedeutet, der Wert `new movie` wird zu `newMovie`. Dieses Format wird *Camel Case* genannt, da die umgesetzte Schreibweise an die Höcker eines Kamels erinnert.

Namenskonversion ins Camel-Case-Format

Schauen wir uns die Semantik der `ActionFixture` im Einzelnen an:

start

Delegation zum Akteur

Die `start`-Aktion bestimmt, welche Klasse die folgenden Kommandos ausführen soll. Dieser so genannte *Akteur* folgt mit vollqualifiziertem Klassennamen in Tabellenspalte zwei: Damit die `ActionFixture` sich eine Instanz dieser Klasse erzeugen kann, muss der Akteur sowohl den Defaultkonstruktor besitzen als auch Unterklasse von `Fixture` sein:

```
public class MovieAdministration extends fit.Fixture {
```

press

Benutzeraktion auslösen

Die `press`-Aktion löst eine Aktion in unserem aktuellen Akteur aus. Welche das sein soll, wird durch den Namen der zu rufenden Methode in der zweiten Tabellenspalte festgelegt: `press` erwartet eine Methode ohne Parameter und ohne Rückgabewert:

```
  public void newMovie() {
  }

  public void save() {
  }
```

check

Werte vergleichen

Die `check`-Aktion vergleicht den Wert in der dritten Tabellenspalte mit dem Resultat der in Spalte zwei benannten Methode: `check` erwartet dazu eine nicht parameterisierte Methode passenden Rückgabetyps:

```
  public int movieNumber() {
    return 0;
  }
```

enter

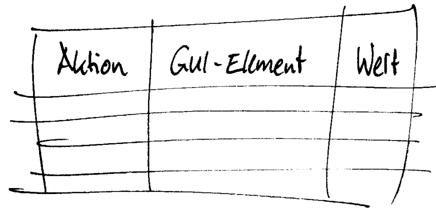
Werte eingeben

Die `enter`-Aktion schließlich sendet den Wert aus Tabellenspalte drei als Parameter an die Methode aus Spalte zwei: `enter` erwartet daher eine Methode mit passendem Parametertyp, aber ohne Rückgabewert:

```
  public void movieTitle(String title) {
  }

  public void priceCategory(String category) {
  }
}
```

Abb. 10–15
Tabellenschema der ActionFixture



Eine Tabellenzeile der `ActionFixture` folgt also diesem Muster:

- **Spalte 1** enthält die auszuführende Benutzeraktion,
- **Spalte 2** das interessierende Element der Benutzeroberfläche und
- **Spalte 3** im Fall von `enter` einen gegebenen Wert und im Fall von `check` den erwarteten Wert.

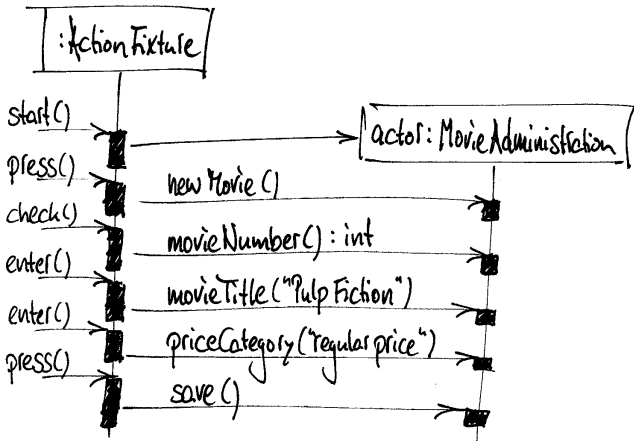
Testszenarien dokumentieren

Alles, was nach der dritten Spalte folgt, ignoriert die `ActionFixture`. Diese Tabellenspalten eignen sich deshalb bestens zur Dokumentation der Testszenarien.

Interpretationsreihenfolge

Tabellen werden zeilenweise abgearbeitet: von oben nach unten und von links nach rechts. Unser Szenario aus Abb. 10–14 spielt sich zum Beispiel so ab:

Abb. 10–16
Resultierendes Testszenario



Wobei sich die Spalten auf unterschiedliche `Fixture`-Klassen beziehen:

- **Spalte 1** benennt Methoden der `ActionFixture`-Klasse selbst,
- **Spalte 2** im Fall des `start`-Befehls die `Fixture`-Klasse des Akteurs des betreffenden Benutzerfensters und im Fall von `press`, `check` oder `enter` die an diesen Akteur zu delegierenden Methoden und
- **Spalte 3** die für diese Methode gegebenen oder erwarteten Werte.

10.7 Richtung peilen, Fortschritt erzielen

Akzeptanztests geben der Entwicklung im Großen ihre Richtung

Wie umfassendere Unit- oder Integrationstests sind uns Akzeptanztests hervorragende Wegweiser: Sie zeigen uns, was eigentlich unser Ziel ist, und verraten uns, wann wir an diesem Ziel sind. Da sie sich dabei an den fachlichen Kundenanforderungen orientieren, zum Beispiel einem Anwendungsfall oder einer Benutzergeschichte, können wir aus ihnen auch die Informationen gewinnen, welche Unit Tests wir tatsächlich schreiben müssen. Das ist viel geschickter, als nur darauf zu vertrauen, dass uns die Testfälle schon auf geheimnisvolle Weise zufallen werden. So entsteht das enge Zusammenspiel von Akzeptanz- und Unit Tests:

Zusammenspiel von Akzeptanz- und Unit Tests

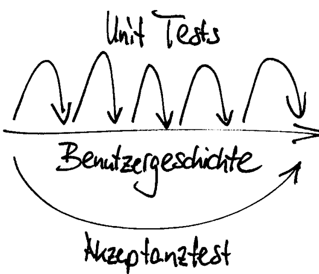


Abb. 10–17 *Testgetriebene Entwicklung im Kleinen und im Großen*

Lassen wir nun FIT über unsere `ActionFixture` laufen, erkennen wir, dass der `check` schief geht. Die restlichen Aktionen sind in Ordnung, da sie nichts weiter prüfen, außer dass keine Exceptions auftreten. Unsere Aufgabe wird es sein, diesen Akzeptanztest zu erfüllen. Dabei kommen erneut die Techniken aus den Kapiteln 5.14-5.17 ins Spiel: Mit »Fake it ’til you make it« von Rot auf Grün in drei Sekunden:

```
public class MovieAdministration...
  public int movieNumber() {
    return 1;
  }
}
```

Unit Tests sichern den Fortschritt im Kleinen

Um unseren Akzeptanztest jetzt wirklich zu erfüllen, brechen wir aus seiner Vorgabe kleine fokussierte Unit Tests heraus und ziehen diese Stück um Stück zur Entwicklung der benötigten Funktionalität heran. Sie erkennen schon die Selbstähnlichkeit:

- Ein **grün➠rot➠grün** auf Akzeptanztestebene schließt eine wichtige Feedbackschleife von einer Kundenanforderung zu ihrer Erfüllung,
- **grün➠rot➠grün** auf Ebene der Unit Tests eine wichtige Schleife von einer Entwurfsentscheidung zu ihrer Erfüllung in getestetem Code.

Die Fixture als erster Verwender

Wie können wir nun ausdrücken, dass wir die Filmnummern eigentlich in aufsteigender Folge vergeben müssen? Wir können uns wünschen, was wir brauchen! Wo wäre ein geeigneterer Ort, als das Design direkt aus den Anforderungen des Akzeptanztests heraus zu entwickeln und die dabei entdeckten Klassen sofort in unsere Fixture einzuspannen? Was uns fehlt, ist eine Art Fabrik zur Erzeugung der Filmnummern:

```
public class MovieAdministration...
  private NumberSequence sequence = new NumberSequence();
```

Unsere Anwender erwarten die Vergabe einer aktuellen Filmnummer, wenn sie die Aktion `new movie` auswählen:

```
  public void newMovie() {
    movieNumber = sequence.nextNumber();
  }
```

Speichern wir also diese Filmnummer:

```
  private int movieNumber;
```

... und liefern sie zum `movie number`-Check aus unserer Fixture zurück:

```
  public int movieNumber() {
    return movieNumber;
  }
}
```

Aus dem Akzeptanztest Unit Tests herausbrechen

Mit diesem Design in der Hand machen wir uns an die Unit Tests:

```
public class NumberSequenceTest extends TestCase {
  public void testSequence() {
    NumberSequence sequence = new NumberSequence();
    assertEquals(1, sequence.nextNumber());
    assertEquals(2, sequence.nextNumber());
    assertEquals(3, sequence.nextNumber());
  }
}
```

... und stürzen uns danach auf die Fabrikklasse:

```
public class NumberSequence {
  private int nextNumber = 1;

  public synchronized int nextNumber() {
    return nextNumber++;
  }
}
```

So! Damit haben wir sowohl die `press`- als auch die `check`-Aktion zum Leben erweckt, unsere Unit Tests sind zufrieden und FIT beschwert sich auch nicht mehr:

fit.ActionFixture		
start	MovieAdministration	
press	new movie	
check	movie number	1
enter	movie title	Pulp Fiction
enter	price category	regular price
press	save	

Abb. 10–18
Erste Grünfläche

Verbleiben die drei abschließenden Aktionen: 2x `enter` und 1x `press`, die für sich genommen zwar nichts prüfen, für Erfolg oder Fehlschlag unseres Tests jedoch mitverantwortlich sind: Zusammen fügen sie dem Verleihsystem erst die neue DVD hinzu, eine Tatsache, die wir später noch mit weiteren FIT-Tests abklopfen wollen.

10.8 Fixture wachsen lassen, dann Struktur extrahieren

Eine eher fortgeschrittene Technik zum gezeigten Vorgehen ist die Idee, die Fixture-Klasse noch stärker als Designwerkzeug zur explorativen Entwicklung auszunutzen: Anstatt ein mögliches Design in der Fixture nur anzudeuten und dann gänzlich außerhalb der Fixture mithilfe gezielter Unit Tests zu entwickeln, lässt sich der erste Code für ein neues Feature oft auch direkt an Ort und Stelle in der Fixture selbst entwickeln und anschließend durch *strukturschaffende* Refactorings extrahieren. Lassen wir uns weiter vom Fixture-Code leiten!

Die Fixture als Designwerkzeug

Durch Refactoring Struktur schaffen

Zunächst merken wir uns den Filmtitel und die Preiskategorie. Zwei Member-Variablen tun den Job:

```
public class MovieAdministration...
  private String movieTitle;
  private String priceCategory;

  public void movieTitle(String title) {
    movieTitle = title;
  }

  public void priceCategory(String category) {
    priceCategory = category;
  }
}
```

Erst mit Betätigung des Speichern-Knopfes wollen wir überhaupt neue Filme anlegen. Die Erzeugung der `Price`-Instanz lagern wir dabei lieber sofort in eine eigene Methode aus:

```
    public void save() {
      Price price = getPrice(priceCategory);
    }
  }
```

Warum? Als Fabrikmethode enthält sie Logik und verdient deshalb ein paar JUnit-Tests:

```
  public class MovieAdministrationTest extends TestCase {
    public void testRegularPrice() {
      Price price = MovieAdministration.getPrice("regular price");
      assertSame(Price.REGULAR, price);
    }

    public void testUnknownCategory() {
      assertNull(MovieAdministration.getPrice("unknown"));
    }
  }
```

Warum die Fixture testen?

Vielleicht staunen Sie, warum wir anfangen, unsere Fixture zu testen? Immerhin handelt es sich um Testcode. Da wir den Code der Fixture später aber extrahieren wollen, entwickeln wir jetzt schon Tests dafür:

```
  public class MovieAdministration...
    public static Price getPrice(String category) {
      return "regular price".equals(category)
        ? Price.REGULAR : null;
    }
```

Als Datenstruktur für die Filme habe ich mich für eine `Map` entschieden. Das ist nicht die einfachste Lösung, die funktionieren könnte, aber in Kürze benötigen wir sowieso den assoziativen Zugriff:

```
    private Map movies = new HashMap();

    public void save() {
      Price price = getPrice(priceCategory);
      Movie movie = new Movie(movieTitle, price);
      Integer key = new Integer(movieNumber);
      movies.put(key, movie);
    }
  }
```

10.9 Nichts als Fassade

Was fällt beim ersten Blick auf die `MovieAdministration`-Fixture auf? Im Testcode haben sich nun Logik und Datenstrukturen breit gemacht. Das war ja auch der erwünschte Effekt, allerdings wollen wir nicht, dass die Fixture für die Verwaltung von Filmen verantwortlich bleibt. Diese Aufgabe fällt ganz klar in die Zuständigkeit der Anwendung. Die Fixture dient vielmehr nur als Adapter zwischen den Testdaten aus der Tabelle und dem System auf dem Prüfstand:

Keine Logik und eigene Datenstrukturen in der Fixture

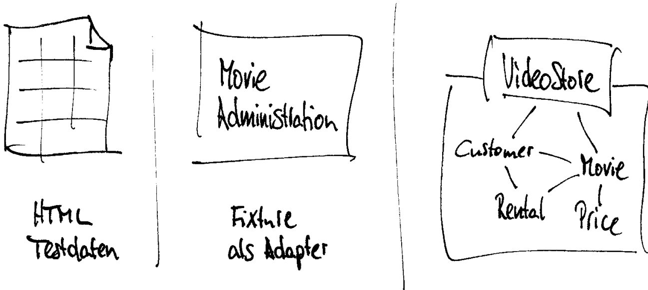


Abb. 10–19
Fixtures sind bloß Adapter

Als solche müssen Fixtures so dumm und so kurz wie möglich bleiben. Mehr als drei Zeilen Adaptercode sind zu viel.

Mehr als drei Zeilen pro Methode sind verdächtig

Fixture-Klassen sollen keinerlei Logik enthalten, sondern nichts als Fassade sein.

In aller Konsequenz impliziert dies auch, dass wir jederzeit Ausschau halten müssen nach verborgenen Abstraktionen in unserem Testcode, die eigentlich ihren Weg in den Anwendungscode hätten finden sollen. Sehen wir unseren Code mit Hinsicht darauf durch:

Versteckte Abstraktionen aufspüren

Unsere Fixture geht offensichtlich mit zwei Klassen schwanger: Die erste würde ich `VideoStore` taufen, da sie uns die Filme verwaltet. Die andere Klasse könnte die neue Heimat unserer `getPrice`-Methode werden. Doch ziehen wir erst einmal eine Fassade hoch:

```
public class MovieAdministration...
  private VideoStore store = new VideoStore();

  public void save() {
    store.newMovie(movieNumber, movieTitle, priceCategory);
  }
}
```

So weit, so gut! Über die Fixture haben wir unser Wunsch-API für den VideoStore gefunden. Durch welchen Test können wir das anstehende Refactoring motivieren?

```
public class VideoStoreTest extends TestCase {
  public void testMovieHandling() {
    VideoStore store = new VideoStore();
    Movie movie = store.newMovie(1, "don't care", "don't care");
    assertSame(movie, store.getMovie(1));
    assertNotNull(movie);
  }
}
```

Anwendungsfassade hochziehen

Um die newMovie-Methode bzw. ihren Seiteneffekt testen zu können, ist es notwendig, das Movie-Objekt zum Rückgabewert zu machen, um so dem VideoStore mittels getMovie Löcher in seinen Bauch zu fragen. Durch Extraktion kommt eine Anwendungsfassade zum Vorschein:

```
public class VideoStore {
  private Map movies = new HashMap();

  public Movie newMovie(int number, String title,
                        String category) {
    Price price = Prices.getPrice(category);
    Movie movie = new Movie(title, price);
    Integer key = new Integer(number);
    movies.put(key, movie);
    return movie;
  }

  public Movie getMovie(int number) {
    Integer key = new Integer(number);
    return (Movie) movies.get(key);
  }
}
```

Die zweite Extraktion, die ich oben mit hineingeschmuggelt habe:

```
public class Prices {
  public static Price getPrice(String category) {
    return "regular price".equals(category)
      ? Price.REGULAR : null;
  }
}
```

10.10 Die Fixture als zusätzlicher Klient

Nach Review der Fixture-Klasse zeigt sich noch ein schwacher Punkt: `MovieAdministration` geht an unserer Fassade vorbei direkt auf die `NumberSequence`, um die nächste Filmnummer zu erfahren. Wenn wir uns aber vor Augen halten, dass unsere Fixtures eigentlich, genauso wie die Benutzerschnittstelle, Klienten des Anwendungskerns sind, wird klar, dass die Logik in die tiefer liegende Schicht wandern muss. In einer Schichtenarchitektur ergibt sich also dieses Bild:

Die Fixture als Alternative zur realen grafischen Benutzerschnittstelle

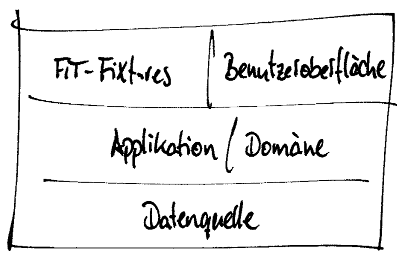


Abb. 10–20
FIT-Tests gegen die Applikations- und Domänenlogik

Natürlich kann FIT auch direkt auf der Benutzeroberfläche aufsetzen, doch dazu kommer wir erst später. Der jetzige Fokus soll sein, für die Oberfläche wie für die Fixtures eine Applikationsfassade auszubauen. Diese Fassade übersetzt die Datentypen zwischen der Benutzungsschnittstelle und der Applikations-/Domänenlogik, minimiert damit die Abhängigkeiten und bietet uns ein High-Level-API zum Test der Systemfunktionalität.

Applikations- und Domänenlogik hinter einer Fassade verstecken

Da die Filmnummer auch für den Benutzer an der Oberfläche sichtbar sein wird und wir in der Benutzerschnittstelle keine direkte Abhängigkeit auf die `NumberSequence` haben wollen, vereinnahmen wir sie hinter der `VideoStore`-Fassade im Anwendungskern:

```
public class VideoStore...
  private NumberSequence sequence = new NumberSequence();

  public int nextMovieNumber() {
    return sequence.nextNumber();
  }
}

public class MovieAdministration...
  public void newMovie() {
    movieNumber = store.nextMovieNumber();
  }
}
```

10.11 Aktion: Neue Aktion

Testsprache um eigene Vokabeln erweitern

Nachdem wir uns eine Weile um das Systemdesign gekümmert haben, kehren wir wieder zum Test zurück. Als Nächstes wollen wir unserer `ActionFixture` ein neues Schlüsselwort beibringen. Über Unterklassen können wir das Standardrepertoire um neue Aktionen erweitern, zum Beispiel um Interaktionsarten wie Selektion oder Drag & Drop.

In unserem Fall wäre es hübsch, wenn wir auch in der Testsprache ausdrücken könnten, dass Preiskategorien nicht aus einer Texteingabe resultieren, sondern aus einer Liste erlaubter Werte zu selektieren sind. Beispielsweise durch eine ausdrückliche `select`-Aktion:

Abb. 10–21
Neues Schlüsselwort: select

ExtendedActionFixture		
press	new movie	
check	movie number	2
enter	movie title	Kill Bill
select	price category	new release
press	save	
press	new movie	
check	movie number	3
enter	movie title	Reservoir Dogs
select	price category	regular price
press	save	

Damit unser `select`-Befehl nun auch noch richtig interpretiert wird, leiten wir unsere eigene `ExtendedActionFixture` von `ActionFixture` ab:

```
public class ExtendedActionFixture extends fit.ActionFixture {
}
```

Ferner müssen wir den Fixture-Namen im Tabellenkopf austauschen. Vergessen wir das, beginnt die unwissende Fixture zu jammern:

```
java.lang.NoSuchMethodException: fit.ActionFixture.select()
```

Aktionen sind Methoden der ActionFixture-Klasse

Das fehlende `select` können wir einfach auf das `enter` zurückführen, da das eine aus dem anderen hervorgegangen ist und sich die beiden Aktionen in ihrer Semantik auch gar nicht voneinander unterscheiden:

```
public class ExtendedActionFixture extends fit.ActionFixture {
  public void select() throws Exception {
    enter();
  }
}
```

In gleicher Art und Weise könnten wir check und press für ähnliche Aktionen missbrauchen. Wie wir in den Aktionen auch mit komplexen Datentypen umspringen können, sehen wir uns noch im Detail an.

Akteur bleibt gesetzt

Was noch auffällig ist: Die start-Aktion konnten wir uns dieses Mal sparen. Unsere MovieAdministration-Fixture ist durch die erste ActionFixture-Tabelle noch als Akteur für die Aktionen gesetzt. Unterbrechen wir unser Testszenario und setzen es in einer späteren Tabelle fort, bleibt der Befehlsempfänger der ActionFixture erhalten, solange die Tabellen am gleichen Testlauf teilnehmen. Eine weitere start-Aktion ist nur notwendig, wenn mehrere Akteure miteinander in Interaktion treten.

Was bleibt? Wir müssen unsere Prices-Fabrik dazu bekommen, dass sie mit neu erschienenen Filmen umgehen kann wie mit regulären. Im selben Zuge benennen wir auch gleich ihre Testfallklasse um von MovieAdministrationTest, wo die Tests von getPrice bislang klebten, in PricesTest:

```
public class PricesTest...
  public void testNewRelease() {
    Price price = Prices.getPrice("new release");
    assertSame(Price.NEWRELEASE, price);
  }
}
```

Schön, wenn man für neue Features nur neue Tests hinzufügen muss! An unserer Fabrikklasse sind einige Sanierungsarbeiten notwendig, wenn wir einen hässlichen if-then-else-Verteiler vermeiden wollen:

```
public class Prices {
  static private Map prices = new HashMap();
  static {
    prices.put("regular price", Price.REGULAR);
    prices.put("new release", Price.NEWRELEASE);
  }

  public static Price getPrice(String category) {
    return (Price) prices.get(category);
  }
}
```

Die JUnit- und FIT-Tests laufen grün! Nach programmierter Fixture und etablierter Testsprache ist es jetzt ein Leichtes, dem Testszenario noch weitere Schritte hinzuzufügen oder bestehende Tests zu variieren.

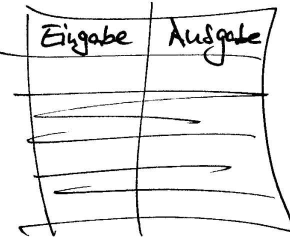
10.12 »ColumnFixture«

Wie testen wir die fachlichen Verarbeitungsregeln?

Logiktest

Die `ColumnFixture` haben wir ja schon im Eingangsbeispiel verwendet. Sie bildet das klassische Eingabe-Verarbeitung-Ausgabe-Schema ab: Jede ihrer Tabellenzeilen ist ein Testfall, bestehend aus Spalten mit gegebenen Eingabewerten und den nach ihrer Verarbeitung erwarteten Ergebniswerten:

Abb. 10–22 *Tabellenschema der ColumnFixture*



Jetzt, wo unser DVD-Verleih die ersten drei Filme im Programm hat, können wir mit dem Kerngeschäft beginnen: Wir platzieren ein paar Ausleihvorgänge im System und berechnen jeweils die Verleihkosten. Je Tabellenzeile spezifizieren wir eine Transaktion: welcher Film für wie viele Tage geliehen wird und wie viele Euro wir dafür kassieren.

Funktionales Testen von Tabelle zu Tabelle

Wir setzen dabei voraus, dass die verwendeten Filmnummern dem System bereits bekannt sind. Charakteristisch für funktionale Tests ist, dass eine Tabelle, im Beispiel unsere `ActionFixture`, den gewünschten Ausgangszustand herstellt, damit andere Tabellen, wie beispielsweise unsere `ColumnFixture`, darauf anschließend aufsetzen können:

Abb. 10–23 *Ausleihen in ColumnFixture-Form*

Verleih

Pro Ausleihvorgang wird die Filmnummer und Leihdauer gespeichert.
Aus Leihdauer und Preiskategorie berechnen sich die Kosten.

RentalEntry		
movieNumber	daysRented	charge()
1	3	1.50
3	5	4.50
2	3	3.75
3	2	1.50
1	1	1.50
3	6	6.00

`ColumnFixture` versucht, für jede ihrer Tabellenspalten entsprechende Felder und Methoden mit gleichem Namen zu finden und zu binden. Dies geschieht anhand der Spaltennamen in der zweiten Tabellenzeile:

Binden des Tabellenkopfs an Felder und Methoden der Fixture

- Instanzvariablen werden als Eingabespalte interpretiert
- Methodennamen (inklusive runder Klammern!) als Ergebnisspalte

Die so benannten Felder und Methoden muss unsere `ColumnFixture` *öffentlich* machen, sonst können sie nicht befüllt bzw. gerufen werden.

Pro Tabellenzeile werden also erst `movieNumber` und `daysRented` mit ihren Zelleninhalten gefüllt und anschließend `charge` überprüft:

```
public class RentalEntry extends fit.ColumnFixture {
  private VideoStore store = new VideoStore();

  public int movieNumber;
  public int daysRented;

  public double charge() {
    Rental rental = store.addRental(movieNumber, daysRented);
    return rental.getCharge().getAmount();
  }
}
```

`charge` delegiert wieder nur an die Fassade. Um alle Ausleihvorgänge zu speichern, erweitern wir dort um ein `addRental`:

```
public class VideoStoreTest...
  public void testRentalHandling() {
    VideoStore store = new VideoStore();
    store.newMovie(7, "Seven Samurai", "regular price");
    Rental rental = store.addRental(7, 42);
    assertEquals("Seven Samurai", rental.getMovieTitle());
  }
}

public class VideoStore...
  private List rentals = new ArrayList();

  public Rental addRental(int movieNumber, int daysRented) {
    Movie movie = getMovie(movieNumber);
    Rental rental = new Rental(movie, daysRented);
    rentals.add(rental);
    return rental;
  }
}
```

10.13 Fixture-Interkommunikation

Probieren wir jetzt einen Testlauf, hagelt es `NullPointerExceptions`. `RentalEntry` findet offenbar einen leeren Laden vor:

```
java.lang.NullPointerException
  at Rental.getCharge(Rental.java:11)
  at RentalEntry.charge(RentalEntry.java:10)
```

Zustand bei funktionalen Tests durchschleppen

Kein Wunder, wenn alle Fixtures ihr eigenes `VideoStore`-Objekt haben. Damit unsere `RentalEntry`-Tabelle auch tatsächlich von dem Zustand profitiert, den die `MovieAdministration`-Tabelle schon aufgebaut hat, müssen wir tabellenübergreifende Zustände von Fixture zu Fixture durchschleppen. Die globale Variable der Objektorientierung ist das Singleton Pattern:

```
public class RentalEntry...
  private VideoStore videostore = SystemUnderTest.instance();
}

public class MovieAdministration...
  private VideoStore videostore = SystemUnderTest.instance();
}
```

Da die Lebenszeit der `Fixture`-Objekte auf ihre Tabelle beschränkt ist, wir den `VideoStore` selbst aber nicht zum Singleton machen wollen, bietet sich eine Extraklasse an:

```
public class SystemUnderTest {
  private static VideoStore instance = new VideoStore();

  public static VideoStore instance() {
    return instance;
  }
```

Möglichkeit zum Setup und Teardown

Diese Klasse enthält die Referenz zum im Test befindlichen System und nimmt mit der Zeit weitere Hilfsmethoden für den Systemtest auf. Häufige Kandidaten sind ein `setUp` und `tearDown` auf Systemebene sowie oft auch eine Möglichkeit zum `reset`:

```
  public static void reset() {
    instance = new VideoStore();
  }
}
```

Natürlich müssen wir uns gegen zukünftige `NullPointerExceptions` besser absichern.

10.14 Negativbeispiele

Viel zu häufig konzentrieren wir uns zu sehr auf die Positivtests, dabei sind Details darüber, wann ein Stückchen Software zu funktionieren aufhören darf oder aufhören soll, ebenso wichtig:

Die Eingabe unbekannter Filmnummern soll zum Fehler führen.

RentalEntry		
movieNumber	daysRented	charge()
42	7	error

Abb. 10–24
Fehlersituationen besser ausleuchten!

Schlüsselwort error zum Testen von Fehlerfällen

Das Schlüsselwort `error` invertiert hier das Verhalten unserer `Fixture`: Erhalten wir eine Exception, wird die Tabellenzelle grün getüncht. Bleibt sie dagegen aus, gibt es Rot. Welche Exception wir in diesem Fall genau erwarten, ist besser in einem Unit Test aufgehoben:

```
public class VideoStoreTest...
  public void testUnknownMovie() {
    try {
      VideoStore store = new VideoStore();
      store.addRental(42, 7);
      fail("renting an unknown movie should fail");
    } catch (UnknownMovieException expected) {
      assertEquals(42, expected.getMovieNumber());
      assertEquals(0, store.numberOfRentals());
    }
  }
}

public class UnknownMovieException extends Exception {
  private int movieNumber;

  public UnknownMovieException(int movieNumber) {
    this.movieNumber = movieNumber;
  }

  public String getMessage() {
    return "unknown movie number: " + movieNumber;
  }

  public int getMovieNumber() {
    return movieNumber;
  }
}
```

Sobald wir der addRental-Methode die neue Exception anhängen, ...

```
public class VideoStore...
  public Rental addRental(int movieNumber, int daysRented)
  throws UnknownMovieException...

  public int numberOfRentals() {
    return rentals.size();
  }
}
```

... beanstandet der Compiler auch unseren vorher geschriebenen Test. Dort können wir von der neuen Abfrage numberOfRentals übrigens guten Gebrauch machen:

```
public class VideoStoreTest...
  public void testRentalHandling()
  throws UnknownMovieException {
    VideoStore store = new VideoStore();
    store.newMovie(7, "Seven Samurai", "regular price");
    Rental rental = store.addRental(7, 42);
    assertEquals("Seven Samurai", rental.getMovieTitle());
    assertEquals(1, store.numberOfRentals());
  }
}
```

Die gewünschte Exception ist dann einfach geworfen:

```
public class VideoStore...
  public Rental addRental(int movieNumber, int daysRented)
  throws UnknownMovieException {
    Movie movie = getMovie(movieNumber);
    if (movie == null)
      throw new UnknownMovieException(movieNumber);

    Rental rental = new Rental(movie, daysRented);
    rentals.add(rental);
    return rental;
  }
}
```

Entsprechende Vorkehrungen sind auch für die Leihdauer notwendig, da wir dort nur echt positive Eingaben erlauben. Ich möchte uns hier aus Platzgründen jedoch den Code ersparen, auch weil das Beispiel ganz ähnlich verlaufen würde. Also hin zu spannenderen Themen ...

10.15 Transformation: Action ➟ Column

Domänenexperten, Kunden und Endanwender sind meist sehr stark in dem Denken verhaftet, sich auf Benutzeroberflächen zu konzentrieren. Ohne ein Coaching und Feedback vonseiten der Entwickler machen die Fachleute deshalb den häufigen Fehler, die Mehrzahl ihrer Tests auf Basis der `ActionFixture` zu beschreiben. Daraus erwächst der Nachteil, dass wir uns viel zu sehr mit den Abläufen der Benutzeroberfläche beschäftigen, anstatt zum wirklichen Kern des Problems vorzustoßen.

Nicht ausschließlich ActionFixture verwenden

Eine tiefere Einsicht in die Domäne unserer Kunden gewinnen wir, indem wir von der Benutzerschnittstelle abstrahieren und uns stärker mit den Einzelheiten ihrer Geschäftsprozesse und -regeln beschäftigen. Das ist die Domäne unserer `ColumnFixture`! Deshalb ist es wesentlich, sich hin und wieder auf die Suche nach den Abstraktionen zu machen, die sich hinter den Szenarien der einzelnen `ActionFixtures` verbergen, um sie dann in die viel kompaktere Form der `ColumnFixture` zu gießen.

Geschäftsprozesse und -regeln mit ColumnFixture dokumentieren

Ein Indiz dafür, dass sich ein Wechsel der Fixture-Form auszahlt, ist die schleichende Duplikation, die sich in den Testdaten breit macht. Schauen wir unter diesem Gesichtspunkt unsere Action-Szenarien aus Abb. 10–14 und Abb. 10–21 durch, werden die Wiederholungen klar: Um die gleiche Eingabeprozedur nicht dreimal als Benutzerszenario durchspielen zu müssen, drängt sich die `ColumnFixture` geradezu auf. Denn die zeilenweise Eingabe von Testdaten plus dem anschließenden Test einer oder mehrerer Systemfunktionen ist ihr Metier:

Duplikation in den Testdaten und -szenarien eliminieren

MovieEntry			
number	title	category	isValid()
1	Pulp Fiction	regular price	true
2	Kill Bill	new release	true
3	Reservoir Dogs	regular price	true

Abb. 10–25
Eingaben in ColumnFixture-Form

```
public class MovieEntry extends fit.ColumnFixture {
  private VideoStore store = SystemUnterTest.instance();

  public int number;
  public String title;
  public String category;

  public boolean isValid() {
    store.newMovie(number, title, category);
    return true;
  }
}
```

Unsere MovieAdministration-Fixture als ColumnFixture auszudrücken gelingt so weit anstandslos. Kopfschmerzen mag uns aber die isValid-Methode verursachen mit ihrem unbeirrbaren Rückgabewert: true.

Bisher waren auch alle Eingabewerte gültig: Die Filmnummer wird aufsteigend vom System vergeben, als Filmtitel ist Freitext erlaubt und die Preiskategorie ist nur aus einer Liste erlaubter Werte selektierbar. Da wir die Filmnummer jedoch in die Eingabespalten der MovieEntry-Fixture befördert haben, sollten wir auch besser dafür Sorge tragen, dass der Versuch gerügt wird, Filmnummern mehrfach einzutragen:

```
public class VideoStoreTest...
  public void testMovieNumberAlreadyInUse() throws Exception {
    VideoStore store = new VideoStore();
    Movie one = store.newMovie(1, "Rashomon", "don't care");
    try {
      store.newMovie(1, "don't care", "don't care");
      fail("movie number should have been assigned already");
    } catch (MovieNumberAlreadyInUseException expected) {
      assertSame(one, store.getMovie(1));
      assertEquals(1, expected.getMovieNumber());
    }
  }
}

public class VideoStore...
  public Movie newMovie(int number, String title,
                        String category)
  throws MovieNumberAlreadyInUseException {
    if (getMovie(number) != null)
      throw new MovieNumberAlreadyInUseException(number);

    Price price = Prices.getPrice(category);
    Movie movie = new Movie(title, price);
    Integer key = new Integer(number);
    movies.put(key, movie);
    return movie;
  }
}
```

Die MovieNumberAlreadyInUseException ist der UnknownMovieException verwandt, deshalb unterschlage ich ihren Code aus Mangel an Platz. Ebenso gehe ich hier einfach darüber hinweg, dass ältere Testfälle jetzt, wo newMovie die Exception werfen kann, selbstverständlich noch um die entsprechende throws-Klausel erweitert werden wollen.

Uns bieten sich nun zwei Alternativen, wie wir mit der Exception umspringen können; abhängig davon hat `isValid` folgendes Resultat:

- `error`, wenn das Problem weder erwartet noch behandelbar ist
- `false`, wenn Fehler dieser Art in jedem Fall zu erwarten sind

Den ersten Fall haben wir uns in Abb. 10–24 gerade erst angeschaut: Für `RentalEntry` sind fehlerhafte Filmnummern schwere Systemfehler, `MovieEntry` dagegen muss durchaus mit ungültigen Eingaben rechnen. Gängig ist dann, die Exception zu fangen und in ein `false` zu wandeln:

```
public class MovieEntry...
  public boolean isValid() {
    try {
      store.newMovie(number, title, category);
      return true;
    } catch (MovieNumberAlreadyInUseException e) {
      return false;
    }
  }
}
```

ColumnFixture zur reinen Eingabe

Soll dagegen nur eine einzige Systemfunktion ausgeführt werden, ungeachtet eventueller Exceptions, können wir die `isValid`-Methode auch gegen eine Besonderheit der `ColumnFixture` eintauschen:

Abb. 10–26 *Transaktionscharakter der ColumnFixture*

MovieEntry		
number	title	category
1	Pulp Fiction	regular price
2	Kill Bill	new release
3	Reservoir Dogs	regular price

```
public class MovieEntry...
  public void execute() throws Exception {
    store.newMovie(number, title, category);
  }
}
```

Zwei Einschubmethoden der ColumnFixture

Die `ColumnFixture` besitzt zwei Einschubmethoden, die zu bestimmten Verarbeitungszeitpunkten ausgeführt werden:

- `reset` vor der ersten Zelle einer Tabellenzeile,
- `execute` vor der ersten Ergebnisspalte oder, falls keine vorhanden, nach der letzten Eingabespalte einer Zeile (wie im obigen Beispiel).

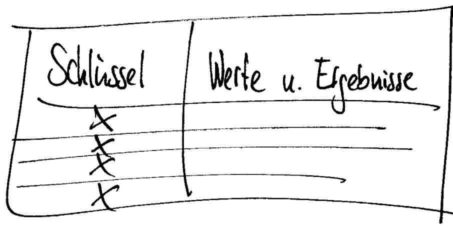
10.16 »RowFixture«

Wie überprüfen wir eine Menge von Objekten?

Die `RowFixture` ist der `ColumnFixture` relativ ähnlich, jedoch mit dem Unterschied, dass sie keine Eingabe-, sondern nur Ergebnisspalten hat. In Summe beschreiben ihre Tabellenzeilen deshalb eine Ergebnismenge von Objekten, die wir anhand eines Schlüssels oder auch mehrerer erkennen und mit den tatsächlichen Resultaten vergleichen wollen:

Mengentest

Abb. 10–27
Tabellenschema der RowFixture



Die `RowFixture` glänzt im Mengentest: also immer, wenn wir mehrere Objekte erwarten, zum Beispiel aus Listen, Reports, Suchergebnissen und ähnlichen Systemanfragen.

... für Listen, Reports, Suchergebnisse u.Ä.

In unserem Beispiel will der Kunde eine einfache Übersicht über sein Geschäft bekommen:

Abb. 10–28
Abfragen in RowFixture-Form

Erlöse

Die Abfrage von Ausleihvorgängen erfolgt nach Filmnummer.
Zur Kontrolle wird der Filmname und die Gesamtzahl Verleihtage ausgegeben.

RentalListing		
movieNumber	movieTitle	totalDaysRented
1	Pulp Fiction	4
3	Reservoir Dogs	13
2	Kill Bill	3

```
public class RentalListing extends fit.RowFixture {
  public Object[] query() throws Exception {
    return new RentalItem[0];
  }

  public Class getTargetClass() {
    return RentalItem.class;
  }
}
```

Die RowFixture definiert zwei *abstrakte* Einschubmethoden und ist deshalb immer durch Vererbung zu spezialisieren:

Zwei Einschubmethoden der RowFixture

- query liefert die tatsächliche Menge von Ergebnisobjekten.
- getTargetClass liefert den Typ der erwarteten Objekte.

Aufgabe der query-Methode ist es, die erwarteten Objekte aufzulesen. Solange wir dazu nicht in der Lage sind, weil die Abfragemöglichkeit dafür im System noch nicht besteht, darf es auch die leere Menge sein. Die RowFixture wird uns dann sagen, was ihr fehlt:

query()

RentalListing		
movieNumber	movieTitle	totalDaysRented
1 *missing*	Pulp Fiction	4
3 *missing*	Reservoir Dogs	13
2 *missing*	Kill Bill	3

Abb. 10–29
Vermisst: Domänenobjekte

Voraussetzung ist, wir erklären der RowFixture mittels getTargetClass, welchen Typs die Objekte sind, die die query-Methode zurückliefert. Aufgrund dieser Information bindet die RowFixture den Spaltenkopf an gleichnamige Felder und Methoden des Zieltyps. Der Mechanismus entspricht dem der ColumnFixture mit einem wichtigen Unterschied: Die ColumnFixture ist ihr eigener Zieltyp; sie reflektiert auf sich selbst, wohingegen die RowFixture stets ein Domänenobjekt zum Zieltyp hat und auf diesem die Reflection anwendet.

getTargetClass()

In unserem Beispiel werden die Zeilen der RentalListing-Fixture durch folgende Klasse beschrieben:

```
public class RentalItem {
  public int movieNumber;
  public String movieTitle;
  public int totalDaysRented;
}
```

Die RowFixture setzt alle ihre Tabellenzeilen auf eine interne Gästeliste und versucht sie dann mithilfe eines *dynamischen* Schlüssels eindeutig zu identifizieren:

Erwartete Objekte auf Gästeliste setzen

- Trifft ein erwartetes Objekt nicht ein, weil kein Schlüssel passte, wird die Tabellenzeile in der ersten Spalte rot als missing markiert. Dies ist in Abb. 10–29 zum Beispiel der Fall, da die Menge leer ist. *missing*
- Trifft dagegen ein nicht erwartetes Objekt ein, wird eine neue Zeile mit dem Fundobjekt in Grau an die Tabelle angehängt und durch surplus markiert. *surplus*

Dynamischer Schlüssel

Hinter den Kulissen läuft hierzu ein Such- und Sortieralgorithmus ab, in dem die Fixture versucht, ihre Gästeliste mit dem `query`-Resultat abzugleichen. Als initialer Schlüssel dient die erste Tabellenspalte:

- Identifiziert der Schlüssel eindeutig ein Objekt aus der Gästeliste, werden die noch folgenden Spalten einfach miteinander verglichen, so wie wir es schon von der `ColumnFixture` gewöhnt sind.
- Ist der Schlüssel dagegen mehrdeutig, wird die nachfolgende Spalte mit in den Schlüssel aufgenommen und der Identifikationsversuch beginnt von vorn.

10.17 Einfacher Schlüssel

Die Reihenfolge der Tabellenzeilen ist für die `RowFixture` unwesentlich, da der Mengenvergleich einzig und allein auf dem Schlüssel basiert. Praktisch verfügen die Zielobjekte einer `RowFixture` immer über einen Schlüssel. Egal ob dieser nun explizit oder nur implizit vorhanden ist, irgendein Eindeutigkeitskriterium liegt immer vor. Denn sonst könnte man selbst nicht einmal eine visuelle Erfolgskontrolle durchführen.

In unserem Fall kann die Filmnummer als Schlüssel einspringen, obwohl `movieNumber` bisher kein Attribut der `Movie`-Klasse darstellte. Macht sich hier unter Umständen ein Designfehler bemerkbar? Sicher! Wobei wir aber berücksichtigen müssen, dass wir im zweiten Kapitel, als Felix und Uli auf der grünen Wiese mit der Entwicklung loslegten, noch nicht entlang von Akzeptanztests gearbeitet haben und dadurch letztendlich vielleicht auch an den wirklichen Anforderungen vorbei.

Wie kommen wir nun aber mit unserem Akzeptanztest vom Fleck? Die `addRental`-Methode wäre geeignet, um sofort an Ort und Stelle die Verleihstatistik zu führen. Mir schwebt eine private Methode vor:

```
public class VideoStore {
  public Rental addRental(int movieNumber, int daysRented)
  throws UnknownMovieException {
    Movie movie = getMovie(movieNumber);
    if (movie == null)
      throw new UnknownMovieException(movieNumber);

    Rental rental = new Rental(movie, daysRented);
    addRentalItem(rental);
    rentals.add(rental);
    return rental;
  }
}
```

So wird für jeden Verleihvorgang die Methode addRentalItem gerufen und wir können dort in einem Abwasch die Buchführung erledigen: Für jeden Film wollen wir die Anzahl seiner Verleihtage mitzählen. Pro Film wird dafür ein RentalItem-Objekt geführt, das wir hinterher in der RentalListing RowFixture inspizieren wollen. Testen können wir die Methode zunächst einmal nur indirekt:

```
public class VideoStoreTest...
  public void testRentalItem() throws Exception {
    VideoStore store = new VideoStore();
    store.newMovie(1, "Koyaanisqatsi", "new release");
    store.addRental(1, 1);
    store.addRental(1, 3);
    List items = store.allRentalItems();
    assertEquals(1, items.size());
    RentalItem item = (RentalItem) items.get(0);
    assertEquals(1, item.movieNumber);
    assertEquals("Koyaanisqatsi", item.movieTitle);
    assertEquals(4, item.totalDaysRented);
  }
}
```

Neben dem abgedruckten Code habe ich der Movie-Klasse auch noch das number-Feld spendiert und Rental ein paar Abfragemethoden:

```
public class VideoStore...
  private Map items = new HashMap();

  private void addRentalItem(Rental rental) {
    int movieNumber = rental.getMovieNumber();
    Integer key = new Integer(movieNumber);
    RentalItem item = (RentalItem) items.get(key);
    if (item == null) {
      Movie movie = rental.getMovie();
      item = new RentalItem(movie);
      items.put(key, item);
    }
    item.addRental(rental);
  }

  public List allRentalItems() {
    return new ArrayList(items.values());
  }
}
```

Fürs RentalItem bleibt nichts weiter zu tun, als einfach die Verleihtage aufzusummieren:

```
public class RentalItem {
  public int movieNumber;
  public String movieTitle;
  public int totalDaysRented = 0;

  public RentalItem(Movie movie) {
    movieNumber = movie.getNumber();
    movieTitle = movie.getTitle();
  }

  public void addRental(Rental rental) {
    totalDaysRented += rental.getDaysRented();
  }
}
```

Wir verwenden hier ein häufiges Muster: Eine ColumnFixture stellt ein paar Transaktionen ins System ein und eine RowFixture stellt am Ende den korrekten Systemzustand sicher. Die ColumnFixture ist in unserem Beispiel RentalEntry (siehe Abb. 10–23), die Transaktionen, für die wir uns interessieren, werden durch RentalItem-Objekte repräsentiert und der unbestechliche Schiedsrichter zum Schluss ist RentalListing. Fehlt uns nur noch, sie alle zum Zusammenspiel zu bewegen:

```
public class RentalListing extends fit.RowFixture {
  private VideoStore store = SystemUnterTest.instance();

  public Object[] query() throws Exception {
    return store.allRentalItems().toArray();
  }

  public Class getTargetClass() {
    return RentalItem.class;
  }
}
```

Und siehe da: Hier sind unsere drei RentalItems:

Abb. 10–30
Identifikation anhand der ersten Tabellenspalte

RentalListing		
movieNumber	movieTitle	totalDaysRented
1	Pulp Fiction	4
3	Reservoir Dogs	13
2	Kill Bill	3

10.18 Mehrfacher Schlüssel

Um mehrspaltige Schlüssel wenigstens einmal ausprobiert zu haben, machen wir ein kleines Experiment: Wir tauschen die Reihenfolge der ersten beiden Spalten und fügen eine Kopie von »Pulp Fiction« hinzu. Fachlich gesehen ist das Unsinn, da `movieNumber` allein als Schlüssel ausreichen täte. Durch den doppelten `movieTitle`-Eintrag erzwingen wir jedoch probeweise die Mehrdeutigkeit des einstelligen Schlüssels:

RentalListing		
movieTitle	movieNumber	totalDaysRented
Pulp Fiction	1	4
Reservoir Dogs	3	13
Kill Bill	2	3
Pulp Fiction	4	1

Abb. 10–31
Mehrstelliger Schlüssel

Mittlerweile haben wir sogar genug Fixtures zusammen, um mit ihnen einfach einmal verschiedene Testszenarien durchzuspielen: Um den neuen Film auch in die Kartei aufzunehmen, müssen wir zuvor unsere `MovieEntry`-Fixture bemühen:

MovieEntry		
number	title	category
4	Pulp Fiction	regular price

Abb. 10–32
Ein neuer Film ...

Auch nicht vergessen dürfen wir den Verleih an sich: Ein Fall für die `RentalEntry`-Fixture:

RentalEntry		
movieNumber	daysRented	charge()
4	1	1.50

Abb. 10–33
... geliehen

Und siehe da: Hier ist unser viertes `RentalItem`:

RentalListing		
movieTitle	movieNumber	totalDaysRented
Pulp Fiction	1	4
Reservoir Dogs	3	13
Kill Bill	2	3
Pulp Fiction	4	1

Abb. 10–34
Identifikation anhand der ersten zwei Spalten

10.19 Abfragemethoden einspannen

Bisher war unser Schlüssel immer ein *öffentliches* Feld des Zielobjekts. Viele Leute betrachten dies als Verstoß gegen geltende Designregeln. Ihre Begründung ist, dass Member-Variablen privat bleiben sollten. Selbst bei Klassen, die sie einzig und allein zu Testzwecken schreiben, drücken sie kein Auge zu. Ich sehe das in dem Fall meist nicht so eng. Im anderen Fall gelten akzeptierte Designprinzipien aber natürlich:

Domänenobjekte inspizieren

In der Regel beschreiben die einzelnen Zeilen einer `RowFixture` ein Domänenobjekt oder einen Ausschnitt davon. In verteilten Systemen könnten es die Datentransferobjekte sein, für die wir uns interessieren, im GUI-Sektor sind es meist Voranzeigeobjekte für Bäume und Listen. Diese Klassen können wir auch über Abfragemethoden einspannen:

Abb. 10–35 *Schwenk auf Abfragemethoden*

RentalListing			
movieTitle()	movieNumber()	totalDaysRented()	totalCharge()
Pulp Fiction	1	4	3.00
Reservoir Dogs	3	13	12.00
Kill Bill	2	3	3.75

```
public class RentalItem {
  private Euro totalCharge = new Euro(0.00);

  public void addRental(Rental rental) {
    totalDaysRented += rental.getDaysRented();
    totalCharge = totalCharge.plus(rental.getCharge());
  }

  public String movieTitle() {
    return movieTitle;
  }

  public int movieNumber() {
    return movieNumber;
  }

  public int totalDaysRented() {
    return totalDaysRented;
  }

  public double totalCharge() {
    return totalCharge.getAmount();
  }
}
```

10.20 »Summary«

Wie erhalten wir eine Zusammenfassung unserer Testlaufergebnisse?

Früher oder später kursieren die Ergebnisse des Akzeptanztests durch verschiedene Hände der Projektgemeinschaft:

- Domänenexperten spezifizieren neue Anforderungen,
- Programmierer entwickeln die Akzeptanztests,
- Kunden verfolgen den Projektfortschritt,
- Projektleiter nehmen den aktuellen Status auf,
- Tester explorieren die Grenzen des Systems,
- Build-Meister bringen das System in Produktion.

Stempel für Testreports

Um für jeden Testreport, der uns irgendwann einmal in die Finger gerät, auch noch feststellen zu können,

- wann die Tests zuletzt gelaufen sind
- und mit welchem Ergebnis,

können wir jedem Dokument einen eindeutigen Stempel verpassen:

fit.Summary

Abb. 10–36
Summary-Fixture

`Summary` ist eine einfache Fixture, die nichts weiter macht, als während des `FileRunner`-Laufs einige statistische Werte zu protokollieren:

fit.Summary	
counts	42 right, 0 wrong, 0 ignored, 0 exceptions
input file	C:\workspace\videostore\testsuite.html
input update	Fri Aug 13 15:02:46 GMT+01:00 2004
output file	C:\workspace\videostore\testsuite-results.html
run date	Fri Aug 13 15:02:50 GMT+01:00 2004
run elapsed time	0:00.33

Abb. 10–37
Testlaufergebnisse

Der `counts`-Status schillert dann bei Fehlschlägen oder Exceptions rot, sonst gewohnt grün.

Wenn Sie wollen, können Sie dem `Summary` in jeder Ihrer Fixtures auch noch weitere Protokolleinträge hinzufügen. Die `summary Map` ist als öffentliches Feld der `Fixture`-Oberklasse von überall zugänglich:

```
summary.put("java version", System.getProperty("java.version"));
summary.put("operating system", System.getProperty("os.name"));
```

Tab. 10–1
Zusammenfassung

ActionFixture	ColumnFixture	RowFixture
Interaktionstests	Logiktests	Mengentests
Aktionstabelle	Ein-/Ausgabetabelle	Ergebnismengentabelle
szenariobasiert	datenbasiert	datenbasiert
Aktion | GUI-Element | Wert 1. Spalte: Aktion (ActionFixture-Methoden) 2. Spalte: GUI-Element (Methoden des Akteurs) 3. Spalte: Wert	Eingabe | Ausgabe Tabellenspalten binden Felder (Eingaben) und Methoden (Ergebnisse) der Unterklasse	Schlüssel | Werte u. Ergebnisse Spalten binden Felder und Methoden des Zieltyps, wobei die ersten Spalten als Schlüssel dienen
Akteure von `fit.Fixture` abzuleiten	durch Unterklassen zu spezialisieren	durch Unterklassen zu spezialisieren

So viel zu den drei Basis-Fixtures. Versuchen Sie, Ihre Tests zunächst durch die geschickte Kombination dieser drei Fixtures aufzustellen, bevor Sie eigene domänenspezifische Fixtures anfangen.

Warum drei Arten von Fixtures?

von Ward Cunningham, Cunningham & Cunningham

Das FIT-Framework erwartet, dass Testdaten in Tabellen gespeichert werden. Die genaue Bedeutung der Testdaten wird durch die in der ersten Zelle benannte Klasse (eine Fixture) bestimmt. Es wird so viele Arten von Fixtures geben, wie es Bedeutungen für Daten gibt, doch wir unterstellen, dass sie in drei grobe Kategorien fallen. Wir nennen sie Row-, Column- und Action-Fixtures und bieten für sie im Framework Unterklassen an. Doch warum diese drei? Bevor ich die Frage beantworte, lassen Sie mich erklären, wie Sie meiner Meinung nach das Format einer Tabelle auswählen sollten.

Wir wollen, dass die Tabellen für alle Projektbeteiligten einfach zu lesen sind, vor allem für jene Personen, die sie auf Genauigkeit prüfen oder in erster Linie schreiben. Das sind die Domänenexperten oder der »Kunde« im XP-Jargon. Manchmal sind die Kunden nicht der Meinung, dass sie Tests schreiben sollten. Ich versuche, sie dann gerne auf diese Weise an den Gedanken zu gewöhnen:

Als Erstes zeige ich großes Interesse für die Einzelheiten der Erläuterungen, die sie mir anzubieten haben. Ich mache mir Notizen und versuche, das Gehörte mit eigenen Worten wiederzugeben.

Dann fange ich an, nach Beispielen zu fragen. Ich frage nach mehr und mehr Details, bis mein Kunde Mühe hat, sich an Ort und Stelle Beispiele auszudenken. Ich sage: »Es würde mir sicher helfen, wenn Sie ein paar detaillierte Beispiele ausarbeiten, so dass ich ganz sichergehen kann, alles richtig verstanden zu haben.« In diesem Moment erwähne ich dann, dass eine Tabellenkalkulationstabelle mit ausgearbeiteten Zahlen großartig wäre.

Tatsächlich jedoch frage ich die Kunden genauso danach, über ihre Daten nachzudenken, wie ich nach brauchbaren Mustern frage. Ich bin viel mehr an den Spalten und Zeilen ihrer Tabellen interessiert als an den tatsächlichen Daten. Ich nehme, was immer sie mir geben, und versuche, die Daten mit entweder der Row-, Column- oder Action-Fixture aufzustellen. Wenn das gelingt, bilde ich die passende Unterklasse. Wenn nicht, versuche ich, eine neue Art von Fixture zu bilden, nur um zu sehen, ob das gelingt.

Schließlich bekomme ich ihre Beispieltabelle mit FIT ins Laufen. Vielleicht muss ich das Format ein wenig abändern, aber ich sorge dafür, dass sie das Ergebnis wiedererkennen. Dann fügen wir neue Fälle hinzu. »Welche weiteren Beispiele passen zu diesem Format?«, bohre ich nach. Relativ schnell übernehmen sie die Verantwortung für die Beispiele, ohne zu bemerken, dass sie Tests schreiben.

Nun zurück zu unserer ursprünglichen Frage: Warum drei Arten?

Die Antwort ist einfach, nämlich dass eine nicht ausreicht. Eine reicht nicht aus, weil Kunden nicht gut darin sind, sich neue Abstraktionen auszudenken. Wenn ich Ihnen nur das eine Rezept für die Arbeit mit dem Kunden gäbe, das ich Ihnen gab, könnten Sie Ihre erste Fixture ziemlich einfach finden, doch Sie bekämen Probleme mit der zweiten. Ihr Kunde wird versuchen, das zweite Beispiel so aufzustellen wie das erste. (Wie gesagt, es fehlt ihnen an Abstraktionsvermögen.) Es gibt drei unterschiedliche Arten von Fixtures in FIT, nur zu Erinnerung, dass es mindestens drei unterschiedliche Tabellenformate gibt.

Ihre Kunden werden Ihnen helfen, die Beispiele zu entwickeln, doch Sie werden ihnen helfen müssen, damit dies auch gelingt. Wenn ihre Tabellen wirklich lang und sie vom Tippen müde werden, helfen Sie, ein besseres Format zu finden. Sie haben drei großartige Beispiele in der Hinterhand: Row-, Column- und Action-Fixture.

10.21 »ExampleTests«

Wie führen wir eine Reihe von Akzeptanztests gemeinsam aus?

FIT-Dokumente in Testsuiten organisieren

Mittlerweile haben sich schon einige `Row`-, `Column`- und `ActionFixtures` angesammelt, so dass es an der Zeit ist, unsere Akteptanztests in einer Testsuite zu organisieren. Dazu möchte ich zwei Fixtures vorstellen, die dem FIT-Framework derzeitig im Beispiele-Package beiliegen.

`ExampleTests` ist ein Beispiel, das Ward Cunningham dazu diente, eine Sammlung von Beispieltests gegen neue FIT-Implementierungen auszuprobieren, und das sich daher besonders gut für Regressionstests eignet. Es handelt sich um eine `ColumnFixture`, die alle Dokumente in der Reihenfolge ausführt, wie sie in ihrer Tabelle aufgezählt sind:

Abb. 10–38
Aufzählung unserer Einzeldokumente

Suite aller Akzeptanztests

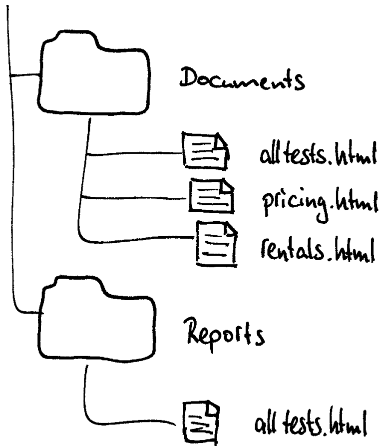
eg.ExampleTests				
file	right()	wrong()	ignores()	exceptions()
pricing.html				
rentals.html				

Ergebnisse des Testlaufs

fit.Summary

Als ersten Schritt zur Testsuite habe ich die `Pricing`-Tabelle aus dem einführenden Kapitelbeispiel ins `pricing.html`-Dokument verschoben. Alle anderen Testfälle sind in das `rentals.html`-Dokument gewandert. Nur die `ExampleTests`-Suite bleibt in der `alltests.html`-Datei zurück:

Abb. 10–39
Aufteilung in mehrere Dokumente und Testreportverzeichnis



Ausführen lässt sich unsere Testsuite auf die gewohnte Art und Weise:

```
java fit.FileRunner Documents/alltests.html
                    Reports/alltests.html
```

Suite aller Akzeptanztests

eg.ExampleTests				
file	right()	wrong()	ignores()	exceptions()
pricing.html	10	0	0	0
rentals.html	32	0	0	0

Ergebnisse des Testlaufs

fit.Summary	
counts	0 right, 0 wrong, 0 ignored, 0 exceptions
counts run	42 right, 0 wrong, 0 ignored, 0 exceptions
input file	C:\workspace\videostore\Documents\alltests.html
input update	Thu Aug 19 23:08:48 GMT+01:00 2004
output file	C:\workspace\videostore\Reports\alltests.html
run date	Thu Aug 19 23:08:51 GMT+01:00 2004
run elapsed time	0:00.33

Abb. 10–40
Ergebnisse der Beispieltests

Sie sehen, die Fixture füllt uns durch den Testlauf die Ergebnisspalten `right`, `wrong`, `ignores` und `exceptions` zur Sichtkontrolle in Grau aus. Wir machen hier von der speziellen Eigenart aller Fixtures Gebrauch, dass leere Zellen in einer Ergebnisspalte besondere Bedeutung haben: Die leere Tabellenzelle ist das Losungswort dazu, den Vergleich von erwartetem und tatsächlichen Wert auszulassen und uns stattdessen, allerdings ausgegraut, das *tatsächliche* Ergebnis mitzuteilen.

Aus diesem Grund stehen auch alle `counts` des `Summary` auf null: Unsere `ExampleTests ColumnFixture` selbst hat schließlich keine Werte vergleichen können, denn die Ergebnisse haben wir alle offen gelassen. Spezifizieren wir die erwarteten Testlaufergebnisse dagegen in der `ExampleTests`-Tabelle, können wir so in Regressionstests sicherstellen, dass die Ergebnisse über mehrere Testläufe unverändert bleiben.

Ideal für Regressionstests

Die Fixture setzt die Verzeichnisstruktur der Distribution voraus: Die Pfade ließen sich zwar anpassen, doch generell werden die unter Tabellenspalte `file` aufgezählten HTML-Dokumente im Verzeichnis `Documents` erwartet und die Reports im Verzeichnis `Reports` abgelegt. Im Testlauf werden jedoch nicht für alle Dokumente Reports erstellt: Nur die, die fehlschlagen, erhalten im `ExampleTests`-Report eine kleine Fußnote mit Verweis auf den Report im Unterverzeichnis `footnotes`.

Verzeichnisse Documents und Reports

Unterordner footnotes

10.22 »AllFiles«

Wie führen wir Verzeichnisse von Akzeptanztests aus?

Mit voranschreitendem Projektfortschritt sammeln sich sehr schnell unzählige Einzeldokumente an. Um in diesem Wust nicht komplett den Überblick zu verlieren, organisieren viele Teams ihre Akzeptanztests je nach Entwicklungsiteration in extra Unterordnern:

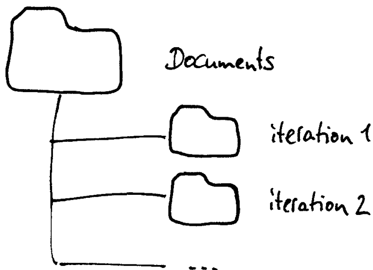


Abb. 10–41
Aufteilung in Unterordner

Das hat den großen Vorteil, dass wir zum Beispiel in Iteration 33 alle für diese Iteration aktuellen Tests auch einzeln ausführen können. Schon nach ein paar wenigen Iterationen wird es nämlich nicht mehr möglich sein, alle gesammelten Akzeptanztests so häufig auszuführen, wie wir es uns unter Umständen wünschen würden. Die Gesamtsuite läuft immerhin nicht wie unsere Unit Testsuite gegen einzelne isolierte Softwarebausteine, sondern testet unser gesamtes System von einem Ende zum anderen Ende. Und das ist häufig viel weiter, als man denkt.

Aus diesem Grund laufen Akzeptanztests, die von ihrer Natur her Integrationstests sind, verglichen mit Unit Tests im Schneckentempo. Sobald die gesamte Akzeptanztestsuite mehr als nur ein paar Minuten schluckt, ist es daher völlig okay, während der Entwicklung nur noch die zurzeit relevanten Tests auszuführen. Auf dem Integrationsserver sollte jedoch regelmäßig die Gesamtsuite laufen. Die `AllFiles`-Fixture hilft uns dann dabei, die Akzeptanztestdokumente anhand Wildcards zu finden und auszuführen:

Abb. 10–42
Bündelung via Wildcards

Suite aller Akzeptanztests

eg.AllFiles
Documents\iteration1*.html
Documents\iteration2*.html

Ergebnisse des Testlaufs

fit.Summary

Legen wir zwei neue Unterverzeichnisse `iteration1` und `iteration2` an und verschieben unsere Dokumente `pricing.html` und `rentals.html` in den ersten Ordner, erhalten wir einen solchen Report:

Suite aller Akzeptanztests

eg.AllFiles	
Documents\iteration1*.html	
rentals.html	32 right, 0 wrong, 0 ignored, 0 exceptions
pricing.html	10 right, 0 wrong, 0 ignored, 0 exceptions
Documents\iteration2*.html no match	

Ergebnisse des Testlaufs

fit.Summary	
counts	2 right, 0 wrong, 1 ignored, 0 exceptions
counts run	42 right, 0 wrong, 0 ignored, 0 exceptions
input file	C:\workspace\videostore\Documents\alltests.html
input update	Sun Aug 22 21:57:36 GMT+01:00 2004
output file	C:\workspace\videostore\Reports\alltests.html
run date	Sun Aug 22 21:57:39 GMT+01:00 2004
run elapsed time	0:00.39

Abb. 10–43
Expandierte Wildcards

Die `AllFiles`-Fixture expandiert zunächst unsere Wildcard-Zeichen, sammelt damit alle Tests zusammen und führt sie nacheinander aus. Für jedes gefundene Dokument hängt die Fixture eine neue Zeile mit dem Namen des Dokuments und des Ergebnisses seines Testlaufs an. Existieren dagegen keine Treffer wie im Falle unseres zweiten Ordners, wird unser Dateipfad ignoriert, ausgegraut und mit `no match` markiert. Als Wildcards verarbeitet `AllFiles` derzeit lediglich das Jokerzeichen, wobei in jedem Element des Dateipfads jeweils jedoch nur ein Joker berücksichtigt wird.

Wie `ExampleTests` erstellt auch die `AllFiles`-Fixture Detailreports nur für Dokumente mit Problemen. Die `counts` des `Summary` betrachten Tests als Erfolg, solange ihnen Fehlschläge und Exceptions fernbleiben. Gruppieren wir mit der `AllFiles`-Suite also andere Untersuiten, zählt die Fixture, wie viele dieser Suiten fehlerlos ausgeführt wurden. Gruppieren wir Testfälle, zählt sie, wie viele von ihnen fehlerlos liefen. Den Kombinationsmöglichkeiten von `ExampleTests`- und `AllFiles`-Tabellen sind also keine Grenzen gesetzt. Und wem es nicht ausreicht, der programmiert sich einfach seine eigene Testsuite.

10.23 Setup- und Teardown-Fixtures

Sobald wir mehrere Akzeptanztests nacheinander ausführen, stellt sich sofort die Frage, wie wir unsere Systemumgebung vor dem Testlauf aufbauen, während des Testlaufs zurücksetzen und nach dem Testlauf wieder abräumen. Am einfachsten spannen wir für solche Aktionen extra `SetUp`- und `TearDown`-Fixtures ein:

Abb. 10–44
TearDown-Fixture

TearDown

So würde für jede `TearDown`-Tabelle ein neues Fixture-Objekt erzeugt und wir hätten nichts weiter zu tun, als im Konstruktor dieser Klasse die nötigen Aufräumarbeiten zu erledigen:

```
public class TearDown extends fit.Fixture {
  public TearDown() {
    SystemUnterTest.reset();
  }
}
```

Oft bilden mehrere Akzeptanztests jedoch eine funktionale Testreihe, was bedeutet, dass sie aufeinander aufsetzen und wir nicht zu Beginn jedes Dokuments ein `SetUp` und zum Ende ein `TearDown` gebrauchen.

10.24 Das FIT-Framework von innen

Akzeptanztests sind so extrem projektspezifisch, dass es in generischen Test-Frameworks meist doch noch irgendwo kneift. So auch mit FIT. Haben Sie erst ein Weilchen mit dem Framework gearbeitet, werden Sie Möglichkeiten erkennen, wie Sie es projektspezifisch anpassen und verbessern können. Sie werden sehen, das Framework ist erstaunlich offen gegenüber neuen Anforderungen.

In den folgenden Kapiteln nehmen wir das Framework näher unter die Lupe und zerlegen es. Wenn Sie FIT eines guten Tages zielgerichtet um Ihre eigenen Bedürfnisse erweitern, benötigen Sie dazu das Wissen, wie die Einzelteile und Mechanismen zusammenhängen:

Kernelemente von FIT

- `FileRunner` führt HTML-Dokumente aus und erstellt Testreports.
- `Parse` repräsentiert grammatikalisch analysierte HTML-Bäume.
- `Fixture` koppelt Tabelleninhalte an das im Test befindliche System und schreibt Prüfvermerke in die HTML-Tabelle zurück.
- `TypeAdapter` konvertiert Datentypen zwischen Java und HTML.

10.25 »FileRunner«

Wie führen wir ein Akzeptanztestdokument aus?

Der `FileRunner` ist uns ja schon mehrfach begegnet. Er führt unsere HTML-Dokumente aus und erstellt eine annotierte Kopie als Report. Im Wesentlichen kann sein gesamtes Wirken auf vier einfache Zeilen Code zurückgeführt werden:

```
Parse tables = new Parse(input); // HTML-Dokument parsen
Fixture fixture = new Fixture();
fixture.doTables(tables); // Tabelleninhalte interpretieren
tables.print(output); // Testreport schreiben
```

Leider ist die Funktion des `FileRunners` nicht idempotent. Das heißt, Sie können ihm seine Ausgabedatei nicht als Eingabe zurückführen! FITs Parser würde sich an seinen eigenen Vermerken verschlucken. Deshalb müssen wir zwischen Ein- und Ausgabedateien unterscheiden: ob durch unterschiedliche Dateinamen oder getrennte Verzeichnisse, das ist reine Geschmacksfrage.

FIT im Build-Prozess

Falls Sie Ihre Testsuite im nächtlichen Build o.Ä. ausführen wollen, ist es noch wissenswert, dass der `FileRunner` als Exit-Code die Summe aus `wrong` und `exceptions` liefert. Nach einem tadellosen Testlauf soll das Ergebnis also `0` sein.

10.26 »Parse«

Wie greifen wir auf die HTML-Tabelleninhalte zu?

Die Framework-Klasse `Parse` zerlegt unser HTML-Eingabedokument gemäß den uns interessierenden HTML-Tags `<table>`, `<tr>` und `<td>`. Der Parser läuft dazu in mehreren Pässen über das gesamte HTML und fischt sich in jedem Lauf nur ein bestimmtes HTML-Tag heraus. Auf diese Weise werden die Testdaten in einer internen Datenstruktur abgebildet, vergleichbar mit dem *Document Object Model (DOM)*, die uns während der Testausführung eine flexible Verarbeitung und Manipulation des HTML-Baums ermöglicht.

Jedes Vorkommen eines der gesuchten HTML-Tags wird durch ein einzelnes `Parse`-Objekt repräsentiert. Jeder dieser Knoten speichert den vom öffnenden und schließenden HTML-Tag eingeschlossenen Text, *Body* genannt, und die das Tag umgebenden Texte, *Leader* und *Trailer*. Wenn Sie umblättern, sehen Sie ein paar Verwendungsbeispiele für den HTML-Parser. Die Lerntests sind mit Ausnahme kleiner Anpassungen FITs gleichnamiger `FrameworkTest`-Klasse entnommen.

Im ersten Testfall parsen wir nur das <table>-Tag:

```
public class FrameworkTest extends TestCase {
  public void testParsing() throws Exception {
    Parse p = new Parse("leader<table>body</table>trailer",
                        new String[] { "table" });
    assertEquals("leader", p.leader);
    assertEquals("<table>", p.tag);
    assertEquals("body", p.body);
    assertEquals("trailer", p.trailer);
  }
```

Sollte das body-Element weitere interessierende HTML-Tags enthalten, unternimmt der Parser einen rekursiven Abstieg und merkt sich das Parse-Resultat des besuchten Unterbaums im Attribut parts, über das wir dann ganz einfach die gesamte Teilstruktur traversieren können. Standardmäßig sucht Parse nach <table>-, <tr>- und <td>-Tags:

```
  public void testRecursing() throws Exception {
    Parse p = new Parse("<table><tr><td>one</td></tr></table>");
    assertNull(p.body);
    assertNull(p.parts.body);
    assertEquals("one", p.parts.parts.body);
  }
```

Sind dagegen im trailer-Element noch weitere gesuchte HTML-Tags enthalten, bahnt sich der Parser seinen Weg durch diesen Unterbaum und speichert den Parse-Wurzelknoten entsprechend im Attribut more. Den Parse-Baum dieses Beispiels sehen Sie rechts in Abb. 10–45:

```
  public void testIterating() throws Exception {
    Parse p = new Parse("<table>"
                        + "<tr>"
                          + "<td>1</td>"
                          + "<td>2</td>"
                          + "<td>3</td>"
                        + "</tr>"
                      + "</table>");
    assertEquals("1", p.parts.parts.body);
    assertEquals("2", p.parts.parts.more.body);
    assertEquals("3", p.parts.parts.more.more.body);
    assertNull(p.parts.parts.more.more.more);
  }
}
```

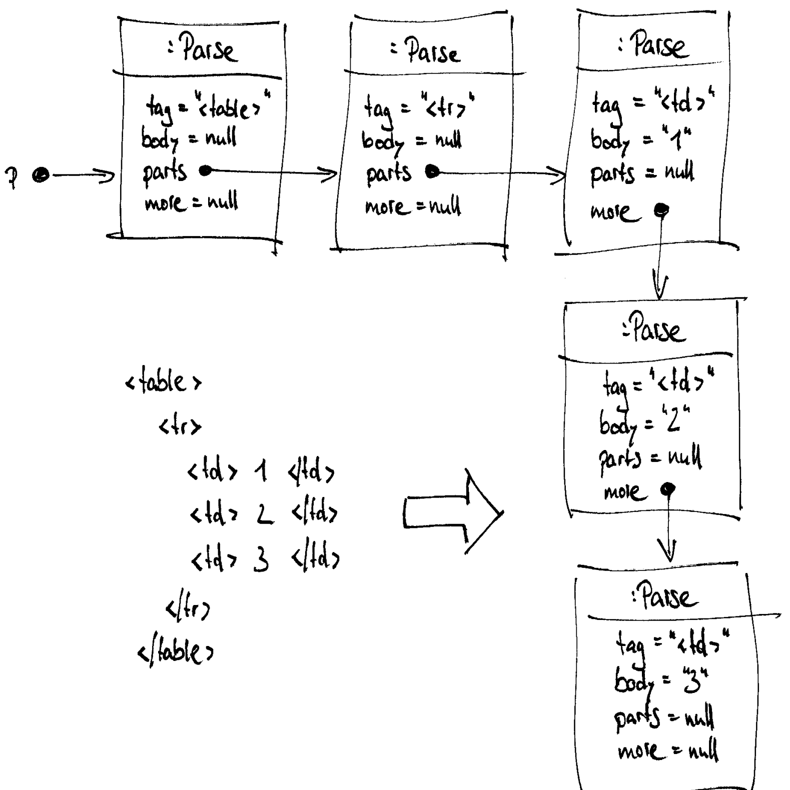


Abb. 10–45
Parse-Baum

FIT-HTML-Parser

Dass FIT keinen der gängigen XML-Parser einsetzt, sondern seinen eigenen Haus-Parser mitbringen muss, hat wahrscheinlich ebenso viele Vorteile wie Nachteile:

Äußerste Vorsicht: HTML-Kommentare ignoriert der Parser (!)

Auf der Minusseite hat der Parser einige ernste Beschränkungen. So sind Tabellen in Tabellen zum Beispiel ein Problem. Auch lässt der Parser sich durch HTML-Tags durcheinander bringen, die eigentlich Zeichenketten oder Kommentaren angehören. Beides kommt daher, weil der Parser äußerst pragmatisch nur mit `indexOf` herumhantiert. Zur Verteidigung dieses einfachen Ansatzes will ich jedoch einwerfen, dass die beschriebenen Grenzen in der Praxis überraschenderweise kein Problem darstellen. Schwerer fällt da vielleicht noch ins Gewicht, dass `Parse` nicht gleichzeitig mit `<th>`- und `<td>`-Tags umgehen kann.

Auf der Plusseite macht sich die Abwesenheit von Collections in erstaunlich elegantem und flexiblem Code in den Fixtures bemerkbar. `Parse`-Objekte werden in die Fixtures hineingereicht, dort interpretiert und mit den Testresultaten annotiert. Wie das alles im Detail abläuft, schauen wir uns als Nächstes an.

10.27 »Fixture«

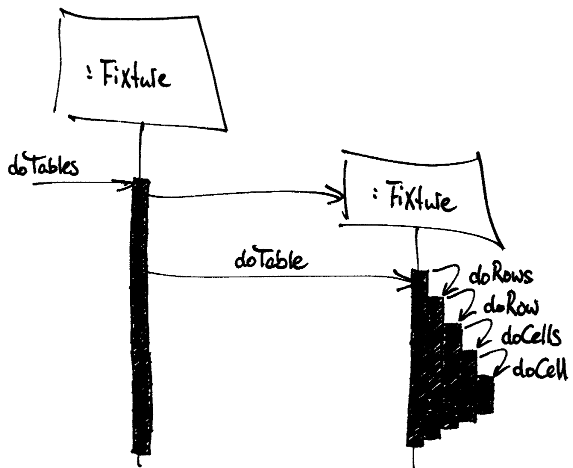
Wie verarbeiten wir die tabellarischen Testdaten?

`Fixture` bildet die zentrale Klasse des FIT-Frameworks. Sie bestimmt, wie die Zelleninhalte unserer HTML-Tabellen interpretiert werden. Aus diesem Grund sind alle Fixtures von ihr als Basisklasse abzuleiten, direkt oder indirekt.

Über- und untergeordnete Fixture

Die Abarbeitung eines kompletten Dokuments teilen sich mehrere Fixtures untereinander. So ist eine Fixture grundsätzlich den anderen übergeordnet, indem sie dem Kopf der aktuell zu verarbeitenden Tabelle zunächst deren Fixture-Namen entnimmt, daraufhin erst die untergeordnete Fixture erzeugen kann und dieser schließlich den Rest der aktuellen Tabelle zur Verarbeitung überlässt:

Abb. 10–46
Traversionsschema



- `doTables` iteriert über alle Tabellen eines Dokuments.
- `doTable` interpretiert eine Tabelle.
- `doRows` iteriert über alle Zeilen einer Tabelle.
- `doRow` interpretiert eine Tabellenzeile.
- `doCells` iteriert über alle Zellen einer Zeile.
- `doCell` interpretiert eine Tabellenzelle.

Lebenszyklus

Vom Lebenszyklus der Fixtures ist wichtig zu verstehen, dass für jede Tabelle ein neues Examplar der untergeordneten Fixture erzeugt wird. Das geschieht selbst dann, wenn Fixtures innerhalb eines Dokuments mehrmalige Verwendung finden. Schauen wir uns einmal den genauen Ablauf an, wie untergeordnete Fixtures die Kontrolle übernehmen.

Es fängt damit an, dass die untergeordnete Fixture in `doTable` die zu verarbeitende Tabelle als `Parse`-Baum erhält, davon die erste Zeile, in der der Fixture-Name steht, überliest und alle restlichen verarbeitet:

doTable()

```
public class Fixture...
  public void doTable(Parse table) {
    doRows(table.parts.more);
  }
```

Die Tabelle wird dann zeilenweise von oben nach unten durchlaufen. Tabellenzeilen, die die Fixture selbst einfügt, werden übersprungen:

doRows()

```
  public void doRows(Parse rows) {
    while (rows != null) {
      Parse more = rows.more;
      doRow(rows);
      rows = more;
    }
  }
```

Eine jede der Tabellenzeilen besteht wiederum aus 0..n Tabellenzellen:

doRow()

```
  public void doRow(Parse row) {
    doCells(row.parts);
  }
```

Die Zellen einer Zeile werden jetzt spaltenweise von links nach rechts durchlaufen. Tritt dabei eine Exception auf, wird diese Zelle markiert:

doCells()

```
  public void doCells(Parse cells) {
    for (int i = 0; cells != null; i++) {
      try {
        doCell(cells, i);
      } catch (Exception e) {
        exception(cells, e);
      }
      cells = cells.more;
    }
  }
```

Das Defaultverhalten besteht darin, den Inhalt der Zelle zu ignorieren:

doCell()

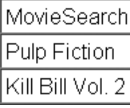
```
  public void doCell(Parse cell, int columnNumber) {
    ignore(cell);
  }
}
```

10.28 Annotationsmöglichkeiten in Dokumenten

Wollen wir mit unseren Tabelleninhalten etwas Sinnvolleres anstellen, als nur alle Zellen in Grau zu tauchen, müssen wir die `doCell`-Methode bei uns überschreiben. In den Blättern des `Parse`-Baums angekommen, sollte schließlich irgendwer die tatsächliche Arbeit übernehmen.

Als kleines Beispiel dafür nehmen wir uns eine einfache Suche vor: Die erste Tabellenzelle benennt wie immer die interpretierende Fixture. Die darauf folgenden Zeilen enthalten die Titel gesuchter DVDs:

Abb. 10–47
Such!



MovieSearch
Pulp Fiction
Kill Bill Vol. 2

Natürlich könnten wir unsere Fixture besser von `RowFixture` ableiten. Hier geht es uns jedoch darum, die Arbeit *einmal* mit eigenen Händen gemacht zu haben. Deshalb setzen wir auf die Basisklasse auf:

```
public class MovieSearch extends Fixture {
  private VideoStore store = SystemUnterTest.instance();
```

text()

Mithilfe des `Parse`-Objekts können wir nun den Inhalt der aktuellen Tabellenzelle ermitteln. Die Methode `text` liefert uns den rohen Text, von umgebenden HTML-Tags befreit und in lesbare Form gebracht. Dazu gehört, dass die *Escape-Sequenzen* <, >, & und   zu <, >, & bzw. Leerzeichen gewandelt werden.

```
  public void doCell(Parse cell, int columnNumber) {
    String movieTitle = cell.text();
    if (store.hasMovie(movieTitle)) {
      right(cell);
    } else {
      wrong(cell);
    }
  }
}
```

Methoden zur Annotation

Zu unserer Bequemlichkeit bietet die Oberklasse `Fixture` schon einige Methoden zur standardisierten Annotierung von Tabellenzellen:

- `right`, wenn das tatsächliche und erwartete Resultat gleich sind,
- `wrong`, wenn tatsächliches und erwartetes Ergebnis abweichen,
- `ignore`, wenn der beabsichtigte Test gerade/noch nicht möglich ist,
- `exception`, wenn während der Abarbeitung eine Exception auftritt.

Das weitere Spielchen kennen Sie ja schon:

```
public class VideoStore...
  public boolean hasMovie(String movieTitle) {
    return false;
  }
}
```

Die `wrong`-Anweisung sorgt in diesem Fall dafür, die Hintergrundfarbe der Tabellenzellen auf Rot zu setzen:

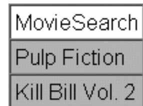


Abb. 10–48
Einfach nur rot

Bei der Titelsuche interessiert uns die Groß-/Kleinschreibung nicht:

```
public class VideoStoreTest...
  public void testMovieSearching() throws Exception {
    VideoStore store = new VideoStore();
    store.newMovie(1, "Tiger and Dragon", "regular price");
    assertTrue(store.hasMovie("tiger AND dragon"));
    assertFalse(store.hasMovie("Hero"));
  }
}
```

```
public class VideoStore...
  public boolean hasMovie(String movieTitle) {
    Iterator i = movies.values().iterator();
    while (i.hasNext()) {
      Movie movie = (Movie) i.next();
      if (movie.getTitle().equalsIgnoreCase(movieTitle)) {
        return true;
      }
    }
    return false;
  }
}
```

Ich hoffe, trotz Schwarz-Weiß-Druck leuchtet das Grün noch durch:

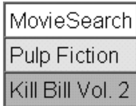


Abb. 10–49
Rot-grün gestreift

Zwei Dinge sollten wir an der MovieSearch-Fixture noch verbessern: Erstens sieht man der HTML-Tabelle nicht unbedingt ihren Zweck an. Eine zusätzliche Spaltenüberschrift könnte hier sicher nicht schaden. Zweitens setzt die Fixture bisher voraus, dass der Filmtitel vollständig und richtig eingetippt wird. Stattdessen wollen wir zukünftig auch die Eingabe von Teilzeichenketten zulassen:

Abb. 10–50
Verbesserte Suche

MovieSearch
search string
I
ill
still

Wie könnte das gehen? Die doCell-Methode holt sich den Suchbegriff, geht damit in die Suche und merkt sich die gefundenen Filme im Feld:

```
public class MovieSearch...
  private List movies;

  public void doCell(Parse cell, int columnNumber) {
    String searchString = cell.text();
    movies = store.searchMovies(searchString);
  }
```

Dabei dürfen wir jedoch nicht vergessen, den eingefügten Spaltenkopf zu überspringen. Dazu fügen wir dem Startpunkt noch ein more hinzu:

```
  public void doTable(Parse table) {
    doRows(table.parts.more.more);
  }
}
```

Der Test für die erweiterte Suche sieht dann beispielsweise so aus:

```
public class VideoStoreTest...
  public void testMovieSearching() throws Exception {
    VideoStore store = new VideoStore();
    store.newMovie(1, "Tiger and Dragon", "regular price");
    store.newMovie(2, "Hero", "new release");
    assertEquals(0, store.searchMovies("*").size());
    assertEquals(1, store.searchMovies("tiger").size());
    assertEquals(2, store.searchMovies("er").size());
  }
}
```

Der Code dafür geht etwas in die Länge:

```
public class VideoStore...
  public List searchMovies(String searchString) {
    String searchTitle = searchString.toLowerCase();
    List result = new ArrayList(movies.values());
    for (Iterator i = result.iterator(); i.hasNext();) {
      Movie movie = (Movie) i.next();
      String actualTitle = movie.getTitle().toLowerCase();
      if (actualTitle.indexOf(searchTitle) == -1) {
        i.remove();
      }
    }
    return result;
  }
}
```

HTML-Baum dynamisch erweitern

Die Liste gefundener DVDs können wir jetzt in der Tabelle anzeigen. Wie machen wir das? Wir können den `Parse`-Baum in unserer Fixture dynamisch verändern, beispielsweise indem wir weitere Tabellenzellen an die Zelle mit dem entsprechenden Suchbegriff anhängen:

```
public class MovieSearch...
  public void doCells(Parse cells) {
    super.doCells(cells);

    for (Iterator i = movies.iterator(); i.hasNext();) {
      Movie movie = (Movie) i.next();
      appendCellTo(cells, movie.getTitle());
      right(cells);
    }
  }
```

Warum überschreiben wir hier die `doCells`-Methode, anstatt die Titel sofort in `doCell` zu verarbeiten? Wenn Sie bitte noch einmal vier Seiten zurückblättern, erkennen Sie, dass die `doCells`-Methode (noch) nicht symmetrisch zu `doRows` implementiert ist: Dynamisch eingefügte Zellen werden demnach in späteren Schleifendurchläufen von `doCells` noch mitverarbeitet. Im schlimmsten Fall ergibt sich eine Endlosschleife. Aufgrund dessen arbeiten wir unsere Tabellenzeile zunächst einmal vollständig ab, indem wir auf die gesehene `doCells`-Implementierung der Oberklasse zurückgreifen. Darauf folgend können wir dann auch neue Zellen in die Zeile einhängen. Seiteneffektfrei.

Spalten einhängen

Wie modifizieren wir jetzt unseren Parse-Baum? Ganz einfach! Parse-Objekte bilden eine rekursive Datenstruktur, in der ein Element auf das nächste zeigt. Sie erinnern sich: Dazu sind die Felder parts und more da. Diese Struktur können wir jederzeit auch manipulieren:

```
private void appendCellTo(Parse cells, String body) {
  cells.last().more = new Parse("td", body, null, null);
}
```

Wir holen uns das letzte Parse-Objekt der aktuellen Zeile und hängen auf gleicher Ebene ein neues <td>-Tag an mit dem DVD-Titel als body, keinen parts- und keinen more-Knoten. So einfach!

Zeilen einhängen

Nach dem gleichen Muster könnten wir auch neue <tr>-Tags und damit weitere Zeilen in die Tabelle einfügen. Aufpassen muss man bei Manipulation rekursiver Datenstrukturen nur immer, dass man keine Endlosschleifen baut oder Elemente unwissentlich durch fehlerhafte Verzeigerung abhängt.

do...-Methoden zum Überschreiben

Als Gimmick wollen wir leere Suchergebnisse noch markieren. Dieses Mal hängen wir uns spaßeshalber bei der doRow-Methode ein. Ich will damit jedoch nur verdeutlichen, dass jede do...-Methode eine Einschubmethode ist, die wir nach Herzenslust überschreiben können:

```
  public void doRow(Parse row) {
    super.doRow(row);

    if (movies.isEmpty()) {
      markRowWithNoMatch(row);
    }
  }

  private void markRowWithNoMatch(Parse row) {
    Parse cell = row.leaf();
    cell.addToTag(" style=\"font-style:italic;\"");
    cell.addToBody(gray(" no match"));
  }
}
```

addToTag()
addToBody()

Mit addToTag fügen wir hier dem <td>-Tag ein style-Attribut hinzu. Mit addToBody bekommt das *Body*-Element Gesellschaft in Grau:

Abb. 10–51
Suchergebnisse

MovieSearch		
search string		
I	Kill Bill	Pulp Fiction
ill	Kill Bill	
still no match		

10.29 »TypeAdapter«

Wie parsen wir domänenspezifische Datentypen?

Bislang haben wir zur Beschreibung unserer Testdaten nur primitive Java-Typen und `Strings` heranziehen können. Viel ausdruckskräftiger und zugleich näher am Problemkern sind jedoch die Geschäftsobjekte, die im Anwendungskern und der Domänenlogik ohnehin schon ihre Verwendung finden. Wünschenswert wäre, wenn sich die Fachlichkeit in dem nötigen Detaillierungsgrad auch in den Testdaten ausdrücken ließe und wir unsere Domänenexperten dazu befähigen könnten, ihre fachlichen Begrifflichkeiten so zu verwenden, wie sie es erwarten:

- 1 kg
- 2 cm
- 3,– Euro
- 5 °C
- 08:13 h
- Reihe 21

Als Nächstes schauen wir uns deshalb an, wie wir unsere Testsprache um Vokabeln der Kundensprache erweitern können und wie sich die fachlich motivierten Klassen, die einen Teil der Domäne modellieren, in die Fixtures einspannen lassen. Zum Beispiel unsere `Euro`-Klasse:

Pricing		
daysRented	regularPrice()	newRelease()
1	EUR 1.50	EUR 2.00
2	EUR 1.50	EUR 2.00
3	EUR 1.50	EUR 3.75
4	EUR 3.00	EUR 5.50
5	EUR 4.50	EUR 7.25

Abb. 10–52
Domänentypen als zusätzliche Beschreibungsmittel

```
public class Pricing extends fit.ColumnFixture {
  public int daysRented;

  public Euro regularPrice() {
    return Price.REGULAR.getCharge(daysRented).getAmount();
  }

  public Euro newRelease() {
    return Price.NEWRELEASE.getCharge(daysRented).getAmount();
  }
}
```

Tief verankert im Framework sitzt der `TypeAdapter` mit der Aufgabe, Datenformate zwischen der untypisierten HTML- und der typisierten Java-Welt hin und her zu transformieren. Dazu gehört unter anderem, die Zellinhalte textuell zu parsen und in entsprechende Java-Typen zu wandeln. Die notwendige Unterstützung, um spezielle Datenformate zu analysieren und zu konvertieren, versucht FIT in dieser Reihenfolge in den `parse`-Methoden dieser zwei Klassen zu finden:

parse()

1. `TypeAdapter`
2. `Fixture`

Weiß jedoch weder der Adapter noch die Fixture, wie aus dem Inhalt einer Tabellenzelle ein Objekt des bestimmten Typs zu konstruieren ist, meckert die `Fixture` die betreffende Zelle gelb an:

```
java.lang.Exception: can't yet parse class Euro
```

`TypeAdapter` und seine Familie von Unterklassen sind für die dynamische Adaption primitiver Typen und ihrer Objektinkarnationen zuständig. Außerdem geht der allgemeine Reflection-Mechanismus, wie einzelne Tabellenspalten an Felder und Methoden der Fixture (für `Column`- und `ActionFixture`) bzw. des Zieltyps (für `RowFixture`) gebunden werden, auf diese Klassenhierarchie zurück.

Unsere eigenen Typen lassen sich in diesen Mechanismus einfach integrieren, indem wir die `parse`-Methode der `Fixture` in unserer Unterklasse erweitern. Das Parsen an sich wird von dort jedoch meist an den zu parsenden Typ selbst übertragen:

```
public class Pricing...
  public Object parse(String text, Class type)
  throws Exception {
    if (Euro.class.equals(type)) {
      return Euro.parse(text);
    }

    return super.parse(text, type);
  }
}
```

Unsere Oberklasse sorgt dann zusätzlich noch dafür, diese Datentypen zu parsen:

- `String`
- `Date`
- `ScientificDouble` (später mehr)

Was muss die parse-Methode nun tun? Objekte müssen sich aus ihrer eigenen String-Repräsentation wieder parsen können:

```
public class EuroTest...
  public void testParsing() throws Exception {
    Euro parsed = Euro.parse("EUR 2.00");
    assertEquals(two, parsed);
  }

  public void testInvariant() throws Exception {
    assertEquals(two, Euro.parse(two.toString()));
  }

  public void testParseException() {
    try {
      Euro.parse("2.00");
      fail("missing currency symbol not noticed");
    } catch (ParseException expected) {
      assertEquals(0, expected.getErrorOffset());
    }
  }
}

public class Euro...
  private static final String CURRENCY_SYMBOL = "EUR ";

  public static Euro parse(String text) throws ParseException {
    if (!text.startsWith(CURRENCY_SYMBOL))
      throw new ParseException("no currency symbol found", 0);

    String number = text.substring(CURRENCY_SYMBOL.length());
    double amount = Double.parseDouble(number);
    return new Euro(amount);
  }

  public String toString() {
    return CURRENCY_SYMBOL + getAmount();
  }
}
```

Gegen alle denkbaren Fehleingaben muss der Parser nicht gefeit sein. Es sei denn, der Code findet im System später ohnehin Verwendung zur Validierung der Benutzereingaben.

Setzen Sie die dynamische Typadaption sparsam ein und testen Sie jeden handgeschmiedeten Parser.

Date

Ein häufig verwendeter Datentyp, deshalb von FIT direkt unterstützt, ist die gute alte Date-Klasse. Wollten wir unseren Ausleihen ein Datum hinzufügen, könnten wir das auf folgende Weise bewerkstelligen:

```
public class RentalEntry...
  public Date dateRented; // not really used, just for demo

  public double charge() throws UnknownMovieException {
    Rental rental = store.addRental(movieNumber, daysRented);
    return rental.getCharge().getAmount();
  }
}
```

Kein Grund zur Wiederholung

Den entsprechenden Umbau vorzuführen, erspare ich uns, um nicht Papier zu vergeuden. Lassen Sie mich stattdessen einen Trick erklären: Wenn sich ein und dasselbe Datum über mehrere Zeilen wiederholt, können wir vom Feature Gebrauch machen, dass die Fixture beim Antreffen leerer Tabellenzellen ihre Schreib-/Leserichtung umdreht: Der Wert der betreffenden Zelle wird jetzt nicht mehr zur Fixture transportiert, sondern aus der Fixture für den Zelleninhalt bestimmt. In unserem Fall wird so der letzte Spaltenwert für die folgenden leeren Zeilen wiederverwendet, wodurch sich die Tabelle besser liest:

Abb. 10–53
Die Zeit läuft ...

RentalEntry			
dateRented	movieNumber	daysRented	charge()
17.09.2004	1	3	1.50
	3	5	4.50
	2	3	3.75
	3	2	1.50
18.09.2004	1	1	1.50
	3	6	6.00

Vorteilhaft ist die Typadaption,

- wenn die Tests dadurch an Lesbarkeit gewinnen,
- wenn die Anhänge und Zusätze selbst wesentliche Informationen mittransportieren, wie zum Beispiel die Einheit,
- wenn die Informationsdichte zunimmt, etwa durch Formelsprache, Abkürzungen, Fachjargon usw.

Nachteilig erweist sich die Adaption vor allem bei aggregierten Typen, wenn also mehrere einzelne Elemente zu einem strukturierten Ganzen zusammengesetzt werden müssen. In diesem Fall ist es häufig besser, mit einer anderen Art und/oder Form von Fixture zu experimentieren.

10.30 »ScientificDouble«

Wie vergleichen wir Fließkommazahlen mit gewisser Präzision?

Fundamentale Datentypen eignen sich besonders gut zur Typadaption. Eine äußerst geschickte Verwendungsmöglichkeit führt uns auch die FIT-Klasse `ScientificDouble` vor, die ich hier nur kurz erwähnen will. Sie stellt Fließkommazahlen mit der Genauigkeit dar, wie sie durch die Anzahl signifikanter Dezimalstellen in den Testdaten spezifiziert wird:

- `3.1416` wird zum Beispiel auf fünf Dezimalstellen,
- `3.14159` dagegen auf sechs Stellen genau verglichen. Sehr nützlich!

10.31 Domänenspezifische Grammatiken

Noch einige Worte zum Parser-Bau. Vergleichen Sie diese zwei Werte:

- ISBN 0-8050-5438-3
- N 45.5 W 122.75

Sowohl die ISBN-Nummer als auch die geografischen Koordinaten lassen sich mit Leichtigkeit von Hand parsen. Dennoch möchte ich Sie dazu ermutigen, ab einer bestimmten Komplexität auf einen gängigen Parser-Generator umzusatteln. Eine so einfache Grammatik wie diese:

```
coordinate   = northOrSouth Num eastOrWest Num;
northOrSouth = "N" | "S";
eastOrWest   = "E" | "W";
```

... lässt sich mit einem objektorientierten Parser so einfach parsen:

```
Alternation northOrSouth = new Alternation();
northOrSouth.add(new CaselessLiteral("N"));
northOrSouth.add(new CaselessLiteral("S"));

Alternation eastOrWest = new Alternation();
eastOrWest.add(new CaselessLiteral("E"));
eastOrWest.add(new CaselessLiteral("W"));

Sequence coordinate = new Sequence();
coordinate.add(northOrSouth);
coordinate.add(new Num());
coordinate.add(eastOrWest);
coordinate.add(new Num());
```

Wer will da noch einen fehleranfälligen Parser von Hand schreiben? Steven Metsker [me_{01}] zeigt Ihnen, wie weit die Möglichkeiten gehen.

10.32 »ArrayAdapter«

Wie parsen wir kommaseparierte Zelleninhalte?

ArrayAdapter ist ein TypeAdapter zur Verarbeitung *eindimensionaler* Arrays beliebigen Typs. Besonders praktisch sind Arrays, um in einer Tabellenzelle eine Reihe von Werten zu parsen oder auch darzustellen. Die einzelnen Werte sind dabei durch Komma zu trennen:

Abb. 10–54
Für alles Listenförmige

RentalEntry		
movieNumber	daysRented	charges()
1	3, 1	EUR 1.50, EUR 1.50
3	5, 2, 6	EUR 4.50, EUR 1.50, EUR 6.00
2	3	EUR 3.75

```
public class RentalEntry...
  public int[] daysRented;

  public Euro[] charges() throws UnknownMovieException {
    Euro[] result = new Euro[daysRented.length];
    for (int i = 0; i < daysRented.length; i++) {
      Rental rental = store.addRental(movieNumber,
                                      daysRented[i]);
      result[i] = rental.getCharge();
    }
    return result;
  }

  public Object parse(String text, Class type)
  throws Exception {
    if (Euro.class.equals(type)) {
      return Euro.parse(text);
    }

    return super.parse(text, type);
  }
}
```

Sind die Tabellenspalten in der Fixture an den Typ Array gebunden, versucht der ArrayAdapter für den Typ der zu parsenden Elemente die entsprechende Unterstützung in TypeAdapter und Fixture zu finden. Leerstelle und Leerzelle tragen dabei besondere Bedeutung:

- Eine leere Tabellenzelle ist Synonym für ein null-Array,
- null-Elemente werden kommasepariert als Leertext dargestellt.

10.33 »PrimitiveFixture«

Wie können wir neue Fixture-Formen aufstellen?

`PrimitiveFixture` ist an sich ein historisches Relikt der FIT-Evolution, entpuppt sich jedoch als erstaunlich nützlich, um prototypisch schnell einmal neue Arten von Fixtures aufzustellen. Sie kommt gänzlich ohne Reflection und ohne `TypeAdapter` aus. Stattdessen bietet sie zum Parsen der Zellinhalte primitive Methoden an: `parseLong`, `parseDouble` und `parseBoolean`. Ebenso muss der Vergleich erwarteter und tatsächlicher Resultate von Hand geschehen, wozu wiederum für alle Grundtypen spezifische `check`-Funktionen parat stehen. Ein Beispiel:

Keine Reflection, kein TypeAdapter

parseLong()
parseDouble()
parseBoolean()
check()

Pricing		
Number of days a DVD is rented	Regular price in €?	How much if it's a new release?
1	1.50	EUR 2.00
2	1.50	EUR 2.00
3	1.50	EUR 3.75
4	3.00	EUR 5.50
5	4.50	EUR 7.25

Abb. 10–55
PrimitiveFixture erlaubt exotischere Formen

Ich muss vorweg sagen, unsere `Pricing`-Fixture gibt nicht das beste Beispiel einer `PrimitiveFixture` her, nur das einfachste, das wir haben, um die nötigen Grundkonzepte zu vermitteln.

Es geht damit los, dass `PrimitiveFixture` von selbst weder Felder noch Methoden bindet, was man auch unseren Spaltennamen ansieht, die wir prompt überlesen. Alles das, was sonst ein `TypeAdapter` macht, führen wir jetzt in einem `switch-case`-Verteiler aus:

```
public class Pricing extends fit.PrimitiveFixture {
  public void doRows(Parse rows) {
    super.doRows(rows.more);
  }

  public void doCell(Parse cell, int columnNumber) {
    switch (columnNumber) {
      case 0 : break;
      case 1 : break;
      case 2 : break;
      default : ignore(cell);
    }
  }
}
```

Die doCell-Methode deckt nun jede der Tabellenspalten für sich ab. Da dazu in der Regel schlicht über den Spaltenindex verzweigt wird, sind primitive Fixtures nicht unabhängig von ihrer Spaltenreihenfolge. Ist hier mehr Flexibilität nötig, müssen die Namen des Spaltenkopfs vorher an den Index gebunden werden. Wir bleiben ganz primitiv:

```
public class Pricing...
  private int days;

  public void doCell(Parse cell, int columnNumber) {
    switch (columnNumber) {
      case 0 :
        days = (int) parseLong(cell);
        break;
      case 1 :
        check(cell, Price.REGULAR.getCharge(days).getAmount());
        break;
      case 2 :
        check(cell, Price.NEWRELEASE.getCharge(days));
        break;
      default :
        ignore(cell);
    }
  }
```

Die Eingabespalten behandeln wir, indem wir den Zelleninhalt mithilfe einer parse-Methode auslesen. Die Ergebnisspalten behandeln wir, indem wir erwartete und tatsächliche Werte über eine check-Methode vergleichen. Zur Illustration benutzt die eine Spalte den Typ double, die andere Euro. Letztere erfordert jedoch einen speziellen Check:

```
  public void check(Parse cell, Euro actual) {
    try {
      Euro expected = Euro.parse(cell.text());
      if (expected.equals(actual)) {
        right(cell);
      } else {
        wrong(cell, actual.toString());
      }
    } catch (ParseException e) {
      exception(cell, e);
    }
  }
}
```

10.34 Domänenspezifische Fixtures

Eine große Anzahl von Tests lässt sich gewöhnlich bereits mit den drei bekannten Basis-Fixtures aufstellen. Will uns dies mit einer Form allein nicht gelingen, erreichen wir dies meistens doch, indem wir die drei Grundformen geschickt miteinander kombinieren. Die bloße Aneinanderreihung einzelner Tabellen führt ab einem gewissen Punkt jedoch dazu, dass die Essenz der Tests in langatmigen verworrenen Dokumenten verloren geht. Ein schlechtes Zeichen:

- **Wiederholungen in den Testdaten** wirken genauso schädlich wie solche von Codestrukturen. Sie entstehen vielfach, wenn wir in den Testdaten keinerlei System erkennen, ihnen kein Muster entlocken können ... oder einfach nur zu beschäftigt sind für ein Refactoring. Durch Zusammenfassung wird der Test wieder kurz und bündig.

Refactoring

- **Geschwätzigkeit der Testsprache** führt dazu, dass der tatsächliche Fokus eines Testfalls im Dickicht von Informationen verloren geht und sich die Testautoren beim Verfassen der Tests zu Tode tippen. Mit einer Sprache, die das fachliche Problem passender beschreibt, können wir dagegen mit jedem Test ein kleine Geschichte erzählen.

... der Testsprache

Ein Ausweg führt zu *domänenspezifischen* Fixtures: Sie erlauben uns, Akzeptanztests wieder auf ihren wesentlichen Kern zu konzentrieren und so Dokumente zu formulieren, die ihre Intention ausdrücken, keine Duplikation enthalten und dabei mit möglichst wenig Tabellen und Tabellenzellen auskommen. Einfache Form durch Refactoring!

Jede Domäne kennt ihren Weg, um die zu lösenden Probleme absolut auf den Punkt zu bringen. Finden Sie heraus, welche Darstellungsform Ihr Kunde am gebräuchlichsten findet.

Auf gebräuchliche Tabellenformate setzen

Eine intuitive Form ist meist die, die eh schon von einer tabellarischen Darstellung Gebrauch macht: Reports, Formulare oder Rechnungen. Nehmen wir zum Beispiel ein Rechnungsschema:

Monatsabrechnung

Kunden erhalten monatlich eine schriftliche Rechnung mit den Details, wann sie welche DVD geliehen bzw. zurückgegeben haben.

Invoice				
movie number	movie title	date rented	date returned	charge
1	Pulp Fiction	20.9.2004	23.9.2004	EUR 1.50
2	Kill Bill	22.9.2004	24.9.2004	EUR 2.00
3	Reservoir Dogs	24.9.2004	26.9.2004	EUR 1.50
total				EUR 5.00

Abb. 10–56
Rechnungsschema

Die Mitarbeiter unseres kleinen DVD-Verleihers dagegen interessiert vielleicht eine Sicht über alle Kunden:

Abb. 10–57 *Stundenplanschema*

Kassenbericht

Kassierer erhalten eine Wochenzusammenstellung mit den Informationen, wann und wie lange einzelne Filme von welchen Kunden geliehen wurden.

WeeklyFinances				
date	#1 Pulp Fiction	#2 Kill Bill	#3 Reservoir Dogs	total
20.9.2004	#560 V. Vega			
21.9.2004	|			
22.9.2004	|	#344 B. Kiddo		
23.9.2004	EUR 1.50	|		EUR 1.50
24.9.2004		EUR 2.00 #105 P. Mei	#69 L. White	EUR 2.00
25.9.2004	#504 E. Villalobos	|	|	
26.9.2004	|	EUR 2.00	EUR 1.50	EUR 3.50
totals	EUR 1.50	EUR 4.00	EUR 1.50	EUR 7.00

Auf gebräuchliche Tabellenformate setzen

Bei so mancher Tabelle dieser Art könnte man berechtigt behaupten, dass sich das wirkliche Geschäftsfeld mit all seinen filigranen Facetten nicht mehr in einem übersichtlichen Format darstellen ließe. Lassen Sie sich jedoch von der Tatsache, nicht alles gründlich testen zu können, nicht davor abschrecken, kleinere und doch realitätsnahe Ausschnitte unter die Lupe zu nehmen.

Positionelle Tabellenformate

Äußerst nützlich ist es, eine solche Tabelle, die eigentlich nur eine Momentaufnahme, ein statisches Bild zeigt, als Drehbuch zu nehmen, um so die viel interessantere Dynamik eines Systems als Geschichte zu beschreiben. Oft können wir so das anvisierte Testziel und die Schritte, die dorthin führen, in einer einzelnen kompakten Tabelle darstellen. Bei Abbildung dynamischer Vorgänge lassen sich auch besonders gut *positionelle Tabellenformate* anbringen, die Zeilen- und Spaltenindex dafür gebrauchen, automatisch zum nächsten Tag, nächsten Film o.Ä. weiterzuschalten.

Hinter den meisten domänenspezifischen Fixtures verbergen sich keine großen Geheimnisse. Ganz im Gegenteil: Die besten sind wie alte Freunde, die man sofort erkennt. Wenn die Domäne noch jung ist, kann die strukturgebende Idee jedoch auch schon einmal länger auf sich warten lassen. Die besten kommen einem dann meist kurz vor Ende des Projektes, wenn die Domäne am tiefsten durchdrungen ist.

10.35 Anschluss finden

Betrachten wir zum Abschluss unsere möglichen Anknüpfungspunkte, um uns mit einer Fixture in das betreffende System einzuklinken:

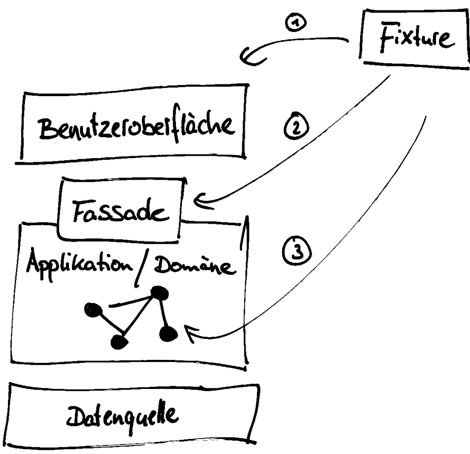


Abb. 10–58
Drei wesentliche Anknüpfungspunkte

Drei Anknüpfungspunkte

Verbindungen können wir auf verschiedenen Systemstufen herstellen:

1. **Benutzeroberfläche,** um Anwendungsszenarien durchzuklicken
2. **Applikationsfassade,** um ganze Geschäftsprozesse anzustoßen
3. **Reine Geschäftslogik,** um Geschäftsregeln direkt abzuhorchen

Effektive Teststrategie

Unsere Teststrategie ist dabei umso effektiver, je weniger wir uns auf einen der Anknüpfungspunkte versteifen und je mehr wir versuchen, eine Balance zwischen ihnen herzustellen. Patentrezepte gibt es nicht. Dafür zeigen Systemarchitekturen einfach zu viel Variationsreichtum. Testgetriebene Entwicklung mithilfe von Akzeptanztests hilft jedoch, das makroskopische Design der Systemarchitektur auf ebenso interessanten Bahnen in Richtung Testbarkeit zu lenken, wie Testgetriebene Entwicklung mit Unit Tests das mikroskopische Design beeinflusst.

Testbarkeit auf mehreren Ebenen

Bei der Wahl von Anknüpfungspunkten ist Einfachheit Trumpf. Lässt sich ein Test lediglich auf der Benutzeroberfläche abwickeln, müssen wir dort automatisieren. Lässt sich ein Test dagegen auf die Ebene von Geschäftsobjekten reduzieren, können wir uns auch über ein Hintertürchen nötigen Zugang zu den Syteminnereien verschaffen. Bei mir wandern die Fixture-Klassen dann auch immer in das Package, das am besten beschreibt, auf welcher Ebene der Akzeptanztest läuft. Sehen wir uns die drei Anknüpfungsmöglichkeiten jetzt im Detail an. Ich gehe sie rückwärts durch aufgrund der langsameren Komplexitätssteigerung.

10.36 Stichproben reiner Geschäftslogik

In der Regel sind Akzeptanztests von ihrer Natur her Integrationstests, oft sogar Systemtests, wenn sie von einem Ende zum anderen gehen. Nichtsdestotrotz spielen feingranulare Geschäftslogiktests eine Rolle, um über die groben Akzeptanzkriterien hinweg eine Vielzahl weiterer Kombinationen abzuklopfen, Negativbeispiele auszuprobieren usw. Für diese Stichproben spricht, dass Fehler, die sich auf der Oberfläche bemerkbar machen, einfach zu entdecken sind, im Gegensatz zu tief im Systemkern verborgenen Fehlern der Geschäftslogik.

Geschäftsfakten in Geschäftssprache

Gewöhnlich tendieren diese Tests in Richtung von Unit Tests und können sich teilweise mit ihnen überschneiden mit dem Unterschied, dass sie Geschäftssprache sprechen und Geschäftsfakten beschreiben. Ein gewisser Grad an Redundanz lässt sich meiner Ansicht nach nicht vermeiden. Wenn Entwickler und Kundenseite aber schon dieselben Tests machen und die Akzeptanztests der Entwicklung der jeweiligen Anforderung vorausgehen, sollten wir aus der Redundanz wenigstens Profit schlagen. Tatsächlich kommen wir mit weniger Unit Tests aus, wenn wir uns zusätzlich durch gezielte Akzeptanztests treiben lassen. Unit Tests schließen dann eine aus Kundensicht unspezifizierte Lücke vom Akzeptanztest zur technischen Realisierung.

Weniger Unit Tests bei guten Akzeptanztests

10.37 Integrationstests gegen Fassaden und Services

Eine zweite Sorte von FIT-Tests setzt direkt unter der Oberfläche an. Diese Tests konzentrieren sich auf die Kernprozesse der Anwendung und sind dazu in aller Regel gegen eine Fassadenschnittstelle gerichtet. Falls unser System anderen Systemen gegenüber Services bereitstellt, können auch diese aus Kundensicht durchleuchtet werden.

Akzeptanztests, die unabhängig von der Benutzeroberfläche sind, haben den großen Vorteil, dass sie nicht angepasst werden müssen, wenn sich die Oberfläche erneut ändert. Diese Tests zeigen ihre Stärke, indem sie abstrakte Sichten auf das tiefere Domänenmodell herstellen. Eine solche Sicht kann sowohl die tatsächliche Benutzungsoberfläche abbilden als auch in Konkurrenz dazu stehend alternative Blickwinkel anbieten. Insbesondere lassen sich so Akzeptanzkriterien spezifizieren, bevor irgendwelche Festlegungen bei der Oberflächengestaltung getroffen sind. Auch können auf diesem Weg eine Vielzahl von Dingen abgesichert werden, die nur im Innern möglich sind, weil sie auf der Oberfläche nicht mehr sichtbar sind.

Virtuelle Benutzeroberfläche

Survival of the FIT Test

von Steffen Künzel, ESG GmbH & Tammo Freese, freier Berater

Nach einer extrem kräftezehrenden, manuellen Testwoche war allen klar: Automatisierte Akzeptanztests mussten her. Sie sollten von unseren Fachtechnikern, aber auch direkt von unserem Kunden bearbeitet werden können, zu jeder Zeit und von überall. Die Testsprache sollte erlauben, ohne großen Aufwand komplexe fachliche Szenarien abzubilden.

Zur Umsetzung der Tests griffen wir auf FIT zurück. Da die zu testende Anwendung auf Eclipse basierte, wir aber auch ohne die Benutzungsoberfläche testen wollten, implementierten wir Adaptercode, der unterhalb der Benutzungsoberfläche direkt auf die fachliche Funktionalität zugriff. Die Tests speicherten wir in einem projektweit verfügbaren Wiki.

Das Ergebnis war ein überwältigender Fehlschlag. Das Editieren von Tests im Wiki wurde von den Fachtechnikern, die komfortable Werkzeuge wie Word und Excel gewöhnt waren, nicht angenommen. Vor allen Dingen das Editieren von Tabellen war zu umständlich. Die Testsprache blieb immer unvollständig, da wir Entwickler mit der Implementierung des Adaptercodes für die schnell wachsenden Funktionalitäten nicht mithalten konnten. Änderungen an der Oberfläche führten manchmal dazu, dass Tests erfolgreich waren, obwohl die Funktionalität auf der Oberfläche nicht funktionierte und umgekehrt.

Aus dem ersten Versuch, automatisierte Akzeptanztests zu etablieren, blieben uns zahlreiche Testfälle, die einen großen Teil der Anforderungen abdeckten. Wegen veralteter oder fehlender Adapter waren sie aber nicht automatisiert durchführbar und damit auch nicht gepflegt. Wieder auf manuelle Durchführung umzusatteln, war für uns nicht akzeptabel. Also wagten wir einen zweiten Anlauf.

Die Fachtechniker forderten ein Werkzeug, mit dem sie sowohl die Tabellen der Akzeptanztests als auch den erklärenden Text auf einfache Weise bearbeiten konnten. Die Wahl fiel auf Microsoft Word, da alle Fachtechniker es regelmäßig nutzten. Um zu erreichen, dass die Testergebnisse die auf der Benutzungsoberfläche verfügbare Funktionalität widerspiegeln, griffen wir mit der neuen Testsprache direkt auf Oberflächenkomponenten zu und beschrieben damit Testfälle als Abläufe von Benutzerinteraktionen. Das gab bei der Testdurchführung ein gutes Feedback, da die Anwendung sichtbar durchgeklickt wurde.

Auch für die Entwickler bedeutete der neue Ansatz einen geringeren Aufwand. Adapter mussten nicht mehr für eine schnell wachsende Zahl von fachlichen Funktionalitäten erstellt werden, sondern nur für eine feste Anzahl von Oberflächenkomponenten.

Im zweiten Anlauf setzten sich die Akzeptanztests durch. Zug um Zug wurden die alten Tests in die neue Testsprache übersetzt. Die Testsuite umfasst über 300 Testfälle mit über 10.000 Einzelschritten, die einen Großteil des Systems in etwa 4 Stunden automatisiert prüfen. Auch unser Kunde erstellt mittlerweile automatisierte Tests mit unserem Werkzeug.

Mit der Testsprache können wir alle Standardoberflächenkomponenten wie Tabellen, Eingabefelder, Schaltflächen und Bäume ansprechen. Dabei ist sie unabhängig von fachlicher Funktionalität geblieben, so dass wir auch in zukünftigen Eclipse-Projekten versuchen werden, sie einzusetzen. Da wir mit der Testsprache auch Tests für nicht vorhandene Funktionalität schreiben können, ist sogar akzeptanztestgetriebene Entwicklung möglich.

10.38 Oberflächentests

Fragile GUI-Tests

Über Tests, die direkt auf der grafischen Benutzeroberfläche aufsetzen, hört man häufig, wie extrem anfällig diese Sorte von Tests gegenüber Änderungen an der Oberfläche sind. Was wahrlich kein Wunder ist, denn selbst wenn wir sämtlichen Code der Benutzeroberfläche fein säuberlich von der Anwendungs- und Domänenlogik trennen, bleiben oberflächlich betrachtet unzählige Aspekte zu testen:

- **Darstellung:** Form, Farbe, Layout
- **Interaktion:** Ein-/Ausblenden/-grauen, Selektion, Drag & Drop, Sichtenaktualisierung
- **Benutzbarkeit:** Ergonomie, Unterstützung, Navigation, Konsistenz
- **Bildschirmübergänge:** Formularabfolge, Sessionmanagement
- **Zugangsrechte:** Benutzerprofile, Rollenwechsel
- **Lokalisierung:** Übersetzungen, landessprachliche Anpassungen

Deshalb müssen wir uns bei Oberflächentests sehr genau überlegen,

- welche Oberflächendetails wir eigentlich absichern wollen,
- was wir automatisiert testen und
- was wir visuell inspizieren können,
- welche Aspekte wir zusammen und
- welche wir getrennt testen sollten.

Zu frühe Fixierung vermeiden

Als Faustregel gilt, Tests nicht verfrüht auf ein konkretes GUI-Design auszurichten. Viele Darstellungsaspekte, obwohl einmal festgelegt, ändern sich später noch einmal. Schlecht, wenn dann unzählige Tests betroffen sind. Eine andere Erfahrung ist, dass sich die notwendige Testsprache nicht herauskristallisieren wird, wenn wir nicht einfach einmal einen Anfang wagen. Denn eine Testsprache, deren Vokabeln uns einzig und allein dazu ermächtigen, »Klicke hier und schaue dort« auszudrücken, führt zu einer wahnsinnigen Duplikation in den Tests.

Gesprächige Tests

Mein Rat ist deshalb, mithilfe der zwei zuvor beschriebenen Sorten von Tests zunächst eine Stabilisierung der Testsprache abzuwarten. Ohne domänenspezifische Testsprache ist aus meiner Sicht nicht an Testgetriebene Entwicklung auf Basis von Oberflächentests zu denken. Das ergibt ein kleines, aber lösbares Henne-Ei-Problem.

Schleichende Duplikation in Testdaten und Testcode ist der ärgste Feind unserer Anstrengungen. Halten Sie sich deshalb das Prinzip des isolierten Testens vor Augen: Jedes Testproblem lässt sich durch eine weitere Indirektionsstufe lösen.

Akzeptanztests von Oberflächendetails entkoppeln

Durch kontinuierliche Abstraktion bildet sich nach und nach eine entkoppelnde Übersetzungsschicht heraus, die in der Lage ist, die Drift zwischen Testsprache und Oberfläche zu einem gewissen Grad abzupuffern.

Finden Sie heraus, an welchen Orten Änderungen an der Tagesordnung sind, und erleichtern Sie Änderungen an diesen Orten.

Zusätzliche Technologieadapter notwendig

Um eine Oberfläche programmatisch über unsere Fixture zu steuern, sind in aller Regel zusätzliche Adapter erforderlich. Diese Adapter übernehmen dann alle nötigen Anpassungen für die Anbindung an eine konkrete Oberflächentechnologie: Für die Bedienung einer Swing-Anwendung müssen wir beispielsweise die Eingabefelder ausfüllen können, Ausgabefelder lesen und hunderte von Mausklicks simulieren. Für eine Webanwendung müssen wir Hyperlinks abgrasen können, dynamische Seiteninhalte prüfen und so weiter. Je nachdem, welche Art von Oberfläche wir ansteuern wollen, existieren verschiedene Open-Source-Bibliotheken, die den Integrationsaufwand erleichtern:

Open-Source-Helferlein

- **JFC/Swing:** Jemmy [jmy] bietet für Swing-Komponenten passende Operatoren für die Fernsteuerung über die Event-Queue.
- **Webanwendungen:** jWebUnit [jwu] emuliert einen Webbrowser, navigiert so die Anwendung und validiert die Seiteninhalte.
- **XML-Dokumente:** XMLUnit [xml] prüft einzelne XML-Elemente, vergleicht ganze Bäume oder wertet XPath-Ausdrücke aus.
- **SWT/JFace:** Traversieren von Composites, Abschicken von Events und Benachrichtigen der Listener sind ohne Umwege möglich.

10.39 Kundenfreundliche Namen

Die `ActionFixture` ist in der Lage, die `press`-, `enter`- und `check`-Namen in Camel-Case-Methodennamen zu konvertieren. Der Grund dafür ist, dass die `ActionFixture` die Brücke zur Welt der GUI-Elemente schlägt, wo die Namenszwänge einer Programmiersprache ungeeignet sind. Die `Column`- und `RowFixture` unterstützen diese für die Kunden ohnehin viel freundlichere Namenskonversion erst seit kurzem. Was zunächst wie ein reines Gimmick aussehen mag, kann auf Kundenseite jedoch empfindlich schnell über die Akzeptanz des Testansatzes entscheiden. Deshalb gehe ich davon aus, dass solche Programmierkonventionen wie die Klammern hinter unseren Methodennamen mit der nächsten FIT-Version nicht mehr bis zum Kunden durchscheinen werden.

Kundenfokus bewahren

10.40 FitNesse

Eine Idee, die Ward Cunningham mit FIT von Anfang an verfolgt hat, ist die Integration mit einem Wiki, dessen Geistesvater er ja auch ist. Für alle, die nicht wissen, was ein Wiki ist: Ein Wiki ist eine Website, die von jedermann direkt geändert werden kann. Auf jeder Webseite prangt ein Knopf zum Editieren des Seiteninhalts. Wikis ermöglichen das *Publizieren auf Knopfdruck*, was ursprünglich Tim Berners-Lees Vision für das *World Wide Web* war. Wiki-Seiten werden automatisch miteinander verlinkt, indem man den Namen einer Wiki-Seite beim Tippen einfach zusammenlaufen lässt, also ohne Leerzeichen und mit Großbuchstaben im Namen schreibt. »WikiWikiWeb« wäre beispielsweise so ein gültiger Wiki-Name. Existiert eine Wiki-Seite mit diesem Namen bereits, wird automatisch dorthin verlinkt. Existiert sie nicht, kann sie auf Knopfdruck neu angelegt und verlinkt werden.

Was ist ein Wiki?

FitNesse [fin] ist ein Wiki-Server mit FIT-Integration. Mit FitNesse können Sie Ihre Akzeptanztests im Handumdrehen im Wiki erstellen, pflegen, organisieren und von dort auch direkt ausführen. Auf der rechten Buchseite sehen Sie eines unserer Testszenarien in FitNesse, bestehend aus ein paar Testtabellen und ein wenig Dokumentation. Links oben haben wir den »Test«-Knopf, um die Seite mit FIT auszuführen. Was Sie rechts sehen, ist das Ergebnis eines solchen Testlaufs. Die etwas dunkleren Tabellenzellen müssen Sie sich in vertrauensvolles Grün gefärbt vorstellen. Ebenfalls links oben finden Sie den »Edit«-Knopf, der uns die Wiki-Seite direkt im Webbrowser editieren lässt. Textinhalte erstellen Sie wie gewohnt. Spezielle Auszeichnungen wie Überschrift und Fettschrift erfolgen über eine *Wiki Markup Language*: Für einen Tabellenstrich tippen Sie zum Beispiel ein Pipe-Symbol: |

VideoStore.

TestVideoStore

Test Results

Tests Executed OK

Test
Edit
Versions
Properties
Refactor
Where Used
RecentChanges
Files
Search

Assertions: 21 right, 0 wrong, 0 ignored, 0 exceptions

Inventar

Zur Neuanlage benötigen wir den Filmtitel der DVD und ihre Preiskategorie.
Die Filmnummern werden aufsteigend vom System vergeben.

Action Fixture		
start	Movie Administration	
press	new movie	
check	movie number	1
enter	movie title	Pulp Fiction
enter	price category	regular price
press	save	

Wir geben zwei weitere Filme ein.

Movie Entry			
number	title	category	valid ?
2	Kill Bill	new release	true
3	Reservoir Dogs	regular price	true

Verleih

Pro Ausleihvorgang wird die Filmnummer und Leihdauer gespeichert.
Aus Leihdauer und Preiskategorie berechnen sich die Kosten.

Rental Entry		
movie number	days rented	charge ?
1	3	1.50
3	5	4.50
2	3	3.75
3	2	1.50
1	1	1.50
3	6	6.00

Erlöse

Die Abfrage von Ausleihvorgängen erfolgt nach Filmnummer.
Zur Kontrolle wird der Filmname und die Gesamtzahl Verleihtage ausgegeben.

Rental Listing			
movie number	movie title ?	total days rented ?	total charge ?
1	Pulp Fiction	4	3.00
3	Reservoir Dogs	13	12.00
2	Kill Bill	3	3.75

[.FrontPage] [.RecentChanges]

Abb. 10–59
Testausführung im FitNesse-Wiki

Wenn Sie aufmerksam hinschauen, sehen Sie, dass FitNesse kundenfreundliche Namen (engl. *Graceful Names*) für Fixture-Klassen, Felder und Methoden gestattet. Als Alternative zu Methodenklammern darf sogar ein Fragezeichen gesetzt werden. Alles extrem kundengerecht!

Kundenakzeptanz

Was leider überhaupt nicht kundengerecht ist, sind die Editiermöglichkeiten der Textarea-Box im Browserfenster. Nach mehr als zehn Jahren Internet hat sich an dessen Primitivität nichts geändert. Kurzer Rede langer Sinn: **Akzeptanztesten beginnt mit *Akzeptanz* und endet mit *Testen*.** Die Personen, die die Akzeptanztests hauptsächlich schreiben und pflegen, die Ansprechpartner auf Kundenseite oder die Domänenexperten der Fachabteilung, müssen mit dem Wiki arbeiten wollen, denn sonst schläft das Akzeptanztesten früher oder später ein. Meist sind diese Personenkreise aber eifrige Office-Nutzer und lehnen das Wiki kategorisch ab, weil es im Vergleich massiv umständlich ist, um etwa Tabellenzeilen zu vertauschen oder neue Spalten einzufügen. Wundern Sie sich also nicht, wenn der Ruf nach Word und Excel laut wird. FitNesse unterstützt zwar auch ein Einfügen von und zu Excel, ein wirklich gangbarer Weg ist dies aus meiner Erfahrung jedoch nicht. Eine *reiche* Weboberfläche auf AJAX-Basis *(Asynchronous Javascript and XML)* wäre sicherlich die Lösung aller Akzeptanzprobleme.

Paste from/to Excel

Für uns Entwickler ist FitNesse allerdings in der Tat ein tolles Spielzeug. FitNesse ist im Gegensatz zum Ur-Wiki [www] hierarchisch, wodurch Testsuiten, Setup- und Teardown-Seiten ermöglicht werden. Wenn Sie die Tests also selber schreiben, weil Ihr Kunde keine liefert, ist FitNesse vielleicht eine gutes Werkzeug. Obwohl Sie eigentlich das Problem lösen müssten, warum Ihr Kunde Ihnen keine Tests liefert. Denn Sie testen dann zwar eifrig, aber unter Umständen die ganze Zeit an den Anforderungen Ihrer Auftraggeber vorbei.

Organisation von Testsuiten und Setup/Teardown

10.41 FitLibrary

Zum Abschluss möchte ich Ihnen noch eine interessante Alternative zur `ActionFixture` vorstellen:

Abb. 10–60 *DoFixture zur Beschreibung von Abläufen*

DoRentals						
client	#560 V. Vega	rents movie	#1 Pulp Fiction			
client	#344 B. Kiddo	rents movie	#2 Kill Bill			
client	#344 B. Kiddo	returns movie	#2 Kill Bill	after	2	days
check	client	#344 B. Kiddo	is charged	EUR 2.00		

Rick Mugridges `DoFixture` [mu$_{05}$] ist Teil seiner *FitLibrary* [fil] und übrigens auch schon in der Distribution von FitNesse mitenthalten. Das Format dieser Fixture ist äußerst wandlungsfähig und die Sprache dieser Tests ist sehr natürlich.

Natürlichsprachliche Akzeptanztests

10.42 Akzeptanztesten aus Projektsicht

Eine oft gestellte Frage lautet: Wann schreiben wir die Akzeptanztests? Da ihre Formulierung der eigentlichen Entwicklung vorausgehen soll, könnte man der Idee verfallen, man müsste zu Projektbeginn zunächst einmal alle Akzeptanztests spezifizieren. Nun, selbst wenn diese Tests als ein Abnahmekriterium für die Vertragserfüllung herhalten können, ist es keinesfalls sinnvoll, über einen so großen zeitlichen Horizont in Vorleistung zu gehen. Es wäre purste Verschwendung. Informationen veralten, Ihr Verständnis wächst, Anforderungen ändern sich.

Pull-Modell

Testgetriebene Entwicklung folgt dem *Pull-Modell*: Just-in-Time-Tests für Just-in-Time-Anforderungen. Wie Unit Tests schreiben wir also auch die Akzeptanztests erst unmittelbar, bevor wir sie tatsächlich benötigen. Das bedeutet, dass sie rechtzeitig zu der Iteration erstellt werden müssen, wenn das neue Feature realisiert werden soll:

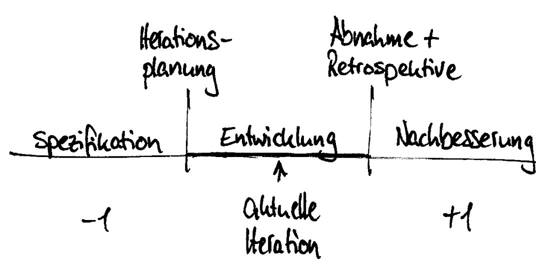


Abb. 10–61 *Zeitgerechtes Anforderungs-management*

Die Spezifikation der Tests geht ihrer Umsetzung also immer um einige wenige Tage voraus, damit sie zur Iterationsplanung auch garantiert zu Rate gezogen werden können. Ein vertrauensbildendes Ritual ist, zum Ende jeder Iteration die neu entwickelten Funktionalitäten vom Kunden selbst vorführen und damit gleichzeitig abnicken zu lassen. Kann etwas nicht akzeptiert werden, weil es nicht fertig geworden ist oder nicht wie erwartet funktioniert, folgen daraus Nachbesserungen in der anschließenden Iteration.

Zur Iterationsplanung auf »Good Day«-Szenarien konzentrieren

In aller Regel müssen die Akzeptanztests zur Iterationsplanung noch nicht alle Anforderungen in vollem Detailreichtum beschreiben. Meistens reicht es aus, wenn wenigstens die »Good Day«-Szenarien abgedeckt sind. Wichtig ist erst einmal, dass das für die anstehende Entwicklung nötige Verständnis der Anforderungen von den Fachleuten zu den Technikern transportiert wird. Der richtige Detaillerungsgrad ist dabei erreicht, wenn es leicht gelingt, die geforderten Features im Iterationsplanungsworkshop in technische Aufgaben zu zerlegen und eine realistische Aufwandsschätzung für sie abzugeben.

Zur Entwicklung durch »Bad Day«-Szenarien verfeinern

Während der Iteration wird dann der genaue Funktionsumfang durch die Hinzunahme weiterer Testfälle und »Bad Day«-Szenarien verfeinert. Natürlich sollte das Entwicklungsteam jetzt nicht mehr vor große Überraschungen gestellt werden, denn das würde nur bedeuten, dass die nötigen Details vorher unzureichend kommuniziert wurden. Anforderungen sind ein Dialog, kein Dokument. Die Akzeptanztests sind lediglich die Resultate dieser Konversation.

Anforderungen sind ein Dialog

Das Formulieren der Akzeptanztests zwingt unsere Kunden dazu, über ihre fachlichen Anforderungen in einer Detailtiefe nachzudenken, die ganz häufig eine Reihe von bisher ungestellten Fragen aufwirft. Akzeptanztests sind also ein wichtiges Kommunikations- und Analysewerkzeug, da sie die Kommunikations- und Wissenslücke zwischen Domänenexperten und Entwicklern schließen. Johannes Link bringt es auf den Punkt: *Testen heißt, die Sprache des Kunden zu sprechen.*

Akzeptanztests als Kommunikations und Analysewerkzeug

Rolle der Tester

Der Tester bekommt dadurch eine ganz neue, viel aktivere Rolle im Entwicklungsprozess. Er kann jetzt den Kunden dabei unterstützen, die Lücke zwischen dem, was das System bereits leistet, und dem, was es einmal leisten soll, zu schließen. Seine Aufgabe ist es, versteckte Annahmen in den Anforderungen zu enthüllen und in automatisierten Tests explizit zu machen. Ferner fällt ihm die Aufgabe zu, noch all die Akzeptanztests zu schreiben, die der Kunde vielleicht vergessen hat, für die er anderweitig zu beschäftigt oder einfach nur zu faul war.

Versteckte Annahmen explizit machen

Zum Iterationsende sollten alle Akzeptanztests erfolgreich laufen. Während wir unsere Unit Tests zu jeder Integration auf Grün bringen, ist diese Forderung für Akzeptanztests selbstverständlich nicht haltbar. Es ist nicht so ungewöhnlich, dass die Entwicklung umfassender neuer Features übergangsweise mit den bestehenden Akzeptanztests bricht. Sorgen sollten Sie sich allerdings machen, wenn eine Iteration keinerlei Aufwärtstrend erkennen lässt.

Akzeptanztests als Metrik für den Projektfortschritt

Für das Projektmanagement stellen featurebasierte Akzeptanztests ein gutes, weil untrügliches Mittel dar, um den tatsächlichen Projektfortschritt messen und damit managen zu können. Stellen Sie deshalb sicher, dass das Akzeptanztesten mit der Entwicklung Schritt hält und umgekehrt. Meist gelingt es nicht, große Vorsprünge später mit den Akzeptanztests wieder aufzuholen. Nachgelagerte Tests verspielen außerdem einen Großteil ihres eigentlichen Wertes, helfen sie doch weder bei der Anforderungsanalyse, noch lenken sie unser Design in Richtung Testbarkeit.

11 Änderbare Software

Wenn wir darüber nachdenken, was Software eigentlich ausmacht, muss man meiner Meinung nach zu dem Schluss kommen, dass es ihre hohe Änderbarkeit ist. Ihrer Flexibilität verdankt sie ihr *wesentlichstes* Qualitätsmerkmal und den Grund dafür, warum wir sie entwickeln: Software ist im Vergleich zu Hardware leicht änderbar.

So weit zumindest die Theorie. In der Praxis zeigt sich jedoch oft ein anderes Bild, in dem Entwicklungsteams von ihrem eigenen Code ausgebremst werden. Von wegen leichte Änderbarkeit ... ganz im Gegenteil! Doch woran hapert's? Nun, natürlich hängt sehr viel vom Entwicklungsprozess ab und der nach wie vor wohl am häufigsten angewandte Prozess ist leider »Code & Fix«.

Tatsächlich halte ich Testgetriebene Entwicklung für den ersten *wirklichen* Softwareentwicklungsprozess. Welcher Prozess macht so viele Aussagen darüber, wie Minute für Minute programmiert wird? Welcher andere Prozess hält die funktionale und strukturelle Qualität so hoch? Welcher Prozess ist näher am Code? In diesem letzten Kapitel möchte ich Ihnen einige Ideen liefern, wo die Zukunft und Chancen dieses Prozesses liegen.

Harte Prozesse führen zu harten Produkten

von Dierk König, Canoo Engineering AG

Wir haben versucht, Software so zu bauen wie Ingenieure. Dabei konstruieren Ingenieure doch Hardware.

Nun stellen wir fest: Unsere Software ist zu schwer anzupassen, sie ist zu »hart«.

Wenn harte Verfahren harte Produkte erzeugen, dann können wir vielleicht mit adaptiven Verfahren auch anpassbare Produkte erzeugen.

11.1 Konstantes Entwicklungstempo

Softwareentwicklung ist ein Marathon, kein Sprint

Die ultimative Metrik, wie gut ein Entwicklungsprozess funktioniert, ist für mich, ob ein Projekt über längere Zeiträume mit konstantem Tempo neue Anforderungen umsetzen kann. Ohne chronisches Überstundenleiden, versteht sich, und natürlich unter der Voraussetzung, dass jederzeit, zumindest aber in kurzen regelmäßigen Releasezyklen ausgeliefert werden kann. Die große Kunst der Softwareentwicklung und des Managements besteht für mich genau darin, dafür zu sorgen, dass eine gleichmäßige Entwicklungsgeschwindigkeit nachhaltig über den gesamten Projektverlauf erhalten bleibt.

Notwendigkeit für ein Turnaround Management

Mir gefällt diese Metrik, weil sie die nötigen Anknüpfungspunkte zur Prozessverbesserung bietet. Zum einen können wir den Ursachen auf den Grund gehen, warum unsere Entwicklungsgeschwindigkeit wie so häufig mit fortschreitender Projektdauer immer weiter absinkt. Steht der Code von gestern den Anforderungen von heute im Weg? Schieben wir einige nötige Refactorings nun schon zu lang vor uns her? Ist der Code *buggy*, da wir aus Zeitdruck die Tests weggelassen haben? Aus der Ursachenforschung erhalten wir wertvolle Hinweise darüber, wie wir einen Trend vielleicht noch umkehren können.

Aufgaben für das Projektmanagement

Zum anderen ermöglicht die Metrik, ein Projekt so zu managen, dass es eben gar nicht erst zu einer Schieflage kommt. Beispielsweise, indem die von uns geschätzten Aufwände für Refactoringaktivitäten, Akzeptanz- und Unit Tests einerseits schon bei der Iterationsplanung in unsere jeweiligen Aufgaben eingerechnet werden und die Planung andererseits auch stets ausreichend *Slack* für Unerwartetes bietet. Langfristiges Handeln zahlt sich aus, denn aufgeschobene Tests und Refactorings entsprechen einem *technischen Kredit*, den wir später mit Zins und Zinseszins abtragen müssen.

Geschwindigkeit ist in der Physik die zurückgelegte Wegstrecke pro Zeit. Wobei der Weg immer *richtungsbehaftet*, also ein Vektor ist, was die Geschwindigkeit ebenfalls zu einer vektoriellen Größe macht. In der Softwarewelt haben wir dieselben Verhältnisse: Unser Projektfortschritt pro Zeiteinheit ist unsere Entwicklungsgeschwindigkeit. Wobei bei uns die Richtung, in der wir Fortschritt erzielen, eine ganz kritische Rolle spielt. Denn es ist ziemlich gleichgültig, wie schnell wir vorwärts kommen, wenn wir dabei in die falsche Richtung laufen. Akzeptanztests sind wichtig, weil sie der Entwicklung ihre *Richtung* geben, während Unit Tests uns den *Fortschritt* auf dem Weg absichern, damit »fertig« bleibt, was »fertig« ist. Da der Kunde an der Erfüllung seiner Akzeptanztests interessiert ist, stellt ihr Erfüllungsgrad auch das ideale Maß für den erzielten Projektfortschritt dar.

Die wenigsten Projekte haben das Problem, zu schnell zu sein. Wenn Sie in Ihrem Projekt daher nur die Beseitigung der Flaschenhälse in Angriff nehmen, setzen Sie den Hebel schon an der richtigen Stelle an. Lassen Sie sich dabei nicht von Rückschlägen entmutigen. Zum Beispiel: Alten Code änderbar zu halten (oder zu machen), ist häufig mehr ein psychologisches als ein technisches Problem.

Alten Code testgetrieben weiterentwickeln

von Juan Altmayer Pizzorno und Robert Wenner, Port25 Solutions

Port25 Solutions entwickelt einen Hochgeschwindigkeits-Mailserver namens PowerMTA (Power Mail Transfer Agent) für Linux, Solaris und Windows NT, der Millionen E-Mails pro Stunde zustellen kann.

Eine Komponente von PowerMTA ist der DNS-Resolver, der die Mailhosts einer Domain ermittelt. Dieser Vorgang läuft in zwei Schritten ab: 1) Auflösen der Domain und Ermitteln der für diese Domain zuständigen Mailserver sowie 2) Ermitteln der IP-Adressen der im vorherigen Schritt erfassten Mailserver. Einige Antworten des DNS beinhalten bereits alle benötigten Informationen, manchmal sind auch mehrere Anfragen nötig. Der Resolver muss asynchron arbeiten, darf nicht blockieren, während er auf eine Antwort wartet, und muss mehrere Anfragen parallel abarbeiten können. Um die Verarbeitung weiter zu optimieren, werden Antworten in einem Cache gehalten und die Last auf alle verfügbaren Nameserver verteilt, indem diese der Reihe nach beschickt werden.

PowerMTA wird seit 1999 in C++ entwickelt und nutzte anfangs keine Exceptions, weil damals die Unterstützung durch Tools noch nicht ausreichend war. Manche Teile des Codes sind eher C mit Klassen oder ein nicht objektorientierter Prototyp, der mit der Zeit gewachsen ist, ohne Refactoring oder (automatisierte) Tests. Um den Resolver weiterzuentwickeln und Fehler zu beheben, benötigten wir Testfälle, die das bisherige Verhalten dokumentierten und die wir immer wieder laufen lassen konnten, um sicherzustellen, dass unsere Änderungen nichts kaputt gemacht hatten. Doch der monolithische Code bot wenige Ansatzpunkte, wo wir Mock-Objekte einbringen konnten. Der Cache des Resolvers war bereits getrennt testbar, aber der eigentliche Resolver hing sehr stark vom Cache ab. Das UDP-Socket, die Netzwerkverbindung, war ebenfalls bereits als eigene Klasse vorhanden und somit austauschbar. PowerMTA nutzt aus Geschwindigkeitsgründen mehrere Threads und aufgrund dieser Parallelverarbeitung war ein Großteil der Funktionalität nicht isoliert testbar.

Unser erster Schritt war daher, den Code aufzubrechen. An besonderen Stellen, an denen bisher einfach ein Objekt erzeugt wurde, änderten wir die alte Resolver-Klasse so, dass sie dieses Objekt als Parameter annahm. Beispielsweise wollten wir das Socket austauschen, um nicht bei jedem Test wirkliche Anfragen über das Netz zu schicken und damit von Problemen im Netz abhängig zu sein. Wir wollten DNS-Pakete simulieren, die wir von unserem »Dummy-Netz« an den Resolver übergaben.

Wir fingen also mit zentralen Objekten an und brachten Testobjekte ein. Diese Umstellung war anstrengend und erforderte viel Konzentration. Wir waren froh, dass wir dies mit Paarprogrammierung in Angriff nehmen konnten und zwei Paar Augen darauf geachtet haben, ob nicht durch diese Änderung etwas kaputt gehen konnte – wir hatten ja noch kein Sicherheitsnetz!

Dieses schufen wir uns im nächsten Schritt: Wir entwarfen Tests auf vergleichsweise abstraktem Niveau, wie Anfragen mit Timeout, Anfragen nach nicht existierenden Domains und Anfragen, die in mehreren Antworten beantwortet wurden. Wir schrieben Tests zur Dokumentation, wie der alte Resolver mit unvollständigen Antworten umging, wie er offensichtlich falsche Antworten behandelte und wie er mit kaputten Antwortpaketen verfuhr. Alle diese Testfälle waren lang (30 Zeilen und mehr) und unhandlich, weil eben immer alles zusammen getestet werden musste. Teilweise waren Verrenkungen nötig, um neue Testobjekte in den Resolver-Code einzubringen, aber das wurde von Mal zu Mal einfacher, denn wenn wir jetzt etwas änderten, um Tests schreiben zu können, hatten wir immerhin schon ein paar andere Tests, um zu prüfen, dass wir nichts dabei kaputt machten. Letztendlich hatten wir eine Testsuite von 25 Tests, welche die wesentlichen Merkmale des bisherigen Resolvers abdeckten.

Anschließend nutzten wir ein Fassaden-Muster, um dem alten Code die gewohnte Schnittstelle zu bieten und dahinter ein völlig neues Design zu erstellen. Dann arbeiteten wir uns von zwei Seiten vor: Auf der einen Seite gingen wir vom Netz in Richtung Anwendung. Hier sind viele Details durch die RFCs (Syntax und Semantik von Anfragen und Antworten) und das Betriebssystem (Netzzugriff) vorgegeben. Wir modellierten ein DNS-Paket und eine DNS-Anfrage. Auf der anderen Seite des Designs beschäftigten wir uns mit den Anforderungen von PowerMTA, des Cachings und der Parallelität. Die einzelnen Klassen entwickelten wir dann testgetrieben, wobei wir teilweise Code des alten Resolvers wiederverwendeten.

Nun ist parallel laufender Code nur schwer zu testen. Daher liefen im Wesentlichen die Tests sequenziell. Um den Code trotzdem in jeglicher Hinsicht zu testen, schrieben wir spezielle Mock-Objekte, die in den Tests an genau vorher bestimmten Stellen einen zweiten Thread simulieren. So konnten wir mit einem Test einen Deadlock simulieren und danach den Code schreiben, der das Problem behebt.

Unsere Überarbeitung machte aus einer unwartbaren Klasse eine Hierarchie von 18 Klassen, von denen 4 Klassen reine Interfaces oder abstrakte Klassen sind. Alle Klassen sind handlich, wartbar und sauber strukturiert. Der neue Resolver leistet zudem mehr als der alte. Während der Entwicklung stießen wir auf eine Situation, die von unseren Tests nicht abgedeckt wurde. Wir schrieben einen Test für den alten Resolver und fanden tatsächlich einen Fehler. Wir schrieben einen Test für den neuen Resolver und fanden auch hier den Fehler. Es war ein Aufwand von einer Stunde, ihn *test-first* zu beheben. Diesen Fehler fanden wir nur, weil wir den Code in kleine Stücke aufgebrochen hatten, die wir in Isolation testen konnten.

Analog konnten wir neue Funktionalität einbauen: Der Cache wurde bisher beim Start initialisiert. Er sollte nun im Betrieb manuell gelöscht werden können. Innerhalb einer halben Stunde hatten wir dieses Feature getestet und implementiert. Zusätzlich kann der Cache pro Domain manuell gelöscht werden – ebenfalls eine halbe Stunde Aufwand. Im alten Code wäre das schätzungsweise (inklusive Tests) ein Aufwand von mehreren Tagen gewesen.

In alten Codebeständen steckt oft ein beträchtlicher Wert. Doch häufig ist dieser *Legacy Code* weder einfach änderbar noch einfach testbar. Dieses Dilemma führt viele zu dem Glauben, dass der Testgetriebenen Entwicklung damit ein Sperrriegel vorgeschoben ist. Dabei eignen sich die Techniken hervorragend, um bestehende Funktionalitäten noch so lange zu erhalten, bis diese abgeschaltet oder ersetzt werden können.

Michael Feathers führt in seinem Buch [fe_{04}] allerlei Techniken an, um störende Abhängigkeiten zu brechen, um Testpunkte einzubringen, um sichere Änderungen durchzuführen. Eine essenzielle Technik ist, vor jeder Änderung zunächst das vorhandene Verhalten des Codes und selbst sein Fehlverhalten mit *charakterisierenden Tests* zu fixieren. Michael hat eine recht extreme Definition von Legacy Code, die auch erklärt, warum das sinnvoll ist: »**Legacy Code ist Code ohne Tests.** [...] Code ohne Tests ist schlecht. [...] Es ist egal, wie gut er geschrieben ist. Mit Tests können wir Code schnell und zuverlässig ändern. Ohne Tests wissen wir praktisch nie, ob der Code besser oder schlechter wird.«

Legacy Code = Code ohne Tests

Die Latte liegt jetzt höher

von Michael Feathers, Object Mentor

Ich bin überzeugt, in unserer Branche breitet sich eine Kluft aus und sie wird immer breiter.

Ich bin es leid, auf Entwürfe zu stoßen, die zerbrechlich und schwer zu ändern sind. Ich kann es nicht mehr mit ansehen, wenn Leute sich durch ihren Code tasten, Kleinigkeiten ausprobieren und dann das ganze System neu bauen, damit sie es manuell starten und nachsehen können, ob es tut, was es ihrer Ansicht nach tun sollte.

Sie glauben, Ihr Entwurf sei gut? Nehmen Sie eine Klasse, irgendeine, und versuchen Sie, sie innerhalb einer Testumgebung zu instanziieren. Früher glaubte ich, meine damaligen Entwürfe wären gut, bis ich diesen Test einzusetzen begann. Wir können uns über Kopplung und Kapselung und solche netten Sachen unterhalten, doch wir wollen nicht nur reden, sondern Taten sprechen lassen. Bekommen Sie diese Klasse außerhalb der Anwendung zum Laufen? Bekommen Sie sie bis zu dem Punkt, dass Sie mit ihr in Echtzeit herumspielen können? Können Sie sie in weniger als einer Sekunde einzeln bauen und Tests hinzufügen, um herauszufinden, wie sie sich wirklich in gewissen Situationen verhält? Allerdings nicht, was Sie vielleicht denken, was passieren könnte, auch nicht, was Sie hoffen, was passieren könnte, sondern wie sie sich wirklich verhält?

Das ist, wo wir im Moment stehen. Einige Teams können so mit ihren Klassen arbeiten. Andere können so nur mit einem kleinen Bruchteil ihrer Klassen arbeiten. Und wenn Sie sie danach fragen, ob sie lieber in dem Code mit oder dem ohne den Unit Tests arbeiten, ist die Antwort klar. Sie können einfach schneller arbeiten, wenn ihre Turnaround-Zeiten kürzer sind. Ihre Schritte sind sicherer, wenn sie wissen, wo sie landen.

Wer mich näher kennt, weiß, dass ich ein eher ruhiger Typ bin. Ich schimpfe nicht über allzu viele Dinge, doch ehrlich gesagt, denke ich, dass in der Softwarebranche momentan einfach zu viel auf dem Spiel steht. Wenn Sie den Unterschied noch nicht selbst gesehen haben, der diesen Arbeitsstil ausmacht, müssen Sie es einmal erleben. Die Latte liegt jetzt höher. Die Best Practices sind in den vergangenen fünf Jahren ein ganzes Stück besser geworden. Erzählen Sie mir nicht, Ihr Entwurf sei gut, wenn Sie sich noch abstrampeln.

11.2 Kurze Zykluszeiten

Aus wirtschaftlicher Sicht halte ich eine weitere Metrik für bedeutsam: Die Zykluszeit vom Zeitpunkt, wo eine neue Anforderung geplant ist, bis zum Zeitpunkt, wo das neue Feature in Produktion gehen kann, gilt es zu minimieren:

Abb. 11–1
Minimale Zykluszeiten

Warum ist es erstrebenswert? Nun, zum einen reduzieren wir dadurch die Zeit zwischen unserer Investition in die Software und unserem Gewinn aus der Software bzw. ihren Einsparungen. Effektiv ziehen wir unsere *Time-to-Market* vor und damit unseren *Return-on-Investment*. Das heißt, wir können früher Geld einnehmen, was eine gute Sache ist. Zum anderen häufen wir nicht so große Stapel *unfertiger* Software auf: Software, die noch nicht in Produktion gehen und somit auch kein Geld erwirtschaften kann, ist vom Kapitelwert her nur reines Inventar. Das heißt, wir können dadurch nicht nur früher Einnahmen erzielen, wir können auch noch langsamer Geld ausgeben.

Früher Einnahmen erzielen

Langsamer Geld ausgeben

Zudem können wir viel früher das für die Weiterentwicklung so kostbare Anwenderfeedback einholen. Der Kunde kann einerseits das Projekt viel einfacher zum Erfolg steuern, indem er uns immer gerade nur an den Features arbeiten lässt, die für ihn den größten Wert haben. Zum anderen kann der Auftraggeber das Projekt aber auch nach dem »Fail Fast«-Prinzip einstellen, wenn sich schon frühzeitig abzeichnet, dass es ohnehin scheitern oder wirtschaftlich wirkungslos sein wird. Wir reduzieren durch das frühe, häufige und regelmäßige Ausliefern also unser Risiko, dass die entwickelte Software zum Lieferzeitpunkt vielleicht schon gar nicht mehr benötigt wird.

Früh und häufig auf Anwenderfeedback reagieren

Kurze Zyklen sind Indiz eines gesunden Prozesses, weshalb auch diese Metrik eine Reihe von Möglichkeiten zum Prozess-Tuning bietet. Wir können zum Beispiel Toyota als Leitbild nehmen: Der Automobilbauer vermeidet Überproduktion und damit teures Inventar dadurch, dass nur das gefertigt wird, was benötigt wird, wenn es benötigt wird und in der Menge, in der es benötigt wird. Gleichzeitig halten sich die Japaner Optionen möglichst lange offen, indem sie Entscheidungen bis zu einem Zeitpunkt hinauszögern, wo sie sie spätestens treffen müssen ... oder ihnen sonst eine mögliche Alternative verloren geht [po03].

Pull-Prinzip in der Just-in-Time-Produktion

Last Responsible Moment

11.3 Neue Geschäftsmodelle

Wertschöpfung

Softwareentwicklung ist Wertschöpfung. Software existiert nur aus dem einen Grund, um mit ihr mehr Geld zu verdienen oder zu sparen, als ihre Entwicklung verschlingt. Unterschiedliche Anforderungen besitzen jedoch unterschiedlichen Geschäftswert. Den Geschäftswert für jede Iteration und jedes Release zu *maximieren*, muss das Ziel jedes wirtschaftlich erfolgreichen Softwareprojekts sein.

Der Geschäftswert von Anforderungen

Business Value First

Ein Projekt geschäftswertorientiert zu planen und durchzuführen bedeutet, *immer* die Anforderung mit der *größten* Wertschöpfung zuerst zu implementieren. Der prognostizierte Return-on-Investment informiert uns dabei über die anzustrebende Entwicklungsreihenfolge. Indem wir gleichzeitig die Releasezyklen von Monaten und Wochen auf Tage und Stunden reduzieren, kann sich die Entwicklung im Idealfall sogar selbst finanzieren. Inkrementelle Entwicklung wird somit zum Synonym für inkrementelle Finanzierung [de$_{04}$].

Software, die sich selbst finanzieren kann

Ein interessantes (Gedanken-)Experiment ist: Stellen Sie sich vor, Sie hätten nur Geld für eine Woche Entwicklungszeit. Zum Ende dieser Woche müssen Sie mit Ihrer entwickelten Software produktiv gehen, um sich darüber die zweite Woche Entwicklung zu finanzieren usw. Klingt extrem? Doch wie würden wir Software in dem Fall entwickeln, damit sie ab Woche eins wertschöpfend wäre und sich somit selbst finanzieren könnte? Wie würden wir entwickeln, wenn wir tatsächlich nicht wüssten, wohin die Reise eigentlich geht, wir also veränderliche Anforderungen als Chance und nicht als Problem auffassen wollten? Meine Behauptung ist, diese Rahmenbedingungen sind gar nicht so abwegig, wie sie sich anhören, beschreiben sie doch die tagtäglichen Herausforderungen in der heutigen schnelllebigen Wirtschaft.

Veränderung als einzige Konstante

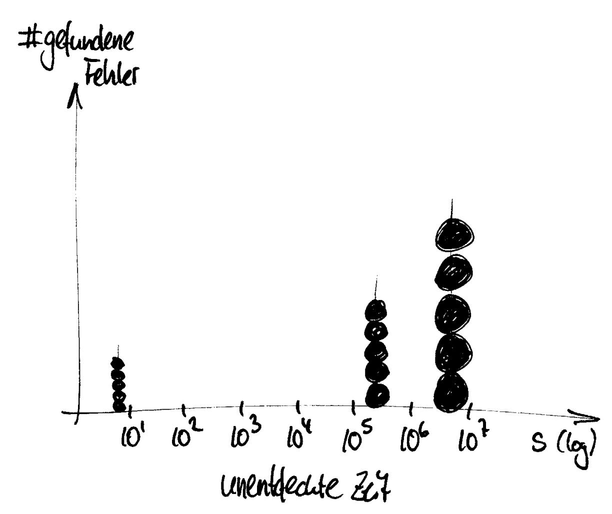
Immer häufiger sehen wir, dass *die* Unternehmen das Geschäft machen, die am schnellsten reagieren und adaptieren. *Agilität* zeichnet sich heute in den Fähigkeiten aus, einerseits auf Veränderungen im Markt schnell und angemessen reagieren zu können und andererseits ebenso schnell die Mitstreiter mit innovativen Ideen aus dem Rennen werfen zu können. Das *Beschleunigungsgesetz von Kurzweil* besagt, dass wir in den kommenden hundert Jahren so viel technologischen Fortschritt machen werden wie in den letzten zwanzigtausend Jahren. Allein in den nächsten zwei Jahrzehnten erwarten uns in etwa so viele Veränderungen wie in den gesamten vergangenen einhundert Jahren. Die Zukunft ist also auch nicht mehr das, was sie einmal war ...

Technische Singularität: www.kurzweilai.net/articles/art0134-html?printable=1

Bug-Trap-Linien

von Michael Hill, Industrial Logic

Eine Möglichkeit, über unsere Testsuiten nachzudenken, besteht darin, sie als Linien in einem Bug-Diagramm zu betrachten.



Die y-Achse des Diagramms stellt den Aufwand zum Debuggen dar, gemessen in gefundenen und behobenen Fehlern. Auf der x-Achse ist die Zeit aufgetragen, gemessen in Sekunden $_{\log 10}$. Jede vertikale Linie im Diagramm repräsentiert eine einzelne Möglichkeit, einen Fehler im Code zu finden: eine Bug-Trap-Linie. Weit auf der linken Seite nahe null Sekunden ist zum Beispiel immer eine vertikale Linie. Sie verkörpert Ihren Compiler oder Interpreter, wie er die Syntax Ihres Codes überprüft. (Natürlich gibt es für die meisten modernen Sprachen auch noch Lint-Tools, deren Analyse über bloße Syntaxchecks hinausgeht, auch sie sitzen ziemlich nahe der Null-Marke.)

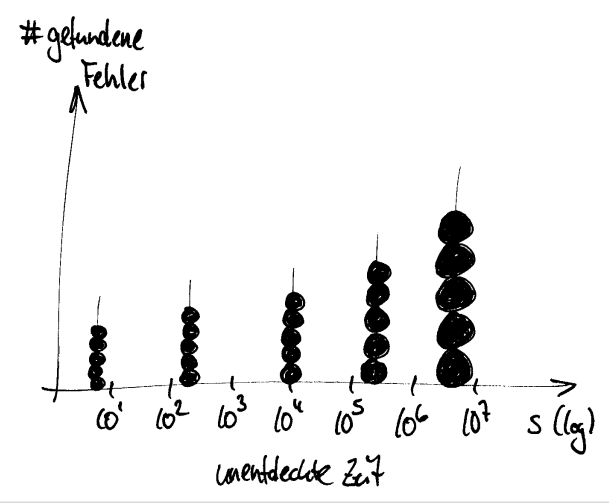
Auf dem anderen Ende der Skala liegt das tatsächliche Release der Software. Wenn Ihr Unternehmen neue Versionen vierteljährlich veröffentlicht, sitzt diese Linie knapp vor 10^7 Sekunden. Diese Bug-Trap-Linie stellt die unglückliche Situation dar, dass Ihr Kunde die Fehler für Sie findet. (Die meisten Firmen führen einmal pro Woche eine Art von Smoke-Tests durch. Diese Linie finden Sie in der Mitte des Diagramms bei $10^{5,5}$.)

Wir nehmen in unserem Beispieldiagramm an, auf jeder Linie wären jeweils fünf Fehler gefunden worden. Die vom Compiler gefangenen Fehler sind jedoch sehr klein gezeichnet, während die vom Kunden gefundenen Fehler sehr groß sind. Um diese Darstellung zu verstehen, müssen wir einen Moment lang über den Lebenszyklus eines Fehlers reflektieren. Bugs erleben vier grundlegende Ereignisse: Kreation, Konsequenz, Korrelation und Korrektur.

- **Kreation** ist der Zeitpunkt, wenn ein Fehler durch jemanden (anderen als Sie natürlich) erzeugt wird, indem er (oder sie) irgendwelchen Code tippt, der nicht ganz richtig ist. Der Fehler bleibt für eine unbestimmte Zeit im Code. Vielleicht findet Lint ihn. Oder einfaches Alphatesten vom Programmierer. Tatsächlich könnte der Fehler jedoch eine Ewigkeit lang unentdeckt bleiben. Wenn Sie in einem schwachen Moment ein Array hart darauf verdrahtet haben, 10.000 Geschäftskontakte zu halten, ohne einen Mechanismus zur Verfügung zu stellen, um mit einem Überlauf umzugehen, könnte das Versehen für Monate unbemerkt bleiben, während Ihre Anwender fleißig Namen und Telefonnummern anhäufen.
- **Konsequenz** ist der Moment, wo der Fehler endlich sein hässliches Gesicht zeigt. Einer Ihrer wichtigsten Kunden tippt den 10.001. Kontakt ein und die gesamte Software fällt auseinander. Unter extrem hohem Druck versuchen Sie, der Ursache auf die Spur zu kommen und das Problem zu beheben. Warum können Sie den Fehler nicht einfach korrigieren, wenn die Konsequenz aufkreuzt? Die Frage behandeln wir, wenn wir den Lebenszyklus abgeschlossen haben.
- **Korrelation** kommt als Nächstes. Wenn Sie ein echtes Programm haben, dann haben Sie eine Menge Code. Wenn Ihre Kunden echte Anwender sind, dann existieren viele verschiedene Pfade, die sie durch diesen Code nehmen könnten. Korrelation ist die Aufgabe herauszufinden, wo genau der fehlerhafte Code steckt. Typischerweise geschieht dies durch ein Gemisch von Tätigkeiten: 1) einen wiederholbaren Ablauf zu finden, der die Konsequenz erzeugt, 2) einen Debugger auf das Programm zu werfen und den Ablauf zu produzieren und 3) auf Ihre Eingebung zu horchen.
- **Korrektur** ist das Ende des Fehlers. Jetzt, wo Sie den defekten Code eingegrenzt haben, können Sie ihn mit größerer oder kleinerer Leichtigkeit ersetzen, je nachdem wie oft der Fehler im Code wiederholt wurde und wie viel abhängiger Code existiert.

Es vergeht stets eine gewisse Zeit zwischen der Kreation eines Fehlers und seiner Konsequenz. Vielleicht Sekunden, vielleicht auch Monate, doch immer schlummert der Fehler für eine Weile in Ihrem Code. Ebenso vergeht immer eine gewisse Zeit zwischen der Konsequenz und Korrelation, wenn der Fehler zwar schon aktiv, aber noch nicht behoben ist. Und hier wird deutlich, warum Fehler, die von Linien weit rechts im Diagramm gefunden werden, durch größere Kuller dargestellt werden: Sie erzeugen größere Aufwände. **Das Verhältnis von dem Zeitpunkt, wo der Fehler schlummert, zu dem Zeitpunkt, wo er sich aktiv zeigt, ist eine steigende und nichtlineare Kurve.** Anders ausgedrückt: Je länger ein Fehler schon schlummert, desto schwerer ist er auch zu finden. Der Aufwand, um eine lokalisierte Fehlerursache abzustellen, hängt hauptsächlich von der Natur des Fehlers ab: Schreibfehler sind ein Kinderspiel, Speicherüberläufe sind schwieriger und Race-Conditions (d.h. Wettläufe zwischen Threads) sind verdammt hart. Doch der Aufwand, die Konsequenz zurück zur Kreation zu korrelieren, ist sehr wahrscheinlich von der Zeitdauer abhängig, die der Fehler schlummern konnte. Generell gilt: Je mehr Zeit inzwischen vergangen ist, desto schwerer wird die Korrelation. Darum sind die Fehler auf der rechten Diagrammseite größer gemalt als die auf der linken.

Was können wir also tun? Betrachten wir das gleiche Diagramm, dieses Mal jedoch mit einer Menge von Unit Tests, die innerhalb von 300 Sekunden ablaufen, kurz hinter 10^2, und einer Menge von Akzeptanztests, die innerhalb weniger Stunden (10^4) ablaufen.



Sie bemerken, wie sich die Linien nun mehr oder weniger gleichmäßig ausbreiten. Diese visuelle Balance stellt für unsere Fehlerfalle auch eine strategische Balance dar. Aufgrund der Nichtlinearität der Kurve können wir nicht nur mehr Fehler finden, wir können sie auch noch schneller finden und beheben. Hier zahlt sich das Investment in Testgetriebene Entwicklung aus. Ich glaube wirklich, dass Testgetriebene Entwicklung nicht nur Code von höherer Qualität produzieren kann, sondern dies tatsächlich auch noch schneller, und der Effekt der Bug-Trap-Linien ist der Grund dafür.

Literatur

be_{96} Kent Beck: Smalltalk Best Practice Patterns.
Prentice Hall, 1996.

be_{02a} Kent Beck: Test-Driven Development by Example.
Addison-Wesley, 2002.

be_{02b} Kent Beck: Test-Driven Development Between Teams. 2002.
http://groups.yahoo.com/group/testdrivendevelopment/files/
(Mitgliedschaft in dieser Yahoo! Group vorausgesetzt)

ber_{02} Stephen P. Berczuk, Brad Appleton: Software Configuration Management Patterns. Addison-Wesley, 2002.

de_{04} Mark Denne, Jane Cleland-Huang: Software by Numbers.
Prentice Hall, 2004.

deu_{01} Arie van Deursen, Leon Moonen, Alex van den Bergh, Gerard Kok: Refactoring Test Code. 2001.
http://homepages.cwi.nl/~leon/papers/xp2001/xp2001.pdf

ev_{03} Eric Evans: Domain-Driven Design. Addison-Wesley, 2003.

fe_{01} Michael Feathers: The »Self«-Shunt Unit Testing Pattern. 2001.
http://www.objectmentor.com/resources/articles/
SelfShunPtrn.pdf

fe_{04} Michael Feathers: Working Effectively with Legacy Code.
Prentice Hall, 2004.

fo_{99} Martin Fowler: Refactoring. Addison-Wesley, 1999.

fo_{00} Martin Fowler: Is Design Dead? 2000.
http://martinfowler.com/articles/designDead.html

fo_{04} Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern. 2004. *http://martinfowler.com/articles/injection.html*

fr_{04} Steve Freeman, Nat Pryce, Tim Mackinnon, Joe Walnes: Mock Roles, not Objects. 2004. *http://www.jmock.org/oopsla2004.pdf*

ga_{95} Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Addison-Wesley, 1995.

hu_{99} Andrew Hunt, David Thomas: The Pragmatic Programmer. Addison-Wesley, 1999.

li_{05} Johannes Link: Softwaretests mit JUnit. dpunkt.verlag, 2005.

ma_{00} Tim Mackinnon, Steve Freeman, Philip Craig: Endo-Testing: Unit Testing with Mock Objects. 2000. *http://www.connextra.com/aboutUs/mockobjects.pdf*

ma_{02} Robert C. Martin: Agile Software Development. Prentice Hall, 2002.

me_{01} Steven J. Metsker: Building Parsers with Java. Addison-Wesley, 2001.

mu_{05} Rick Mugridge, Ward Cunningham: Fit for Developing Software. Prentice Hall, 2005.

pa_{02} Stephen R. Palmer, John M. Felsing: A Practical Guide to Feature Driven Development. Prentice Hall, 2002.

po_{03} Mary und Tom Poppendieck: Lean Software Development. Addison-Wesley, 2003.

ro_{04} Stefan Roock, Martin Lippert: Refactorings in großen Softwareprojekten. dpunkt.verlag, 2004.

sc_{01} Peter Schuh, Stephanie Punke: Object Mother. 2001. *http://www.xpuniverse.com/2001/pdfs/Testing03.pdf*

th_{98} David Thomas, Andrew Hunt: Tell, Don't Ask. 1998. *http://www.pragmaticprogrammer.com/ppllc/papers/1998_05.html*

we_{05} Frank Westphal. *http://www.frankwestphal.de/TestgetriebeneEntwicklungmitJUnitundFIT.html*

Werkzeugkasten

ant Apache Ant.
http://ant.apache.org

cc CruiseControl.
http://cruisecontrol.sourceforge.net

cl Clover.
http://www.cenqua.com/clover/

cvs Concurrent Versions System.
http://www.nongnu.org/cvs/

eas EasyMock.
http://www.easymock.org

ec Eclipse.
http://www.eclipse.org

fil FitLibrary.
http://fitlibrary.sourceforge.net

fin FitNesse.
http://www.fitnesse.org

fit Framework for Integrated Test.
http://fit.c2.com

int IntelliJ IDEA.
http://www.jetbrains.com/idea/

je Jester.
http://jester.sourceforge.net

jmy Jemmy.
http://jemmy.netbeans.org

juh JUnit Home.
http://www.junit.org

jus JUnit SourceForge.
http://sourceforge.net/projects/junit/

juy JUnit Yahoo! Group.
http://groups.yahoo.com/group/junit/

jwu jWebUnit.
http://jwebunit.sourceforge.net

mo MockObjects.
http://www.mockobjects.com

ref Refactoring.
http://www.refactoring.com

svn Subversion.
http://subversion.tigris.org

tdd Test-Driven Development Yahoo! Group.
http://groups.yahoo.com/group/testdrivendevelopment/

www Ward's Wiki.
http://c2.com/cgi/wiki

xml XMLUnit.
http://xmlunit.sourceforge.net

xpf Deutschsprachiges XP-Forum.
http://de.groups.yahoo.com/group/xp-forum/

xu xUnit.
http://www.xprogramming.com/software.htm

Index

A

Abfragemethode 225, 240
Abhängigkeiten 175–176
  brechen 313
  explizite 224, 232
`ActionFixture` 243–246, 306
  Akteur 245, 255
  Aktionen 243
  Aktionen hinzufügen 254
  in `ColumnFixture` wandeln 261
Adapterfunktion 81
`addToBody()` 288
`addToTag()` 288
`@After` 46
`@AfterClass` 47
Agilität 316
Akzeptanztest 2, 233, 235, 242, **307–308**
  Automatisierung 233
  natürlichsprachlich 306
`AllFiles` 276–277
`AllTests` 36
Analyse 55, 308
Änderbarkeit 5, 309
Anforderungen 232, 233, 235, 242, 247, 306, 307–308, 310, 315, 316
  Analyse 242, 308
  Änderungen 14, 91, 307, 316
  Dokumentation 234
  versteckte Annahmen 308
Anforderungsdokument
  ausführbar 234, 242
Anforderungsmanagement 307
Annotation 44
Ant 116–119
  `build.xml` 116–119
  `junitreport`-Task 118
  `junit`-Task 118
  Target 116–117
  Testlauf 118
  Testreport 118
Anwenderfeedback 315
`ArrayAdapter` 294
`Assert` **27–28**, 35, 42
  `String`-Parameter 30, 165
`assertEquals()` 27–28, 49
`assertFalse()` 27
`AssertionError` 49
`AssertionFailedError` 29–30, 42
`assertNotNull()` 28
`assertNotSame()` 28
`assertNull()` 28
`assertSame()` 28
`assertTrue()` 27
`assert()` 49
Assoziationsobjekt 102
Attrappe, *siehe* Stub-Objekt
Aufwandsschätzung 307, 310
Ausdruck
  regulärer 149

B

Befehlsmethode 225
`@Before` 46
`@BeforeClass` 47
Benutzergeschichte 235
Black-Box-Test 136, 233
`body` 280, 288
Bottom-up 71, 95–96, 221
Branches 115
Bug
  Datenbank 171
  Report 171
Build 20, 110
  Automation 113
  Erstellung 113
  Management 116
  nächtlicher 38
  Prozess 113
  Schleife 122
  Skript 110, 113, **116–119**
  Status 122

Build (Fortsetzung)
  Tuning 121
  Überwachung 122
  Verifikation 113
Business Value First 316

C

Camel Case 244, 304
`check` 243–246
`check()` 295–296
Clover 155–156
Code Smells 75
  Duplicated Code 83
  Primitive Obsession 79
Code & Fix 309
Codeabdeckung 60, 154
Codeeigentum
  gemeinsames 111, 172
Codegerüche, *siehe* Code Smells
`ColumnFixture` 238–241, 243, 256–263
  Eingabespalte 257
  Einschubmethoden 263
  Ergebnisspalte 257
  Transaktionen 263
`counts` 271
Coverage-Analyzer 154
CruiseControl 122
CVS 116

D

Datentransferobjekt 124, 270
Defekt 169
  einstreuen 157
Dependency Injection 187, 232
Dependency Inversion 187, 232
Design 56
  einfaches 12
  evolutionäres 14, 56, 91, 107
  Strategie 12, 91
  Verbesserungen 73
Design Patterns
  Adapter 212
  Builder 134
  Collecting Parameter 232
  Composed Method 220
  Composite 37
  Decorator 137
  Factory 134
Design Patterns (Fortsetzung)
  Fassade 212, 229, 251, 253
  Mediator 229
  Singleton 162, 258
  Template Method 88
  Visitor 232
Direktive
  dritte 3, **109**
  erste 2, **51**
  zweite 3, **73**
`doCells()` 282–283, 287
`doCell()` 282–284, 296
`Documents` 275
`DoFixture` 306
Domäne 297–298
Domänenexperte 242, 261, 289, 306, 308
Domänenmodell 300
Domänenobjekt 270
`doRows()` 282–283, 287
`doRow()` 282–283, 288
`doTables()` 282
`doTable()` 282–283
Dummy, *siehe* Stub-Objekt
  Crashtest 207

E

EasyMock 208–211
  Class Extension 209
  `createControl()` 210
  `createNiceControl()` 210
  `createStrictControl()` 210
  `expectAndReturn()` 210
  `MockControl` 208
  `replay()` 209
  `setReturnValue()` 210
  `setThrowable()` 210
  `verify()` 209
Eclipse 24, 26, 57
Einfache Form 3, 14, 52–53, 77, 88
`enter` 243–246
Entropie 5
Entwicklungsgeschwindigkeit 310
Entwicklungsprozess 309, 310
Entwurfsmuster, *siehe* Design Patterns
Erinnerungstest 131
`error` 259
Erwartungen 193–204
Erwartungsklassen 197

`ExampleTests` 274–275
Exception
  Checked 43
  simulieren 207
  Unchecked 43
`exception()` 284
`execute()` 263
`ExpectationList` 193
Exploration 242

F

Fail Fast 315
`fail()` 42
Fake it 'til you make it 98, 247
Feature
  Request 171
  Team 124
Fehler
  Behebung 169–171
  Kategorien 142
Fehlschlag
  frühzeitiger 194
`FileRunner` 235, 271, 278–279
FIT 233–306
  Anknüpfungspunkte 299
  Annotationsmöglichkeiten 284
  Basis-Fixtures 243, 272
  Datentypen parsen 289
  Dokument 233–234, 242, 274–277, 297
  Download 234
  `fit.jar` 234
  Fließkommazahlen vergleichen 293
  Framework 235–243, **278**
  im nächtlichen Build 279
  im Wiki 304
  Installation 234
  Integration 304
  kommaseparierte Zelleninhalte 294
  Parser 236, **279–281**
  positionelle Tabellenformate 298
  Report 236, 271
  Spalten einhängen 288
  Testlauf 235
  Testlaufergebnis 271
  Testsuite 274–277
  Wiki 234
  Zeilen einhängen 288
FitLibrary 306
FitNesse 304–306
`Fixture` 237–238, 278, **282–283**, 284
  domänenspezifisch 297–298
  Interkommunikation 258
  Lebenszyklus 258, 282
  Schreib-/Leserichtung 292
  `SetUp` 278
  `TearDown` 278
  Typadaption 290
Flaschenhälse 311
`footnotes` 275
Framework for Integrated Test, *siehe* FIT

G

Gastautoren
  Andy Hunt 92
  Bastiaan Harmsen 180
  Christian Junghans 171
  Dave Thomas 92
  Dierk König 141, 232, 309
  Ivan Moore 205
  Johannes Link 43
  Juan Altmayer Pizzorno 311
  Lasse Koskela 125, 173
  Michael Feathers 72, 314
  Michael Hill 317
  Moritz Petersen 214
  Olaf Kock 112, 171
  Robert Wenner 311
  Sabine Embacher 20
  Steffen Künzel 301
  Steve Freeman 205
  Tammo Freese 67, 71, 81, 301
  Tim Mackinnon 205
  Ward Cunningham 272
Geschäftswert 5, 316
Gesetz
  von Demeter 228
  von Kurzweil 316
`getTargetClass()` 265
Graceful Names 305
Grammatik
  domänenspezifisch 293
Guard Clause 42, 170

H

Häufige Integration 3, 20, 51, **109–126**
    Integrationszüge 110
HTML
    Dokument 235
    Export 236
    Kommentare 281
    Tabelle 239
    Tags 279–281, 284

I

`@Ignore` 48
`ignore()` 284
Implementierungssubstitution 78–79
Import
    statischer 45
Indirektion
    zusätzliche 185
Integration 109
    Bigbang 110, 124
    Build 122
    häufige, *siehe* Häufige Integration
    Integrationsserver 110, 122, 124, 276
    Konflikt 111–112
    Konflikte 20
    Staging 124
    teamübergreifend 124
    Token 112
    Umgebung 124
Integrationstest 213, 300
IntelliJ IDEA 24, 57
Inversion of Control 231
Iteration 307–308
    Planung 307, 310
Iterativ-inkrementelle Entwicklung 3

J

Jemmy 303
Jester 157–158
JUnit 21–43
    ältere Tests 24
    Benutzeroberfläche 38
    `ClassLoader` 38
    Download 21
    EKG 106
    Erweiterungen 22
    Exception Handling 43
JUnit (Fortsetzung)
    Framework 27
    in Eclipse 26
    Installation 22
    JUnit 4 44–50
    `junit.jar` 22
    mit Ant 118
    Probleme 22
`junit`-Ant-Task 118
`JUnitCore` 49
`JUnit4TestAdapter` 48
Just-in-Time-Produktion 315
jWebUnit 303

K

Kombinatorische Explosion 148
Kommunikation 242, 308
Komponenten
    vertrauenswürdige 179
Kunde 261, 306, 308
Kunden 233
Kundenakzeptanz 304, 306

L

Last Responsible Moment 315
Legacy Code 313
Lerntest 139, 242, 279
Lieferfähigkeit 115
Locking 112

M

Merging 111
Metapher vom Klettern 4
Metrik 106, 156, 159, 308, 310, 315
Mikrointegrationstest 175, 177–178
`missing` 265
mockobjects-Bibliothek 193, **197**
    `addActualMany()` 197
    `addActual()` 193
    `addExpectedMany()` 197
    `addExpected()` 193
    `ExpectationCounter` 197
    `ExpectationDoubleValue` 197
    `ExpectationList` 197
    `ExpectationMap` 197
    `ExpectationSegment` 197
    `ExpectationSet` 197
    `ExpectationValue` 197

mockobjects-Bibliothek (Fortsetzung)
Mock-Implementierungen 208
MockObject 204
setActual() 197
setExpected() 197
setExpectNothing() 197
setFailOnVerify() 197
Verifier 200
verify() 193, 197, 200
Mock-Objekt 201–204
Designstil 215
dynamisches 208
generieren 208
übertrieben 211
verify() 201, 204
more 280, 288
Mutation Testing 157–158

N

Negativtest 142, 259

O

Oberflächentest 302
Orthogonalität 144–149, 225

P

Parse 278, **279–281**, 288
parseBoolean() 295
parseDouble() 295
parseLong() 295
Parser-Generator 293
parse() 290–291
parts 280, 288
press 243–246
PrimitiveFixture 295–296
Prinzip des isolierten Testens 185, 212, 303
Programmierepisode 10
Programmierstandards 111
Programmierzüge 16
Programming by Intention 104
Projekt-Dashboard 122
Projektfortschritt 308, 310
Projektmanagement 308, 310
Protokoll 198, 211
Proxy
dynamisch 208
Prozess-Tuning 315
Prozessverbesserung 310
Pull-Modell 307
Pull-Prinzip 315

Q

Qualität
funktionale 4
strukturelle 4
query() 265

R

Refactoring 3, 6, 17, 51, 53, 60, 66, **73–108**
Durchbrüche 108
Durchsicht 90
Kandidaten 66, 121
Katalog 76
Refactoringzüge 74
Route **78**, 80–82
von Testcode 160–161
Refactorings
Collapse Hierarchy 90
Extract Class 84
Extract Interface 183
Extract Method 80
Extract Superclass 87
große 108
Inline Method 81
Inline Temp 81
Introduce Explaining Variable 79
Rename Method 82
Replace Data Value with Object 79
Substitute Algorithm 87
Refaktorisieren, *siehe* Refactoring
Referenzobjekt 22
Regressionstest 52, 274–275
Releasezyklus 310, 316
Reports 275
reset() 263
Return on Investment 315, 316
right() 284
Rollen 232
RowFixture 243, 264–270
Abfragemethode 270
einfacher Schlüssel 266
Einschubmethoden 265
mehrfacher Schlüssel 269
RSS-Kanal 122

S

Schnittstellen 177
- entdecken 218
- Entwurf 172
- Evolution 78, **80–82**
- extrahieren 183
- publizierte 124, 172
- Saum zwischen Teams 172
- schmale 223

`ScientificDouble` 293
Self-Shunt 191–193, 204
`setUp()` 33–34, 43, 163, 258
Singularität
- technisch 316

Slack 310
Spezifikation
- am Beispiel 235
- ausführbar 234

Staging-Umgebung 124
`start` 243–246, 255
Stub 172
Stub-Objekt 183–189, 204
- anonyme und innere 189
- Einsatz 184
- mit EasyMock 210
- von konkreter Klasse 189

Subversion 116
`suite()` 36, 48, 151
`Summary` 271
`surplus` 265
Systemtest 233, 300
Szenarien
- Bad Day 129, 308
- Good Day 129, 307

Szenarientest 166

T

Teamkoordination 111
`tearDown()` 33–34, 43, 258
Technologieadapter 303
Tell, don't ask 215–216, 225
`Test` 37
`@Test` 44–45
- `expected`-Parameter 47
- `timeout`-Parameter 48

Test Smells 161
Testbarkeit 3, 308, 313
`TestCase` 23, **31–35**, 37
Test-Code-Refactoring-Zyklen 20, **52**
Testdaten
- Generierung 134–136

Testen
- an den Systemgrenzen 212
- auf Objektgleichheit, *siehe* `assertEquals()`
- auf Objektidentität, *siehe* `assertSame()`
- auf Wertgleichheit, *siehe* `assertEquals()`
- automatisiertes 6
- des Tests 58, 153–159
- durch Indirektion 185–188
- effektives 2
- funktionales 163, 256
- grafischer Benutzeroberflächen 302–303
- isoliertes 175–213
- von Ausnahmen und Fehlern 207
- von Benutzerszenarien 243
- von Exceptions 42
- von Fehlerfällen 142
- von innen 193
- von Legacy Code 313
- von logischen Bedingungen, *siehe* `assertTrue()`
- von Methoden ohne Rückgabewert 191
- von Mittelsmännern 190
- von Objektmengen 264
- von Objektreferenzen, *siehe* `assertNull()`
- von privaten Methoden 173
- von Protokollen 198
- von Verarbeitungsregeln 256
- von Zeitabhängigkeiten 173
- von Zufällen 173

Testergebnis
- Error 43
- Failure 43

Testfall
- Aufbau 23, 127–128
- Ausführung, *siehe* `TestRunner`
- Benennung 129
- charakterisierend 313
- Dokumentation 30
- Ermittlung 134, 166–167
- Gruppierung, *siehe* `TestCase`
- implementierungsunabhängig 136

Testfall (Fortsetzung)
  indirekt 71, 85
  Isolierung 34–35, 162
  Lebenszyklus 34
  offen 71
  Organisation, *siehe* `TestSuite`
  orthogonal 144–149
  parameterisierbar 150–152
  Qualität 153
  refaktorisieren 160–161
  reihenfolgeunabhängig 162
  sabotieren 157
  selbsterklärend 164–165
  überspezifiziert 211
  verschieben 85
Test-Fixture 31–33, 127–128, 237–238
  minimale 140
Test-Framework 11
Testgetriebene Entwicklung 2, 20, 51, 309
  Entwicklungszyklus 52–60
  im Kleinen und im Großen 247
Testgetriebene Programmierung 2, 11, **51–72**
  Ausnahmefälle 61–71
  Der erste Schritt 55, 130
  Der nächste Schritt 60, 130
  Programmierzüge 53
Testhelferklasse 134–136, 161, 168
Test-infected 9
Testkombinatorik 177–178
Testliste 55
Testpunkte 313
Testreport 120–121
`TestRunner` 24, 38
`TestSetup` 137
Testspezifikation 233
Testsprache 161, 168, 303, 306
Teststrategie 299
`TestSuite` **36–37**
Testszenarien 234
  Dokumentation 246
Testumgebung, *siehe* Test-Fixture
  Freigabe, *siehe* `tearDown()`
  Konfiguration, *siehe* `setUp()`
text() 284
Time to Market 315
Top-down 71, 95–96
Toyota Produktionssystem 171, 315
`trailer` 280
Turnaround Management 310
Typ
  aggregiert 292
`TypeAdapter` 278, 289–292, 294, 295
Typen 232
  domänenspezifisch 289

U

Unit Test 2, 21
  strikter 121, 175–213, 215
Ursachenforschung 310

V

Versionierung 114–115
Versionsverwaltung 20, 110, 116
Vertrag
  equals() 40
  hashCode() 41

W

Werte
  erwartete 131
Wertobjekt 22, 39–40
Wertschöpfung 316
W-Fragen 171
White-Box-Test 21, 136
Wiki 304, 306
  Server 304
`wrong()` 284

X

XMLUnit 303

Z

Zykluszeit 315