Von der Methode bis zur technischen Infrastruktur

Deutsche Originalausgabe

Agile Java-Entwicklung

in der Praxis

O'REILLY®

Michael Hüttermann

Agile Java-Entwicklung in der Praxis

Michael Hüttermann

O'REILLY®
Beijing · Cambridge · Farnham · Köln · Paris · Sebastopol · Taipei · Tokyo

Die Informationen in diesem Buch wurden mit größter Sorgfalt erarbeitet. Dennoch können Fehler nicht vollständig ausgeschlossen werden. Verlag, Autoren und Übersetzer übernehmen keine juristische Verantwortung oder irgendeine Haftung für eventuell verbliebene Fehler und deren Folgen.
Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Der Verlag richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Das Werk einschließlich aller seiner Teile ist urheberrechtlich geschützt. Alle Rechte vorbehalten einschließlich der Vervielfältigung, Übersetzung, Mikroverfilmung sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Kommentare und Fragen können Sie gerne an uns richten:
O'Reilly Verlag
Balthasarstr. 81
50670 Köln
Tel.: 0221/9731600
Fax: 0221/9731608
E-Mail: kommentar@oreilly.de

Copyright der deutschen Ausgabe:
© 2008 by O'Reilly Verlag GmbH & Co. KG
1. Auflage 2008

Die Darstellung einer Mondspinnermotte im Zusammenhang mit dem
Thema Agile Softwareentwicklung ist ein Warenzeichen von O'Reilly Media, Inc.

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.

Lektorat: Christine Haite, Köln
Fachliche Unterstützung: Daniel Lehmann, Bonn & Ralph Pöllath, Köln
Korrektorat: Eike Nitz, Köln
Satz: reemers publishing services gmbh, Krefeld; www.reemers.de
Umschlaggestaltung: Edie Freedman, Boston & Michael Oreal, Köln
Produktion: Karin Driesen, Köln
Belichtung, Druck und buchbinderische Verarbeitung:
Druckerei Kösel, Krugzell; www.koeselbuch.de

ISBN-13 978-3-89721-482-8

Dieses Buch ist auf 100% chlorfrei gebleichtem Papier gedruckt.

Inhalt

Prolog XI

Einleitung XV

Teil I: Die Methodik agiler Softwareentwicklung

1 Einführung in die agile Entwicklung 3
Geschichte der agilen Entwicklung 4
Das Agile Manifest 4
Die Prinzipien 6

2 Die Vorteile agiler Entwicklung 13
Die Kundensicht 13
Programmierersicht 16
Die Projektsicht 19
Soziale und betriebswirtschaftliche Sicht 21

3 Agile Ansätze und traditionelles Vorgehen – eine Abgrenzung 23
Prozessmodelle 23
Traditionelle Modelle 24
Was agile Entwicklung nicht ist 26
Ausprägungen agiler Entwicklung 26

4 Dimensionen agiler Entwicklung 45
Motivation 45
Kommunikation 57
Team 64

Wissen 69
Zeit 76
Die größten Verschwender 79

5 Die Durchführung 81
Aufbauorganisation 81
Das Release 83
Die Iteration 91
Pair Programming 93
Stand-up-Meetings 94
Konfigurationsmanagement 95
Test-Driven Development 97
Continuous Integration (CI) 99
Refactoring 102
Standards 102
Tracking 105
Abschluss 109

6 Agile Prozesse einführen 112
Wer überzeugt werden muss 112
Einführungsstrategien 114
Fallstricke 117
Besondere Situationen 120

Teil II: Eine agile technische Infrastruktur 123

7 Ant-Grundlagen 125
Installationshinweise 126
Terminologie und Konzept 126

8 Versionskontrolle mit Subversion 128
Versionskontrolle 128
Subversion 129
Installation und Konfiguration 131
Nutzung 136
Revisionen 142
Versionierung (Tags) 143
Separate Entwicklungszweige (Branches) 145

Locking 148
Automatisierte Nutzung mit Ant 150
Zusätzliche Werkzeuge 154
Zusammenfassung 156

9 Komponententests mit JUnit 157
Komponententests 158
JUnit 158
Ein erstes Beispiel 159
JUnit 3.x vs. JUnit 4.x 164
Testanatomie 167
Instanzen und Abhängigkeiten 170
Testaggregation und Nebenläufigkeit 171
Welche Artefakte sollte ich testen? 176
Testautomatisierung und kontinuierliche Integration mit Ant 178
Test-Driven Development mit JUnit 185
Zusammenfassung 194

10 Testabdeckung mit EMMA und JUnit 195
Installation und Beispiel 195
Automatisierte Nutzung via Ant 198
Testabdeckung 202
Zusammenfassung 206

11 Der Einsatz von Mock-Objekten mit EasyMock 207
Von Mocks und Dummys 207
EasyMock installieren 209
Erstes Szenario: »Wegmocken« von Callbacks 210
Zweites Szenario: »Wegmocken« einer ressourcenintensiven DB-Schnittstelle . 218
Zusammenfassung 222

12 Komponententests mit TestNG 223
Steckbrief 223
Ein Beispiel: Agile, nebenläufige, gruppierte Verarbeitung 225
Aufruf der Tests: Das Build-Skript 236
Zusammenfassung 246

13 GUIs testen mit Jemmy **247**
Swing 247
Einführung in Jemmy 252
Ein ausführliches Beispiel: Testen der Filterung bei Tabellen 253
Zusammenfassung 264

14 Webanwendungen testen mit Selenium **265**
Architektur von Webanwendungen 265
Die Arbeitsweise von Selenium 268
Selenium IDE 269
Selenium RC – Remote Control 273
Kontinuierliche Integration mit Ant 278
Zusammenfassung 282

15 Akzeptanztests mit Fit **283**
Akzeptanztests 283
Einführung in Fit 284
Vorgehen 284
Die verschiedenen Fixtures 285
Ein Rundgang: ColumnFixture 286
Zusammenfassung 291

16 Akzeptanztests für Webanwendungen mit WebTest **292**
Überblick 292
Installation 293
Ein erstes Beispiel: Der WebTest-Selbsttest 293
Ein weiteres Beispiel: Auf Googles Spuren 296
Wiederverwendung 299
Reporting 300
Kontinuierliche Integration 305
Zusammenfassung 311

17 Projektkommunikation mit Trac **316**
Wikis 316
Trac 318
Zusammenfassung 324

18 Ant – Teil 2: Praktische Einsatzszenarien **326**
Das fundamentale Ant-Gerüst für einen Entwicklungszyklus 327
Erstes Szenario: »Continuous Integration« auf dem Integrationsrechner 331
Zweites Szenario: »Continuous Integration« auf einem Arbeitsplatzrechner . . . 331
Drittes Szenario: »Single Code Basis«-Redundanzen eliminieren unter CVS . . . 332
Viertes Szenario: Coding Guidelines überprüfen 338
Fünftes Szenario: HotSwap 345
Best Practices und Ant-Tipps 347
Zusammenfassung 349

19 Kontinuierliche Integration mit CruiseControl **351**
Der pragmatische Ansatz: cron 351
Der umfassende Ansatz: CruiseControl 352
Zusammenfassung 360

20 Der Einsatz von Maven für Build-Prozesse **362**
Funktionsumfang 363
Architektur 364
Prinzipien 365
Die ersten Schritte 367
Das POM 367
Phase und Goal 370
Lifecyles 371
Agile Entwicklung mit Maven 372
Zusammenfassung 383

Anhang: Weiterführende Literatur **387**
Zur Methodik 387
Peopleware – der Faktor Mensch 388
Testen 388
Fit 388
Refactoring 388
TestNG 388
Ant, Maven 389
Subversion 389

Index **391**

Prolog

Während meiner kurzen Zeit als Java-Entwickler, die gerade mal zehn Jahre dauerte, musste ich so einige Projektmanagementstile aushalten, vom klassischen Wasserfall bis hin zu agilen Projekten.

Die frustrierendste Erfahrung habe ich in dem Projekt gemacht, das am wenigsten agil war. Einmal die Woche erstellten wir einen Build, was jeden einzelnen Entwickler im Team mindestens einen halben Tag kostete, weil man damit beschäftigt war, Dateien in Microsoft Visual Source Safe hineinzuziehen und eine ganze Reihe von Batch-Dateien laufen zu lassen. Zwei oder drei Systemtechniker versuchten dann, das ganze System zusammenzubauen, wobei sie die typischen »nicht-agilen« Probleme hatten. Es dauerte dann etwa zwei Tage, bis das System komplett stand. Aber das war nur der Anfang. Die Testprozedur bestand darin, dass einige arme Teufel sich mühsam durch eine Reihe von Word-Dokumenten hindurcharbeiteten und unseren Build mit ihnen abglichen. Die Testmannschaft brauchte immer mindestens drei Wochen, um unser System zu prüfen! Und wir machten wirklich jede Woche einen Build! Das bedeutete natürlich, dass die Bugs fröhlich von einem Release zum nächsten mitwanderten. Es dauerte Jahre und kostete einen Haufen Geld, alles richtig hinzukriegen.

Wäre das ein Einzelfall in einem fernen Land, bräuchte es dieses Buch nicht zu geben. Aber noch heute fahren Firmen hinaus auf die Straßen der Softwareentwicklung, ohne vernünftige Landkarten mitzunehmen, die ihnen dabei helfen könnten, nicht dauernd in Sackgassen und Einbahnstraßen landen. Praktische Leitfäden sind leider Mangelware. Dieses Buch, das auf Michael Hüttermanns großer Projekterfahrung aufbaut, soll dazu beitragen, diesen Mangel zu beheben. Stellen Sie sich einfach einen Reiseführer vor, etwa eine »Hilfe zur Orientierung in der agilen Entwicklung«, der Ihnen verrät, um welche Kneipen Sie lieber einen Bogen machen sollten. In etwa so zeigt Ihnen Michael, wo er schon gewesen und welche Wege er gegangen ist, damit Sie wissen, wie Sie die finsteren Ecken meiden können und wo Sie den besten Hummer bekommen.

Der typische Softwareentwickler ist hochintelligent, individualistisch, introvertiert, phantasievoll und eigensinnig. Auch wenn es einige unserer deutschen Leser schockieren mag: Bei der Arbeit an Projekten mit deutschen Firmen habe ich es so empfunden, dass

die Eigenschaft »eigensinnig« in Deutschland noch stärker ausgeprägt ist als in anderen Ländern. Womöglich liegt das an der höheren formellen Bildung in Deutschland – wo sonst findet man so viele gewöhnliche Programmierer mit einem Doktor in Informatik? Oder vielleicht ist es einfach die Mentalität, gleich beim ersten Mal alles richtig machen zu wollen, koste es, was es wolle?

Die erste Eigenschaft, Intelligenz, ist zweifelsohne für die Entwicklung komplexer Lösungen erforderlich. Man braucht einen gewissen IQ, um Probleme zu erfassen, sich Lösungen auszudenken und sie in so einem abstrakten Medium wie Software zu verwirklichen. Wenn aber hohe Intelligenz und ein gewisser Eigensinn zusammenkommen, kann es dazu kommen, dass man in einem Team Monate damit verbringt, verschiedene Gesichtspunkte der Software zu diskutieren, ohne zu einer funktionierenden, marktfähigen Lösung zu kommen. Das wird bei der agilen Vorgehensweise durch Scrum-Meetings und die Konzentration auf lauffähige Software verhindert.

Die zweite Eigenschaft, Individualismus, ist bei Programmierern lustig zu beobachten. Von außen betrachtet sehen Programmierer im Grunde alle gleich aus. Sie finden das neueste technische Gimmick spannender als weltweite Sportereignisse. Der normale Programmierer ist in seinen eigenen Augen aber ganz anders als alle anderen! Das liegt an den Erfahrungen, die er seit seiner Jugend in seinem sozialen Umfeld gemacht hat: Selbst heute ist der typische Computer-Geek in der Schule doch eher weniger beliebt. Auch wenn er es ist, dem bald die halbe Stadt gehört, bevorzugen die hübschen Mädchen im Zweifel den sonnengebräunten, muskulösen Surfertyp gegenüber dem Java-Ass, das ein Ergebnis von 100% beim »Java Black Belt«-Wettbewerb schafft. Vielleicht passt sich die Evolution nicht schnell genug der Wirklichkeit an – bedeutet »der Stärkste« zu sein immer noch rohe Gewalt, oder hat es endlich auch etwas mit Köpfchen zu tun? Der typische Programmierer wächst jedenfalls in dem Bewusstsein auf, anders als die meisten Mitschüler zu sein, und er nimmt dieses Bewusstsein mit ins Berufsleben. Hier hilft der agile Ansatz, indem er Individuen und Interaktionen berücksichtigt. Wir verschließen uns nicht der Realität – nämlich dass Programmierer Individualisten sind, die lieber kündigen würden, als Eclipse als IDE zu benutzen –, sondern wir konzentrieren uns darauf, sie zum Interagieren zu kriegen.

Was für einen schöneren Beruf könnte es geben, als einen, bei dem man acht Stunden am Tag allein und nur von Büchern umgeben in einer Kabine sitzen und auf einen großen 21-Zoll-Monitor blicken darf? Besser geht's nicht! Lachen Sie nicht, viele von uns Geeks sehen das so. Stellen Sie sich vor, Sie versuchen, eine Radtour für die Softwareentwicklungsabteilung zu organisieren. Vermutlich wird ein Quake-Tournament, bei dem jeder mit seinem Aschenbecher und Dutzenden leerer Pappbecher in seinem Kabinchen sitzen bleiben darf, wesentlich mehr Anklang finden. Trotzdem bleibt es dabei: Die Interaktion ist so essenziell wichtig in einem Softwareentwicklungsteam, dass eine Radtour doch die bessere Methode sein dürfte, die Programmierer zu einer echten Zusammenarbeit zu bekommen.

Einer der sympathischsten Aspekte der agilen Softwareentwicklung ist die Tatsache, dass sie die besonderen Eigenschaften von Programmierern anerkennt und *mit* ihnen arbeitet, nicht gegen sie. Anstatt die Entwickler zu zwingen, Tausende von Seiten technischer Dokumentation zu schreiben, die sowieso nie jemand lesen wird, ist die Hauptsache laufende Software.

»Agil« bedeutet auch, davon auszugehen, dass immer Veränderungen stattfinden. Je länger sich die Fertigstellung eines Projekts ohne Beteiligung des Kunden hinzieht, desto wahrscheinlicher wird es, dass das Ergebnis komplett danebengeht. Wir müssen unsere Vorgehensweise ständig an das anpassen, was der Kunde will. Oft weiß ja er ja erst, was er wirklich will, wenn er einen Teil der vorgeschlagenen Lösung gesehen hat.

Ein Kunde, der Teil des Gesamtprozesses ist, hilft dabei, die Kosten zu senken, indem er Features, die nicht unbedingt notwendig sind, einfach auf das nächste Release vertagt. Für den Kunden kommt ein Stück Software dabei heraus, das genau das tut, was er will – nicht mehr und nicht weniger. Er muss nicht für Features bezahlen, die er sowieso nie benutzen wird. Was nicht unbedingt weniger Geld für die Entwicklungsfirma bedeutet: Meist kann sie höhere Stundensätze berechnen, weil der Kunde kein Geld für unnötige Features verschwendet.

Michael ist ein hervorragender Autor. Er reißt Sie mit seinem Erzählfluss mit und sorgt so dafür, dass Sie das Buch nicht weglegen können, bevor Sie ein paar Abschnitte gelesen haben. Und seine Sprache ist nicht so kompliziert, dass sie von den Inhalten ablenkt.

Mir persönlich hat der Abschnitt besonders gut gefallen, in dem Michael beschreibt, was »agil« nicht ist. Er beschreibt die Situation, wo der Kunde die Telefonnummer des Programmierers hat und ihn dauernd anruft, um die Anforderungen zu ändern. Michael schreibt: »Das ist nicht agil, das ist Chaos!«, und damit hat er so recht ...

Die 15-minütigen Scrum-Meetings sind eine der Sachen, die ich an agiler Entwicklung am liebsten mag. Michael erläutert richtigerweise, dass »scrum« aus dem Rugby kommt, das in meinem Heimatland Südafrika sehr beliebt ist. Bevor ich dieses Buch gelesen hatte, wusste ich nicht, dass Scrum-Meetings wohl vom Rugby kommen. Der Scrum wird durchgeführt, um das Spiel nach einem kleineren Foul oder einem Ausball wiederaufzunehmen. Das Wort kommt von *skirmish*, zu Deutsch »Gefecht«. Das Rugby-Gefecht ist in Wirklichkeit eine ziemlich gefährliche und manchmal gewalttätige Angelegenheit. Ich habe mit eigenen Augen gesehen, wie sich im Jugendsport Spieler dabei die Hälse gebrochen haben. Auch wenn »Scrum« ein Ausdruck der mannigfachen Fähigkeiten sein soll, die die verschiedenen Leute in einem agilen Team mit an den Tisch bringen: Mit wem fechten *wir*? Hoffentlich nicht mit den Kunden!

Das Kapitel über Kommunikation ist sehr wichtig. Besonders gut fand ich den Teil, in dem es darum geht, wie die Entwickler in Meetings oft den Manager statt des Teams ansprechen. Wenn Meetings auf diese Art ablaufen, führt das eher dazu, dass die Entwickler unter Druck gesetzt werden, vor ihren Kollegen Ergebnisse zu versprechen, anstatt dazu, dass die Entwickler von ihren konkreten Problemen berichten und man so zu Lösungen kommen kann. Mir läuft es kalt den Rücken herunter, wenn ich höre, wie

ein Manager einen Programmierer fragt, wie lange etwas dauern wird, weil die Antwort allermeistens nur ein Schuss ins Blaue ist. In einem Meeting muss man miteinander reden, nicht die ganze Zeit den Chef ansehen.

Im weiteren Verlauf des Buchs werden verschiedene Werkzeuge genauer unter die Lupe genommen, die bei der Unterstützung von agilen Prozessen hilfreich sein können. Auch hier muss betont werden, dass der Schreiber über große Erfahrung in der Anwendung dieser Tools verfügt und seine Kenntnisse in einer verständlichen Weise herüberbringt. Michael behandelt eine Menge Werkzeuge und gibt Ihnen dabei genau so viele Informationen, wie Sie brauchen, um einen leichten Einstieg in ihre Benutzung zu haben. Wenn Sie genauere Informationen benötigen, können Sie sie in der Dokumentation des jeweiligen Tools nachlesen.

Das hier ist ein hervorragendes Buch, dem ich viel Erfolg auf dem deutschsprachigen Markt wünsche!

Dr. Heinz Kabutz
The Java Specialists' Newsletter
http://www.javaspecialists.eu

Einleitung

Was heißt eigentlich »agil«? Sind wir nicht alle ein wenig agil? Wie würden Sie auf die Frage »Arbeiten Sie agil?« antworten? Ich bin mir recht sicher, dass Ihre Antwort lauten würde »Ja, aber sicher, ich arbeite agil.« Agil (von lateinisch *agilis*) zu arbeiten bedeutet tätig, geschäftig sein. Jeder, der diese Frage gestellt bekommt, wird alles daran setzen, *nicht* den Eindruck zu erwecken, *nicht* agil zu sein – denn gerade in der Arbeitswelt ist schon der Verdacht von Lethargie, melancholischer Versunkenheit oder irgendeiner anderen Charaktereigenschaft, die der vom Leben zeugenden Agilität entgegensteht, gefährlich und sollte vermieden werden.

Aber was macht agile Softwareentwicklung aus?

Agile Softwareentwicklung basiert im Wesentlichen auf den vier 2001 von einigen auf diesem Gebiet führenden Persönlichkeiten im »agilen Manifest« festgehaltenen Werten:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

(»Einzelpersonen und Interaktionen haben Vorrang vor Prozessen und Werkzeugen, funktionierende Software hat Vorrang vor ausführlicher Dokumentation, die Zusammenarbeit mit dem Kunden hat Vorrang vor Vertragsverhandlungen, auf Veränderungen einzugehen ist wichtiger als einen festen Plan zu verfolgen«.)

Im Zentrum stehen dabei Kommunikation und Zusammenarbeit mit dem Kunden und die Zielsetzung, Software zu erstellen, die ihm wirklichen Nutzen bringt. Außerdem ist eine offene, flexible Herangehensweise beim Umgang mit neuen und geänderten Anforderungen elementar. Agile Entwicklung bedeutet dabei, nicht chaotisch, sondern diszipliniert vorzugehen, so dass die Arbeit anhand von vielen kontinuierlichen Interventionspunkten gut verfolgt und kontrolliert werden kann.

Ist das neu?

Die Bedeutung des menschlichen Faktors – also gegenseitigen Vertrauens und kollegialer Zusammenarbeit und Kommunikation – wurde schon früher thematisiert. Tom DeMarco stellte u. a. in seinem 1997 erschienenen Werk »Deadline« die These auf, ein Mitarbeiter zahle sich bereits aus und sei der Produktivität des ganzen Projekts zuträglich, wenn er ausschließlich als »Katalysator« eingesetzt wird, der das Team zusammenhält, bei der Lösung zwischenmenschlicher Probleme hilft und für Motivation sorgt. 1998 widmete der Psychologe Gerald M. Weinberg der Psychologie des Programmierers mit »The Psychology of Computer Programming« ein ganzes Buch. Noch früher, bereits 1974, arbeitete Frederick P. Brooks die wahren Kräfte, Einflussfaktoren und Interdependenzen in der Softwareentwicklung heraus und setzte mit seinem Buch »The Mythical Man-Month« einen Meilenstein.

Parallel dazu gab es mit *Lean Development* und später mit *Extreme Programming* (»XP«) und *Scrum* agile Ansätze, Projekte erfolgreich zu Ende zu bringen und sich auf die wirklich wichtigen Dinge – wie hohe Qualität und Kundenzufriedenheit – zu konzentrieren. Die Agile Entwicklung mit ihren Ausprägungen hat all diese Strömungen in sich vereint und in dem agilen Manifest auf einen Nenner gebracht.

Agile Entwicklung ist kein Allheilmittel. Einerseits wurde auch mit traditionellen Prozessen bereits Software hergestellt. Andererseits gibt es Teams und vermeintliche Experten, die sich große agile Entwicklungsfähigkeiten auf die Fahnen schreiben, aber den Kern der agilen Entwicklung nicht verstanden haben. Doch die bloße Sensibilisierung für die kritischen Punkte bei der Softwareerstellung ist bereits ein Schritt nach vorn, und viele Projekte haben mit agilen Projekten große Erfolge erzielt. Agile Entwicklung ist erwachsen geworden. Nachdem viele »Early Adopters« XP und Scrum, die beiden bekanntesten Ausprägungen, erfolgreich eingesetzt haben, nutzen nun immer mehr – auch konservativere – Unternehmungen wie Sabre und SAP diese Ansätze und profitieren davon. Software agil zu entwickeln steigert die Wahrscheinlichkeit, sie erfolgreich bereitstellen zu können!

Aber Agilität ist nicht mehr nur auf die IT-Abteilung beschränkt, sondern es ist ein neuer Managementansatz, dem sich immer mehr Marktteilnehmer weltweit verschreiben. Mit »Kaizan« liegt z. B. beim japanischen Konzern Toyota ein seit 50 Jahren perfektioniertes auf stetigen Wandel konzentriertes Managementkonzept vor.

Zielgruppe dieses Buchs

Grundsätzlich muss in einem agilen Projekt jeder Beteiligte die agile Entwicklung als Prozessmodell verstehen. Dies ist ein Buch, das einerseits die agile Softwareentwicklung hemdsärmelig erarbeitet sowie neueste Erkenntnisse einbringt und andererseits neue Akzente setzt – das also als Leitfaden für den praktischen Einsatz durch Architekten, Entwickler/Programmierer, Tester, Projektmanager, Manager und Kunden von IT-Projekten dienen kann. Es liefert in der Praxis bewährte Rezepte und Antworten auf Fragen zu Themen, bei denen herkömmliche Bücher längst aufhören, wie beim konkreten Nutzen der

agilen Entwicklung, der Abschätzung von Anforderungen oder bei der Illustration einer technischen Infrastruktur. Das gilt sowohl für Leser, die ihre bestehenden agilen Prozesse schärfen wollen, als auch für interessierte Projektbeteiligte, die den möglichen Einsatz agiler Prozesse prüfen.

Während der erste allgemeine Teil also für alle Beteiligten relevant ist, wird der technische Teil vor allem diejenigen interessieren, die mit der konkreten Java-Entwicklung beschäftigt sind. Mir war es wichtig, eine funktionierende technische Infrastruktur vorzustellen, damit Sie verstehen, welche Werkzeuge was leisten und wie sie richtig einzusetzen sind, um die agile Methodik erfolgreich anwenden zu können. Dabei könnte jedem hier vorgestellten Werkzeug ein separates Buch gewidmet werden. Der Blick auf weiterführende Literatur sei jedem ans Herz gelegt, sobald das Grundverständnis eines Werkzeugs gelegt ist. Allein durch das Studium dieses Buchs werden Sie aber schon die meisten Anwendungsszenarien erfolgreich angehen können.

Es gibt viele auf Einzelthemen fokussierte Bücher. Es liegt Ihnen hier ein kartographisches Werk vor, das dem Wanderer und Wandler den Weg weist.

Voraussetzungen

Das vorliegende Buch ist für Einsteiger in die Softwareentwicklung gleichermaßen geeignet wie für den erfahrenen IT-Profi. Dem Einsteiger wird die agile Thematik detailliert vorgestellt und er wird mit verhältnismäßig einfachen Beispielen an die Nutzung geeigneter Werkzeuge herangeführt. Der Erfahrene wird aus den Ausführungen zur agilen Methodik hoffentlich viele neue wertvolle Impulse und Anleitungen mitnehmen. Im technischen Teil dann kann auch er sich voll und ganz auf die Werkzeuge konzentrieren, ohne sich durch überlange Codebeispiele oder Theorie zu quälen. Die Werkzeuge stehen hier im Mittelpunkt, nicht die beispielhaften Anwendungen. Ich denke, die Ausführungen zu den Beispielen lassen sich dann auf komplexere und individuelle Szenarien übertragen. Vom Leser des Methodikteils werden keine Programmierfähigkeiten erwartet. Der Leser des Werkzeugteils sollte über Java- und XML-Grundkenntnisse verfügen.

Viele Projekte nutzen mittlerweile Linux, und das nicht nur auf zentralen Servern. Die Installationshinweise zu den Werkzeugen basieren auf Linux, lassen sich aber sehr einfach anhand der jeweils angeführten Hinweise oder Quellen auf Windows übertragen. Die Nutzung der vorgestellten Werkzeuge ist dann grundsätzlich unabhängig vom Betriebssystem. Gleiches gilt für die IDE-Integration. An einzelnen Stellen gehe ich auf IntelliJ IDEA aus dem Hause JetBrains ein, das, insbesondere in den USA breite Verbreitung genießt. Ob Sie das Werkzeug nun auch mit IDEA nutzen wollen oder mit Eclipse, spielt für das Buch eine untergeordnete Rolle. Das Buch hat den Fokus auf der IDE-unabhängigen Nutzung der Werkzeuge.

Nachdem lange Zeit Cobol oder auch Smalltalk die erste Wahl bei der Realisierung großer Softwareprojekte war, ist seit einigen Jahren Java die Sprache, die weltweit eine Führungsrolle eingenommen hat. Ist sie auch vielleicht nicht die sauberste, eleganteste

Sprache, so hat sie doch – nicht zuletzt durch eine riesige Marketingkampagne von Sun Microsystems – die Welt erobert: Sie wird nicht nur bei der Erstellung »irdischer« Applikationen eingesetzt, wie auf Handys oder im Rahmen betriebswirtschaftlicher Software unterschiedlichster Größe, auch der Mars Rover Spirit wurde 2004 auf seiner Marsmission von Java gesteuert. Obwohl im Unternehmensbereich mit Microsofts .NET eine Alternative zu Java und der Java-Komponentenarchitektur JEE existiert und neue Alternativen wie »Ruby on Rails« auf sich aufmerksam machen, setzt dieses Buch bei der technischen Unterstützung aufgrund seiner weltweiten Verbreitung, Bedeutung und Popularität konsequent auf Java.

Aufbau dieses Buchs

Der erste Teil des Buchs befasst sich mit der agilen Methodik, ihren Bestandteilen, einzelnen konkreten Ausprägungen wie dem Extreme Programming oder Scrum und dem Nutzen agiler Prozesse. Wir gehen ferner einen agilen Entwicklungsprozess exemplarisch von Anfang bis Ende durch und ich gebe Ihnen einige nützliche Hinweise zur Einführung agiler Arbeitsweisen.

Der erste Wert des agilen Manifests, »Individuals and interactions over processes and tools«, also dass Interaktionen und Individuen wichtiger sind als Prozesse und Werkzeuge, unterstreicht die größere Bedeutung des Menschen. Das bedeutet aber nicht, dass Prozesse und Werkzeuge gänzlich unwichtig wären. Ganz im Gegenteil: Ich werde Ihnen im zweiten Teil des Buchs eine Reihe von Werkzeugen empfehlen, mit denen Sie Software automatisch erstellen, testen und dokumentieren können, die die Verteilung des kollektiven Wissens und die Zusammenarbeit fördern und testgetriebene Entwicklung ermöglichen. Diese Werkzeuge bleiben allerdings immer Mittel zum Zweck, elementare Hilfsmittel, um die Software überhaupt erst effizient erstellen, testen und verteilen zu können.

Immer mehr in Java entwickelte Werkzeuge wurden in den letzten Jahren bereitgestellt, von denen viele kostenlos nutzbar sind. Dies spiegelt einen Trend wider, der dazu geführt hat, dass Sun Microsystems selbst nicht nur weite Teile der Programmiersprache Java in enger Zusammenarbeit mit den Nutzern gemeinsam entwickelt, sondern 2007 den Quellcode als Open Source komplett in das Allgemeineigentum übertragen hat.

Dieser Tatsache trägt das Buch Rechnung: Ich stelle Ihnen Open Source-Werkzeuge vor, die entweder bereits De-facto-Standards sind oder aufgrund ihrer Verbreitung und ihrer überragenden Eigenschaften auf dem besten Wege sind, es zu werden. Diese Werkzeuge kombinieren und integrieren wir, um durch Synergien den maximalen Nutzen zu erreichen und über eine vollständige konsistente Infrastruktur verfügen zu können.

Teil I – Die Methodik agiler Softwareentwicklung

Kapitel 1, *Einführung in die agile Entwicklung*

Dieses Kapitel erläutert das Agile Manifest, die Grundlage einer jeden agilen Arbeitsweise.

Kapitel 2, *Die Vorteile agiler Entwicklung*

In diesem Kapitel wird der Nutzen diskutiert, den agile Projekte liefern. Hier beschreibe ich den Nutzen aus Kundensicht, aus Programmierersicht, aus Projektsicht und aus einer sozial- und betriebswirtschaftlichen Sicht.

Kapitel 3, *Agile Ansätze und traditionelles Vorgehen – eine Abgrenzung*

Hier werden traditionelle Vorgehen wie das Wasserfallmodell agilen Prozessen gegenübergestellt. Ich gehe auf agile Ausprägungen wie Extreme Programming, Scrum und Lean Development ein.

Kapitel 4, *Dimensionen agiler Entwicklung*

Als »Dimensionen agiler Entwicklung« bezeichne ich die für agile Projekte essentiellen Stellschrauben. Hierunter fallen Motivation (und wie ich sie fördere oder zerstöre), Kommunikation (wie kommunziere ich richtig, wie falsch), das Team in seiner Zusammensetzung und mit seinen Bedürfnissen, das gesamte Wissen (wie generiere ich Wissen und stelle es zur Verfügung) und die Zeit als kritische Größe.

Kapitel 5, *Die Durchführung*

In diesem Kapitel gehe ich ein typisches agiles Projekt chronologisch durch. Angefangen bei der aufzusetzenden Aufbauorganisation diskutiere ich Releases und deren Iterationen. Wie plane ich diese? Wie komme ich zu den Features, die umzusetzen sind? Wie schätze ich den Aufwand, der mit ihrer Umsetzung verbunden ist? Wie hoch ist der Durchsatz des Entwicklerteams? Wie wird die Funktionalität umgesetzt? In diesem Zusammenhang gehe ich auch auf Stand-Up-Meetings, Konfigurationsmanagement, testgetriebene Entwicklung, kontinuierliche Integration, Standards und Refactorings ein. Dieses Kapitel schlägt damit eine Brücke zum Werkzeugteil.

Kapitel 6, *Agile Prozesse einführen*

Hier bekommen Sie Hinweise, wie Sie agile Prozesse einführen können. Vermutlich werden Sie anfangs etwas Überzeugungsarbeit leisten müssen, für die ich Ihnen einige Tipps geben möchte. Konkrete Einführungsstrategien, Fallstricke und besondere Situationen für agile Projekte runden das Kapitel ab.

Teil II – Die Werkzeuge

Kapitel 7, *Ant-Grundlagen*

In diesem Grundlagenkapitel beschreibe ich die wesentlichen Merkmale von Ant und lege die Grundlagen für seine Nutzung in Kombination mit den folgenden Werkzeugen.

Kapitel 8, *Versionskontrolle mit Subversion*

Subversion ist ein Versionskontrollsystem, das das Rückgrat Ihres agilen Projektes darstellen sollte. In diesem Kapitel führe ich in Subversion ein.

Kapitel 9, *Komponententests mit JUnit*

In diesem Kapitel illustriere ich, wie mit JUnit Komponententests entwickelt werden können. Die Tests werden schließlich automatisiert mit Hilfe von Ant durchgeführt. Anschließend stelle ich anhand eines einfachen Beispiels das übliche Vorgehen bei der testgetriebenen Entwicklung mit JUnit dar.

Kapitel 10, *Testabdeckung mit EMMA und JUnit*

EMMA dient dazu, die Testabdeckung des Codes zu prüfen. In diesem Kapitel erläutere ich die Nutzung von EMMA im Zusammenspiel mit JUnit und die automatisierte Nutzung via Ant.

Kapitel 11, *Der Einsatz von Mock-Objekten mit EasyMock*

EasyMock ist ein Rahmenwerk, das Objekte simuliert und Tests isoliert und vereinfacht. Anhand typischer Szenarien führe ich in seine Nutzung ein.

Kapitel 12, *Komponententests mit TestNG*

TestNG ist ein Werkzeug zur Durchführung von Komponententests. Ich grenze es von JUnit ab und zeige Einsatzszenarien, bei denen sich die Nutzung von TestNG bewährt hat.

Kapitel 13, *GUIs testen mit Jemmy*

Mit Jemmy ist es möglich, Java-GUIs zu testen. Ich erläute das Werkzeug anhand eines konkreten Beispiels.

Kapitel 14, *Webanwendungen testen mit Selenium*

In diesem Kapitel zeige ich Ihnen, wie Sie mit Selenium Webanwendungen testen können.

Kapitel 15, *Akzeptanztests mit Fit*

Fit hat sich als Rahmenwerk zur Bereitstellung und Durchführung von Akzeptanztests bewährt. Ich skizziere seine Einsatzregeln.

Kapitel 16, *Akzeptanztests für Webanwendungen mit WebTest*

WebTest ist ein mächtiges Werkzeug, um für Webanwendungen Akzeptanztests zu erstellen und durchzuführen.

Kapitel 17, *Projektkommunikation mit Trac*

Jedes agile Projekt sollte über ein Wiki verfügen. Ich stelle Ihnen Trac vor und erläutere seine Vorteile.

Kapitel 18, *Ant – Teil 2: Praktische Einsatzszenarien*

Dieses Kapitel zeigt, wie Ant Ihren Entwicklungsprozess umfassend unterstützt. Darüber hinaus demonstriere ich in unterschiedlichen Szenarien, wie Sie mit Ant Ihre Software-Artefakte in einem CVS-System verwalten und mit Checkstyle Ihre Standards überprüfen können.

Kapitel 19, *Kontinuierliche Integration mit CruiseControl*

CruiseControl hilft Ihnen, Ihre Ant-Skripte automatisiert zu wohldefinierten Zeitpunkten laufen zu lassen und das Ergebnis verfügbar zu machen.

Kapitel 20, *Der Einsatz von Maven für Build-Prozesse*

Maven 2 ist eine Alternative zu Ant. Ich stelle Ihnen Maven 2 vor, erläutere seine Konzepte und Einsatzmöglichkeiten.

Hinweise zur Lektüre

Das Buch ist so geschrieben, dass Sie nicht nur linear von vorne nach hinten lesen, sondern auch einzelne Kapitel direkt angehen können. Die ersten Kapitel schärfen das Verständnis, was agile Entwicklung wirklich bedeutet. Auf das Lesen dieser Kapitel sollten Sie auf keinen Fall verzichten.

Dem Werkzeugteil ist ein Ant-Kapitel vorangestellt, das rudimentär in Ant einführt. Ant wird in den meisten anderen Werkzeugkapiteln genutzt und vertiefend erklärt. Kennen Sie Ant bereits, können Sie dieses Kapitel übergehen.

Obwohl die Codebeispiele teilweise aufeinander aufbauen bzw. wiederverwendet werden, können Sie die Werkzeugkapitel unabhängig voneinander lesen. Auf eine komplexe Geschäftsanwendung, die sich durch viele bzw. alle Kapitel durchzieht, habe ich bewusst verzichtet.

Hinweise zur Verwendung des Codes

Die Beispiele in diesem Buch sind bewusst einfach konzipiert, so dass Sie sich voll und ganz auf das Werkzeug konzentrieren können und nicht eine komplexe Beispielanwendung nachvollziehen müssen. Unter *http://huettermann.net* finden Sie weitere Hinweise zum Buch und Code- und Nutzungsbeispiele zu den Werkzeugen. Die herunterladbaren Beispiele sind frei nutzbar.

Typografische Konventionen

Dieses Buch verwendet die folgenden typografischen Konventionen:

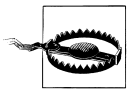
Kursiv
: Wird für wichtige Begriffe, Programm- und Dateinamen, URLs, Ordner und Verzeichnispfade und zur Hervorhebung verwendet.

`Nichtproportionalschrift`
: Wird für Programmcode verwendet.

`Fette Nichtproportionalschrift`
: Verwende ich, um wichtige Stellen im Code herauszustellen, auf die ich mich im erläuternden Fließtext beziehe.

KAPITÄLCHEN
: Kennzeichnen Menüeinträge, Buttons und andere GUI-Schaltflächen.

Dies ist ein praktischer Hinweis darauf, wie Sie Dinge einfacher und schneller durchführen können.



Mit diesem Symbol wird auf Besonderheiten hingewiesen, die zu Problemen führen oder ein Risiko darstellen können.

Danksagung

Ich danke allen, die mir in meinen langen IT-Jahren durch ihr Verhalten in vielen gemeinsamen Projekten so viel Material für dieses Buch geliefert haben. Dies gilt im Positiven wie auch im Negativen: Ich traf exzellente Manager und Projektleiter, phantastische Entwickler und Kunden, aber auch Beteiligte, die sich weniger oder gar nicht agil verhalten haben, auch wenn sie dies von sich behaupteten. Sie alle haben mein Bewusstsein immer mehr geschärft, was in IT-Projekten wirklich wichtig ist.

Ich danke allen, mit denen ich während der Zeit der Bucherstellung Diskussionen führen durfte und von denen ich Tipps bekam.

Ich danke Wikipedia und seinen Editoren. Dort zu lesen und zu schreiben prägt.

Ich danke meinem Notebook, dass es so tapfer in verschiedensten Städten Deutschlands meine Gedanken aufgenommen hat.

Ein herzlicher Dank geht an den O'Reilly-Verlag, dass er das Buch überhaupt ermöglicht hat. Insbesondere danke ich meiner Lektorin Christine Haite für ihre tatkräftige, immer ermunternde Unterstützung.

Schließlich danke ich meiner Frau und meinen Söhnen für ihre Unterstützung. Nun reicht es nicht mehr, dass er Tag und Nacht vor dem Rechner sitzt, jetzt schreibt er auch noch ein Buch, das ihn deutlich mehr als zwölf Monate kostet!

TEIL I

Die Methodik agiler Softwareentwicklung

Im ersten Teil dieses Buchs wird es um das Fundament der agilen Entwicklung gehen: das Agile Manifest und die grundlegenden Werte und Praktiken, die sich daraus ableiten lassen. Ich zeige Ihnen, welche Vorteile Ihnen agile Prozesse bringen, wie Sie diese einführen und wie sich ein typisches agiles Projekt darstellt. Kurz: Dieser Buchteil wird Ihnen das nötige Rüstzeug vermitteln, um dann im zweiten Teil mit den konkreten Werkzeugen richtig loszulegen.

KAPITEL 1

Einführung in die agile Entwicklung

Der Begriff *Agile Entwicklung* beschreibt einerseits ein Wertesystem, andererseits ist es ein Oberbegriff für konkrete Methodiken, die teilweise schon recht lange genutzt werden. »Agile Entwicklung« ist eine Philosophie, ein Metamodell, auf das verschiedene konkrete Ausprägungen aufbauen, unter anderem die im weiteren Verlauf vorgestellten Methoden *Extreme Programming*, *Scrum* und *Lean Development*.

Diese Einteilung ist nicht frei von Zündstoff. Einige Vertreter von Extreme Programming sehen Ihr Vorgehen nicht als »Methodik« an, da dieser Begriff doch ein eher starres Vorgehen suggeriert. Da diese einzelnen Methodiken jedoch Aussagen treffen, wie Software hergestellt werden soll, scheint mir der Begriff legitim – auch wenn die »Methodik« bei diesen agilen Methoden im Vergleich zu schwergewichtigen Prozessmodellen sehr leichtgewichtig ist.

Weiterer Zündstoff liegt in der Einstufung von Lean Development als agile Methodik. Es ist eine Philosophie, die die Bewegung der agilen Entwicklung maßgeblich geprägt hat. Ferner existieren viele Wechselwirkungen zwischen Lean Development und agiler Entwicklung. Eine abstrakte Herangehensweise wäre es, Lean Development als Metamodell zu bezeichnen und agile Entwicklung als Meta-Metamodell, worauf ich im weiteren Verlauf der Einfachheit wegen aber verzichte.

Die einzelnen Ausprägungen sind Baukästen. Sie bieten unter dem Oberbegriff agiler Entwicklung und unter Einhaltung der maßgeblichen Werte zahlreiche Lösungen an, die in der Praxis häufig methodikübergreifend miteinander kombiniert werden. Die einzelnen Methodiken füllen die Philosophie mit Leben und bieten für die Praxis anwendbare Leitfäden.

Ich werde Ihnen in diesem Kapitel die Grundlagen der agilen Entwicklung nahe bringen. Das Wertesystem der agilen Entwicklung basiert auf dem agilen Manifest, das durch Prinzipien konkretisiert wird. Am Ende des Kapitels ist das Fundament für den Rest des Buchs gelegt. Sie verstehen, was ich in allen späteren Kapiteln mit agiler Entwicklung meine, nämlich die Entwicklung von Software im Einklang mit dem agilen Manifest und den begleitenden Prinzipien.

Geschichte der agilen Entwicklung

Im Jahr 2000 organisierte Kent Beck einen Workshop, an dem Vertreter einzelner agiler Methodiken teilnahmen.

In der weltweiten Software-Community war es zu diesem Zeitpunkt immer offenkundiger geworden, dass unterschiedliche vorgeschlagene iterative Ansätze in die gleiche Richtung gingen. Diese Ansätze waren insbesondere im Rahmen der objektorientierten Entwicklung entstanden. Das hatte mit der Programmiersprache Smalltalk begonnen und ab Mitte der 90er Jahre mit Java enorm an Fahrt aufgenommen.

Während des Treffens einigten sich die Anwesenden darauf, ein weiteres Treffen mit den führenden Vertretern der Szene abzuhalten, um ausschließlich dieses Thema zu diskutieren. Als es dann so weit war, wollten 17 Vertreter die Ansätze der jeweils anderen besser kennen lernen. Während des Treffens kam man zu der Einsicht, einen Oberbegriff für die einzelnen Ausprägungen finden zu müssen. »Agile« war geboren und mit dem agilen Manifest auf dem Treffen erstmals mit Leben gefüllt.

Eine neue Philosophie der Softwareerstellung brach sich Bahn. Die 17 Vertreter sahen sich als Urväter der Bewegung, die den Ball zum Laufen gebracht hatten. Sie haben die Ansätze im weiteren Verlauf selbst angewandt, und die weltweite Community hat diese Ansätze aufgenommen.

Als Ergebnis haben einige der Urväter die *Agile Alliance* (siehe *http://www.agilealliance.org*) gegründet, eine nicht auf Gewinn ausgerichtete Organisation, die der Erforschung und Förderung des agilen Ansatzes dienen soll. Interessenten haben damit eine zentrale Anlaufstation. Die Agile Alliance bietet umfangreiche Literatur rund um die agile Entwicklung, richtet einmal jährlich eine Konferenz aus und fördert User Groups und regionale, agile Aktivitäten weltweit.

Das Agile Manifest

Das Agile Manifest hat folgenden Wortlaut:

1. Individuen und Interaktionen haben Vorrang vor Prozessen und Werkzeugen.
2. Lauffähige Software hat Vorrang vor umfassender Dokumentation.
3. Zusammenarbeit mit den Kunden hat Vorrang vor Vertragsverhandlungen.
4. Auf Änderungen einzugehen ist wichtiger als einen Plan zu verfolgen.

Das Manifest besteht aus vier grundlegenden Aussagen, die jeweils zwei Kategorien gegenüberstellen und bewerten. Sie sind so zu verstehen, dass der rechte Ausdruck zwar wichtig ist, der linke aber wichtiger.

Individuen und Interaktionen vor Prozessen und Werkzeugen

Individuen und die Interaktionen zwischen ihnen sind wichtiger als Prozesse und Werkzeuge. Dies bedeutet nicht, dass Prozesse und Werkzeuge unwichtig sind, ganz

im Gegenteil: Sie werden sogar intensiv genutzt. Im Zweifel ist ein persönlicher Austausch jedoch immer einem formalem Vorgehen vorzuziehen. Der Mensch steht im Mittelpunkt: Ein Prozess muss zwar sein, aber nicht um des Prozesses willen. Software wird *von* Menschen *für* Menschen gemacht. Die meisten Projekte scheitern an sozialen Schwierigkeiten oder zwischenmenschlichen Problemen, nicht an technischen Herausforderungen, die nicht zu meistern wären.

Lauffähige Software vor umfassender Dokumentation

Der zweite Punkt befasst sich mit der Dokumentation. Ein Softwareprodukt ganz ohne Dokumentation ist zwar nicht erstrebenswert, aber es ist wichtiger, lauffähige Software zu entwickeln und anbieten zu können. Im Zweifel soll bewusst auf eine umfassende Dokumentation verzichtet werden, schließlich kauft der Kunde keine Dokumentation, sondern ein Produkt, das laufen und fachlich das unterstützen muss, was der Kunde benötigt.

Zusammenarbeit mit den Kunden vor Vertragsverhandlungen

Der dritte Punkt stellt ebenfalls den Menschen in den Mittelpunkt. Der Kunde soll offen und ehrlich in die Entwicklung der Software einbezogen werden. Es existiert eine natürliche Spannung zwischen den beiden Parteien Kunde und Softwarehersteller: Der Kunde möchte für wenig Geld möglichst viel bekommen, der Hersteller möglichst viel Geld verdienen. Dennoch: Ohne Zusammenarbeit geht es nicht. Es muss selbstverständlich ein Vertrag aufgesetzt werden. Sich aber im Laufe des Projekts immer wieder darauf zu berufen, zu verhandeln und zu streiten wird die Software nicht besser machen. Wenn sich die beiden Parteien gegenseitig respektieren und offen miteinander umgehen, ist das Endergebnis deutlich besser. Der Kunde soll während der gesamten Entwicklungsarbeit aktiv mitarbeiten. Es findet ein stetiger, vertrauensvoller Austausch zwischen Kunde und Entwicklungsteam statt, der durch permanente Erreichbarkeit des Kunden erst ermöglicht wird.

Auf Änderungen eingehen vor einen Plan verfolgen

Der letzte Punkt des Manifests befasst sich mit den Änderungen. Ein agiles Projekt ist auf Änderungen eingestellt: Neue Anforderungen können während der Entwicklung aufgenommen werden, bestehende Anforderungen werden konkretisiert oder verworfen. Es wird bewusst mit der Tatsache umgegangen, dass der Kunde zu Beginn des Projekts seine Anforderungen an die Software noch nicht komplett kennt. Hält sich das Entwicklerteam stur an die zu Beginn fixierten Anforderungen, ohne Änderungen zuzulassen, bekommt der Kunde schließlich eine Software, die für ihn nicht optimal ist. Agile Projekte zeichnen sich zwar durch diszipliniertes Vorgehen aus, und es existieren durchaus Pläne für Releases und Iterationen – der offene Umgang mit Änderungen ist jedoch inhärenter Bestandteil dieser Vorgehensweise.

Die Prinzipien

Die vier Kernaussagen des Manifests werden durch zwölf Prinzipien konkretisiert und erklärt. Ich werde diese im Folgenden etwas ausführen und auch immer das Negativ dazu beschreiben, also wie man es nicht machen sollte. Es handelt sich dabei allerdings jeweils um den Idealfall und sein extremes Pendant. Das wirkliche Leben ist natürlich nicht so schwarz-weiß, hier soll nur das Prinzip deutlich werden.

»Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.«

Die höchste Priorität bei der Erstellung der Software ist, den Kunden zufrieden zu stellen. Der erste Teil der Aussage hält die Softwarehersteller an, sich bewusst zu sein, dass es der Kunde ist, der die Software möchte, und er nur zufrieden ist, wenn er die Software bekommt, die er sich wünscht. Doch wie kann der Kunde zufrieden gestellt werden? Dazu ist es notwendig, dass das Entwicklerteam frühzeitig und kontinuierlich lauffähige Software bereitstellt. Der Kunde bekommt kontinuierlich ein Produkt, das er sich anschauen kann und mit dem er arbeiten kann. Dadurch sieht er den aktuellen Stand der Entwicklung und kann sich ein Bild von den bereits implementierten Anforderungen machen. Er kann zeitnah intervenieren, falls die Entwicklung in die falsche Richtung geht oder Anforderungen sich geändert haben. Nur eine lauffähige Software in einer gewissen Ausbaustufe informiert den Kunden ausreichend. Die in traditionellen Projekten eingesetzten ausführlichen Dokumentationen, Architektur- und Anforderungsdokumente können eine lauffähige Software nicht ersetzen.

»Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.«

In traditionellen Projekten wird ein Großteil der Architektur und Planung vor der Implementierung durchgeführt. Die Entwicklung folgt einem starren Prozess, der in Phasen abläuft. Die Phasen Analyse, Design, Entwicklung und Test folgen aufeinander bzw. sind nur eingeschränkt verzahnt. Anforderungsänderungen in späten Phasen finden keine Berücksichtigung. Ein häufig zu beobachtendes Szenario ist, dass der einmal aufgestellte Projektplan eisern bis zur Auslieferung verfolgt wird und geänderte und neue Anforderungen abgeblockt werden, so dass schließlich ein Produkt entsteht, das die dann aktuellen Anforderungen nicht erfüllt oder am Markt nicht mehr konkurrenzfähig ist. In agilen Projekten sind die einzelnen Phasen sehr eng verwoben und laufen fast gleichzeitig ab. Hier wird es nicht nur geduldet, sondern sogar begrüßt, wenn Änderungen an den Anforderungen während der eigentlichen Entwicklung bekannt werden. Um den Wert der erstellten Software zu halten oder zu steigern bzw. sich veränderten Marktsituationen zu stellen, ist es notwendig, die Entwicklung der Software jederzeit anpassen zu können.

»Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.«

Agile Projekte arbeiten sowohl *iterativ* als auch *inkrementell*. Eine *Iteration* ist dabei ein einzelner Entwicklungszyklus, der sämtliche Phasen der traditionellen Softwareentwicklung enthält (siehe oben). Die Software wird zudem *inkrementell* entwickelt. Das Team konzentriert sich immer auf einzelne Funktionalitäten, die in späteren Iterationen weiter ausgebaut und ergänzt werden. Die jeweiligen Inkremente, also die lauffähigen Zwischenlösungen, werden in kurzen, regelmäßigen Zeitspannen dem Kunden zur Verfügung gestellt. Traditionelle Projekte richten ihre Entwicklung dagegen gewöhnlich am Endtermin aus. Vor diesem Termin, an dem die Software dem Kunden übergeben werden soll, gibt es oft keine lauffähigen Zwischenlösungen, die mit den Kundenwünschen synchronisiert werden.

»Business people and developers work together daily throughout the project.«

Bei agilen Projekten ist es üblich, dass das Entwicklerteam und der Kunde sehr eng zusammenarbeiten. Die Entwickler kennen sich ausgezeichnet mit der Technik aus, während der Kunde natürlich der »Domain Expert« ist – er kennt sich mit den fachlichen Anforderungen aus. Der Kunde benötigt die Hilfe der Programmierer von Anfang an. Sie müssen nicht nur kontinuierlich den Aufwand schätzen, Anforderungen auf Machbarkeit prüfen und Inkremente bereitstellen. Die Programmierer helfen dem Kunden auch, seine Vorstellungen zu präzisieren. Zu Beginn weiß der Kunde seine Produktidee nur vage auszudrücken. Der Programmierer hilft ihm, im Laufe des Projekts ein immer klareres Bild von der Software zu bekommen. Kunde und Entwickler sprechen dabei in der fachlichen Sprache des Kunden. Kontinuierlicher Austausch hilft Missverständnisse zu vermeiden und Anforderungen zu konkretisieren. Dieser Austausch gelingt nur, wenn der Kunde täglich verfügbar ist.

Traditionelle Projekte dagegen bauen durch ihren Entwicklungsprozess (wenn auch unbewusst) Wände zwischen Kunde und Programmierer. Anforderungsänderungen in späten Phasen beispielsweise werden den Kunden negativ angelastet. Der Kunde seinerseits ist während der Entwicklung häufig nicht verfügbar. Das Entwicklungsteam implementiert dann einfach die zu Beginn entworfene Lösung ohne Rückkopplungen mit dem Kunden; dieses Vorgehen nennt man auch BDUF (*Big Design Up Front*, also vollständiges Design der Lösung zu Beginn). Sich erst während der Entwicklung stellende Fragen bleiben unbeantwortet, Inkonsistenzen in der Software unbehandelt. Programmierer schaffen sich oft eine ganz eigene Welt und kommunizieren in einer für den Kunden unverständlichen Sprache. Zwischen Kunde und Entwicklungsteam kann sich so kein Vertrauen entwickeln.

Das traditionelle Entwicklerteam wirft dann die fertige Software bildlich über eine Mauer. Ich habe einmal an einem nicht-agilen Projekt mitgearbeitet, in dem versucht wurde, die Anforderungen vollständig vor Beginn der Implementierung zu spezifizieren, was natürlich nicht gelang. Das Projekt hatte eine fest geplante Laufzeit. Das Entwicklerteam durfte und wollte die Anforderungskonkretisierungen während der Entwicklung nicht aufnehmen, was häufig Verzögerungen im Projekt-

plan zur Folge hatte. Man wollte nicht als Schuldiger dastehen, wenn das Produkt zu spät bereitgestellt würde. Das führte soweit, dass Entwickler, die eine Anforderung nicht verstanden und so nicht umsetzen konnten, sich heimlich für wenige Minuten mit Vertretern des Kunden trafen. Es war ein internes Projekt, bei dem die Programmierer durch ihren inoffiziellen Einsatz den Erfolg der Software retteten – entgegen den Vorgaben der Vorgesetzten, die von der Richtigkeit ihres Vorgehens überzeugt waren. Der Argwohn der beiden Parteien war so groß, dass sich die Programmierer schließlich rechtfertigen mussten, wenn Sie einen Vertreter des Kunden auch nur grüßten.

»Build projects around motivated individuals, give them the environment and support they need and trust them to get the job done.«

Agile Projekte bestehen aus Individuen und jeder Programmierer mit seinem individuellen Charakter ist ein Teil des Teams. Menschen sind schließlich unterschiedlich und beleben in ihrer Vielfältigkeit das Team. Unterschiede bei den Charaktereigenschaften, Interessen und Fähigkeiten werden daher akzeptiert. Die Programmierer werden durch hohes ihnen entgegengebrachtes Vertrauen motiviert und durch die Möglichkeit, sich aktiv einzubringen und Verantwortung zu übernehmen. Sie haben die besten Werkzeuge, die sie zum Erreichen der Ziele benötigen. Schließlich erfahren sie jederzeit jede Form von Unterstützung, die sie benötigen.

In nicht-agilen Projekten dagegen herrscht häufig Gleichschaltung; Charakterunterschiede bei den Programmierern werden nicht als Chance, sondern als Gefahr begriffen. Die Gleichschaltung, also das Zwängen in linienförmige Verhaltensmuster und der Mangel an Möglichkeiten, sich aktiv einzubringen und Verantwortung zu übernehmen, führt zu Frust. Demotivierend ist es auch, wenn Prozesse nicht transparent sind oder Informationen verheimlicht werden. Auch die Nutzung von für eine Aufgabe ungeeigneten Werkzeugen ist nicht nur unproduktiv, sondern wirkt demotivierend – schließlich möchte ein guter Programmierer auch mit guten Werkzeugen arbeiten. Weitere negative Beispiele sind Fälle, in denen der Vorgesetzte Entscheidungen nicht delegieren kann oder der Programmierer nicht respektiert wird. Mangelhafter Respekt kann sich zum Beispiel darin äußern, dass der Vorgesetzte die Aufwandsabschätzungen seiner Programmierer anzweifelt, obwohl er nicht die gleichen technischen Kenntnisse hat.

»The most efficient and effective method of conveying information with and within a development team is face-to-face conversation.«

Agile Projekte zeichnen sich dadurch aus, dass ein Großteil der Kommunikation *face-to-face*, also persönlich stattfindet. Persönliche Kommunikation schafft Vertrauen, und Missverständnisse und Schwierigkeiten können direkt geklärt werden. Durch unmittelbare Interaktion der Gesprächspartner wird gewährleistet, dass die Kommunikation so verläuft, wie es beide erwarten. Innerhalb des Entwicklerteams und zwischen Entwickler und Kunde herrscht also ein kontinuierlicher, direkter Austausch.

Nicht-agile Projekte verzichten auf den kontinuierlichen persönlichen Austausch. Sie kommunizieren überwiegend über elektronische Medien oder schriftliche Dokumente. Im Extremfall setzen Entwickler die schriftlich festgehaltenen Spezifikationen um, ohne auch nur ein einziges Mal mit dem Kunden gesprochen zu haben. Auch innerhalb des Entwicklerteams gibt es teilweise keinen regelmäßigen, direkten Austausch.

»Working software is the primary measure of progress.«

Die Software ist das Artefakt, das der Kunde in Auftrag gibt und das er so geliefert haben möchte, dass er es einsetzen kann. Alle anderen Artefakte, allen voran Spezifikation und Dokumentation, sind nur Mittel zum Zweck. Mit einer Spezifikation kann der Kunde nicht arbeiten. Kann der Kunde in kurzen Zyklen mit einem lauffähigen Softwareinkrement arbeiten, so ist das für ihn das einzige Mittel zu erkennen, wie der Entwicklungsstand ist. Denn Software lügt nicht. In agilen Projekten *zeigt* man, was man bisher entwickelt hat, anstatt es zu erklären oder zu beschreiben.

Bei nicht-agilen Projekten existiert auch nach langer Entwicklungszeit noch kein lauffähiges Produkt, sondern nur Akten voll mit Dokumentation (es soll auch schon nicht-agile Projekte gegeben haben, wo selbst die fehlt), so dass der Kunde nicht weiß, wie der Entwicklungsstand ist. Er hat keine Gewissheit, ob die Entwickler ihn richtig verstanden haben und die Anforderungen so umsetzen, wie er es möchte. Das hat zur Folge, dass er die Software zunächst ausschließlich vor dem geistigen Auge hat. Fachliche Inkonsistenzen kann er somit nur erahnen. Bei nicht-agilen Projekten macht man sich den Aufwand, mit einem Malprogramm ein Bild der Anwendungsoberfläche anzufertigen, anstatt Ihnen ein konkretes Nutzen und Testen der Funktionalitäten zu ermöglichen.

»Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.«

Bei agilen Projekten ist es unerlässlich, dass die Beteiligten in einem gleichmäßigen Tempo arbeiten. Die Arbeitsbelastungen der Programmierer werden auf das ganze Projekt verteilt. Dazu gehört auch die Vermeidung von Überstunden. Jede Stunde – die allererste Stunde in einem Projekt, aber auch die letzte – sind gleich wichtig und werden auch von allen als gleich wichtig wahrgenommen. Im Laufe des agiles Projekts wird die Arbeit weniger: Die Software läuft stabil und ist bereits frühzeitig gut getestet. Es gibt keine späten Überraschungen. Vor Abschluss des Projekts kann der eine oder andere sogar Urlaub machen, da es in ruhigem Fahrwasser läuft.

Haben Sie schon einmal einen Kurzstreckensprinter gesehen, der in seinem üblichen Tempo die 800 Meter laufen will? Ich nicht. Ich vermute allerdings, dass er nach 110 Metern umfallen wird. Ein 100-Meter-Lauf ist eine anaerobe Belastung. Der Körper benötigt viel Energie, die sehr schnell produziert werden muss – die unter Belastung stehenden Muskeln stellen Energie ohne Sauerstoff bereit. Bei Belastungen geringerer Intensität, also beispielsweise Gehen oder langsames Joggen, verbrennt der Körper Sauerstoff, den er einatmet und zu den unter Belastung stehenden Muskeln transportiert. Das ist aerobe Belastung.

Genauso ist es mit der Softwareentwicklung: Agile Projekte erstellen Ihre Software unter aerober Belastung, die die Beteiligten nicht näher an einen Herzinfarkt oder einen Familienzwist bringen. Nicht-agile Projekte arbeiten hingegen unter anaerober Belastung.

Bei nicht-agilen Projekten werden die ersten Stunden (bzw. Tage, Wochen oder gar Monate) des Projekts nicht selten verschlafen. Es hat zu Beginn bei den Kunden oder Entscheidungsträgern des Entwicklungsteams keine Wichtigkeit. Das Entwicklungsteam kann sich nicht selbst organisieren oder wird dadurch ausgebremst, dass wichtige Informationen über die zu realisierende Software fehlen. Andererseits denken aber auch die Entwickler häufig, sie hätten noch viel Zeit, und schieben ihre Aufgaben hinaus. Es existieren keine regelmäßigen Synchronisationspunkte, an denen sich Kunde und Entwicklungsteam ausrichten können.

Im Laufe des Projekts, wenn der Liefertermin näher rückt, nimmt dann die Arbeitsbelastung zu. Nun gewinnt das Projekt an Aufmerksamkeit, insbesondere die Entscheidungsträger erhöhen den Druck. Der Projektplan kann nicht eingehalten werden, da die Programmierer erst spät auf Schwierigkeiten stoßen oder Anforderungen nicht verstehen. Softwareintegrationen liefern permanent neue Fehler. Weder der Entwicklungsprozess noch die Software sind so konzipiert, dass sie mit dieser Situation umgehen können. Gewöhnlich folgen Überstunden, um die falschen Aufwandsabschätzungen und die falsch eingeschätzte Komplexität der Lösung in den Griff zu kriegen. Nach diversen Wochenendschichten machen die Programmierer durch ihre täglichen ungeplanten Eingriffe mehr kaputt, als sie reparieren. Jede Form von koordinierter Arbeit wird schließlich aufgegeben, nur um einen Endtermin einzuhalten, der nicht einzuhalten ist. Alle Beteiligten sind demotiviert.

»*Continuous attention to technical excellence and good design enhances agility.*«

Agile Projekte zeichnen sich durch hohe Ansprüche an die Qualität aus. Die Software wird einerseits unter Zuhilfenahme bester Werkzeuge erstellt. Andererseits wird auch das Design der Software stetig optimiert. Finden die Programmierer während der Realisierung der Anforderungen Defekte am Design, wird es augenblicklich verbessert. Dazu finden kontinuierlich *Refactorings* statt, die vom ganzen Team durchgeführt werden. Durch kollektives Eigentum an der ganzen Software ist nämlich jeder angehalten, direkt auf Designdefekte aufmerksam zu machen bzw. sie zu beseitigen. Es sind die kontinuierlichen Synchronisierungspunkte auf verschiedenen Ebenen – beispielsweise *Pair Programming*-Sitzungen, testgetriebene Entwicklung und häufige Iterationen –, die die Qualität (und somit das Design) verbessern. Es kommt dabei eine Wechselwirkung zum Tragen: Nur wenn die Qualität der Software sehr hoch ist, kann tatsächlich agil entwickelt werden. Software mit einem guten Design und konzeptioneller Integrität lässt sich nämlich leicht ändern oder erweitern und hat somit eine verlängerte Lebenszeit.

In nicht-agilen Projekten kommt der Qualitätssicherung manchmal zuwenig Bedeutung zu. Hier ist ausschließlich das wichtig, was der Kunde sieht und heute haben möchte. Diese kurzsichtige Herangehensweise ist selbstverstärkend: Ist das Design erst einmal von einer bedenklich niedrigen Qualität, fließt kaum noch Aufwand in seine

Verbesserung. Modifikationen und Erweiterungen an der Software sind dann nicht nur schwierig und kostenintensiv, sie verschlechtern das Design auch noch zusätzlich. Das läuft so ab wie bei einem Auto, das länger in einer Seitenstraße abgestellt ist: Zunächst bleibt es lange unversehrt. Sobald aber der erste Schaden am Auto eintritt, wenn zum Beispiel eine Fensterscheibe bei einem Unwetter zu Bruch geht, nimmt das Unglück seinen Lauf. Kinder werden nicht mehr zögern, an dem Auto ein, zwei Experimente durchzuführen. Andere finstere Gestalten benötigen vielleicht ein Ersatzteil... und in wenigen Tagen ist das Auto nicht mehr wiederzuerkennen. Aber zurück zur Software des nicht-agilen Teams: Wenn das Team einmal damit anfängt, die Software zu »patchen« und die ersten »Krücken« einzubauen, wird das Design bald nicht mehr zu retten sein. Zukünftige Änderungen und Erweiterungen haben dann zahlreiche negative Auswirkungen, sind fehleranfällig und dauern länger. Lange Reaktionszeiten aber gehen natürlich auf die Wettbewerbsfähigkeit und kosten so bares Geld.

»Simplicity – the art of maximizing the amount of work not done – is essential.«

Es ist eine Kunst, mit minimalem Aufwand ein optimales Ergebnis zu erreichen. Agile Entwicklungsteams konzentrieren sich darauf, eine für eine konkrete Aufgabenstellung minimale Lösung anzubieten, die gerade eben die Anforderung erfüllt. Diese »einfache« Lösung verringert die Komplexität der Anwendung und fördert gutes Design.

Nicht-agile Projekte hingegen entwickeln Funktionalität oft auf Vorrat. Sie erstellen Software, die mehr kann, als der Kunde benötigt. Häufig werden generische Lösungen und ganze Frameworks erstellt, die die zukünftigen Kundenwünsche in sämtlichen Abwandlungen vorwegnehmen möchten. Dies macht das Design unnötig komplex. Unnötige Funktionalität oder generische Lösungen schränken die Agilität deutlich ein. Zukünftige Änderungen und Erweiterungen sind dann schwer möglich.

»The best architectures, requirements and designs emerge from self-organizing teams.«

Agile Projekte profitieren von selbstorganisierten Teams. Solche Teams verteilen Aufgaben selbstständig untereinander. Architektur und Design werden optimiert durch enge Zusammenarbeit und kontinuierlichen Wissensaustausch. Auch Anforderungen werden im Teamgespräch oft klarer. Selbstorganisierte Teams helfen sich gegenseitig, respektieren sich und bereinigen zwischenmenschliche Probleme selbst. Was sie benötigen, ist ein Container, also eine Umgebung, in der sie in Ruhe arbeiten können. Der Manager muss diese Umgebung schaffen.

Nicht gewünscht ist, dass Programmierer die ihnen gestellten Anforderungen umsetzen, ohne die Möglichkeit zu haben, sich aktiv an ihrer Gewinnung und Fortschreibung zu beteiligen. Durch fehlenden Kontakt mit dem Kunden verstehen sie dann nämlich die Anforderungen vielleicht nicht. Sie setzen sie zwar um, ihnen selbst fallen allerdings keine fachlichen Inkonsistenzen auf. Anforderungen werden in solchen Projekten zudem zwecks Realisierung teilweise Programmierern zugewiesen, die für diese Implementierung nicht bestens geeignet sind. Die Architektur der Anwendung wird von oben herab vorgegeben und die unterschiedlichen Fähigkeiten und Erfahrungen der Mitglieder des Teams werden nicht genutzt.

»At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.«

Ein Grundgedanke der agilen Entwicklung ist das kontinuierliche Lernen. Dies betrifft insbesondere auch die Arbeit des Teams als Ganzen. Wie kann das Team noch effektiver arbeiten? Was kann verbessert werden? Wo gab es zuletzt Probleme? Alle Beteiligten treffen sich regelmäßig zu einer Art Manöverkritik und diskutieren offen und sachlich über die zuletzt aufgetretenen Schwierigkeiten. Solche Gespräche finden in kurzen Abständen statt und sollten mindestens nach jeder Iteration durchgeführt werden. In der nächsten Iteration werden die angesprochenen Schwierigkeiten gemeinsam behoben. Auch tägliche *Stand-up-Meetings* sind eine Form der Retrospektive.

Nicht-agile Teams führen oft keine Manöverkritiken durch. Nicht selten behaupten Sie von sich und der Arbeitsweise des Teams, es gäbe nichts mehr zu verbessern. Wo agile Teams immer Verbesserungspotenzial suchen, ruhen sich traditionelle Teams auf dem Status Quo aus. Sie verteidigen vehement ihre Arbeitsweise, auch wenn diese kontinuierlich zu Problemen führt. Gelegentlich gibt es auch bei konventionellen Projekten Retrospektiven. Diese werden allerdings zu unregelmäßig durchgeführt, wodurch die Chance vertan wird, während des Projekts justierend einzugreifen. In diesem Fall kommt die Retrospektive einem Nachruf oder einer Bestandsaufnahme gleich. Andere nicht-agile Teams führen zwar vielleicht regelmäßige Manöverkritiken durch, allerdings falsch – wenn sie in einer Atmosphäre der Unsachlichkeit und der gegenseitigen Schuldzuweisungen stattfinden, werden die Chancen solcher Runden nicht genutzt, im Gegenteil: Das Fundament des Misstrauens ist gelegt, das die weitere Arbeit nur erschwert.

KAPITEL 2

Die Vorteile agiler Entwicklung

Agile Softwareentwicklung hat gegenüber der klassischen Entwicklung für alle Beteiligten zahlreiche Vorteile. Im Folgenden beleuchte ich den Nutzen agiler Entwicklung aus verschiedenen Blickwinkeln. Wir betrachten den Nutzen aus Kundensicht, aus der Sicht des einzelnen Programmierers, aus der Projekt- bzw. Unternehmenssicht und aus einer wissenschaftlicheren Sicht.

Die Kundensicht

Kunde ist im Folgenden ein Sammelbegriff für die Personen, die das Softwareprodukt in Auftrag geben und/oder mit dem entwickelten Produkt arbeiten werden, also die Nutzer der Software sind.

Anforderung und Anwendung entwickeln sich synchron

In agilen Projekten kann der Kunde jederzeit auf das Projekt einwirken. Dies ist besonders bei lange laufenden Projekten von Bedeutung. Der Kunde kennt häufig zu Beginn noch gar nicht die genauen Anforderungen an das zu entwickelnde Produkt. Erst im Lauf der Zeit formt sich mit den ersten Zwischenlösungen auch sein Bild, wie er die Anforderungen genau umgesetzt haben möchte. Durch die kontinuierlichen Feedbackschleifen können Anforderungen jederzeit verfeinert, geändert oder verworfen werden.

Lauffähige Zwischenlösungen

In agilen Projekten kann der Kunde zusammen mit der Entwicklungsmannschaft über die bisher durchgeführten Arbeiten kontinuierlich reflektieren: Die Zwischenlösungen sind immer lauffähig und für den Kunden sichtbar. Diese Lösungen beinhalten zwar nicht die komplette, endgültige Funktionalität, sind aber zu jedem Zeitpunkt viel besser greifbar als eine Dokumentation oder ein Konzept, das in Papierform vorliegt. Nur wenn »richtige« Nutzer mit dem System arbeiten, werden tatsächlich alle Probleme zu Tage treten. Je früher das passiert, desto besser.

Auch und gerade bei der Umstellung eines Legacy-Systems können agile Methoden helfen. Meine Erfahrung ist, dass man Altsysteme in kleinen Schritten ablösen sollte, nicht in einem einzigen großen. Werden vom agilen Prozess frühzeitig Inkremente geliefert, können diese bei der Umstellung direkt helfen.

Kurze Reaktionszeit auf Marktentwicklung

Damit einher geht der strategische Vorteil, sich schneller den Entwicklungen am Markt anpassen und auf Entscheidungen von Marktbegleitern reagieren zu können. Schnelle Reaktionen auf die Aktionen der Konkurrenten, die sich mit der agil entwickelnden Unternehmung in Wettbewerb befinden, sind entscheidend, um sich am Markt behaupten zu können. Nehmen Sie hier beispielsweise die Entwicklung der Handysparte des Siemens-Konzerns: Vor Jahren noch ein marktführender Handyanbieter, hat Siemens die rasante Entwicklung in der Branche versäumt. Konkurrenten wie Nokia erkannten zeitnah die Bedürfnisse der Kunden und entwickelten sehr schnell neue Handymodelle und brachten sie auf den Markt. Die fehlende Nähe zum Kunden und die daraus resultierende falsche Unternehmensstrategie sowie lange Entwicklungszeiten haben den Rückstand von Siemens konstant vergrößert. Es waren letztendlich Managementfehler, die für den Untergang verantwortlich waren. Schließlich hat Siemens das defizitäre Handy-Geschäft abgestoßen, was für diesen führenden Elektronikkonzern einen immensen Imageschaden darstellt.

Mit einer agilen Entwicklung können auf der anderen Seite Gelegenheiten, die sich im Lauf der Prozess- und Marktzyklen neu auftun, sofort aufgenommen und umgesetzt werden, mit einer bemerkenswerten *time to market*. Mit der Veröffentlichung eines Produkts können nicht nur aktuelle Entwicklungen aufgenommen, sondern sogar neue Marktlücken geschlossen werden.

Vertrauen und Verständigung

Der permanente Austausch zwischen Kunde und Entwicklungsteam und die aktive Einbeziehung des Kunden schafft zudem Vertrauen. Potenzieller Skepsis oder Vorbehalten kann frühzeitig entgegnet werden. Typische Kundensorgen sind »Hat der Entwickler mich richtig verstanden?«, »Was steht da eigentlich in den ganzen Dokumenten?« und »Die sprechen eine ganz andere Sprache!«

Im agilen Projekt darf der Kunde seine Wünsche in seiner eigenen Sprache in Form von *User Stories* formulieren. Diese werden dann von den Entwicklern realisiert. Der Kunde bewegt sich also in »seiner Welt« mit »seiner Sprache«. Letztendlich kann er selbst am Besten ausdrücken, was er umgesetzt haben möchte. User Stories auf *Story Cards* sind dabei nur Gedächtnisstützen. Das Wissen ist in erster Linie im Kopf der Beteiligten. Ist den Entwicklern eine Kundenanforderung unklar, fragen Sie beim Kunden direkt nach. Das ist möglich, da der Kunde bei agilen Projekten während der Entwicklung vor Ort oder zumindest ständig erreichbar ist.

Agile Entwickler verstricken sich nicht in Implementierungen, die der Kunde gar nicht haben möchte. So ist es in konventionellen Projekten durchaus üblich, dass Entwickler Funktionalität »auf Vorrat« realisieren. Diese so genannte *Featuritis* wird bei der agilen Entwicklung durch tägliche Rückkopplungen vermieden. Die konkreten umzusetzenden Aufgaben werden dabei in Tasks formuliert und genauso transparent gemacht wie die User Stories. Der Kunde und alle anderen Beteiligten sehen also jederzeit, wer an was arbeitet und wie der Fortschritt ist.

Höherer Return On Investment

Ein weiterer, nicht unerheblicher Vorteil ist, dass früher Rendite erwirtschaftet wird, die dann letztendlich in ihrer Gesamtheit höher ist. Durch die inkrementelle Vorgehensweise stehen schnell reife Versionen zur Verfügung, die gegebenenfalls poduktiv schon genutzt werden können. Die Software wirft für den Kunden also schneller Gewinn ab. Die Rendite wäre zwar formal höher, würde das Produkt in einem einzigen Inkrement bereit gestellt. Dies dauert aber deutlich länger, und in diesem Zeitraum wird erst einmal überhaupt kein Geldrücklauf erwirtschaftet. Bei agilen Projekten ist damit der sogenannte ROI (Return on Investments) deutlich höher.

Dieser errechnet sich durch Division des Gewinns durch den Kapitaleinsatz. In agilen Projekten wird die Rendite zudem durch ein einfaches, schlankes Design der Anwendung erreicht. Generische Lösungen oder Funktionalitäten, die zum entsprechenden Zeitpunkt eigentlich nicht benötigt werden, senken hingegen die Rendite. Konzentriert sich das Team, wie bei agilen Arbeitsprozessen üblich, auf die wesentlichen Punkte, ist das Produkt deutlich schneller am Markt. Bei traditionellen Projekten kommen nicht nur die einmaligen Kosten der Erstellung einer generischen oder zu komplexen Lösung zum Tragen: Auch die Wartung und Weiterentwicklung dieser Software ist deutlich fehleranfälliger und zeitintensiver – und somit teurer.

Höhere Qualität des Produkts

Nicht zuletzt profitiert der Kunde natürlich von der höheren Qualität der Software. Das Design der Software unterliegt – wie erläutert – ständig einem gemeinschaftlichen Review. Durch die Entwicklung von Akzeptanztests, funktionalen Tests und Komponententests vor den fachlichen Klassen und *Pair Programming Sessions* wird das Design der Software verbessert. Häufige Integrationen und automatisierte Tests identifizieren Fehler früh. Testen nimmt einen ganz neuen Stellenwert ein: Das bei konventionellen Projekten übliche nachgelagerte Testen *findet* Fehler, das vorgelagerte und begleitende Testen bei agilen Projekten *verhindert* sie.

Stetiges Refactoring und viele Tests (neben den bereits angesprochenen Tests auch Integrationstests) tragen dazu bei, dass die Software stabil und von gutem Design ist. Muss ein Kunde dann für die Wartung selbst aufkommen, kann ihm eine agile Entwicklung erhebliche Kosten sparen. Bei nicht wenigen traditionellen Projekten hingegen kann er nämlich Schwachstellen erst entdecken, wenn es für ihn eigentlich zu spät ist – und er muss die Folgekosten selbst tragen:

Geringeres Risiko für Folgekosten

Aus Sicht des Risikomanagements ergibt sich eine höhere Wahrscheinlichkeit des wirtschaftlichen Erfolgs eines Produkts auch daraus, dass fehlerhafte Implementierungen durch frühe Akzeptanztests, testgetriebene Entwicklung und kontinuierliche Integration frühzeitig auffallen und behoben werden können (»technisches Risiko«). Technisch anspruchsvolle Implementierungen und Betreten von technischem Neuland werden explizit berücksichtigt. Hier stellen agile Projekte häufig eine zeitlich abgeschlossene Auswertungszeit den eigentlichen Iterationen voran. Erst wenn das technische Risiko überblickt werden kann oder eine Machbarkeitsstudie erfolgreich verlaufen ist, beginnen die eigentlichen Iterationen. Erstellte Prototypen werden häufig als »Spikes« bezeichnet. Gelegentlich werden vorbereitende Iterationen, Iterationen bevor die eigentliche Fachlichkeit implementiert wird, auch als »Iteration 0« bezeichnet.

Neben der Verminderung des technischen Risikos erlaubt agile Entwicklung das frühe Aufdecken eines fachlichen Risikos. Widersprüchliche Anforderungen können frühzeitig am laufenden System aufgedeckt werden. Es liegt im Interesse des Kunden, mit Abschluss der Entwicklungsarbeiten ein Produkt zu bekommen, das seinen Vorstellungen entspricht. Dies kann er aber nur an dem lauffähigen System überprüfen. In traditionellen Projekten dienen häufig Schriftstücke als Medium des Austauschs zwischen Entwickler und Kunde. Der Kunde möchte aber kein Dokument kaufen, er möchte eine Software einsetzen. Nur am lauffähigen System fallen ihm fachliche Diskrepanzen, Inkonsistenzen oder Defizite auf. Je später Defekte an einer Anwendung entdeckt werden, desto teurer wird ihre Eliminierung. Das trifft auch auf fachliche Defekte zu. Durch stetige Rückkopplungen und kurze Interaktionen minimieren agile Projekte das fachliche Risiko drastisch.

Geringeres Risiko für das Scheitern des gesamten Projekts

Letztendlich ist das Scheitern eines Projekts auch für den Kunden immer ärgerlich, auch wenn der Auftragnehmer daran Schuld ist und dem Kunden keine Kosten entstehen. Bei agilen Projekten sehen sich alle Beteiligten als ein Team, das ein gemeinsames Ziel erreichen will. Schließlich wollen alle das Gleiche, nämlich lauffähige Software erstellen.

Programmierersicht

Auch aus Sicht der einzelnen Programmierer hat eine Entwicklung nach agiler Methodik Vorteile.

Chance auf gut durchdachtes Design

In agilen Projekten wird häufig in Zweierteams entwickelt. Dadurch wird das Design der Software verbessert. Die Beteiligten lernen nämlich voneinander und reflektieren mehr über die Entwicklung. Probleme werden schneller erkannt und können zusammen gelöst werden. Während der eine die Tastatur hat und Quellcode schreibt, kann der andere nicht nur über diesen neuen Quellcode, sondern auch über das ganze Design reflektieren.

Keine Wissensmonopole und einseitigen Aufgabenprofile

Es existiert keine strikte Rollentrennung bei der Entwicklung, wie es in konventionellen Projekten üblich ist. In agilen Projekten geschieht die Aufgabenverteilung abhängig von Situation und Fähigkeiten. Dadurch, dass das Wissen über die Software kollektives Eigentum ist, wird einerseits der Druck auf den Einzelnen vermindert (er ist nicht mehr der Einzige, der sich mit dieser oder jener Komponente auskennt und darum z. B. nie krank werden oder in den Urlaub fahren darf), andererseits gibt es keine »schwarzen Löcher«, d. h. Implementierungen, die nur eine Person kennt. Der stetige Austausch beugt also Wissensmonopolen vor. Durch sich ändernde Aufgabengebiete steht man zudem immer wieder vor neuen technischen Herausforderungen, was willkommene Abwechslung bieten kann.

Kurzer Draht zum Kunden

Der agile Programmierer versteht außerdem den Kunden besser. Durch den stetigen Austausch mit ihm kann er Fragen stellen, und Unklarheiten können zeitnah beseitigt werden. Häufig haben die Entwickler andernfalls Angst, dass Sie im Rahmen von schwergewichtigen Prozessen die Design-Sessions oder Kunden-Workshops falsch interpretieren, und den Kunden möglicherweise falsch verstanden haben.

Teilhabe an der Projektverantwortung

Ein weiterer positiver Aspekt ist, dass der Programmierer aktiv in die Planung eingebunden ist: Er unterstützt den Kunden beim Entwerfen der User Stories, trägt also Verantwortung. Wenn der Programmierer merkt, dass man ihm vertraut und dass er eine Verantwortung hat, steigert das sein Selbstwertgefühl. Vertrauen zu spüren ist eine elementare Voraussetzung für gute Arbeit, wie wir später noch diskutieren werden. Agile Projekte fördern aktive Partizipation im gleichberechtigten Team und profitieren von ihr. In herkömmlichen Projekten stehen häufig die organisatorischen Hierarchien im Vordergrund: Der Vorgesetzte beauftragt einen Mitarbeiter, einen bestimmten Auftrag zu erfüllen. In agilen Projekten wählt der Programmierer aus dem Pool an zu realisierenden technischen Aufgaben eine aus. Das führt ihn heraus aus einer defensiven Haltung, in der er ausschließlich die ihm aufgetragenen Aufgaben umsetzen soll, hin zu eigenverantwortlichem Arbeiten. Er fühlt, dass er ein wichtiger Teil des Teams ist.

Eigenverantwortliche Voranschläge

Der Programmierer akzeptiert die Verantwortung, eine Aufgabe erfolgreich zu realisieren. Verantwortung kann nicht zugewiesen werden, sie muss akzeptiert werden. Eine weitere wichtige Dimension der aktiven Teilnahme an agilen Projekten ist das Fällen von Entscheidungen im Team. Ein Beispiel dafür sind die Aufwandsschätzungen, die in traditionellen Projekten meist von den Vorgesetzten durchgeführt werden. Obwohl ein Vorgesetzter gewöhnlich nicht die gleichen technischen Kenntnisse wie ein Spezialist hat, gibt

er vor, wie lang die Realisierung einer Aufgabe dauern wird. Er tut das häufig durch die Anwendung von indirektem Druck. Das Verheerende ist darüber hinaus, dass der Programmierer auch noch für die von ihm nicht verantworteten falschen Voranschläge zur Rechenschaft gezogen wird, wenn eine Entwicklung nicht in der vorgegebenen Zeit gelingt. Dies gilt auch für technische Lösungskonzepte aller Art, die er von einem Nichttechniker vorgesagt bekommt. Letztendlich hat das ganze Team und das Projekt nichts davon: Das Projekt verläuft nicht erfolgreich oder verzögert sich, und im Team entsteht ein Vertrauensverlust, insbesondere, wenn der Manager die Schuld für seine Fehlentscheidungen bei einem einzelnen Programmierer sucht.

In agilen Projekten vertraut der Vorgesetzte den Schätzungen der Spezialisten, die sich gemeinsam über die notwendigen Aufwände für anstehende Arbeiten einigen. Daneben entscheidet das Team über die Art der technischen Lösung. In agilen Projekten bestimmt das Team weitestgehend autonom und organisiert sich selbst (wie es beispielsweise durch die selbstständige Wahl der nächsten Aufgabe geschieht). Die gemeinschaftlichen Entscheidungen des agilen Teams befreien den Einzelnen von einem großen Druck und somit auch von der Angst, Fehler zu machen. Ferner werden keine unlösbaren Vorgaben von »außen« an den Entwickler herangetragen.

In traditionellen Projekten existiert häufig ein Architekt, ein Programmierer mit besonderer fachlicher Befugnis. Dieser ist für grundlegende Entscheidungen über die Architektur der Lösung und ihre langfristigen Konsequenzen allein verantwortlich. Er entwirft die Software am Reißbrett. Nachdem die erste Anwendung fertiggestellt ist, wird die zweite entworfen. Ein häufig auftretendes Problem dabei ist, dass die Architektur der zweiten Anwendung viel zu umfangreich und kompliziert entworfen wird. Agile Projekte verteilen die Architekturaufgaben generell auf die Schultern aller Programmierer. So ist es auch beim zweiten agilen Projekt nahezu ausgeschlossen, dass das ganze Team eine Anwendung konzipiert, die unnötig komplex ist.

In agilen Teams steht jederzeit die Leistung des ganzen Teams im Vordergrund. Wenn ein agiles Team mit hochqualifizierten, motivierten Personen besetzt ist, kann ein einzelner Programmierer niemals die Schuld daran haben, wenn es mit einer Sache nicht vorangeht oder etwas falsch realisiert wird. In solchen Fällen ist es das Problem des ganzen Teams, dem technisches oder fachliches Wissen fehlt oder den Entwicklungsprozess mangelhaft durchgeführt hat.

Keine extreme Arbeitsbelastung

Schließlich wirkt sich die Tatsache auf die Leistungsfähigkeit und Motivation des Programmierers aus, dass keine Überstunden gemacht werden sollen. Der Einzelne ist durch einen geregelten Arbeitstag ohne Überstunden produktiver. Während Überstunden werden häufig mehr Fehler gemacht. Unter häufigen Überstunden leidet nicht nur die Motivation der Programmierer, Überstunden sind auch ein Team-Killer: Machen einzelne Programmierer Überstunden und andere nicht, werden Spannungen zwischen ihnen erzeugt, die zu Vorwürfen und Schuldzuweisungen führen können, falls sich aufgrund der Belastung des Privatlebens Probleme ergeben.

Die Projektsicht

Der Vorgang der Erstellung von Software ist eine Mischung aus Ingenieurdisziplin, kreativer Kunst und einer Menge Soziologie.

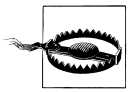
Risikobewusstsein

Agile Projekte leugnen das hohe mit der Erstellung von Software verbundene Risiko nicht und gehen offen damit um. Auch in anderen, nicht-agilen Vorgehensmodellen wird mit Risiken umgegangen. In konventionellen Projekten gibt es gewöhnlich Risikolisten, die einzelne Risiken aufführen und bewerten. Das ist schon einmal ein sehr guter Schritt. Die Vielzahl der Risiken zu ignorieren, die mit Softwareentwicklung verbunden sind, wäre nämlich fahrlässig. Erstellen sie eine Liste in dieser Form:

Tabelle 2-1: Risikoliste

Risiko	Eintrittswahrscheinlichkeit	Auswirkung	Verminderungsmaßnahmen	Eventualmaßnahmen
Entwickler erkrankt	gering	hoch	Pair Programming Session	Tasks neu verteilen
Integration schlägt fehl	hoch	hoch	Continuous Integration	keine neuen Funktionen ins Repository einchecken, Integration fixen
Anforderung falsch umgesetzt	hoch	hoch	kontinuierlicher Austausch mit dem Kunden, frühe Iterationen	Prozess agil ausrichten

Risikolisten machen die Risikoarten transparent. Ist die Liste offen verfügbar und einsehbar, schafft das ein Bewusstsein dafür, mit welchen Risiken das Projekt täglich konfrontiert ist und welche Konsequenzen ein Eintreten des Risikofalls hat. Vorbeugende Maßnahmen können durchgeführt werden, um die Auswirkungen eines eventuell eingetretenen Problemfalls zu minimieren. Diese Maßnahmen erhöhen zwar zunächst den Aufwand, ziehen also erhöhte Kosten nach sich, minimieren aber diejenigen Kosten drastisch, die im Schadensfall auftreten würden. Notfallmaßnahmen dagegen sind Maßnahmen, die durchgeführt werden, wenn der Fall bereits eingetreten ist. Es ist verwunderlich und bedenklich, wie viele Projekte sich gar nicht mit den tagtäglichen Risiken beschäftigen. Diese Projekte hängen an einem seidenen Faden.



Der Einsatz einer Risikoliste oder jeder anderen Art von Dokumentation bringt nur Mehrwert, wenn das Team bzw. der Manager ihren Einsatz beherrscht. Ergänzen Sie eine Risikoliste heimlich für sich oder verfassen Sie eine Dokumentation, um sich in irgendeiner Form zu schützen, so lassen Sie das mit Risikoliste lieber sein. Ein zerrüttetes Vertrauen können Sie mit solchen Dokumenten garantiert nicht verbessern!

Risikominimierung

Agile Projekte gehen aber noch weiter und erstellen nicht nur eine Risikoliste: Ihre ganze Arbeitsweise ist auf Risiko*minimierung* angelegt. So wird der Kunde definitiv Software erhalten, die ihn nicht überrascht. Kurze Iterationen und stetige Akzeptanztests fördern das Vertrauen des Kunden in das Produkt. Es gibt kein »Big-Bang-Meeting«, bei dem nach einem halben Jahr Arbeit der Kunde sein Produkt vorgestellt bekommt, nur um festzustellen, dass er die Hälfte anders haben wollte.

Im Gegensatz zum traditionellen *expliziten* Vorgehen pflegen agile Projekte einen *impliziten* Umgang mit dem Risiko: Der Umgang mit dem Risiko ist inhärenter Bestandteil des gesamten Prozesses. Agile Projekte schreiben die Risiken nicht explizit auf. Beispielsweise sind stetige Iterationen und kontinuierliche Rückkopplungen mit dem Kunden eine Methode, um Fehlentwicklungen vorzubeugen und sich neuen oder geänderten Anforderungen zeitnah zu stellen. Ein agiles Projekt kommt ohne Risikoliste aus. Der gesamte agile Prozess ist vielmehr auf Risikominimierung ausgerichtet.

Auch das stetige Review ist in agilen Projekten eine Risikoverminderungsmaßnahme. Der gesamte Prozess steht jederzeit auf dem Prüfstand: Das Design der Software wird durch Refactorings stetig optimiert, der Prozess wird von Rückblicken begleitet. Nach jeder Iteration findet eine Retrospektive statt, in der alle Beteiligten den Verlauf der Iteration kritisch hinterfragen. Offen und konstruktiv werden Verbesserungsmöglichkeiten identifiziert und mit der nächsten Iteration direkt angewendet. Nicht-agile Teams verzichten auf diese begleitenden Rückblicke: Ähnlich der großen »Big-Bang-Iteration« am Schluss, in der erst sehr spät Probleme offenkundig werden, und der »Big-Bang-Auslieferung« der Software in einem Stück am Ende des Projekts werden Retrospektiven, wenn überhaupt, nur einmal am Ende des Projekts durchgeführt. Dann ist es aber schon zu spät, um justierend auf den Entwicklungsprozess einzuwirken und Probleme zu lösen.

Dies alles kann durch regelmäßige Rückblicke verhindert werden. Wenn sich alle Beteiligten in einer offenen, respektvollen und konstruktiven Art und Weise in kurzen Abschnitten austauschen, verläuft der Entwicklungsprozess transparenter und effektiver.

Keine Risikokonzentration

Ein weiteres Beispiel für dieses Vorgehen ist der Umgang mit dem Ausfall einzelner Personen in agilen Teams. Ein derartiger Ausfall kann durch die Grundstruktur und den gelebten Prozess leicht kompensiert werden: Es existieren keine Wissensmonopole, so dass im Krankheitsfall andere Entwickler einspringen können. Alle Entwickler kennen sich (nahezu) gleich gut in der Anwendung aus. Auch eine eventuelle Kündigung eines Entwicklers kann so besser aufgefangen werden.

Realistische Aufwandsschätzungen

Auch Aufwandsschätzungen werden verlässlicher, da sie im Team getroffen und ständig einer kritischen Überprüfung (Review) unterzogen werden. Die Schätzung einer einzelnen Person kann schon einmal falsch sein – denn eine Schätzung bleibt immer eine Schätzung.

In vielen Projekten gelingt es nicht, das Produkt in der gewünschten Zeit im gewünschten Umfang und mit den geplanten Ressourcen fertigzustellen. In agilen Projekten soll nur das verwirklicht werden, was tatsächlich einen Nutzen für den Kunden hat. Durch ständigen Austausch mit dem Kunden und Prioritätsanalysen sollen die *unbedingt* zu erstellenden Funktionen identifiziert werden. Dabei sollte mit den Funktionen begonnen werden, die den größten Nutzen haben und das größte (technische) Risiko darstellen. In agilen Projekten ist es daher viel unwahrscheinlicher, dass die Zeitplanung völlig aus dem Ruder läuft. Und wird ein Projekt, zum Beispiel aus Kostengründen, vorzeitig eingestellt, besteht durch die regelmäßigen Iterationen dennoch ein einsatzfähiges Produkt. Was will man mehr?

Soziale und betriebswirtschaftliche Sicht

Auch Erkenntnisse der Sozialwissenschaften werden zur Bewertung agiler Entwicklung genutzt und die agile Entwicklung vor dem Hintergrund der Psychologie betrachtet. Traditionelle Projekte sehen den Mensch nämlich als austauschbaren Faktor und ignorieren zwischenmenschliche Probleme. Agile Projekte hingegen stellen den Menschen in den Mittelpunkt – schließlich wird Software *von* Menschen *für* Menschen erstellt.

Mitarbeiterzufriedenheit

Das Personalwesen in Unternehmen betrachtet agile Entwicklung insbesondere im Hinblick auf ihre Auswirkungen auf die Mitarbeiterzufriedenheit. Die Programmierer sind durch die akzeptierte Verantwortung, durch ein hohes Maß an ihnen entgegengebrachtem Vertrauen und gewisse Freiheiten beispielsweise bei der Auswahl der als Nächstes zu implementierenden Aufgaben hoch motiviert.

Kontinuierliche Weiterbildung und kollektiver Wissenszuwachs

Ferner trägt agile Entwicklung zum kooperativen Lernen bei. Für das Personalwesen ist dieses Verfahren also besonders aus Sicht der Personalentwicklung interessant. Gerade in der schnelllebigen IT-Branche mit ihren immer neuen technischen Herausforderungen müssen die Mitarbeiter, allen voran die Programmierer, ein Bewusstsein für stetige Weiterbildung haben. In nicht-agilen Projekten wird häufig umfassendes Wissen bei jedem Einzelnen vorausgesetzt, was jedoch bei der sich exponentiell weiterentwickelnden Technik utopisch ist. Häufig wird in IT-Unternehmen über Weiterbildungsmaßnahmen nachgedacht und die Programmierer werden jährlich auf eine Weiterbildung geschickt.

Unternehmen, die ausschließlich so mit Wissen und Weiterbildung umgehen, haben den Kern der Problematik nicht erkannt. Stattdessen müssen ein kollektives Teamwissen und ein kontinuierliches gemeinsames Lernen etabliert werden. Falls Sie mit dem Star Trek-Universum vertraut sind, setzen Sie Ihr Team mit den »Borg« gleich – aber bitte nur im positiven Sinn: Gehen Sie morgen nicht in ein fremdes Arbeitszimmer und versuchen, ein anderes Projekt zu assimilieren!

Höhere Gesamtproduktivität

Durch höhere Mitarbeiterzufriedenheit und die Vermeidung von Überstunden soll die gesamte Produktivität erhöht werden. Die Praxis des Pair-Programming lässt zwar – rein mathematisch betrachtet – Gegenteiliges vermuten. Im Endeffekt aber ist die Gesamtproduktivität höher. Denn die Vermeidung von Spezialistentum und individuellem Besitz von Passagen der Software dient der kollektiven Wissenskonstruktion, indem sie den Austausch von Entwickler zu Entwickler verbessert.

Über agile Entwicklung soll eine größere Einbindung des Kunden in den Wertschöpfungsprozess möglich sein. Dies gewinnt umso mehr an Bedeutung, wenn man sich klarmacht, dass Software nur ein Mittel zum Zweck ist. Der Kunde möchte durch die Nutzung der Software etwas erreichen. Die Software ist unter diesem Gesichtspunkt eher ein Vorprodukt, also ein Produkt, das zur Herstellung (oder Verwaltung) eines anderen Produkts (oder Prozesses) dient.

Adäquate Informationsmedien

Agile Entwicklung muss auch aus Sicht des Informationsmanagements betrachtet werden, das den Aufwand für den notwendigen Informationsaustausch festlegt und ökonomisch bewertet. Dazu werden Erkenntnisse der Informations- und Kommunikationswissenschaft genutzt. Dabei kann insbesondere die *Medienreichhaltigkeitstheorie* eingesetzt werden: Weil die Sachverhalte, die es zu kommunizieren und zu diskutieren gilt, in der Regel sehr komplex sind, werden auch sehr komplexe, reichhaltige Kommunikationsmedien gewählt: direkte, persönliche Gespräche.

Bei konventionellen Projekten unterliegt man dem Trugschluss, komplexe Sachverhalte ausschließlich in Papierform dokumentieren zu müssen. Es werden ganze Schränke mit Spezifikationen gefüllt. Bei agilen Projekten weiß man, dass das Verständnis der komplexen Sachverhalte ausschließlich in den Köpfen der Beteiligten verfügbar ist. Auch wenn er vorher eine ganze Bürowand mit Spezifikationsdokumenten gefüllt hat, weiß der Kunde erst zu dem Zeitpunkt, an dem er die fertige Anwendung in der Hand hat, ob er das bekommen hat, was er wollte. Das bedeutet nicht, dass es in agilen Projekten keine Dokumentation gäbe. Diese besteht jedoch eher aus Gedächtnisstützen. Auch die User Stories beschreiben nur die wesentlichen Eigenschaften eines fachlichen Szenarios. Erst in stetigen begleitenden Gesprächen werden diese Gedächtnisstützen mit Leben gefüllt.

KAPITEL 3

Agile Ansätze und traditionelles Vorgehen – eine Abgrenzung

Wir haben in den vorangegangenen Kapiteln bereits damit begonnen, die agile Entwicklung von Software einer traditionellen Methodik gegenüberzustellen. Wir wissen nun, was agile Entwicklung ist und welche Vorteile sie mit sich bringt. In diesem Kapitel behandle ich detailliert die traditionellen Ansätze und die Frage, was agile Entwicklung ist, was sie nicht ist, und welche populären Ausprägungen agiler Entwicklung es gibt. Mit diesem Kapitel schließt sich der Kreis: Sie verstehen den Mehrwert der agilen Entwicklung, können sie von traditionellen Vorgehensweisen abgrenzen und kennen nun mit Scrum, Extreme Programming und Lean Development drei Ausprägungen. In späteren Kapiteln erfahren Sie genauer, wie Sie in Ihrem agilen Projekt ganz konkret vorgehen können.

Prozessmodelle

Ein konventionelles, schwergewichtiges IT-Prozessmodell ist eine strikte Anleitung, wie Software erstellt werden soll. Es definiert die Rollen sämtlicher am Prozess beteiligten Personen, also wer wann welche Verantwortung hat, was macht und welche Artefakte erstellt, die eine genaue Form haben müssen. Sind agile Vorgehen gleichsam ein Prozessmodell? Sie haben bereits gesehen, dass agile Projekte nach einer »Methodik« arbeiten – sind sie aber auch ein umfassendes Prozessmodell?

Agilität bedeutet, bewusst und implizit mit Änderungen und Unsicherheiten umgehen, sich bei der Softwareerstellung auf das »Was« zu konzentrieren, weniger auf das »Wie«. Es werden Praktiken angewendet, die der eigenen Projektsituation angepasst werden sollen. Sicherlich gibt es bereits seit einiger Zeit Prozess-Rahmenwerke, auf die das ebenfalls zutrifft, z. B. den *Rational Unified Process* (RUP). Dennoch ist bei agiler Entwicklung der Schwerpunkt ein anderer: Die Werte, die im agilen Manifest artikuliert werden, stehen im Vordergrund. Es ist sicherlich möglich, auch nicht-agile Projekte in gewisser Weise agil voranzubringen bzw. einzelne Muster der agilen Welt anzuwenden. Andersherum gibt es auch vermeintlich agile Projekte, die unzählige agile Muster anwenden, aber in einer Atmosphäre arbeiten, in der agile Werte wie Respekt und Offenheit keine Bedeutung haben – solch ein Projekt mag auf dem Papier »agil« arbeiten, ist aber zum Scheitern verurteilt.

Agile Entwicklung ist eine Philosophie, eine Denkweise. Konkrete Anwendungen der agilen Welt, z. B. Extreme Programming oder Scrum, setzen diese Denkweise um. Man könnte diese Anwendungen als »Prozessmodelle« bezeichnen, aber jede Zuordnung zu einem starren Prozessmodell wird dem dynamischen, generischen Ansatz schwerlich gerecht. Womöglich wäre der Begriff »Rahmenwerk« angebrachter. Viele Anhänger der agilen Praktiken stehen solchen Formulierungen sehr skeptisch gegenüber und scheuen sogar den Begriff »Methodik«.

Einigen wir uns für den weiteren Verlauf darauf, dass agile Entwicklung als Ganzes eine Philosophie darstellt, und die konkreten Ausprägungen Methodiken und Rahmenwerke sind. Prozessmodelle sind sie nicht, hat der Begriff doch einen zu schwergewichtigen Nachklang. Wie auch immer wir es bezeichnen, für Projekte ist es wichtig, agil zu arbeiten.

Traditionelle Modelle

Bei traditionellen Projekten, allen voran dem *Wasserfallmodell*, durchläuft die Entwicklung in sequenziell vordefinierten Stufen (siehe Abbildung 3-1). So werden beim Wasserfallmodell die Schritte der Softwareerstellung wie Projektbegründung, Anforderungsanalyse, Design, Codierung und Test strikt getrennt und nacheinander ausgeführt. Ist man erst einmal auf einer Stufe angekommen, kann man nicht mehr (oder nur noch sehr eingeschränkt) auf eine vorherige zurück. Das Modell geht davon aus, dass die Anforderungen zu Beginn alle klar sind und sich im Laufe der Zeit kaum noch ändern. Der Projektfortschritt lässt sich durch lieferbare Artefakte messen, z. B. Anforderungsdokumente, Designdokumente und Testpläne. Einzelne Komponenten werden gegen Ende integriert und als Ganzes ausgeliefert.

Auch wenn das starre Gerüst der Abfolge der Entwicklung in Derivaten wie dem *Spiralmodell* zugunsten von mehr Flexibilität aufgeweicht ist, bleibt der Prozess dennoch schwergewichtig und prozess- und dokumentenzentriert. Das besonders in Deutschland verbreitete *V-Modell* bzw. *V-Modell XT*, das u. a. bei Softwareprojekten der öffentlichen Hand eingesetzt wird, weicht das starre Vorgehen ebenfalls auf. Hier werden die Entwicklung der fachlichen Funktionalität und das Testen der Software als zwei unabhängige Aktionsstränge gesehen. Obwohl das ein großer Vorteil gegenüber dem Wasserfallmodell ist, lässt es doch zahlreiche Eigenschaften agiler Projekte vermissen.

Agile Methoden auf der anderen Seite liefern in sehr kurzen Abschnitten durchgetestete und integrierte Software, die direkt produktiv einsetzbar ist. Kleine, greifbare Arbeitspakete werden fortlaufend entwickelt und zu einem Gesamtsystem integriert. In agilen Projekten werden die traditionellen Schritte Design, Implementierung, Test und Wartung in einem Schritt durchgeführt. Auch die Anforderungen werden stetig überprüft und der aktuellen Situation angepasst.

In konventionellen Projekten ist ein enormer Anstieg der Kosten über die Zeit erkennbar. Der eigentlichen Implementierung nachgelagerte Tests, Wartungsfälle der Produktivumgebung und zahlreiche auftretende Fehler lassen die Kosten exponentiell steigen. Die Kosten der Implementierung machen nur einen recht kleinen Prozentsatz an den

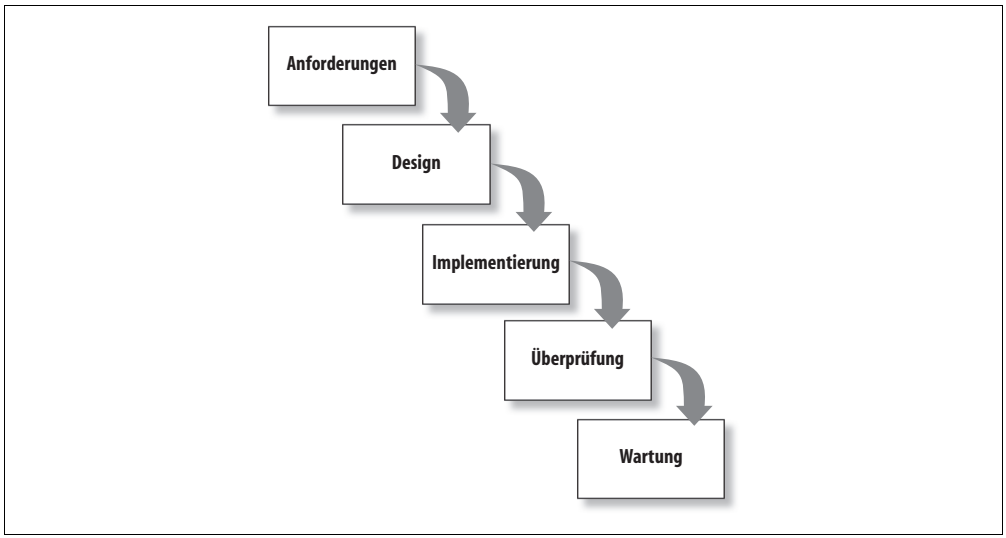


Abbildung 3-1: Die Traditionelle Entwicklung in Stufen

Gesamtkosten aus. Durch umfangreiche Planung zu Beginn versuchen die Projekte, die Unsicherheiten und die Kosten zu minimieren.

Agile Projekte gehen einen anderen Weg. Sie harmonisieren den Aufwand und somit die Kosten, die in die Software fließen (siehe Abbildung 3-2). Zu Beginn wird bewusst kein Wert darauf gelegt, alle Anforderungen umfassend zu erfassen. Agile Maßnahmen wie begleitende Tests, testgetriebene Entwicklung und häufige Integrationen verhindern Fehler, noch bevor sie entstehen. Agile Projekte haben keine ausführliche nachgelagerte Testphase. Diese Kosten werden zum großen Teil gespart bzw. auf frühere Phasen umgelegt. Befindet sich die Software in Produktion, ist sie im Optimalfall fehlerfrei. Nun auftretende Fehler zu lokalisieren und zu beheben, ist ausgesprochen teuer. Diese Kosten treten in agilen Projekten nicht auf.

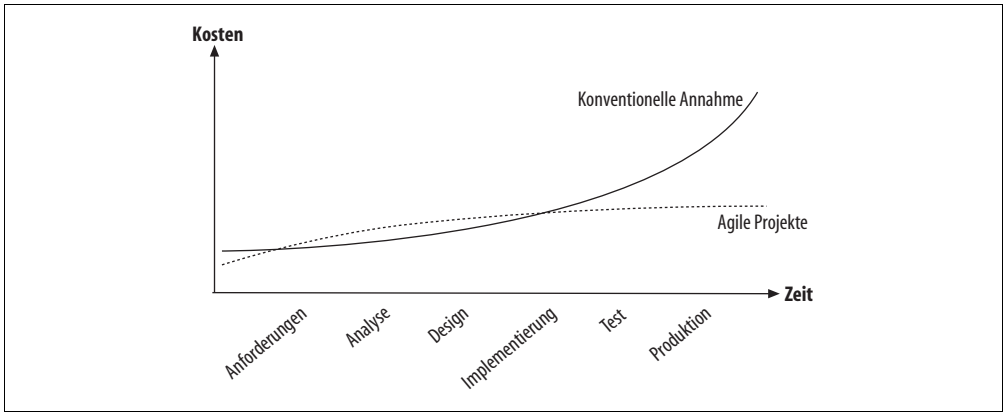


Abbildung 3-2: Die Kostenentwicklung bei traditionellen und agilen Projekten

Was agile Entwicklung nicht ist

Ich habe häufig erlebt, dass Chaos und Unwissenheit in einem Projekt als »agiles Vorgehen« bezeichnet wurden. Agile Entwicklung ist nicht gleichzusetzen mit Anarchie, in der jeder macht, was er will, und das entwickelt, was der Kunde gerade zuruft. Obwohl es stetigen Austausch gibt und Anforderungen nicht als ewig fix angesehen werden, sind die Inhalte einer einzelnen Iteration doch unveränderbar. Wie sonst könnte die Teamleistung (als Vergleich der geplanten mit der tatsächlich erbrachten Leistung) innerhalb der Iteration gemessen werden, wenn sich während der Iteration die Rahmenbedingungen – insbesondere die Anforderungen – ändern? Wenn Sie auf Zu- bzw. Anruf von Kunden neue Anforderungen umsetzen und alles Geplante über den Haufen werfen, dann ist das nicht-agil, sondern Chaos.

Wenn Sie ein Werteschema haben, das mit dem agilen Manifest übereinstimmt, wenn also der Mensch im Mittelpunkt steht, ständige Kommunikation stattfindet, Informationen offen zugänglich sind und dennoch ein klares Vorgehen mit einem hohem Maß an Disziplin existiert, dann arbeiten Sie vielleicht schon agil.

Ausprägungen agiler Entwicklung

Wie bereits gesagt wurde, ist der Begriff »agile Softwareentwicklung« ein Überbegriff, ein Mantel um eine Vielzahl einzelner Ansätze. Einige Ausprägungen sind erst nach der Formulierung des agilen Manifests entstanden. Die Mehrzahl aber existierte bereits vorher. Einzelne Einflüsse haben ihren Ursprung fernab von der Softwareentwicklung, nämlich in Fertigungsprozessen (z. B. das Lean Development). Im Folgenden führe ich in die bedeutendsten Ausprägungen und Implementierungen der agilen Philosophie ein, die für die späteren Kapitel wichtig sind. So wird das Kapitel Dimensionen agiler Entwicklung die einzelnen Ansätze aufnehmen und diskutieren, das Kapitel Die Durchführung zeigen, wie ein Projekt in der Praxis erfolgreich agil umgesetzt werden kann. In der Praxis werden häufig Mischformen angewendet bzw. Methodiken den individuellen Verhältnissen angepasst, was auch ausdrücklich erwünscht ist. Dieses Kapitel stellt das Grundgerüst der einzelnen Methoden so vor, wie sie »schulbuchmäßig« dokumentiert und von den Urvätern propagiert werden.

Scrum

Seit den 80er Jahren wurde diese Methodik hauptsächlich von Ken Schwaber und Jeff Sutherland im Rahmen objektorientierter Entwicklung entworfen. Der Begriff »Scrum« kommt aus dem Rugby und bezeichnet dort eine Absprache (also quasi ein Meeting) vor einem bestimmten Spielzug.

Scrum konzentriert sich auf die Facetten des (Projekt-)Managements und erstellt ein grobes organisatorisches Grundgerüst, das die eigentliche Entwicklung umhüllt (siehe Abbildung 3-3). Darüber, wie die technische Realisierung auszusehen hat, macht Scrum keine Aussagen.

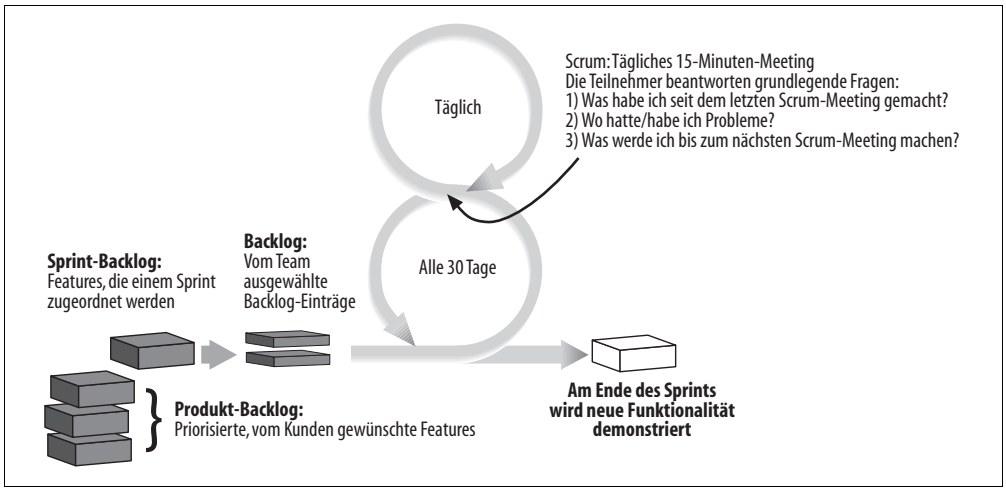


Abbildung 3-3: Die Softwareentwicklung mit Scrum

Scrum zeichnet sich durch eine durchstrukturierte zyklische (iterative/inkrementelle) Vorgehensweise aus. Die Entwicklung wird dazu in Iterationen von jeweils 30 Tagen unterteilt (30 aufeinander folgende Kalendertage), die *Sprints* genannt werden. Die Kundenanforderungen werden mit jeweiligen Aufwandsabschätzungen in einem *Product-Backlog* verwaltet. Je höher die Anforderung dort eingestuft ist, desto wichtiger ist sie und desto genauer sind ihre Aufwandsabschätzungen. Für das Product Backlog ist der *Product Owner* verantwortlich, derjenige, der an dem Projekt ein Nutzungsinteresse hat, im Allgemeinen also der Kunde.

Aus dieser hierarchisierten Liste von Aufgaben werden die in einer Iteration zu erledigenden Arbeitspakete vom Produktmanager ausgewählt und in ein *Sprint-Backlog* geschoben. Dazu wird ein *Sprint Planning Meeting* durchgeführt, das einen Tag dauert und in zwei Hälften aufgeteilt ist: In den ersten vier Stunden stellt der Product Owner die hoch eingestuften Einträge des Product-Backlog vor und diskutiert mit dem Team, wie viel davon in der kommenden Iteration umgesetzt werden kann. Im zweiten Teil des Tages erarbeitet das Team das Sprint-Backlog. Das Team ist in hohem Maß selbstorganisierend und interdisziplinär.

Im Sprint-Backlog sollten dann nur noch einzelne *Tasks* mit maximal 16 Stunden Arbeitsaufwand vorliegen. Das Sprint-Backlog wird kontinuierlich fortgeschrieben und stellt den aktuellen Stand dar, z. B. wer an einem Task arbeitet und wie viel Restzeit dafür verbleibt.

Bei Scrum werden tägliche Meetings, so genannte *Daily Scrums*, durchgeführt, die nicht länger als 15 Minuten dauern sollten. Dabei berichtet jeder Entwickler reihum von seiner Arbeit seit dem letzten Daily Scrum, z. B. über eventuelle Schwierigkeiten und was er bis zum nächsten Meeting am Folgetag umsetzen möchte.

Während der Iteration sollen die Entwickler ungestört arbeiten und der *Scrum-Master* über den reibungslosen Ablauf wachen. Der Scrum-Master kennt sich mit der Scrum-Methodik sehr gut aus und ist für ihre korrekte Anwendung verantwortlich. Nach der Iteration wird ein vom Scrum-Master geleitetes dreistündiges *Scrum Retrospective Meeting* abgehalten, in dem verbesserungsbedürftige Punkte identifiziert und besprochen werden. Ebenfalls am Ende der Iteration wird das Inkrement in einem *Sprint-Review* dem Product Owner vorgestellt.

Scrum sieht sich selbst als Mantel um andere, konkretere Softwareentwicklungspraktiken. Es kommt häufig vor, dass die Managementansätze von Scrum mit den technischeren Ansätzen des Extreme Programming kombiniert werden. Dort wo Scrum keine konkreten Aussagen über die Implementierung macht, werden Praktiken aus Extreme Programming herangezogen.

Scrum setzt den Fokus darauf, den Entwicklungsprozess durch Reduzierung des organisatorischen Overhead zu optimieren und durch die schnellen Inkremente nah am Markt sein und mit schnell wechselnden Anforderungen umgehen zu können. Alle 30 Tage werden nutzbare Inkremente geliefert, die eingesetzt werden können.

Trotz konkreter Vorgaben und eines disziplinierten Vorgehens (Meetings und Iterationen sind *time-boxed*, also in der Dauer festgeschrieben) besticht Scrum durch seine Einfachheit sowohl im eigentlichen Prozess als auch bei der Einführung: Scrum kann jederzeit eingeführt werden, also auch während ein Projekt läuft. Durch die allgemeine Art kann es auch dort direkt zum Einsatz kommen, wo schnell ein agiler Prozess eingeführt werden soll, man jedoch vor Praktiken wie Pair Programming zurückschreckt.

Extreme Programming

Extreme Programming (XP) wurde Ende der 90er Jahre im Wesentlichen von Kent Beck, Ron Jeffries und Ward Cunningham im Rahmen des *Chrysler Comprehensive Compensation System*-Projekts entwickelt. An dem Projekt wurde von 1995 bis 2000 gearbeitet. Es hat verschiedene alte Abrechnungssysteme bei Chrysler abgelöst.

XP ist einerseits eine Sammlung von *Best Practices*, andererseits eine Methodik. Die Gründer identifizierten für die Softwareerstellung elementare Bestandteile und wendeten sie exzessiv, eben *extrem*, an.

XP ist maßgeblich von Scrum inspiriert. Man könnte sagen, XP ist ein *Fork* von Scrum, also eine Abzweigung der Entwicklung in einen autonomen Bereich, der selbstständig weiterentwickelt wird. Im Gegensatz zu Scrum konzentriert sich XP hauptsächlich (aber nicht nur) auf Faktoren, die den eigentlichen Implementierungsprozess betreffen. Dabei basiert es grundlegend auf den agilen Werten Respekt, Mut, Kommunikation, Feedback und Einfachheit und interpretiert diese durch eigene Prinzipien und Praktiken.

Extreme Programming unterteilt die Aktivitäten in Releases, Iterationen und Tasks (siehe Abbildung 3-4). Ein *Release* ist eine in sich geschlossene Version der Software, die in dieser Zusammenstellung endgültig genutzt und dem Nutzer bereitgestellt werden soll. Das

Release beinhaltet mehrere kurze *Iterationen*. Diese stellen einen lauffähigen Teil der geplanten Funktionalität bereit. Bei der Planung von Releases und Iterationen wird mit *User Stories* gearbeitet: In kurzen, formlosen Sätzen wird eine Funktionalität beschrieben. User Stories werden mit *Story Points* abgeschätzt. Innerhalb der einzelnen Iterationen werden User Stories implementiert: Dazu zerteilt der Programmierer eine User Story in kleine technische *Tasks*. Der Durchsatz des Teams, also wie viele Story Points in einer Iteration implementiert werden können, wird als *Velocity* bezeichnet. Das ganze Vorgehen ist *time-boxed*: Die Laufzeiten von Releases und Iterationen sind streng fixiert. Ist eine Funktionalität bis zum Ende der Planungseinheit nicht fertig, wird sie nicht mit ausgeliefert. Es müssen auch immer *Tests* mitgeliefert werden, die alle erfolgreich laufen müssen. Es wird stetig Software geliefert: Nach jeder Iteration steht lauffähige Software bereit, so dass der aktuelle Projektstatus ersichtlich ist.

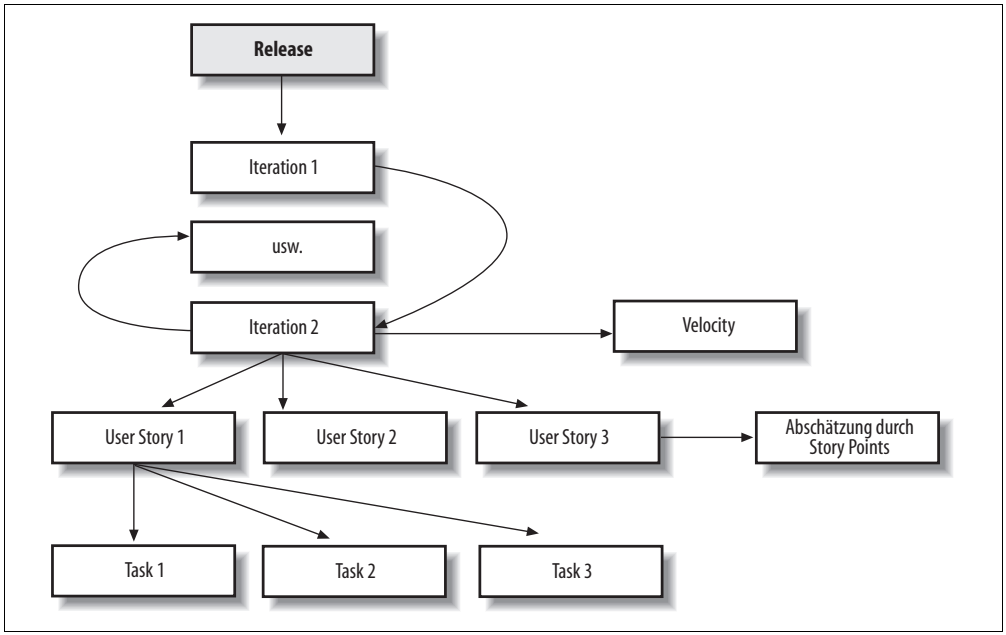


Abbildung 3-4: Extreme Programming

Traditionelles und Evolutionäres XP

XP wurde über das Buch *Extreme Programming – Embrace Change* im Jahr 2001 einem breiten Publikum bekannt gemacht. 2004 folgte dann die zweite Ausgabe des Buchs, die neue Impulse lieferte und viele neue Erkenntnisse vorstellte. Ich unterscheide hier gern traditionelle und evolutionäre Praktiken. Die evolutionären – also die in der zweiten Ausgabe dokumentierten – Praktiken nehmen verschiedene neue Erkenntnisse aus der jahrelangen Anwendung von XP auf. Sie verfeinern oder modifizieren die ursprünglichen Praktiken geringfügig und machen damit die Nutzung klarer und verständlicher.

Die zweite Ausgabe beschäftigt sich insbesondere auch vermehrt mit der Anwendung von XP in großen, verteilten Projekten. Die Arbeitsmethode *Metapher* stellte sich hingegen als schwer vermittelbar heraus und wurde aufgegeben. *Coding Standards* ihrerseits werden als selbstverständlich vorausgesetzt und nicht mehr explizit erwähnt.

Prinzipien

Die 14 Prinzipien von XP sind eine Brücke zwischen abstrakten Werten und den konkreten, anwendbaren Praktiken.

Menschlichkeit
: Software wird von Menschen für Menschen erstellt und ist letztendlich nur Mittel zum Zweck. Der Mensch steht im Mittelpunkt.

Beidseitiger Vorteil
: Alle Beteiligten müssen jederzeit an einen Strang ziehen. Grundsätzlich sind Win-win-Situationen anzustreben.

Selbstgleichheit
: Wenn etwas einmal funktioniert hat, nutzt man es wieder. In der Praxis bewährte Muster werden erneut eingesetzt und auch auf andere, vergleichbare Situationen übertragen.

Verbesserungen
: Der gesamte Prozess ist darauf ausgerichtet, dass zu Beginn nicht alles perfekt entwickelt oder entworfen werden kann. Stetige Refactorings verbessern Design und Code, Retrospektiven den Gesamtprozess.

Mannigfaltigkeit
: Menschen sind verschieden. Das Projekt hat die Chance, davon zu profitieren.

Reflexion
: Durch ständige Reflexion wird kontinuierlich Verbesserungspotenzial gefunden.

Fluss
: Das ganze Team, der ganze Prozess, befinden sich im Fluss. Es wird kontinuierlich Software geliefert. Das Projekt hat einen *Herzschlag*.

Gelegenheit
: Probleme werden als Chancen begriffen.

Redundanz
: Bewährte Vorgehensweisen werden wiederholt. Komplementäre, sich ergänzende Vorgehensweisen dienen demselben Zweck.

Fehlschlag
: Fehlschläge sind ganz normal, es gibt kein *Fingerzeigen*, stattdessen lernt das ganze Team aus Fehlern.

Qualität
: Qualität ist ein fester Bestandteil: Die Software muss jederzeit von hoher Qualität sein.

Kleine Schritte
: In einem iterativen, inkrementellen Vorgehen liefern viele kleine Schritte Sicherheit und bringen das Projekt kontinuierlich sichtbar voran (*baby steps*). Wer in kleinen Schritten läuft, stolpert seltener.

Akzeptierte Verantwortung
: Aufgaben werden nicht delegiert, sie werden bewusst ausgewählt und akzeptiert.

Die Praktiken lassen sich unterteilen in Primärpraktiken und Folgepraktiken.

Primärpraktiken

Diese Praktiken sind elementar für ein erfolgreiches XP-Projekt.

Beieinander Sitzen
: Das Team soll räumlich nah zusammen sitzen, im Optimalfall in einem Büro. Dies fördert die direkte Kommunikation.

Ganzes Team
: Das Team als Ganzes hat die gemeinschaftliche Verantwortung. Es arbeitet ohne starre Rollenverteilung zusammen. Die Mitglieder sind fachlich und technisch versiert. Bringen sich alle Experten der unterschiedlichen Disziplinen gleichermaßen ein, profitieren das Projekt und das ganze Team davon (*cross-functional teams*).

Informative Arbeitsumgebung
: Die Arbeitsumgebung (also die Büroräume und ihre Ausstattung) gibt jederzeit Aufschluss über den aktuellen Projektstand. Dies betrifft die Anforderungen, die Tests und den Projektfortschritt, deren aktueller Stand jederzeit sichtbar dokumentiert ist.

Energiegeladene Arbeit
: Die Teammitglieder sind voller Tatendrang. Diese hohe Motivation wird erreicht durch akzeptierte Verantwortung, kontinuierlichen sichtbaren Projektfortschritt, hohe Qualität und geregelte Arbeitszeiten.

Programmieren in Paaren (Pair Programming)
: Zwei Entwickler sitzen an einem Rechner. Der eine ist der Fahrer, der die Tastatur hat und aktiv entwickelt, der andere ist der Beifahrer, der sich ebenfalls aktiv einbringt. Die Rollen tauschen.

Geschichten
: Die Anforderungen werden in Geschichten, den User Stories, formuliert. Unterstützt vom Programmierer schreibt der Kunde selbst die Anforderungen auf Kärtchen und ordnet sie nach ihrer Priorität.

Wochenzyklus
: Eine Iteration soll in etwa einer Woche durchlaufen sein. Dies ist als grober Maßstab zu sehen, nicht als fixer Richtwert.

Quartalszyklus
: Ein Release dauert drei Monate und repräsentiert als Ergebnis ein in sich geschlossenes Produkt. Auch hier ist das Zeitintervall als Richtwert zu sehen.

Das Team

Das XP-Team setzt sich aus Experten der verschiedenen Disziplinen zusammen. Ein XP-Projekt kennt einen *Tester*, der beim Design und der Implementierung von übergreifenden Tests unterstützt. Er kann auch dem *Kunden* bei der Erstellung von Akzeptanztests helfen. *UI-Designer* konzentrieren sich auf die User Stories und ihre Umsetzung. Sie überprüfen die Bedeutung der Stories am laufenden System und identifizieren ggf. neue Stories, die für spätere Iterationen notwendig sind. *Architekten* kümmern sich um übergreifende, grundlegende Entscheidungen, übergreifende Tests und Refactorings. Ein *Project Manager* optimiert die Kommunikation innerhalb des Teams und zum Kunden hin. Er ist auch der *Tracker*, der über den Projektfortschritt wacht.

Planung ist bei XP eine fortlaufende Aktivität, keine Phase. *Produktmanager* vertreten das Produkt, sind also dafür verantwortlich. Sie wählen die zu entwickelnden User Stories aus. *Manager*, die disziplinarisch Vorgesetzten, sollen das XP-Team im Fluss halten, mit Informationen versehen, motivieren und einen arbeitsfreundlichen Mikrokosmos schaffen. *Nutzer*, der mit dem Produkt arbeitet, kann der Kunde sein, aber ebenso gut der Produktmanager.

Die *Entwickler* (im Buch mit *Programmierern* gleichgesetzt) implementieren die Tests und die Fachlichkeit. Sie unterstützen den Kunden bei der Erstellung und Bewertung der Stories und geben Aufwandsabschätzungen ab. Gewöhnlich vereint der Entwickler viele Rollen in sich, so ist er auch UI-Designer, Architekt usw. Eine starre Rollenverteilung existiert nicht, was gerade am Entwickler und seinen praktischen täglichen Aufgaben deutlich wird; häufig übernimmt er zusätzlich auch Projektmanagementaufgaben.

Freiraum

Die Entwickler benötigen Freiraum, um nicht dauerhaft ein zu hohes, erschöpfendes Tempo gehen zu müssen. In diesem Freiraum evaluieren sie neue Technologien und kümmern sich um projektrelevante Aufgaben, die bisher liegen geblieben sind. Der Freiraum kommt direkt oder indirekt dem Projekt zu Gute. Manche Unternehmen wie Google reservieren grundsätzlich einen Tag der Arbeitswoche dafür.

Zehn-Minuten-Build

Ein Build der Software, die Übersetzung der Klassen und Tests und deren Durchführung, soll nicht länger als zehn Minuten dauern. Teilt man große Systeme in Teilsysteme ein, ist diese Zeitspanne auch für komplexe Anwendungen realistisch. Laufen die Tests zu lange bzw. gibt es zu viele davon (kann es wirklich jemals zu viele Tests geben?), kann ein *Staging* eingeführt werden: In zehn Minuten wird ein *Smoke*-Test durchgeführt, der das Wesentliche testet. Die vollständigen Tests laufen erst später.

Fortlaufende Integration

Mehrfach täglich wird die Software automatisch oder manuell integriert. Die Entwickler führen ihre lokal getestete Software der Versionsverwaltung zu. Ein ausgewählter Programmierer stößt einen Build an: Quellcode und Tests werden übersetzt,

Tests durchgeführt, die Anwendung in eine Laufzeitumgebung überführt. Das kann auch automatisch geschehen, sogar so häufig, dass jede Änderung im zentralen Repository (jeder Entwickler-Commit) einen Build auslöst.

Testgestützte Entwicklung
: Tests steuern die Entwicklung. Bevor mit der Entwicklung der fachlichen Klassen begonnen wird, werden Tests geschrieben. Sie und die Komponenten, die die eigentliche Fachlichkeit enthalten, werden iterativ und inkrementell entworfen und implementiert. Die Tests werden dann automatisiert und verbessern so kontinuierlich das Design der fachlichen Klassen.

Refactoring

Was in der ersten Ausgabe des XP-Buchs noch explizit formuliert wurde, ist nun an vielen Stellen implizit enthalten: Refactoring. Dieses fortlaufende Redesign samt Optimierung von Architektur, Design und Code ist heute elementarer Bestandteil der Softwareentwicklung. Durch werkzeugunterstützte Restrukturierung wird das System stetig den aktuellen Anforderungen und Erkenntnissen angepasst. Durch den stetigen Review, beispielsweise durch Tests und Pair Programming werden Anti-Patterns fortlaufend behoben und die innere Struktur optimiert, ohne das äußere Verhalten (bei Klassen das *Public API*) zu verändern.

Folgepraktiken

Die Folgepraktiken setzen die Primärpraktiken voraus. Es sind im Einzelnen:

Inkrementeller Entwurf
: Während einer Iteration werden immer nur die Anforderungen umgesetzt, für die sich der Kunde für diese eine Iteration entschieden hat. Es wird eine möglichst einfache Lösung angestrebt und keine Technologie oder Funktionalität auf Vorrat entwickelt.

Echte Kundenbeteiligung
: Der Kunde ist räumlich in der Nähe der Entwicklung oder zumindest immer erreichbar. Er arbeitet aktiv mit, erstellt die User Stories, ordnet sie nach Prioritäten, wählt die für die Iteration durchzuführenden User Stories aus und erstellt Akzeptanztests.

Inkrementelles Deployment
: Deployment ist die Überführung der Software in eine Laufzeitumgebung, gewöhnlich die Produktivumgebung, aber auch eine Test- oder Integrationsumgebung. Frühzeitig arbeiten echte Nutzer mit der neuen Software, so gibt es früh Feedback und es kann schneller auf Anforderungsänderungen eingegangen werden. Der Return on Investment ist besser.

Teamkontinuität
Das Team ist in seiner Zusammenstellung fest und in seiner Zielausrichtung bzw. den Aktivitäten klar begrenzt. Das bedeutet dass die Entwickler nur an einem Projekt arbeiten, nicht an mehreren gleichzeitig. Die Fluktuation ist niedrig.

Schrumpfende Teams
Nach diversen Iterationen und Releases wird das Team immer produktiver. Die gleiche Anzahl von Personen erreicht einen höheren Durchsatz. Möglichst soll dann ein Entwickler aus dem Team herausgenommen und in ein anderes Team integriert werden, wo er als Coach dienen kann (gerade in einem noch nicht-agilen Team).

Ursachenanalyse
Ergeben sich Probleme während des Projekts – was ganz normal ist –, muss immer das erste ursächliche Problem identifiziert und gelöst werden.

Planungsspiel

In der ersten Version wurde XP noch explizit mit einem Planungsspiel assoziiert. Hier sitzen Kunde und Entwickler zusammen, um die User Stories zu verfassen, abzuschätzen und nach Priorität zu ordnen. Der Kunde erhält die Information, welche Stories wie viel Entwicklungsaufwand nach sich ziehen und wie viel Stories in der nächsten Iteration realisiert werden können. Wir werden uns noch intensiv damit befassen, wenn wir uns ein konkretes agiles Vorgehen anschauen.

Gemeinsamer Code
Das Team hat eine gemeinsame Verantwortlichkeit, den kompletten Quellcode gemeinschaftlich zu erstellen und weiterzuentwickeln. Niemand ist nur für eine Komponente verantwortlich.

Code und Test
Der Kunde zahlt für lauffähige Software, deshalb muss sich die Entwicklung auf die Erstellung von Quellcode und Tests konzentrieren.

Eine einzige Codebasis
Der Code wird an einer Stelle zentral gehalten und hier zentral weiterentwickelt.

Tägliches Deployment
Der Kunde soll täglich eine neue Version der Software erhalten.

Vertrag mit verhandelbarem Umfang
Festpreise sind für agile Projekte nicht die optimale Lösung. Vielmehr sollte das Budget fix sein und der zu entwickelnde Funktionsumfang variabel.

Bezahlung pro Benutzung
Eine weitere Möglichkeit, um vom starren Fixpreismodell wegzukommen, ist die Zahlung jeder Funktionalität pro Nutzung.

Industrial XP – IXP

Das aus XP entstandene *Industrial XP,* siehe *http://industrialxp.org/*, hat sich zum Ziel gesetzt, XP weiterzuentwickeln und insbesondere für große Organisationseinheiten weiter anzupassen. Es basiert auf den gleichen Werten wie XP und verfeinert die Praktiken und fügt neue hinzu, um insgesamt ein Rahmenwerk anzubieten, das noch genauer die in der Praxis häufig vernachlässigten Facetten der Softwareentwicklung aufgreift.

IXP nennt beispielsweise die Arbeitsmethode »Retrospektive« beim Namen: Die in sehr vielen agilen Projekten genutzte Praktik findet im herkömmlichen XP keine direkte Erwähnung. Ferner achtet IXP noch stärker auf das stetige Lernen und auf die Qualität. Es werden am Projektmanagement orientierte Praktiken wie *Test-driven Management* eingeführt. Der bei XP implizite Umgang mit dem Risiko wird von IXP mit *Continuous Risk-Management* explizit gemacht. Auch die *Coding Standards* erleben eine Wiedergeburt. *Refactorings* werden zu einer festen Praktik.

Ein neuer Trend wurde mit dem *Domain-driven Design* (DDD) als Praktik mit aufgenommen. Bei diesem Ansatz dreht es sich darum, das Design des Systems (also der ganzen Software) von der Fachlichkeit abhängig zu machen. Dies wurde auch schon früher mit einer saubereren Architektur angestrebt, in dem ein System in voneinander entkoppelten Schichten *gelayert* (in entkoppelte, in sich konsistente Schichten geschnitten) wird. Eine gängige Definition von *Domänenmodell* ist »ein Objektmodell der Domain, das sowohl das Verhalten wie auch die Daten abbildet«. Es ist also ein konzeptionelles Modell des Systems, das uns etwas über die Entitäten (einzelne Einheiten des Systems) und ihre Beziehungen untereinander erzählt. Es wird häufig angefertigt, um die Schlüsselkonzepte des Systems besser zu verstehen und ein einheitliches Verständnis und Vokabular zu schaffen.

Domain-driven Design begreift die Domain als Satz von Regeln und Interaktionen, die auch auf prozeduralen Code angewendet werden können. Der Code ist dabei ein Ausdruck der Domain. Ein weiterer Ausdruck wird durch UML-Artefakte bereitgestellt, die für wichtige Bereiche erstellt werden. Sie greifen einen Bereich nicht vollständig auf, sondern betonen ihn lediglich komplementär. UML-Diagramme sollen immer durch Fließtext erläutert und der Leser dadurch geführt werden. Auf der anderen Seite ist nicht empfehlenswert, zu einer Komponente Klassendiagramme erstellen, die alle Informationen über die Klasse enthalten: Der Quellcode selbst besitzt bereits diese Informationen. Die Klassendiagramme sollen diese Informationen ergänzen und aufbereiten, so dass wirklicher Mehrwert entsteht.

Das Domain Model ist das Ziel der Domain, unabhängig von der Implementierungssprache. Die Funktionalität ist dabei wichtiger als die Sprache. Die Architektur lässt sich in DDD besser als »strategisches Design« bezeichnen, da sie über lange Zeit fest bleibt. DDD stellt ein Rahmenwerk für Designentscheidungen und Patterns dar und liefert ein Vokabular, das von allen Beteiligten gleichermaßen genutzt werden kann.

Als Beispiel für nicht nach den Ansätzen des Domain-driven Design entworfene Domains sind die Datum- und Zeit-Klassen von Java zu nennen, zu denen es eine DDD-Alternative gibt, die als »Referenzimplementierung« für den neuen konzeptionellen Ansatz dient.

XP@Scrum

Wir haben bereits herausgearbeitet, dass Scrum einen Managementrahmen definiert, während XP sich primär um die technische Implementierung der Software kümmert. Entwickler des Scrum-Ansatzes propagieren verstärkt die Funktion von Scrum als *Management Wrapper* (Mantel) um XP. Scrum übernimmt dabei die Verantwortung für das Management des agilen Projekts, XP für die Entwicklungspraktiken (siehe Abbildung 3-5).

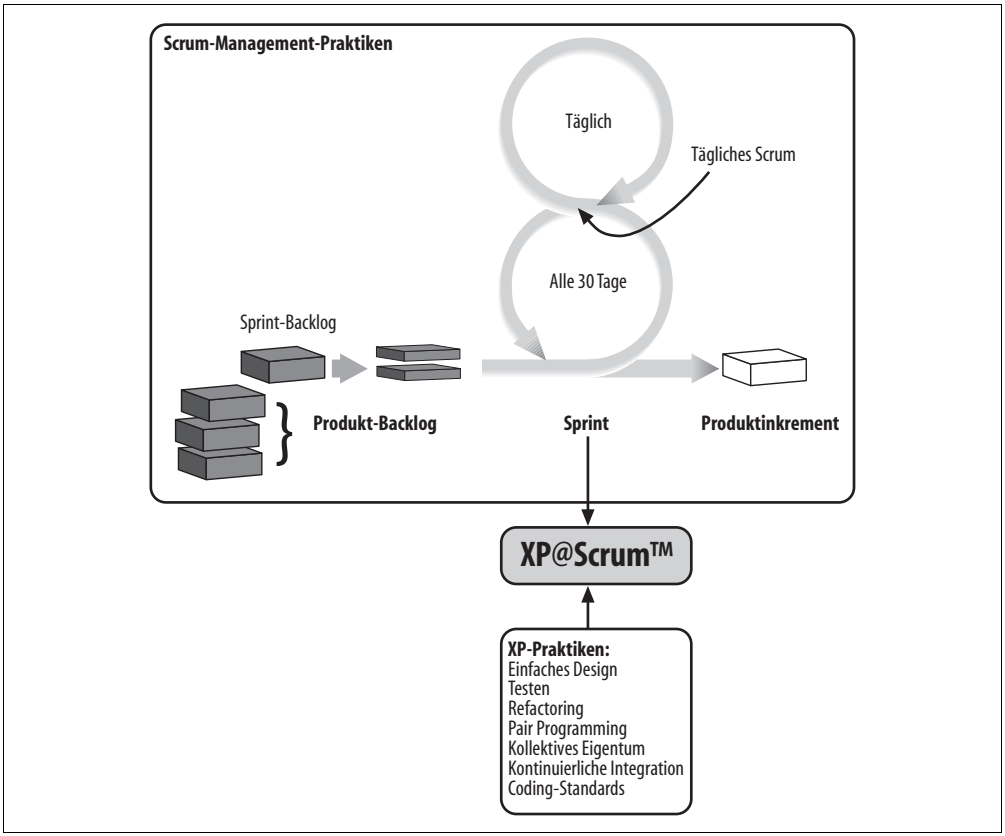


Abbildung 3-5: Ein kombinierter Ansatz: XP@Scrum

Von dieser von den Befürwortern von Scrum propagierten Mischung der Ansätze verspricht man sich eine noch bessere Anwendung von XP, eine bessere *Skalierung* in großen Projekten. Von einem praktischen Blickwinkel aus ist das allerdings kritisch zu sehen, weil es faktisch kaum ein reines XP-Projekt oder ein reines Scrum-Projekt gibt.

Häufig werden die Methoden den eigenen Projektanforderungen und der individuellen Organisationssituation angepasst. Das wird von XP auch ausdrücklich gefordert. Andererseits ist es so, dass XP eigene Managementpraktiken besitzt und klare Aussagen über Releases, Iterationen, ihre Länge etc. fällt. Vor diesem Hintergrund ist der Ansatz XP@Scrum also eher als Bestandsaufnahme zu sehen, wie XP sich in der täglichen Praxis bewährt und angepasst wird.

Lean Development

Lean Development (»schlanke Entwicklung«) hat seinen Ursprung in der Fertigungsindustrie, lässt sich aber ausgesprochen gut auf die Softwareentwicklung anwenden. Um sich gegen US-amerikanische Wettbewerber mit ihrer Massenproduktion durchzusetzen, legte Kiichiro Toyoda ab 1950 den Grundstein für eine Methode, die als *Toyota Production System* bekannt werden sollte. Das System zeichnet sich dadurch aus, dass es jede Form von überflüssiger Aktivität eliminiert. Seine zwei Stützen sind *Just-in-Time Flow* und *Stop-the-Line*.

Just-in-Time Flow

Projekte bzw. ganze Unternehmen sollen nicht mehr auf Vorrat produzieren. Alle Komponenten werden in kleinen Häppchen durch die Wertschöpfungskette geführt. Einzelne Knoten, organisatorische Stellen, in denen das Produkt verarbeitet wird, können sehr schnell umfunktioniert werden und auch andere Aufgaben übernehmen. Bei Just-in-Time wird die lokale Effizienz in den einzelnen Arbeitsschritten nicht mehr maximiert (für sich betrachtet könnte jeder Knoten ohne Just-in-Time mehr produzieren). Durch Just-in-Time wird aber der Gesamtausstoß maximiert.

Stop-the-Line

Das gesamte Produktionssystem wird kontinuierlich *implizit* auf Fehler überprüft. Es existiert keine Qualitätsstelle, die zu vorbestimmten Zeitpunkten die Qualität des erstellten Produkts überprüft. Ein kontinuierlich auf Fehler überprüftes Produkt benötigt das nicht. Doch wie geschieht diese kontinuierliche implizite Überprüfung? Wird irgendwo im Gesamtsystem ein Fehler gefunden, wird »die ganze Linie angehalten« und der Fehler direkt behoben. Insbesondere wird also keine neue Produktion durchgeführt, bis der Fehler behoben ist. Alle Mitarbeiter sind also jederzeit mit der Fehlerfindung bzw. der Überprüfung der Qualität beauftragt.

Ab 1990 wurde das Toyota Production System auch als *Lean Production* bekannt. Verschiedene Unternehmen haben die Anwendungen erprobt. Dabei wurde zunehmend festgestellt, dass das Gesamtsystem optimiert werden muss, also auch der Prozess inklusive der Zulieferer. Die Prinzipien der Lean Production wurden nicht nur auf die gesamte Wertschöpfungskette ausgeweitet, sondern auch auf einzelne Produktentwicklungen wie die Softwareentwicklung. *Lean Software Development* zeichnet sich durch sieben Prinzipien aus, die kaum Aussagen zu konkreten Methoden machen. Die Prinzipien sind eher

grundsätzlicher Natur. Im Gegensatz zu konkreten, in der Praxis anwendbaren Methoden, die die Prinzipien in bestimmten Situationen anwenden, ändern sich Prinzipien über die Zeit nicht. Es ist elementar, die Prinzipien zu verstehen, um Methoden wiederverwenden zu können.

Überflüssiges eliminieren

Um Überflüssiges (*Waste*) eliminieren zu können, ist der erste Schritt, das Überflüssige lokalisieren und identifizieren zu können. Dazu muss bekannt sein, was den eigentlichen Mehrwert ausmacht. Alle Aktivitäten, die keinen Kundenwert mit sich bringen, sind überflüssig. Oft ist beispielsweise halbfertige Arbeit Verschwendung. Auch Anforderungen können zu unnützem Aufwand werden: Werden Anforderungen lange vor der Umsetzung erfasst, werden sie sich bis zur Umsetzung ändern. Der anfangs betriebene Aufwand ist verschwendet.

Ein weiterer Fall für eine ungünstige Zeitspanne (Abstand zwischen Begründung und Durchführung) ist das Testen: Wird die Software erst nach der Implementierung getestet, handelt es sich dabei um Verschwendung. Der wesentliche Nutzen von Tests als treibender Kraft, die das Design verbessert, wird nicht erreicht.

Häufig sind diese einzelnen Verschwendungen nur Vorboten einer weit größeren Verschwendung, einer verzögerten Integration. Hält man sich vor Augen, dass gewöhnlich nur ca. 20% der Funktionalität einer Anwendung genutzt werden, können wir auch hier von einer Verschwendung reden. Die überflüssige Funktionalität wird dennoch mitgeschleift und verursacht Fehler, muss gewartet werden und erhöht die Komplexität der Anwendung unnötig.

Qualität implizit machen

Befassen sich bestimmte Personen mit qualitätssichernden Aktivitäten, ist das ein sicheres Zeichen, dass die Qualität nicht ausreichend im Gesamtprozess verankert ist. Sie wurde nicht implizit gemacht.

Inspektionen im Quellcode werden aber besser nicht *nach* der Lokalisierung eines Fehlers durchgeführt, sondern bereits *vorher*, um genau diese Fehler zu verhindern. Wenn das nicht immer gelingt, muss zumindest nach jedem noch so kleinen Schritt inspiziert werden. Auf diesem Weg werden Defekte direkt identifiziert, zeitnah zu ihrem Auftauchen. Außerdem werden sie direkt behoben. Dazu wird zunächst keine neue Funktionalität erstellt, es wird die »gesamte Linie gestoppt«.

In dieser extremen Form sind sogar Tracking-Systeme, die der Verwaltung von Fehlern dienen, überflüssig. Jeder Fehler wird sofort behoben, und somit ist ein Tracking-System Verschwendung, ist es doch eine Ansammlung von halbfertigen Aktivitäten. Hier spielt Test-driven Development eine große Rolle: Entwickler schreiben Unit Tests und Akzeptanztests, bevor sie den eigentlichen fachlichen Quellcode schreiben. Nachdem der neue Code in das System integriert ist, werden alle Tests durchlaufen. Wird dann ein Fehler festgestellt, wird erst der existente, fehlerhafte Quellcode angepasst, es wird keine neue Funktionalität geschrieben.

Auf diesem Weg wird die Zeit reduziert, die Fehler in einer eigens dafür herzurichtenden Software zu verwalten. Daneben erspart sich das Projekt zu einem späteren Zeitpunkt ein unproduktives, langes Suchen von Fehlern und Debugging (das Durchlaufen von Software und Quellcode, um Fehler zu finden, ist extrem kostenintensiv). Tests sind primär dazu da, Defekte zu verhindern, nicht um sie zu finden. Werden viele Fehler spät gefunden, so lässt das häufig auf einen Managementfehler schließen. Der Entwicklungsprozess scheint falsch abzulaufen. Es wurden offenbar keine Tests geschrieben, die die spät aufgetretenen Fehler früher hätten finden bzw. ganz vermeiden können. Test-driven Development und Continuous Integration sorgen dabei direkt von Anfang an dafür, dass es richtig gemacht wird und die Verschwendung minimiert wird: Es soll möglichst wenig Quellcode geschrieben werden. Anschließend wird so lange weitere Funktionalität ergänzt, bis der Nutzen weiterer Zusätze die Kosten nicht mehr rechtfertigt.

Wissen generieren

Wissen hat viele Ausprägungen. Eine davon sind Anforderungen. Softwareerstellung ist ein Wissen schaffender Prozess. Konstant Änderungen zu akzeptieren bedeutet, sich veränderten Situationen stellen können, sie zeitnah zu verstehen und darauf einzugehen. Dazu ist es notwendig, dem Kunden früh einige wenige Funktionen bereitzustellen, eine Auswertung zu ermöglichen und Feedback einzufordern. Nur so kann gelernt und Wissen generiert werden.

Um Wissen zu erlangen, sind tägliche Build-Prozesse und direktes Feedback von Integrationstests ebenso notwendig wie ein gut ausgebildetes Team, das die richtigen Entscheidungen fällt. Der gesamte Prozess muss permanent überdacht und optimiert werden. Probleme sind ganz normal, sie müssen aktiv angegangen und die ursprüngliche Ursache lokalisiert werden. Retrospektiven helfen dabei, Wissen zu erlangen und für zukünftige Iterationen anwendbar zu machen.

Vorhersagen, die lange im Voraus erstellt wurden, sind nicht aussagekräftig. Ist das System komplex, die vorherzusehende Zeit zu weit in der Zukunft und existieren zu viele unsichere Details in der komplexen Umgebung, können keine verlässlichen Aussagen getroffen werden. Das Wissen ist nicht verfügbar. Lean Development kehrt ab von Vorhersagen und reduziert stattdessen die Antwortzeit, um auf neue Erkenntnisse und Rahmenbedingungen direkt eingehen zu können.

Entscheidungen bis zum spätestmöglichen Punkt schieben

Es existiert immer, insbesondere auch in Softwareprojekten, ein Zeithorizont, innerhalb dessen eine Entscheidung getroffen werden muss. Wird die nötige Entscheidung erst im letzten Augenblick gefällt, hat der Entscheider den aktuellsten Informationsstand. Der letzte Augenblick ist der letzte zu verantwortende Augenblick, bevor die Entscheidung zu spät gefallen ist. Für die Softwareentwicklung bedeutet das, in frühen Projektphasen keine unnötigen Entscheidungen zu treffen, die nicht mehr umkehrbar sind. Wird zu Beginn eines Projekts eine Architektur erstellt, so wird der Architekt sich nicht gleich

anfangs unwiderruflich auf bestimmte starre Nutzungsszenarien fixieren. Er wird die Architektur flexibel halten bis zu dem Moment, in dem konkrete Anforderungen eine Umsetzung erfordern. Planung heißt dabei, sich nicht auf etwas festlegen zu müssen.

Liefern, so schnell es geht

Eine Software (in Inkrementen) schnell zu liefern hat zahlreiche Vorteile. Allen voran wird der Kunde zufrieden gestellt. Er kann die Software in der Ausbaustufe direkt nutzen. Auch frühes Bereitstellen auf einem Test- oder Integrationssystem ist für den Kunden sehr hilfreich. Nach einem täglichen Deployment kann der Nutzer zeitnah und sehr »nah an der Entwicklung« Feedback geben und Verbesserungsvorschläge machen. Jetzt, wo er die umgesetzte Anforderung am Rechner sieht, fallen ihm möglicherweise noch Punkte ein, die einer Verbesserung bedürfen. Es können ferner früher fachliche Defekte identifiziert werden.

Software muss so schnell bereitgestellt werden, dass der Kunde möglichst nicht mehr die Gelegenheit hat, seine Meinung bis zur Bereitstellung zu ändern. Die Anforderung wird umgesetzt und ist verfügbar.

Falle Release-Slot

Ich habe einmal in einer Unternehmung gearbeitet, in der Software nur selten produktiv gestellt werden durfte. Dieser Zeitpunkt war auch noch sehr eng umrissen (wenige Stunden bis ein Tag). Wurde es nicht geschafft, die Software in diesem kurzen Zeitfenster (*Slot*) fehlerfrei produktiv zu stellen, musste die Software überarbeitet werden und der Produktivgang wurde Monate später erneut probiert. Das war der Standardfall, denn die Abteilung, die die Software produktiv stellte, hatte kein sonderliches Interesse daran, durch die neue produktive Software mehr Arbeit zu haben. Die Zeit bis zum nächsten Slot war so groß, dass die Software auch funktionell erweitert werden musste (um neue Anforderungen aufzunehmen). So kam es vor, dass die Software auch beim nächsten Slot nicht produktiv gehen konnte. Das Ergebnis war, dass eine Unternehmung, für die eigentlich kurze Reaktionszeiten sehr wichtig gewesen wären, lange Zeit keine einzige neue Funktionalität bereitstellen konnte und sich dadurch selbst blockierte.

Leute respektieren

Alle Teammitglieder müssen das volle Vertrauen und den Respekt des Managers genießen. Stellen Sie sich einen Manager vor, der einem hochqualifizierten Entwickler eine Aufgabe gibt. Nach einer Woche ist die Aufgabe fast gelöst, aufgrund ihrer Komplexität ergeben sich allerdings weitere Aktivitäten, die der Vorgesetzte vorher nicht abgesehen hat. Der Manager versteht die bei der Umsetzung entstandenen Schwierigkeiten auch nicht im Detail, da er ja ganz andere Aufgaben als der Entwickler hat. Weil er mit dem Mehraufwand (im Vergleich zu seiner ursprünglichen Abschätzung) nicht einverstanden ist, fragt er den Entwickler nicht nach den Gründen, sondern setzt ihn unter Druck. Er

fragt ihn, warum die Arbeiten so »lange« dauern. Solche Aussagen sind häufig polemisch und suggerieren dem Entwickler, seine Arbeit sei nicht gut genug.

Es gibt häufig verschiedene Lösungen für ein Problem. Es kann also davon ausgegangen werden, dass sich der Entwickler bei dem von ihm eingeschlagenen Weg etwas gedacht hat. Kann darüber nicht sachlich diskutiert werden, muss sich der Entwickler ständig rechtfertigen – offensichtlich mangelt es an Vertrauen. Der Entwickler wird nicht respektiert. Negative Folgen derartiger Szenarien sind das Einstellen von Eigeninitiative, Demotivation und ein zeitlich verzögerter Entwicklungsprozess. Werden Mitarbeiter hingegen respektiert, werden Persönlichkeiten gefördert, die sich Problemen stellen – aktive, hochmotivierte Leute, gute Führungspersönlichkeiten, die exzellente Produkte erstellen. Respekt bedeutet aber auch, dass die Personen sich weitestgehend selbst organisieren und selbstständig ein gegebenes Ziel erreichen können. Nur frei denkende Personen finden in einem Prozess Optimierungspotenzial, das es ja immer gibt. Den Beteiligten Freiraum zu geben, Fehler zu tolerieren, zu erlauben, Entscheidungen selbst zu fällen, das alles zeugt von Respekt.

Respekt hat schließlich auch etwas mit Expertise zu tun: Möchte die Firma oder das Projekt gegen Konkurrenz bestehen, müssen die beteiligten Personen hochqualifiziert sein. Sie benötigen das Wissen, das sie von Wettbewerbern unterscheidet. Wird Expertise nur gekauft, so kann sie auch jeder andere kaufen. Respekt fördert *Wissensaufbau*.

Gegensätzliche Abteilungsinteressen

Ich habe einmal in einem Unternehmen gearbeitet, die für jede Person in jeder Abteilung eine leistungsabhängige Vergütung vorsah. Grundsätzlich ist gegen das Festlegen von Zielen und die Messung des Vorankommens nichts zu sagen – man nennt diese Vorgehensweise *Management by Objective* (MBO). Wird MBO allerdings unsensibel und falsch angewendet, ist es ein sicherer Teamkiller und kann sogar den Prozess der ganzen Unternehmung hemmen: Die Mitglieder der Entwicklungsabteilung hatten andere Performance-Ziele (Leistungsziele) als die Mitglieder des Betriebs, also der Abteilung, die die entwickelte Software schließlich übernimmt und produktiv stellt.

Das Entwicklerteam hatte das Ziel, Software auszuliefern, die über weniger Defekte verfügen sollte als bisher (die Software war relativ fehlerhaft, was aber zu einem großen Teil an Prozess- und Managementfehlern lag). Der Betrieb hatte auf der anderen Seite das Ziel, nur noch fehlerfreie Software produktiv zu stellen. Die Grundidee war gar nicht schlecht. Leider wurde dabei vergessen, dass eine Unternehmung aus Menschen besteht: Das Resultat war nämlich, dass der Betrieb kein einziges Release vom Entwicklerteam übernahm und produktiv stellte. Man sah sich die Releases gar nicht an und gab die Software wieder an die Entwicklermannschaft zurück. Da es sich zu dem Zeitpunkt um einen schwergewichtigen Prozess handelte, verbrachte die Software wieder einige Zeit im Entwicklungsteam, bis das Spiel von vorn begann.

→

Aus der Sicht der Betriebler eine einfache Lösung: Indem lange gar keine Version produktiv gestellt wurde, erreichten sie ihr Ziel: eine Produktivumgebung, die keine Fehler, aber leider auch keine neue Software produziert. Für die Unternehmung eine verheerende Entwicklung.

Was war aber die Konsequenz dieses Managementfehlers? Die Entwicklungsabteilung bekam nicht den vollen Bonus und es machte sich allgemeines Misstrauen breit, das in einem Kampf der Abteilungen endete, da es kein Vertrauen und keinen Respekt mehr gab.

Statt die einzelnen Abteilungen autonom zu betrachten, hätte man die Performance-Ziele erweitern müssen: Der Betrieb soll der Entwicklung aktiv helfen, die Qualität der Software zu erhöhen. Die anfängliche Ursache für ein Problem und eine übergreifende ganzheitliche Lösung müssen gefunden werden.

Dieses Beispiel lässt sich auch auf eine Vielzahl von individuellen Performance-Zielen einzelner Personen übertragen. Häufig stehen diese individuellen Ziele dem eigentlichen Projektziel im Weg, indem sie sich zu widersprechen scheinen. Insbesondere passiert dies, wenn zwischen den Entwicklern eine Konkurrenzsituation hergestellt wird, die einen offenen Umgang und Teamarbeit verhindert. Das Ganze zu optimieren bedeutet also immer das Ganze zu sehen: Die ganze Organisationseinheit, die ganze Unternehmung und die komplette Wertschöpfungskette.

Das Ganze optimieren

Lean Development handelt von Optimierungen. Im gesamten Entwicklungsprozess existiert die Atmosphäre, dass immer noch etwas verbessert werden kann. Diese ständigen Verbesserungen sind auch unter dem Begriff *Kaizen* bekannt, das ist ein japanisches Managementkonzept. Lean Development setzt sich dabei aber immer zum Ziel, das Ganze zu optimieren. Es existieren keine Mikrooptimierungen.

Feature-driven Development (FDD)

Dieses von Jeff DeLuca Ende der 90er Jahre entwickelte System, siehe *http://www.featuredrivendevelopment.com/* und *http://www.nebulon.com/fdd/index.html*, stellt das *Feature* in den Mittelpunkt, eine Funktionalität, die den Wert des Produkts steigern soll. Das inkrementelle, auf kurze Iterationen setzende Feature-driven Development (FDD) zeichnet sich durch ein modellbasiertes (architekturzentriertes) Vorgehen aus. Das bedeutet, dass, bevor mit der eigentlichen Implementierung begonnen wird, der Architekt die Software weitreichend entwirft und ein Grundgerüst konzipiert.

Die Entwicklungsschritte sind recht starr vorgegeben. Die Entwicklung sieht fünf Aktivitäten vor, von denen die ersten drei sequenziell durchlaufen werden. Über die beiden letzten wird für jedes Feature iteriert:

- Das übergreifende Gesamtmodell und die Architektur entwickeln
- Feature-Liste erstellen (Chefprogrammierer zerteilt das Gesamtsystem in einzelne Features)
- Features planen und nach Prioritäten ordnen
- Design der Features und Arbeitspakete definieren
- Entwicklung der Features (Unit Tests nicht zwingend notwendig)

Obwohl das Design und die Entwicklung iterativ verlaufen, ist FDD doch schwergewichtiger als andere agile Vorgehensweisen. So werden die drei ersten Aktivitäten einmalig und sequenziell durchlaufen. Ferner sieht FDD Rollen vor: Es gibt einen Chefprogrammierer, der zusammen mit den Entwicklern die Features mit Hilfe von UML-Diagrammen entwirft. Die Entwicklung der Features soll einen Zeitraum von zwei Implementierungswochen nicht überschreiten.

Den einzelnen Entwicklern werden Klassen zugewiesen. Sobald für die entwickelte Klasse die Komponententests erfolgreich durchlaufen sind und auch der Chefprogrammierer die Klasse begutachtet hat, kann das Feature dem aktuellen *Build* zugewiesen werden. Wöchentlich wird ein Release-Meeting abgehalten.

FDD zeichnet sich durch ein *Big Design Up Front* (BDUF), stetige Inspektionen, Rollenverteilung und eine disziplinierte Nutzung des Werkzeugs zum Konfigurationsmanagement aus.

Wichtig ist mir, hier darauf hinzuweisen, dass es viele unterschiedliche agile Ansätze gibt, deren Grad an Agilität unterschiedlich ist. Einige Methoden wie FDD durchlaufen zunächst auch schwergewichtigere Phasen, bevor es an die agile Entwicklung geht. Ebenso ist eine striktere Rollentrennung in manchen Methodiken durchaus üblich.

Zusammenfassung

Agile Vorgehensweisen versprechen Lösungen für Herausforderungen der Softwareentwicklung. Sie setzen den Menschen in den Mittelpunkt und streben mit Ehrlichkeit und gegenseitigem Respekt den maximalen gegenseitigen Vorteil an.

Es existerien verschiedenste agile Ansätze, die alle die Werte und Prinzipien des agilen Manifests implementieren. Wir haben uns Scrum und das populäre Extreme Programming sowie Mischformen angeschaut sowie mit dem Feature-driven Development eine Methode kennen gelernt, die schwergewichtiger ist als andere.

Steht mit Scrum eine Managementmethode zur Verfügung, konzentriert sich Extreme Programming primär auf Prinzipien und Praktiken, die von der eigentlichen Implementierung handeln. Allen gemein ist die Ausrichtung an elementaren, menschlichen Werten, ohne deren Verfolgung die erfolgreiche Entwicklung von Software unmöglich ist. Im Rahmen des Lean Development haben wir uns zahlreiche Prinzipien angeschaut, die bereits seit langer Zeit in der industriellen Massenfertigung Erfolg haben.

In der Praxis finden in der Regel Mischformen aus einzelnen Ansätzen Anwendung. Die einzelnen agilen Ansätze werden kombiniert und den individuellen Projekt- und Organisationsgegebenheiten angepasst.

Ein Beispiel einer individuellen Mischform, die populär geworden ist, ist der »Eclipse Way«, ein agiler Entwicklungsprozess aus der Eclipse Foundation. Eine weitere Mischform aus der »Web 2.0-Generation« ist der Weg der »37signals«, siehe *37signals.com*.

KAPITEL 4

Dimensionen agiler Entwicklung

In den letzten Kapiteln haben Sie die agile Entwicklung kennen gelernt, den Nutzen und Mehrwert agilen Vorgehens verstanden und einzelne seiner konkreten Ausprägungen betrachtet. In diesem Kapitel nun gehe ich auf die treibenden Momente für eine erfolgreiche Umsetzung ein. Die im weiteren Verlauf vorgestellten fünf zentralen Dimensionen sind *Motivation*, *Kommunikation*, *Team*, *Wissen* und *Zeit*. Diese Dimensionen, elementare Faktoren und Bausteine für die agile Entwicklung, greifen ineinander und konkretisieren die notwendigen Facetten einer erfolgreichen agilen Entwicklung. Am Ende des Kapitels wissen Sie, was ein Softwareprojekt erfolgreich macht und kennen die Zutaten für eine agile Entwicklung. Mit diesem Wissen können wir im nächsten Kapitel ein konkretes agiles Vorgehen und seinen zeitlichen Verlauf darstellen.

Motivation

Ein agiles Projekt steht und fällt mit der Motivation des Teams und jedes Einzelnen. Doch was ist Motivation, und was beeinflusst sie? Als Motivation wird gewöhnlich eine innere Kraft beschrieben, die als Antrieb dient, Dinge zu verrichten und Ziele zu erreichen. Der psychische Wille, die Bereitschaft, ein Ziel zu erreichen, ist elementar für die Erstellung von Software. Meine langjährigen Erfahrungen sagen mir, dass so gut wie jeder Entwickler grundsätzlich hochmotiviert ist, zumindest ist er es am Anfang eines jeden Projekts. Im Laufe der Projekte und der Lebensdauer einer Organisationseinheit wirken sehr viele Einflüsse auf den Einzelnen. Diese können helfen, die ursprüngliche Motivation zu halten, sie können die Motivation wenig motivierter Mitarbeiter erhöhen, sie können die Motivation allerdings auch reduzieren und bekämpfen. Der letzte Fall ist in der Praxis weit verbreitet.

Entwickler sind gewöhnlich die zentrale Basis, die Anforderungen umsetzen, neue Impulse liefern und Wissen speichern und weitergeben. Häufig übernimmt ein Team auch Führungsverantwortung, indem es einen kompetenten Manager unterstützt oder einen nicht kompetenten Manager ersetzt. Ein Top-down-Ansatz, in dem ausschließlich Führungskräfte suboptimale Prozesse und technische Probleme identifizieren und Lösungen finden, funktioniert nicht. Eine Organisation ist so gut wie die Entwickler, die

von innen heraus Impulse liefern. Sie allein wissen, wie genau gearbeitet wird (leider verfälscht sich das Bild der Realität häufig proportional zur Hierarchiestufe im Unternehmen aufwärts). Wenn die Motivation des Einzelnen sinkt, ist das Team nicht mehr agil. Das Team, die Führung und die ganze Unternehmung erhalten keine dringend notwendigen neuen Verbesserungsvorschläge von innen heraus.

Wir schauen uns nun an, was die Motivation in einem agilen Team genau ausmacht, wie die Entwickler als ausführende Arbeitskräfte motiviert werden und vor allem, was andere Entwickler und Manager (disziplinarische Manager und Projektmanager) falsch machen können, um ganz sicher zu gehen, dass die Motivation der Mitarbeiter bis gegen null sinkt und das ganze Unterfangen auf diesem Wege zu einem quälenden Horrorprojekt wird.

Dienst nach Vorschrift

Haben Sie schon einmal jemanden aus Ihrer direkten Umgebung sagen hören, er mache jetzt »Dienst nach Vorschrift«? Lief es Ihnen da auch eiskalt den Rücken herunter? Insgeheim wissen wir alle, dass ein starrer, durchorganisierter, nicht-agiler IT-Prozess nicht funktionieren kann. IT ist eine überaus komplexe Disziplin, in der keine zwei Projekte identisch sind. Es werden mitdenkende Individuen gebraucht. Softwareentwicklung ist zum größeren Teil eine Kunst, ein schöpferischer Akt, der von Motivation lebt, sie ist keine repetitive Ingenieurwissenschaft. Im Folgenden stelle ich die wichtigsten für die Motivation notwendigen Komponenten vor, wie sie als Bedürfnispyramide in Abbildung 4-1 dargestellt sind (vergleiche auch »Bedürfnispyramide nach Abraham Maslow«, *http://de.wikipedia.org/wiki/Abraham_Maslow*).

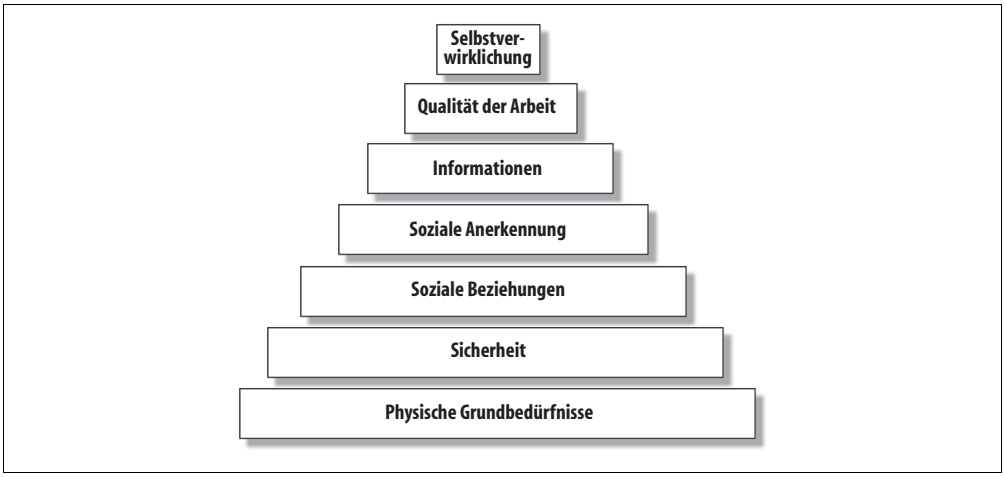


Abbildung 4-1: Die elementaren Bedürfnisse von Mitarbeitern

Die Bedürfnispyramide geht auf alle Faktoren ein, die für die Motivation des Einzelnen entscheidend sind und »Dienst nach Vorschrift« vermeiden helfen.

Physische Grundbedürfnisse

Die unterste Ebene wird durch die physischen Grundbedürfnisse definiert, beispielsweise Essen und Trinken. In einem agilen Projekt sind jederzeit die grundlegendsten menschlichen Bedürfnisse erfüllt. Das fängt damit an, dass die Unternehmung Getränke kostenlos bereitstellt. Die Entwickler haben jederzeit ausreichend zu trinken, insbesondere an heißen Tagen im Sommer. Für Hunger und Appetit gibt es eine Kantine, die eine gewisse Mindestqualität haben sollte. Ich kenne eine Menge Leute, die nur wegen einer ausgezeichneten Kantine den Arbeitgeber nicht wechseln, also unterschätzen Sie diesen Punkt nicht. In agilen Projekten ist auch jeder jederzeit in der Lage, seine Notdurft zu verrichten. Ferner müssen die Teammitglieder unter vertretbarem Aufwand die Arbeitsstelle erreichen können. Die Büros müssen so ausgestattet und ausgelastet sein, dass die dort sitzenden Personen unter angenehmen Umständen arbeiten können. Diese elementaren Dinge klingen selbstverständlich, sind es aber oft nicht.

Möglicherweise fühlen Sie sich durch diese Ausführungen bestätigt oder durch die Missstände in Ihrer Arbeitsumgebung angesprochen. Falls nicht, gebe ich Ihnen nun ein paar aus der Praxis stammende Beispiele, wie eine nicht-agile Arbeitsumgebung aussehen könnte: Es könnte damit anfangen, dass sich Ihr Arbeitsplatz im obersten Stockwerk befindet und der Aufzug jeden zweiten Tag kaputt ist. Den Treppengang zu nutzen ist nicht immer so einfach, da dort regelmäßig gearbeitet wird (»Achtung! Frisch gestrichen!«). Das Büro ist hoffnungslos überbesetzt, so dass Sie sich insbesondere im Sommer fragen: Vielleicht machen wir uns jetzt einen Kräuteraufguss? Getränke gibt es keine, was gerade im Sommer auf die Kondition geht. Oder: Es *werden* Getränke angeboten, aber sie kosten Geld. Sie dürfen während der Arbeitszeit nicht auf die Toilette. Wenn Sie es dann in der Pause tun, finden Sie eine zerstörte Toilettentür vor. Es existieren für die ganze Abteilung zwei Urinale, wobei der Abfluss des einen chronisch verstopft ist. Die Toiletten sind in einem jämmerlichen Zustand.

Was sagen uns solche Zustände? Die Teammitglieder können ihre physischen Grundbedürfnisse nicht befriedigen. Sie sind einerseits nicht in der Lage, körperliche Höchstleistungen zu erbringen – im Gegenteil, Sie werden vermutlich viele Fehler machen und recht unproduktiv arbeiten. Andererseits sind sie demotiviert. Das nicht-agile Unternehmen lässt die Arbeitsumgebung verkommen. Die nicht-agile Arbeitsumgebung ist aber die Umgebung, in der die Mitarbeiter den größten Teil ihrer Wachzeit verbringen. Wenn also der Mitarbeiter merkt, dass das Unternehmen für die Angestellten eine so geringe Wertschätzung hat, wie sollte dann der Mitarbeiter über das Unternehmen denken? Möglicherweise sagt die Firma: »Wir stellen die beste Software her! Wir kümmern uns um unsere Mitarbeiter!« Glauben das die Mitarbeiter auch? Oder ist die Software vielleicht genauso schlecht wie die Sanitärräume? Denkt das Unternehmen genauso über die Mitarbeiter, wie es über den chronisch defekten Aufzug denkt?

Agile Projekte sind eingebettet in ein Unternehmen, das sich um die Mitarbeiter kümmert. Die Mitarbeiter müssen sich über so elementare Dinge gar keine Gedanken machen. Alle Bedürfnisse werden ausreichend befriedigt. Neben der totalen Vernachlässigung der Bedürfnisse kann es auch genau andersherum sein: Viele meiner Bekannten

arbeiten in (nicht-agilen) Projekten und sind über den permanent miserablen Projektverlauf frustriert – sie suchen sich aber keine neue berufliche Herausforderung, weil die physischen Bedürfnisse übererfüllt werden. Keiner sollte nur wegen einer guten Kantine an seinem Arbeitsplatz hängen. Dann stimmt etwas nicht. In einem solchen Fall ist die größte Motivation, auf die Mittagspause zu warten.

Sicherheit

Ein weiteres elementares Bedürfnis jedes Mitarbeiters ist Sicherheit. Spürt der Einzelne in einem Projekt keine Sicherheit, so wird er defensiv agieren, er wird Dienst nach Vorschrift machen. Auch wenn ihm etwas Verbesserungsbedürftiges auffällt oder sogar ein Fehler in einer Anforderung oder in einer Software – fühlt er sich nicht sicher, wird er nicht darauf hinweisen. In einer Atmosphäre der Angst und des Misstrauens, in der Fehler personalisiert und ihrem Entdecker in die Schuhe geschoben werden, möchten die Entwickler nicht auffallen. Hier enden die Entwickler nicht selten in einem Daseinszustand, den ich einmal als »Kokon« beschrieben habe.

Atmosphäre der Angst

Oft ist es das Management, das für eine verängstigte, defensive Atmosphäre sorgt. In einer Kultur voller Angst kann aber kaum etwas Konstruktives entstehen (wie in einem Fußballteam, das nur das eigene Tor verteidigt, ohne zu versuchen, selbst ein Tor zu schießen). Auch Mitarbeiter, die gute Arbeit leisten und sich etwas vorgenommen haben, brechen ihre Vorhaben ab, weil sie davon ausgehen, dass sie sowieso verlieren oder die gute Lösung nicht honoriert bzw. als schlecht dargestellt wird. Unternehmen unter Angst wirken gelähmt und versäumen es, neue Märkte und Lösungen aktiv anzugehen. Sowohl ganze Firmen als auch Abteilungen oder Einzelpersonen verlieren so ihre Wettbewerbsfähigkeit. Angst, durch Erfolge aufzufallen und so den Argwohn der Kollegen oder des Managements auf sich zu ziehen (nicht selten sehen schlechte Manager in sehr guten Angestellten eine Konkurrenz und somit eine Gefahr), verhindert ebenfalls, dass Mitarbeiter und Unternehmen ihre volle Leistungsfähigkeit abrufen. Eine typische Aussage ist: »Ich mache das heimlich. Es ist zwar die beste Lösung, ich will aber nicht, dass der Chef davon erfährt.«

Agile Projekte streben eine andere, konstruktive Arbeitsatmosphäre an:

Keine Angst vor Fehlern

In agilen Projekten werden Fehler akzeptiert. Jeder macht Fehler und in jedem IT-Projekt werden Probleme auftreten. Nur wenn Sie den gleichen Fehler mehrfach machen oder Fehler verschwiegen oder Einzelpersonen für den Fehler an den Pranger gestellt werden, machen Sie etwas falsch. Sie müssen die Mitarbeiter durch eine gesunde Atmosphäre von gegenseitigem Respekt und Menschlichkeit dazu bewegen, Fehler aktiv anzugehen und sie beim Namen zu nennen.

Das Management als Puffer

Eine wunderbare Maßnahme, um diese Sicherheit zunichte zu machen, ist, wenn der Manager äußeren Druck, den er selbst von einer höheren Stelle in der Unternehmung bekommt, direkt ungefiltert an seine Entwickler weitergibt. Einen Entwickler unter Druck zu setzen, damit er einen Fehler schneller findet, ist das Schlechteste, was ein Manager machen kann. Ein Manager muss die Entwickler in ihrem empfindlichen Ökosystem vor äußeren Einflüssen schützen!

In der Ruhe liegt die Kraft

Der Entwickler sollte von seinem Manager auch in dieser Hinsicht geschützt werden, dass er sich nicht zu sehr in eine Aufgabe hineinsteigert. Wenn der Druck zu groß wird, weil z. B. ein nicht einhaltbarer Termin immer näher rückt, kann das auch zu Übermotivation führen. Die Entwickler wollen mehr arbeiten und versuchen, ohne Planung und Durchdenken der Konsequenzen nur schnell eine neue Version einer Software herauszubringen. Dann bekommen sie »endlich« das Release fertig, auf das alle warten (das tun häufig nur die Manager, die ja die unrealistische Terminvorgaben machen), was erst einmal vermeintliche Sicherheit gibt. Als Manager müssen Sie damit sehr behutsam umgehen. Nehmen Sie bewusst Druck raus. Schicken Sie z. B. die Mitarbeiter nach der regulären Arbeitszeit nach Hause. Wird die Übermotivation nicht gedrosselt, oder wird der Entwickler nicht vor äußeren Einflüssen geschützt, kippt die Motivation schnell in das genaue Gegenteil. Der Entwickler ist demotiviert. Schließlich haben Sie im schlimmsten Fall nur noch eine Batterie von Arbeitsbienen, die ohne freien Willen leer und verbraucht Dienst nach Vorschrift machen.

Es gibt ein Leben nach der Arbeit

Agile Prozesse sehen keine Überstunden vor. Das ist eine ehrenwerte Bestrebung, die aber in der Praxis gelegentlich nicht zu halten ist. Ein Dauerzustand sollten Überstunden jedoch nicht sein. Überarbeitete Mitarbeiter machen mehr Fehler, erst recht in Überstunden. Aber was noch schlimmer ist: Überstunden oder gar Wochenendarbeit (vielleicht sogar über einen längeren Zeitraum) sind immer ein Zeichen eines Planungsdefekts oder eines Managementfehlers. Es ist der Anfang vom Ende, wenn die Entwickler merken, dass sie die Fehler anderer ausbaden müssen. Leidet dadurch das Privatleben, macht der Mitarbeiter direkt seine Kollegen oder einfach das Projekt verantwortlich. Die Motivation ist dahin.

Das Auf und Ab gehört dazu

Ein wichtiger Punkt ist außerdem, dass der Entwickler nicht jeden Tag die gleiche Leistung bringen kann und auch die natürliche Motivation Schwankungen unterliegt. Auch wird die Motivation zu Beginn eines Projekts noch eine andere sein als gegen Abschluss. All diese Schwankungen sind ganz normal. Nur wenn die Motivationskurve dauerhaft nach unten zeigt, weil die Projektmitglieder keine Sicherheit bei Ihrer Arbeit verspüren, wird es bedenklich.

Soziale Beziehungen

Ein weiterer für die Motivation wichtiger Punkt sind die sozialen Beziehungen. Der Mensch hat ein natürliches Bedürfnis nach sozialen Kontakten. Werden sie ihm verwehrt, ist auch das ein gutes Mittel, um Motivation zu zerstören. Was spricht dagegen, dass sich zwei Entwickler in einem agilen Team bei einer Tasse Kaffee unterhalten? Das wird leider gern als unproduktive Pause wahrgenommen. Das Gegenteil ist der Fall: Eine gemeinsame Kaffeepause ist häufig ein Indiz, dass es im Team stimmt. Durch persönliche Gespräche wird Vertrauen aufgebaut. Wenn alle nur an der Konsole sitzen und jeder seinen eigenen Weg geht, dann ist das kein Team, geschweige denn ein agiles Team. Es existieren Unternehmen, die sich dieser Problematik leider gar nicht bewusst sind. Andere geben sehr viel Geld aus, um das Miteinander der Kollegen zu stärken. Warum Geld ausgeben, wenn man es auch (beinahe) kostenlos haben kann?

Ein Gespräch zwischen mehreren Kollegen auf der Arbeit ist ein soziales Ereignis, es wird so gut wie immer auch über die aktuellen Arbeitsaktivitäten gesprochen. Warum sollte man das unterbinden? Mir ist sogar ein Fall bekannt, bei dem ein Geburtstagskind die Abteilung zu einem Kuchen einladen wollte, um den fröhlichen Anlass im Kreise der Kollegen zu feiern. Das wurde verboten, da die Personen dann ja unproduktiv seien. Dass so eine Aktivität die Motivation steigert und auch bei solchen Anlässen über die Arbeit gesprochen wird, war den Vorgesetzten wohl nicht klar (abgesehen davon, dass sie dem Geburtstagskind gegenüber mangelnden Respekt zeigten). Wenn dann noch der Chef selbst regelmäßig selbst auf »fremden« Feierlichkeiten unterwegs ist und somit die Konsistenz seiner Aussagen untergräbt, ist das ein sicheres Mittel, die Mitarbeiter zu demotivieren. Wenn die Feierlichkeiten zu viel werden, dann teilen Sie die Abteilung auf. Übrigens sollten auch externe Mitarbeiter nicht rigoros ausgegrenzt werden. Sie sind, zumindest zeitweilig, Bestandteil des Teams und prägen es mit. Teilen Sie das Team nicht in zwei Klassen ein. Es ist besser, einen hochmotivierten Externen zu haben, der 15 Minuten einer Feierlichkeit beiwohnen darf, als einen unmotivierten, der in dieser Zeit allein im Büro sitzt, sich ausgegrenzt fühlt und Löcher in die Wand starrt. Das wird Auswirkungen auf seine Identifikation mit dem Projekt, auf seine Motivation und seine konkrete Arbeit haben.

Agile Prozesse setzen dagegen einen wesentlichen Schwerpunkt auf soziale Beziehungen. Das beginnt mit einem respektvollen, vertrauensvollen Umgang miteinander und geht bis hin zur Förderung von Face-to-Face-Gesprächen. Auch einzelne agile Praktiken leben von sozialen Beziehungen, wie es bei Pair-Programming-Sitzungen der Fall ist, in denen bewusst zu zweit an einem Rechner entwickelt wird. Agile Projekte sind wie große Familien – sie leben von Geborgenheit und zwischenmenschlichem Austausch.

Soziale Anerkennung

Motivation entsteht auch aus sozialer Anerkennung. Diese sollte in der täglichen Arbeit im Team erfahren werden. Wichtig ist, dass in agilen Projekten die Arbeit der Entwickler zur Kenntnis genommen und darüber gesprochen wird. Google beispielsweise stellt in

regelmäßigen Abständen die Erfolge der Projekte und ihrer Beteiligten vor. Der Einzelne fühlt sich dadurch belohnt: Er merkt, dass seine Leistung wahrgenommen und honoriert wird. Durch die erfolgreiche gemeinschaftliche Arbeit an einem Projekt schöpft das agile Team bereits aus der täglichen persönlichen Arbeit miteinander soziale Anerkennung.

Auf keinen Fall kann soziale Anerkennung auf Kosten anderer erreicht werden. Ich habe häufig gehört: »Entwickler A ist viel besser als B.« Das kann nicht funktionieren und drückt die Motivation. Agile Projekte stellen die Leistung des Entwicklers A heraus, ohne sie in einem Atemzug der Leistung von B gegenüberzustellen. *Alle* Entwickler in einem agilen Projekt leisten sehr gute Arbeit.

Ein Mittel zur (monetären) Anerkennung von Leistungen sind die Zielvereinbarungen. Wie wir bereits festgestellt haben, stehen diese Ziele allerdings häufig einer konstruktiven, offenen Arbeit im Team entgegen. Die Ziele zweier Parteien stehen sich nicht selten diametral gegenüber. Es muss vielmehr ein *Teamziel* herausgegeben werden, denn nur das ganze Team kann insgesamt erfolgreich sein.

Zielvereinbarungen sind normalerweise mit Bonusleistungen verbunden. Da in einer nicht-agilen Umgebung der Manager häufig gar nicht willens ist, die potenziellen Bonuszahlungen auszuzahlen, wird nach spätestens der zweiten Runde die Motivation des Entwicklers auf einer »Dienst nach Vorschrift«-Höhe angekommen sein. Leistungen, egal wie gut, werden nicht honoriert. Mir sind Fälle bekannt, in denen im Rahmen dieser regelmäßigen Gespräche, in denen die vergangene Leistung diskutiert werden, recht fadenscheinige Gründe für die Nichtzahlung eines Bonus angeführt werden. Bedenken Sie ferner, dass monetäre Anerkennung niemals soziale Anerkennung ersetzen kann.

Der Entwickler merkt sofort, ob ein Manager ehrlich und selbstkongruent ist und ihn respektiert. Wenn das so ist, ist er dem Manager loyal verbunden und geht mit ihm durch dick und dünn. Es entsteht ein für agile Teams so wichtiges Vertrauensverhältnis. Ist der Manager allerdings nicht ehrlich oder fehlt die Selbstkongruenz, weil der Manager z. B. den Entwicklern zu unterschiedlichen Zeiten widersprüchliche Dinge erzählt, so wird die Motivation leiden. Nimmt der Manager während eines Gesprächs mit dem Entwickler Telefonanrufe entgegen, unterbricht es, wenn jemand anderes etwas von ihm will, oder lässt er den Entwickler warten, so zeugt das von mangelndem Respekt und fehlender Anerkennung.

In agilen Teams sind grundsätzlich Win-win-Situationen anzustreben. Jeder muss mit seiner eigenen Persönlichkeit und seinen individuellen Wünschen ernstgenommen werden. Fragen Sie Kollegen und folgen Sie ihren Ratschlägen. Vertrauen Sie anderen.

Informationen

Agile Projekte leben davon, dass notwendige Informationen jederzeit zur Verfügung stehen. Ich habe es häufig erlebt, dass in nicht-agilen Projekten wichtige Informationen nicht rechtzeitig oder gar nicht weitergeleitet wurden. Manager erfahren in der Regel viel früher von Organisations- oder Projektänderungen und sonstigen Entwicklungen als die

Entwickler. Manche Informationen dürfen sie vielleicht nicht weitergeben oder zögern es mit gutem Grund hinaus, gewöhnlich versäumen sie aber einfach nur, das Team in Kenntnis zu setzen. Schlimm ist, wenn der Manager denkt, die Entwickler bräuchten etwas nicht zu wissen oder sollten es nicht. Oft werden Informationen aus machtpolitischen Erwägungen zurückgehalten, denn mehr zu wissen als andere bedeutet Macht. Wer aber hilfreiche Informationen nicht bekommt, wird das als frustrierend und demotivierend empfinden.

In agilen Projekten werden Informationen an alle Beteiligten weitergeleitet. Durch aushängende Projektfortschrittsberichte am Arbeitsplatz, tägliche Stand-up-Meetings, kontinuierliche Akzeptanztests und regelmäßige Integrationen sind alle Projektteilnehmer auf dem Laufenden und wissen, wo das Projekt gerade steht. Jeder Entwickler ist gleichzeitig Datenbankexperte, Programmierer und Architekt und arbeitet an der ganzen Codebasis, für die er sich so auch komplett verantwortlich fühlt. Es gibt keine Wissensmonopole: Alle Entwickler haben Informationen darüber, wie es in allen Komponenten der Software aussieht. Die Entwickler tauchen sich nicht nur untereinander aus, sondern regelmäßig auch mit dem Kunden.

Der Manager kümmert sich nicht um technische Detailfragen, denn dafür hat er seine Spezialisten. Er sorgt dafür, dass alle Teammitglieder jederzeit mit den Informationen versorgt sind, die sie benötigen. Die Entwickler werden zusätzlich motiviert, wenn sie nicht nur mit Informationen versorgt werden, sondern auch mitbekommen, dass es dem Manager wichtig ist, ihnen diese Informationen zu besorgen, und dass es ihn sogar Aufwand kostet.

Qualität der Arbeit

Was die Qualität der Arbeit angeht, fallen mehrere Ebenen zusammen. Der Mitarbeiter will möglichst gute Arbeit leisten. Das spiegelt sich in der Qualität der Software wieder. Jeder möchte stolz auf sein Produkt sein. Andersherum hat es direkte Auswirkungen auf die Motivation der Mitarbeiter, wenn die Qualität der Software nicht optimal ist. Darüber hinaus besitzt die Qualität des Entwicklungsprozesses ebenfalls eine »Qualität«, die Einfluss auf die Motivation hat. Beispielsweise verschiebt niemand gern Termine. Auch was den Kunden angeht, ist Qualität wichtiger, auch die angestellten Entwickler wollen, dass der Kunde zufrieden ist. In agilen Projekten achtet man auf den Zusammenhang zwischen der Qualität der Arbeit (also der Software, des Prozesses und der Rahmenbedingungen) und versucht, sie zu maximieren, um dadurch die Motivation der Entwickler zu optimieren.

Ein Entwickler möchte dazu immer unter den besten Bedingungen arbeiten, mit den besten Mitteln und Werkzeugen, und die besten fremden Softwarekomponenten einbinden, um perfekte Software zu erstellen. Für den Entwickler ist es demotivierend, wenn ihm eine gute Arbeitsausstattung verwehrt wird.

Das magische Viereck der Softwareentwicklung setzt Qualität, Zeit, Kosten und Funktionsumfang in Beziehung (siehe Abbildung 4-2). Der Kunde hat die Möglichkeit, die

Form des Rechtecks zu bestimmen, seine Fläche bleibt allerdings immer gleich, egal an welcher Ecke er zieht. In nicht-agilen Softwareprojekten ist häufig zu beobachten, dass dem Kunden ein abenteuerlicher Termin zugesichert wird, zu dem eine bestimmte Funktionalität bereitgestellt werden soll.

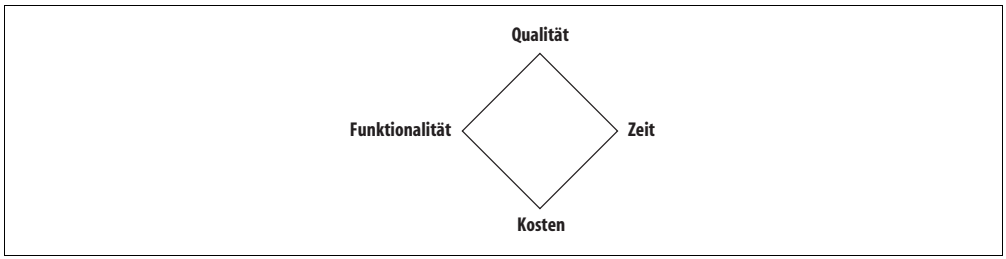


Abbildung 4-2: Das magische Viereck der Softwareentwicklung

Die Kosten für diese Lieferung sollten in der Regel fest sein. Es sei denn, es wird durch Einbringen von weiteren Ressourcen versucht, die sportliche Terminabschätzung zu halten. Zusätzliche Entwickler in ein Projekt zu positionieren wird die Projektsituation allerdings weiter verschlimmern. Die neuen Kollegen müssen schließlich zunächst eingearbeitet werden, was die Projektproduktivität der alten Kollegen reduziert.

Ein agiles Projekt wird aufgrund des kontinuierlichen Austauschs, der stetigen Lieferung von Software durch Time-Boxing und der offenen, respektvollen Kommunikation der Aufwände untereinander und mit dem Kunden in eine solche Situation nicht kommen.

Wenn in einem agilen Projekt während einer Iteration festgestellt wird, dass eine gewünschte Funktionalität (und die entsprechenden Tests) nicht fehlerfrei realisiert werden können, wird die Funktionalität aus der Iteration herausgenommen. Der Kunde entscheidet bei der Planung der nächsten Iteration mit den dann vorliegenden Erkenntnissen erneut, wie zu verfahren ist und was nun die User Stories sind, die die höchste Priorität haben und umgesetzt werden sollen.

Bei traditionellen Projekten hat man gewöhnlich die Einstellung, dass die »versprochene« Funktionalität zum zugesagten Liefertermin auch geliefert werden muss, egal wie. Der Umfang der Funktionalität bleibt konstant – seine Reduzierung ist häufig mit einem Imageschaden verbunden: Da das Entwicklungsteam seine Versprechung nicht einhalten konnte, ist es vermeintlich »inkompetent«. Welche Ecke des Rechtecks ist für traditionelle Projekte am ehesten variabel? Die Qualität. Es wird auf Komponententests komplett verzichtet, da »dafür die Zeit nicht da ist«. Es wird an allen Ecken gepfuscht, nur um zum fixierten Zeitpunkt mit der Funktionalität produktiv gehen zu können.

Dieses Vorgehen ist für die Software verheerend. Sie hat eine schlechte innere Struktur und viele Fehler. Der Kunde ist unzufrieden, da er eine Software von geringer Qualität bekommt. Stellen Sie sich vor, Sie bekommen ein neues Auto. Sie freuen sich, dass es zum vereinbarten Termin geliefert wird. Sie gehen zum Autohändler, der stolz auf die pünktliche Lieferung hinweist. Sie setzen sich ins Auto und stellen fest, dass das Lenkrad fehlt. In der Hektik der ausgelasteten Auftragsbücher wurde dessen Einbau in der Fabrik

vergessen. Läuft es in der Praxis so ab? Nein, Sie bekommen die Information, dass das Auto erst in zwei Monaten lieferbar ist. Das ist zwar ärgerlich, aber der einzig vertretbare Weg, um den Kunden tatsächlich zufriedenzustellen. Schließlich bleibt bei dem nicht-agilen Team noch die Motivation der Entwickler auf der Strecke: Sie haben durch zahlreiche Überstunden kurz vor dem Lieferzeitpunkt der Software alles gegeben, und doch ist die Software fehlerhaft. Sie bekommen die Vorwürfe des Kunden zu hören, der mit dem Produkt nicht zufrieden ist.

In agilen Projekten ist Qualität stets inhärenter Bestandteil des ganzen Prozesses und der erstellten Software. Über ihre Existenz kann nicht diskutiert werden! Die Software enthält immer Komponententests, die alle erfolgreich durchlaufen werden müssen. Schlägt zum Abschluss der Iteration ein Komponententest fehl, so wird die komplette zugrunde liegende Funktionalität nicht ausgeliefert. Ein »bisschen« Qualität gibt es in agilen Projekten nicht.

Zur Qualität gehört auch die verlässliche Aussage von Projektplänen. Agile Projekte sind immer Time-Boxed und harmonisieren die Arbeitsbelastung. Wenn nicht-agile Projekte die Arbeitsbelastung dauerhaft hoch halten und kein Ende nehmen, aber immer vom kurz bevorstehenden Ende geredet wird, ist das demotivierend. Das Team kann nicht dauerhaft sprinten. Eine Urlaubssperre für den kompletten Sommer, weil ein Projekt beendet werden muss, zeugt nicht von guter Organisation.

Agile Entwickler wollen nicht nur eine optimale Arbeitsausstattung, sie wollen auch das Gefühl haben, etwas Besonderes zu leisten und eine besonders gute Software zu schaffen. Nur so kann die Motivation des Teams hoch gehalten werden.

Death Sprint und Death March in traditionellen Projekten

Ein »Death Sprint«-Projekt ist ein Projekt, dem ein überhitzter Projektplan zugrunde liegt. Software wird iterativ bereitgestellt, allerdings in einer viel zu kurzen Zeitspanne. Nach außen sieht das Projekt zunächst sehr erfolgreich aus: Immer wieder werden neue Versionen mit neuen Eigenschaften abgeschlossen. Allerdings leidet die Qualität des Produkts sowohl technisch als auch nach außen sichtbar, was allerdings nur der Entwickler erkennt. Die Qualität nimmt mit jeder »erfolgreichen« neuen Iteration ab. Es werden keine Refactorings durchgeführt. Die Arbeitsbelastung ist nicht harmonisiert.

Ein »Death March«-Projekt dagegen zieht sich ewig hin. Im Optimalfall werden zwar Vorabversionen bereitgestellt, sie sind aber von schlechter Qualität. Der Misserfolg ist objektiv sichtbar. Es können keine Milestones gehalten werden bzw. es existieren gar keine. Die schlimmstmögliche Konsequenz daraus ist, dass das Projekt kein Projekt mehr ist, sondern nur eine zeitlich nicht abgeschlossene Aneinanderreihung von Aktivitäten. Es fehlen konkrete Zusagen für Termine und Lieferung von Eigenschaften, was auch bewusst in Kauf genommen werden kann, um von Defiziten in der Organisation abzulenken und Entwicklungen zu verschleppen (und so lange an etwas entwickeln, bis eine nicht genau spezifizierte Eigenschaft in irgendeiner Form subjektiv funktioniert).

→

Wenn es weder Anforderungs- noch Änderungsmanagement gibt und das Projekt keins mehr ist, schlenkert das Entwicklungsprodukt orientierungslos herum und seine Qualität nimmt stetig ab. Das kann interessanterweise auch mit einem Death Sprint kombiniert werden, um nach außen von Planungslosigkeit und Defiziten im Inneren abzulenken. Es wird dann Funktionalität als neu bereitgestellt, die schon lange existiert, oder es gibt keine Kontrollinstanz, die Notwendigkeit, Relevanz, Form und Korrektheit von bereitgestellter Funktionalität bewertet – »Neuanforderungen von gestern sind die Fehler von morgen«. Das kommt häufig vor, wenn es keine Stakeholder gibt, die Interesse am Produkt haben, oder wenn der in das Produkt einfließende Aufwand oder sogar das ganze Produkt letztendlich keine Bedeutung hat. In diesem Fall beschäftigt sich die Entwicklungsabteilung nicht selten (mit sich) selbst.

Ein Projekt zeichnet sich unter anderem dadurch aus, dass es zeitlich beschränkt ist. In beiden Fällen (Death Sprint und Death March) wird kein agiles Vorgehen praktiziert, insbesondere fehlt Time-Boxing.

Selbstverwirklichung

In agile Projekte bringt sich jeder Entwickler aktiv ein: Er prägt das Projekt maßgeblich und identifiziert sich mit den Aufgaben. Sich auf diese Weise selbst verwirklichen zu können, ist ebenfalls ein hoher Motivationsanreiz.

Verantwortung übernehmen

Wir haben bereits erläutert, was es für einen Unterschied ausmacht, ob ein Entwickler ein Arbeitspaket selbst auswählt und Verantwortung akzeptiert oder ihm eine Aufgabe lediglich aufgetragen wird. In agilen Projekten ist der Entwickler im gesamten Software-Lebenszyklus aktiv gefordert. Er erstellt mit dem Kunden zusammen die User Stories und nimmt nicht einfach nur die Anforderungen des Kunden an. Vom Beginn des agilen Projekts bis zum Ende ist er eingebunden und kann die Software wachsen sehen und bei der Entstehung begleiten. Das fördert die Identifikation ungemein.

Ergebnisse sehen

Auch die kurzen Iterationszyklen helfen dem Entwickler, Selbstverwirklichung zu finden. Er muss sehen können, wie das Projekt, an dem er mitgearbeitet hat, abgeschlossen wird und die Software produktiv geht. In agilen Projekten hat er das in einem kurzen Wochenrhythmus. Nach Abschluss der Iteration darf dann auch gern etwas gefeiert werden (auch wenn das Feiern in agilen Projekten manchmal zur Gewohnheit wird). Es ist etwas sehr Gutes geglückt: Die Software ist produktiv, das Team hat bewiesen, dass es wieder einmal erfolgreich zusammengearbeitet hat, und es kann sich neuen Aufgaben stellen. Das ist sehr motivierend.

Abwechslung

Nicht nur die prägende Teilnahme der agilen Entwickler am gesamten Softwarezyklus erhöht die Motivation, auch die fehlende Rollentrennung in der Implementierungsphase hilft bei der Selbstverwirklichung: Die Entwickler bringen sich ein und helfen den Kollegen da, wo es gerade notwendig ist. Das abwechslungsreiche Aufgabengebiet ist für den Entwickler sehr spannend. Er bringt sich in der ganzen Tiefe und Breite der technischen Aufgabenstellung ein.

Freiräume

Selbstverwirklichung ist nicht nur mit Entfaltungsmöglichkeiten verbunden, sondern auch mit Freiraum. Wenn Sie Manager sind, gönnen Sie Ihren Mitarbeitern eine gewisse Zeit, um Dinge zu tun, die auf den ersten Blick vielleicht nicht in direkter Beziehung zum aktuellen Projekt stehen. In den allermeisten Fällen werden die Entwickler ihren Freiraum so nutzen, dass auch das Team oder das Projekt etwas davon hat, wenn das auch nicht direkt sichtbar sein mag: So kann z. B. der Erwerb neuen Wissens in zukünftigen Situationen eingebracht werden. Die Mitarbeiter entwickelt sich weiter, sind motiviert und generieren mit großer Wahrscheinlichkeit einen bemerkenswerten Nutzen für da Unternehmen.

Abhängig von der Atmosphäre in einem nicht-agilen Team haben Entwickler möglicherweise nicht immer den Mut, auf dringend notwendige Aktivitäten hinzuweisen. Sie gehen ihnen dann allein nach, wenn Sie den Freiraum dafür haben. Denn: Grundsätzlich will der Entwickler mit allen Mitteln das Projekt erfolgreich beenden und arbeitet teilweise sogar heimlich für diesen Erfolg. Durch Erfolg kann er sich selbst verwirklichen.

Viele Entwickler, die ich kennengelernt habe, aktualisieren in kurzen Verschnaufpausen ihres nicht-agilen Projekts ihren Entwicklungsrechner, installieren sich ein neues Werkzeug oder führen dringend notwendige Refactorings an der Software durch. Schade, wenn solche wichtigen Aktivitäten heimlich stattfinden müssen. In agilen Projekten sind das ganz normale Aktivitäten, die, wenn sie gerade notwendig sind, zeitnah offiziell angegangen werden.

Eine freie Zeiteinteilung und ausreichend Pufferzeiten sind auch notwendig, um über eine Aufgabe wirklich ernsthaft nachzudenken. Und sie erhöhen die Flexibilität: In einem agilen Projekt kann das Team jederzeit auf neue Situationen eingehen und zeitnah handeln, was in einem nicht-agilen Projekt nicht möglich ist.

Neun Monate

Motivation ist elementar für den Erfolg eines Projekts und die nachhaltige Wettbewerbsfähigkeit des Unternehmens. Die Beziehung zwischen Manager und Entwickler ist ein Geben und Nehmen. Kritisch wird es, wenn der agile Manager nur nimmt, aber nichts gibt, wenn es nur noch Restriktionen gibt und keinen Freiraum und keine Mitbestim-

mung mehr. Kann sich der Entwickler mit seinem Arbeitsplatzrechner ein kleines Reich aufbauen und beispielsweise frei entscheiden, welches Betriebssystem und welche Werkzeuge er nutzt, hat der Mitarbeiter einen Freiraum, der motiviert.

In agilen Projekten ist der aktuelle Status jederzeit transparent. Erfolgreiche Tests signalisieren: Das Team leistet gute Arbeit. In nicht-agilen Projekten ist der Projektfortschritt nicht transparent. Sind sie Entwickler und haben Sie einen Projektmanager, der noch nie programmiert hat, die technischen Zusammenhänge nicht versteht oder verstehen will, keine ausreichende Kenntnis der technischen Umgebung hat oder haben will und der Sie vorwurfsvoll fragt: »Warum dauert die Umsetzung der aktuellen Aufgabe, an der Sie gerade sitzen, so lange?«

Entgegnen Sie ihm eine der zwei folgenden Antworten, die Sie auch kombinieren können. Entweder sagen Sie: »Durch Ihre implizite Annahme, dass ich langsam arbeite, was immer das auch heißen mag, haben Sie gerade meine Motivation gedrückt. Wenn Sie möchten, erkläre ich Ihnen, was alles zur Aufgabe gehört. Und vielleicht sollten wir mal darüber nachdenken, unsere Software agil zu entwickeln.« Oder Sie antworten: »Ihre Aussage hat soeben meine Motivation gedrückt. Es dauert nun mal so lange, und Druck wird die Dauer höchstens erhöhen. Eine Schwangerschaft dauert normalerweise neun Monate, Sie können nicht nach der Hälfte der Zeit fragen, warum es so lange dauert. Was halten Sie eigentlich davon, unsere Software zukünftig agil zu entwickeln?«

Kommunikation

Wir haben bereits festgestellt, dass Software von Menschen für Menschen gemacht wird. Kommunikation ist dabei ein elementarer Bestandteil. Das agile Team als soziologisches Objekt interagiert in unterschiedlichen Weisen mit sich selbst und anderen. Welche Arten von Interaktion gibt es? Wie kann eine Information optimal übermittelt werden? Was hat schlechte Interaktion für Auswirkungen?

Kommunikation, was ist das?

Kommunikation ist ein wechselseitiger *Austausch* in Sprache, Gestik (Bewegungen insbesondere der Arme und Beine), Mimik (Bewegungen der Gesichtsoberfläche), Schrift und/oder Bild. Es gibt *verbale* (ausgesprochene) und *nonverbale* (beispielsweise Mimik) Kommunikation, sowie *synchrone* (direkte) und *asynchrone* (zeitlich versetzte) Kommunikation. Ein typisches Beispiel für asynchrone Kommunikation ist der Austausch von E-Mails.

Kommunikation kann dem reinen Informationsaustausch dienen oder eine Aktion auslösen. Die Kommunikation ist eingebettet in einen situativen Kontext, eine Rahmenumgebung. Zwei oder mehr Personen tauschen sich über einen *Informationskanal* aus. Über dieses Medium werden *Nachrichten* hin und her übertragen (siehe Abbildung 4-3).

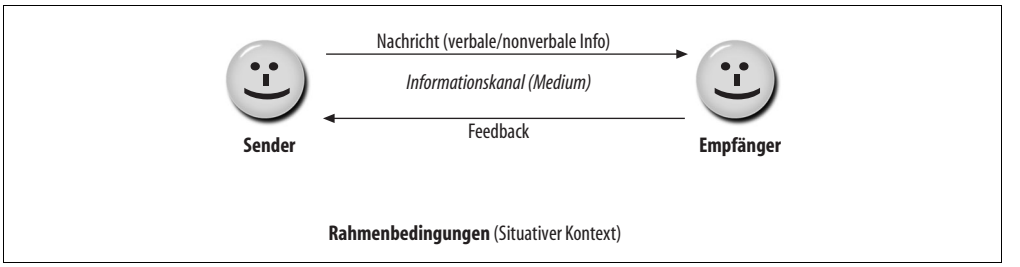


Abbildung 4-3: Schematische Darstellung von Kommunikation

Gewöhnlich wird der Gehalt der Nachricht unterschieden nach Sachebene (rational-logischer Teil), Beziehungsebene (Sympathie, Antipathie, Körpersprache), Befürfnisebene (individuelle Bedürfnisse aller Art wie Selbstverwirklichung) und Appell (indirekte Aufforderungen).

Viele Bücher widmen sich dem insbesondere für Projektarbeit wichtigen Thema Kommunikation. Ich empfehle jedem Team, vor allem nicht-agilen Teams, ein entsprechendes Buch zu lesen oder eine Weiterbildung zu besuchen. Agile Teams haben bereits die notwendigen Fähigkeiten aufgebaut, sie beispielsweise von einem Coach für agile Entwicklung gelernt.

In agilen Projekten wird die Kommunikation besonders groß geschrieben, viele der Praktiken leben von der Kommunikation. Werte wie Respekt und Vertrauen lassen sich nur mit der richtigen Kommunikation verfolgen. Sie ist inhärenter Bestandteil eines agilen Teams.

Regeln für eine funktionierende Kommunikation

Kommunikation in agilen Teams ist immer offen, ehrlich und direkt. Ich hatte einmal einen Büronachbarn, der jeden Tag Spezialisten aus unterschiedlichen Projekten zu sich einlud, um von ihrem Wissen zu profitieren. Diese Spezialisten waren schon länger im Unternehmen und kannten die besonderen gewachsenen Eigenheiten der Projekte. Auch waren diese Spezialisten hochqualifizierte Personen, die viele gute Impulse lieferten. Was tat der Büronachbar nun? Nachdem er sich von diesen Kollegen inspirieren und sich technische Sachverhalte erklären lassen hatte und diese Personen das Büro verlassen hatten, waren es plötzlich *seine eigenen* Ideen. In anschließenden Telefonaten und Gesprächen mit anderen Personen verkaufte er die ihm gerade erst erläuterten Ideen als seine eigenen. Das sprach sich natürlich herum. So eine unehrliche Arbeitsweise zerstört Vertrauen und ist ein Teamkiller. In agilen Projekten wird respektiert, dass nicht jedes Mitglied in jeder Disziplin gleich versiert ist. Jeder hat Wissen, von dem andere profitieren können. Ein Teammitglied darf also offen zugeben, wenn es gerade zu einer Fragestellung keine Antwort weiß. Nicht-agile Teams lassen das nicht zu – einmal eine Schwäche zuzugeben kann einen bleibenden Makel verursachen.

Die richtige Dosis

Kommunikation muss dosiert sein. Bei einem weiteren Fall schaffte ein hochbezahlter und hochqualifizierter Angestellter es nicht, seine Arbeit allein zu machen. Er ließ alle seine Aufgaben von anderen erledigen. Hochqualifiziert und gleichzeitig unfähig? Oder einfach nur zu faul? So oder so ist es bedenklich, wenn man mit jeder Kleinigkeit den Kollegen auf den Füßen steht. Einzelne Punkte lassen sich auch sammeln oder manchmal sogar allein lösen. Viele Personen *wollen* sogar eine Aufgabenstellung allein lösen und sind frustriert, wenn man ihnen nicht nur eine Aufgabenstellung, sondern auch direkt ihre Lösung vorsetzt.

Direkte Kommunikation

Kommunikation sollte immer klar und direkt geschehen und Respekt durch angemessene Zurückhaltung erweisen. Wenn Sie mit einer Handlung eines Kollegen nicht einverstanden sind, dann sprechen Sie mit ihm direkt und unter vier Augen. Kommt jemand zu ihnen und möchte mit Ihnen unter vier Augen sprechen, dann tun sie das, und werden sie in dem Gespräch nicht lauter, wenn andere Personen dazukommen. Wenn Sie Vorgesetzter sind, dann müssen Sie auch hier mit gutem Beispiel vorangehen. Ist der Vorgesetzte offen, ehrlich und direkt, dann ist es auch das Teammitglied. »Senior-Entwickler«, also erfahrene Kollegen, können und müssen Unerfahrene führen, was aber immer von Vorgesetzten gesteuert und kontrolliert werden sollte.

Aber bitte konstruktiv!

Ein häufiges Problem in der direkten Kommunikation ist, dass ein Kommunikationsteilnehmer von falschen Annahmen ausgeht. Beginnen Sie in einem Gespräch damit, einen gemeinsamen Nenner zu finden, den alle mittragen und von dem aus das Gespräch gestartet werden kann. Auch sollte das Gespräch am Ende noch einmal zusammengefasst werden, damit alle die gleiche Vorstellung davon haben, was entschieden wurde. Nicht zuzuhören oder den Gesprächsverlauf zu diktieren macht das Gespräch natürlich ebenfalls zu einem unproduktiven, frustrierenden Erlebnis.

Ich habe häufig erlebt, dass in einem Gespräch zu einer Fragestellung keine Lösung gefunden wurde, nein: gefunden werden *sollte*. Die Personen redeten aneinander vorbei: Die eine wollte ein Problem ansprechen und lösen, die andere, häufig ein Manager, wollte von dem Problem nichts hören. Dieses nicht-agile Vorgehen hat mehr mit Statusdenken und Machtdemonstration zu tun als damit, sich konstruktiv einer Herausforderung zu stellen.

In pathologischen Unternehmen kann es sogar Meetings geben, die eigens dazu dienen, einer Person öffentlich die Schuld für etwas zu geben. Das ist dann kein Brain Storming, sondern *Blame Storming*. Dies kommt leider bei vielen nicht-agilen Projekten vor: Ein Abteilungschef hat primär das Ziel, die eigene Abteilung als die beste herauszustellen. Die eigenen Leute sind unfehlbar, alle anderen sind schlecht. Das ist pathologische Politik, um die wir uns später noch genauer kümmern werden.

Gesteuert von einem jederzeit mit gutem Beispiel vorangehenden Vorgesetzten haben agile Teams ein hohes Maß an Selbstorganisation – auch bezüglich der Kommunikation – und Selbstheilung bei Konflikten. Wenn das Team die ersten Konflikte gemeinschaftlich überwunden hat und gestärkt daraus hervorgeht, ist die Harmonie im Team größer als vorher.

Arbeitsumgebung

Die Arbeitsumgebung trägt direkt zur Kommunikation bei. Durch sie kann sich das Team gut (oder schlecht) austauschen und alle Personen den aktuellen Projektstatus verfolgen. Sitzen die Kollegen zu weit weg oder vielleicht zu nah? Kann man »gut kommunizieren«?

Als es weiter oben um den Nutzen von agilem Handeln ging, habe ich herausgestellt, dass sich agile Teams bewusst direkter Kommunikation bedienen, um komplexe Sachverhalte zu diskutieren. Schwierig wird das, sitzen die Kollegen in unterschiedlichen Büros, so dass kein direkter Austausch möglich ist. Anderswo teilen sich andererseits viele Entwickler ein zu kleines Büro. Wenn nach zehn Minuten intensiver Arbeit der komplette Sauerstoffvorrat im Büro aufgebraucht ist, sollten Sie Ihre Arbeitsumgebung überdenken. In überfüllten Büros kann der einzelne Entwickler nicht vernünftig arbeiten. Stetige Ablenkungen und laute Gespräche im Büro können nerven. Es ist ein schmaler Grat zwischen einem agilen Gemeinschaftsbüro und einem nicht-agilen Basar!

Die Arbeitsumgebung eines agilen Teams macht nicht nur kurze Kommunikationswege möglich. Sie zeichnet sich insbesondere auch dadurch aus, dass bestimmte Medien genutzt werden (siehe unten).

Die Wahl des richtigen Kommunikationsmediums

Die Mitarbeiter bei agilen Projekten wählen das zu einer Situation passende Kommunikationsmedium. Im Optimalfall sprechen die Beteiligten persönlich miteinander, aber es gibt Ausnahmen.

Kommunikationsmedium E-Mail

E-Mails sind aus dem Arbeitsalltag nicht mehr wegzudenken und ein nützlicher Kommunikationsweg. Mails werden beispielsweise geschrieben, wenn eine direkte Kommunikation nicht möglich ist. Ein Beispiel dafür ist die räumliche Trennung zwischen Kunde und Entwickler. Auch bei der Nutzung von Mails sollten Sie ein paar Punkte beachten, um das Medium nicht zu missbrauchen.

Mails sollten dosiert geschrieben werden. Sie sollten Ihre Kollegen nicht zuspammen. Mailverteiler sollten vorsichtig genutzt werden. Haben Sie einen regen internen Mailverkehr, der dem Informationsaustausch dient? Dann sollten Sie darüber nachdenken, stattdessen ein Wiki einzusetzen. Wenn es zu viele Mails gibt, werden diese weniger gründlich oder gar nicht mehr gelesen. Nutzen Sie keine technischen Funktionen, die

überprüfen, ob Ihre Mails vom Empfänger gelesen wurden. Das ist unhöflich und zerstört Vertrauen.

Denken Sie außerdem daran, dass Empfänger, die eine Mail lediglich als Kopie erhalten (CC), sie möglicherweise als für sich selbst nicht so relevant einstufen. Sie können sich genauso wenig darauf verlassen, dass die Empfänger die Mail wirklich lesen. Das ungelesene Löschen von Mails ist in agilen Teams allerdings eigentlich tabu.

Berücksichtigen Sie ferner, dass Mails ein asynchrones Kommunikationsmedium sind und der Zeitpunkt, zu dem die Mail gelesen wird, mehr oder weniger weit in der Zukunft liegt. In agilen Projekten ist es entsprechend tabu, Mails zu schreiben mit einem Inhalt wie »In fünf Minuten starte ich den Server neu, wenn keiner was dagegen hat.« Das setzt voraus, dass alle Beteiligten die Mail tatsächlich *sofort* lesen. Das ist aber nicht der Sinn von Mails: In ihnen werden Dinge angesprochen, die auch bis morgen Zeit haben.

Kommunikationsmedium Telefon

Es existieren viele Szenarien, in denen ein Telefongespräch einer Mail vorzuziehen ist. Komplexe Sachverhalte lassen sich besser persönlich darstellen und erläutern, als es in einer Mail möglich ist. Zu lange Mails und Mails mit langen Sätzen und ausführlichen Erläuterungen werden eventuell nicht gelesen. In einem persönlichen Gespräch können Sie außerdem direkt auf Nachfragen oder Verständnisprobleme des Gegenübers eingehen.

Auch die Nutzung des Telefons sollte nicht gedankenlos geschehen. Sammeln Sie, falls möglich, erst einige Punkte und rufen dann an. Häufiges Anrufen wirkt planlos und unkoordiniert. Nutzen Sie das Telefon nur für wichtige Fragen, die persönlich besser zu klären sind. Muss eine Information den Empfänger nicht synchron erreichen, dann schreiben Sie eine Mail.

Kommunikationsmedium Wiki

Für agile Projekte ist ein Wiki ein Muss. Schreiben Sie ein zentrales Wiki, um Informationen zentral bereitzustellen, die nicht nur punktuell von Interesse sind. Dieses Wiki ist von jedem Mitarbeiter nutzbar und fortschreibbar. Dort hinterlegte Informationen sind jederzeit abrufbar. Typische Beispiele für solche Informationen sind Infrastrukturinformationen (z. B. Webadressen, Login-Daten und Servernamen), Anwendungshandbücher und User Stories.

Meetings

Ein agiles Team kommuniziert auch, um den aktuellen Stand für alle transparent zu machen. Dazu dienen verschiedene Arten von Meetings.

Tägliche Stand-ups

Es werden tägliche Stand-up-Meetings durchgeführt (auch bekannt als »Daily Scrum« oder »tägliche Synchronisationspunkte«), in denen jeder Teilnehmer drei Fragen beantwortet:

1. Was habe ich seit gestern gemacht?
2. Was gab es dabei für Probleme?
3. Was mache ich bis morgen?

Auf diese Weise hat jedes Teammitglied den Überblick darüber, was die anderen machen und was das möglicherweise für Auswirkungen auf die eigene Arbeit hat. Das Meeting sollte nicht länger als 15 Minuten dauern und am besten direkt zu Arbeitsbeginn stattfinden. Auch der Ort sollte immer derselbe sein.

Technische Details und Lösungsstrategien werden hier nicht besprochen. Das wird *nach* dem Meeting im kleineren Kreis gemacht. Es ist enorm wichtig, dass alle Teilnehmer des täglichen Stand-ups auch wirklich *stehen*. So fasst sich jeder kürzer und kommt schneller zum Punkt als im Sitzen. In einem Kreis zu stehen, fördert zielgerichtete Kommunikation.

Auch hier sollte in gerade wachsenden agilen Teams ein Coach die ersten Gehversuche überwachen. Häufig zu beobachten ist nämlich, dass viele Gruppen zwar auf den ersten Blick die agilen Praktiken richtig anwenden, aber im Detail doch entscheidende Fehler machen, wenn z. B. nicht alle Teilnehmer stehen oder das Meeting zu lang ist. Es muss erlaubt sein, das Meeting nach 15 Minuten zu verlassen. Ein weiterer Fehler ist, auf einzelne Personen zu warten. Fangen Sie immer pünktlich an und denken Sie über eine Strafe für Zuspätkommer nach. Aber Vorsicht: Wählen Sie die Sanktion mit Bedacht aus. Ein kleiner Geldbetrag, der einem guten Zweck zukommt, ist deutlich wirkungs- und sinnvoller als ein Obulus an die Kaffeekasse, von dem der Verspätete indirekt dann doch noch etwas hat. Eine falsche Ausführung von Stand-ups, wie ich sie auch persönlich erleben durfte, ist, wenn der Vorgesetzte das Meeting maßgeblich treibt. In nicht-agilen Teams berichten die Teilnehmer dann einzeln dem Vorgesetzten. In agilen Teams berichten die Teilnehmer einzeln der *Gruppe*. Dies kann oder sollte sogar von einem Coach oder Scrum Master begleitet werden, ein disziplinarischer Vorgesetzter ist allerdings nicht direkter Bestandteil des Teams. Das verhindert auch eine zu starke Ausrichtung auf den Vorgesetzten und eine Degeneration des Stand-up zu einem gewöhnlichen Statusmeeting: Ist der Boss noch in einem anderen Meeting, warten alle Beteiligten, bis er da ist. Noch schlimmer ist es, wenn er den Start auf unbestimmte Zeit verschiebt. Das Team steht auf Abruf bereit, so dass sich die Mitglieder nicht auf ihre Arbeit konzentrieren können, weil sie ja jede Sekunde zum Meeting gerufen werden könnten. Wenn das Meeting dann auch gegen Abend noch nicht stattgefunden hat, ist die Frage berechtigt, warum das Meeting nicht einfach ganz abgesagt wurde.

Andere Meetings

Neben den Stand-ups gibt es auch Retrospektivmeetings, um beispielsweise nach Iterationen Optimierungspotenzial zu diskutieren. Auch diese Meetings folgen einem klaren, disziplinierten Vorgehen, um so nicht zuletzt die Transparenz und die Planbarkeit zu steigern.

Daneben wird es sicherlich noch andere Meetings bei Ihnen geben. Diese sollten allerdings möglichst kurz gehalten werden und nur die Leute einbeziehen, die von der besprochenen Thematik direkt betroffen sind. Zu Meetings eingeladen zu werden, zu denen man nichts beitragen kann, raubt nur die Zeit und ist frustrierend. Ferner sollte zu Meetings immer förmlich eingeladen werden. Das kann so aussehen, dass man eine Terminanfrage schickt, oder noch besser eine einfache Mail, wann denn der Kollege Zeit hat. Das ist allemal höflicher, als in sein Büro hineinzuplatzen und seine Konzentration zu stören.

In »regulären« Meetings gilt das Gleiche wie im Stand-up: Fühlt sich jemand in der Runde deplatziert, kann er sofort gehen. Auch auf eine Agenda sollte Wert gelegt werden, denn der Mensch mag die Sicherheit. Wenn die Agenda bekannt ist, können sich die Teilnehmer auf die Runde einstellen und vorbereiten. Fehlende Agenden, eine nicht sichtbare Teilnehmerliste und das Vorenthalten von Informationen, um dem Gastgeber einen Wissensvorsprung zu bringen oder einen Überraschungseffekt zu erzielen, zeugen häufig von einer pathologischen Unternehmenspolitik.

Keine Ad-hoc-Meetings!

Persönliche, direkte Kommunikation ist außerordentlich wichtig. Ad-hoc-Meetings kommen selbstverständlich immer mal wieder vor, wenn sie sich aber häufen, zeugt das von blindem Aktionismus. Es sollen vielleicht schnelle Reaktion und Flexibilität suggeriert werden, aber häufig sind schlechte Organisation und Mangel an Respekt die Gründe für so ein Vorgehen. Nicht-agile Teams verstehen nicht, dass Entwickler kreative Arbeit leisten und Zeit benötigen, um in eine Phase des produktiven Arbeitens zu gelangen. Bei kontinuierlichen Störungen ist das nicht möglich und die Entwickler sind frustriert und schlecht motiviert (siehe Abbildung 4-4).

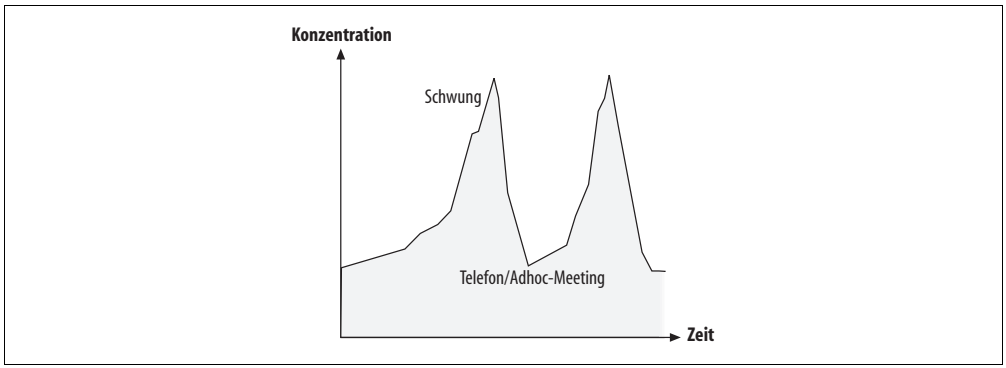


Abbildung 4-4: Die Konzentration eines Entwicklers mit Störungen

Team

Die Gruppe, die das Projekt erfolgreich durchläuft, ist mehr als eine einfache Organisationseinheit. Die motivierten Individuen müssen sich mit dem Projektziel identifizieren, die Verantwortung der Zielumsetzung akzeptieren. Es gibt niemanden, der abseits des Teams im Elfenbeinturm allein an einer Lösung arbeitet. Das ganze Team zieht in dieselbe Richtung und ist mehr als die Summe seiner Teile. Ein Team entwickelt seine eigenen Rituale – die von Außenstehenden gelegentlich auch als lächerlich angesehen werden – und überdauert ein Projekt. Selbst wenn ein Projekt eine totale Katastrophe war, war es dennoch ein gewisser Erfolg, wenn es geholfen hat, das Team zusammenzuschweißen und den Grundstein für weitere Projekte zu legen.

Einige Aspekte der erfolgreichen Teamarbeit sehen wir uns im Folgenden näher an.

Der Vorgesetzte

Es ist in traditionellen Projekten häufig der Fall, dass ein Entwickler sich in der Gegenwart des Chefs anders verhält und anders mit Kollegen spricht, als wenn der Chef nicht dabei ist, z. B. um sich vor ihm zu profilieren. Das kann in einem agilen Team dadurch aufgeweicht werden, dass der Manager tatsächlich seine Managementaufgaben erfüllt – er kümmert sich um die *Team-Performance*. Wie kann das gesamte Team performanter und wie die einzelnen Entwickler gefördert werden? Aus Sicht des Managers ist der Ansatz in einem agilen Projekt also eher ein Pull-System, in dem er fragen sollte, ob die Entwickler in einem speziellen Bereich eine Förderung benötigen oder etwas im Team nicht stimmt, um gegebenenfalls Abhilfe zu schaffen. Dies ist nämlich nicht nur eine Bringschuld der Entwickler.

Zu dieser Aufgabe des Managers gehört, dass er vorbeugend Konkurrenzdenken im Team abpuffert, das Team und seine Geschlossenheit fördert und – falls nötig – auch interveniert, wenn Einzelne sich zu sehr in den Vordergrund stellen wollen. Hier ist Führung gefragt, die der Vorgesetzte aber auch ins Team delegieren kann und muss. Um dabei allerdings gute Rahmenbedingungen zu schaffen, muss er die Leute bei Laune halten und ihre individuellen Interessen genau kennen, also was der Einzelne erreichen will und was ihn motiviert und demotiviert.

Der Vorgesetzte hat die Aufgabe, das Team zusammenzustellen und zu fördern. Werden für das Projekt neue Werkzeuge gebraucht? Der Vorgesetzte fragt das Team, welche Werkzeuge die besten sind und kauft sie, ohne Kompromisse einzugehen, nur um ein paar Euro zu sparen. Fehlt dem Team eine Information? Ist eine Person nicht gut genug ausgebildet für eine spezielle Aufgabe? Der Vorgesetzte besorgt die Information oder schickt den Entwickler auf eine Schulung, falls das Wissen intern nicht vorhanden ist. Aber Vorsicht: In einem versierten Team gibt es sicherlich jemanden, der sich auch damit auskennt und die Schulung intern geben kann. Ihr Wissen einbringen zu dürfen, motiviert diese Person wiederum ungemein.

Der Vorgesetzte muss dabei alle im Team gleich behandeln, ohne Einzelne zu bevorzugen. Er muss sachlich und nüchtern auftreten. Ich habe schon erlebt, dass ein Vorgesetzter einzelne Personen bevorzugt und ihren Aussagen mehr Gewicht gibt. Einzelne Personen mögen sich vielleicht mehr als andere, das ist ganz normal, der Vorgesetzte muss aber dafür sorgen, dass niemand aus dem Team herausragt. Das agile Team kann sich sehr gut selbst regulieren, dafür braucht es Unabhängigkeit. Es werden sich im Laufe der Projekte ganz von allein Führungskräfte innerhalb des Teams herauskristallisieren, die von anderen Kollegen gefragt werden.

Nicht-agile Projekte haben häufig Schwierigkeiten mit der Konstellation, dass der Vorgesetzte gleichzeitig Projektmanager ist und als solcher bei allen Details die Verantwortung allein übernehmen will. Sicher, er hat letztendlich die Verantwortung, aber er muss sie delegieren können. In solchen nicht-agilen Umgebungen kommt es häufig zu Gesprächen zwischen Vorgesetzem und Entwickler, in denen der Entwickler vom Chef nicht verstanden wird. Der Entwickler ist ein Spezialist und mit der Materie bestens vertraut. Der Vorgesetzte behauptet dies auch von sich, aber zu unrecht: Er kann niemals so viel Detailwissen haben wie sein Spezialist. Versteht der Chef nun in einem Gespräch die Erläuterungen des Entwicklers nicht, zweifelt er an dem Entwickler. Zweifel und Unterstellung aus Unkenntnis heraus können die Stimmung im Team verschlechtern: Der Entwickler möchte vielleicht zeigen, dass er doch gut ist, und handelt gegen das Team, indem er z. B. auf Fehler anderer hinweist.

Ein Vorgesetzter in einem agilen Projekt hinterfragt sich selbst und gibt offen zu, wenn er einen Sachverhalt technisch nicht vollständig einschätzen kann. Ich erinnere mich nur zu gern an einen Vorgesetzten, der überhaupt kein Problem damit hatte, Fehler oder Unkenntnis von technischen Details zuzugeben. Das machte ihn mir menschlich und sympathisch und förderte meine Motivation.

In agilen Teams finden Sie Offenheit und Ehrlichkeit vor. Und das Vertrauen, dass der Vorgesetzte die Aufgaben an das ganze Team delegiert, das weitestgehend eigenverantwortlich und selbstorganisierend das Ziel verfolgt.

Ursachenanalyse

Ein besonderes Verhalten hat immer einen Grund. Wenn Sie den Grund nicht kennen, dann fragen sie einfach persönlich nach. Sind Sie wirklich an einer ehrlichen Antwort interessiert, dann bekommen Sie sie auch. Das bringt Ihnen mehr, als mit Unterstellungen und Spekulationen zu arbeiten. Häufig lässt sich ein Verhalten nicht direkt erklären und ist eine zeitverzögerte Reaktion auf ein Ereignis aus einer früheren Phase. Führen Sie eine Ursachenanalyse durch.

Zusammensetzung des Teams

In agilen Teams sind ganz andere Fähigkeiten gefragt als in klassischen Projekten. Wie kann eine Personalabteilung herausfinden, wer die richtigen Personen für agile Projekte sind? Ich bin der Meinung, dass ein Entwickler nicht jedes technische Detail eines aktuellen Hypes kennen muss. Das ist bei der technischen Vielfältigkeit und Komplexität so gut wie gar nicht möglich. In agilen Teams muss er offen Wissenslücken zugeben können und lernen können und wollen. Wenn ich einen Entwickler in einem Auswahlgespräch habe, der seit zehn Jahren C++ entwickelt, dann wird er auch schnell Java lernen und erfolgreich anwenden können. Genauso ist es mit einem erfahrenen Java-Entwickler, der unzählige Frameworks kennt, aber ein spezielles vielleicht nicht. Er hat bereits bewiesen, dass er sich komplexen Aufgaben stellen kann.

Ich habe sehr gute Erfahrungen damit gemacht, die Bewerber *persönlich* zu sprechen. Natürlich muss eine gewisse Vorauswahl getroffen werden, aber erst bei persönlichen Gesprächen wird erkennbar, ob das Gegenüber genügend Feuer hat und begeistert bei der Arbeit sein wird. Das Gespräch sollte sich dabei nicht ausschließlich um technische Themen drehen. Die Personalauswahl in traditionellen Projekten berücksichtigt das zwar häufig auch, aber agile Projekte messen den anderen Aspekten eine höhere Priorität bei. Manche Leute vertreten die Einstellung, Bewerber sollten in ihren Vorstellungsgesprächen Source Code entwickeln. Es mag interessant sein, wie der Bewerber mit der Aufgabe umgeht, um zu überprüfen wie er denkt. Für agile Projekte ist eine Überprüfung der sozialen Eigenschaften allerdings viel wichtiger. Mit entsprechender Ausbildung und Projekterfahrung sowie Referenzen können wir gewöhnlich von vorhandenen Hard Skills ausgehen und uns auf die *Soft Skills* konzentrieren. In der sehr facettenreichen Java-Welt ist es z. B. nahezu unmöglich, dass ein Bewerber in allen technischen Disziplinen Erfahrung besitzt. Es ist in einem Personalauswahlgespräch viel wichtiger, die Fähigkeit beim Bewerber zu erkennen, Fragen zu stellen, offen mit den Kollegen umzugehen und dazulernen zu wollen.

Die Soft Skills sollten auch bei den kontinuierlichen Mitarbeiterbewertungen (falls solche durchgeführt werden) eine übergeordnete Rolle spielen. Ein Mitarbeiter, der ausschließlich anderen Leuten hilft und darüber hinaus auch noch bei zwischenmenschlichen Konflikte im Team vermittelnd und harmonisierend eingreift, hat nach einer harten Messmetrik deutlich weniger (oder gar nichts) geleistet im Vergleich zu jemanden, der anderen nicht hilft und allein vor seinem Rechner seine Aufgaben erledigt. Diesen kurzsichtigen Blickwinkel vieler konventioneller Projekte gibt es bei agilen Projekten nicht. Hier steht die Teamleistung im Vordergrund, die im geschilderten Fall deutlich höher ist, als wenn alle Mitarbeiter ganz allein ihre Aufgaben umsetzen. Wie können die Soft Skills bei Mitarbeiterbeurteilungen gemessen werden? Fragen Sie die Mitarbeiter! Führen Sie regelmäßige *Peer Reviews* durch: Das ganze Team bewertet sich selbst und jeden Einzelnen.

Agile Teams arbeiten intensiv zusammen. Manche Unternehmen haben gute Erfahrungen damit gemacht, dass der Bewerber jedem einzelnen Kollegen vorgestellt wird. Erst wenn alle aktuellen Teammitglieder ihr OK geben und sich mit der Einstellung einver-

standen erklären, wird der Bewerber eingestellt. Teamplay kann zwar nicht erzwungen werden, aber wenn die Mitarbeiter in spe sich alle bereits kennengelernt haben und gut miteinander auskommen, ist die Chance deutlich höher, dass sich ein richtiges Team bildet bzw. der neue Kollege erfolgreich in das Team integriert wird.

Vorgesetzte agiler Projekte sind selbstverständlich auch im Vorstellungsgespräch ehrlich. Wenn sie es nicht sind, ist das der erste Schritt zu fehlendem Vertrauen. Ein unglücklicher Start ohne Vertrauen kann bereits die weitere Arbeitsatmosphäre nachhaltig stören. Unter diesen Umständen kann es manchmal auch für den Entwickler das Beste sein, sein Recht auf Auflösung der »gegenseitigen Probezeit« wahrzunehmen. Wenn Sie als Entwickler in einer derartigen Situation sind und überzeugt sind, dass der Schritt in das neue Unternehmen falsch war, dann zögern Sie nicht, von sich aus den Schritt zurückzunehmen. Es ist besser, Sie tun es, als zu warten, bis es dann vielleicht später der Arbeitgeber tut.

Die agile Philosophie verbreitet sich auch außerhalb der IT-Abteilung: Der agile Personaler hinterfragt die Fachabteilung. Längerfristig zahlen sich Ehrlichkeit und Motivation insbesondere dadurch aus, dass die Mitarbeiterfluktuation niedrig ist. Eine große Fluktuation ist ein eindeutiger Indikator, dass in der Abteilung etwas Grundsätzliches nicht stimmt. Obwohl im agilen Team das Wissen verhältnismäßig gleichmäßig verteilt ist, ist doch das Ausscheiden jedes Einzelnen ein ungemein großer Verlust. Er hat während der Projekte viele Erfahrungen gesammelt, die er bei neuen Projekten mit dem Team wieder hätte einbringen können. Davon unabhängig ist es teuer, einen ausscheidenden Mitarbeiter zu ersetzen. Eine Neueinstellung birgt insbesondere das Risiko, dass der Neue nicht so gut in das Team passt.

Aktive Pause

Wie können Sie ein produktives Team formen und fördern? Sie müssen den Spaß miteinander fördern. Wenn die Entwickler gemeinsam lachen, abends etwas zusammen unternehmen, zusammen aktive Pausen machen, dann ist das Team gesund. Was sind aktive Pausen? Eine aktive Pause ist, wenn ein Kollege den anderen zu einem Kaffee einlädt. Dabei ergeben sich Gesprächsthemen, die das aktuelle Projekt betreffen. Die Entwickler finden Abstand und Ablenkung von Ihrer Konzentrationsphase am Rechner. Doch thematisch befassen sie sich immer noch mit dem Projekt. Auch nicht projektspezifische Themen sind förderlich für das Teambuilding. Eine passive Pause auf der anderen Seite ist ein einfaches Ausruhen, ein Herunterschalten. Bitte verstehen Sie mich nicht falsch: Passive Pausen sind absolut notwendig. Aber agile Teams arbeiten darüber hinaus ergänzend ausgiebig mit aktiven Pausen.

Teamkiller

Die Einflüsse vieler auf die Entwickler wirkender Impulse auf das Team werden erst auf dem zweiten Blick ersichtlich. Ein Beispiel sind hier die Firmenziele. Diese werden von den Entwicklern sehr genau verfolgt. Auch sie haben Einfluss auf das Team!

Im weiteren Verlauf fasse ich einzelne Teamkiller zusammen. Es sind wiederkehrende Muster, die Ihr Team garantiert zerstören bzw. dem Aufbau eines agilen Teams im Weg stehen werden.

Management
: Der Chef vertraut seinen Mitarbeitern nicht. Er hat immer Recht, die Kollegen nicht. Fehler dürfen nicht gemacht werden. Es wird kein Verbesserungspotenzial gesucht, denn angeblich läuft das Projekt schon optimal.

Angstkündigungen
: Gute Personen werden gekündigt. Das schlechte Management hat Angst, dass die guten Mitarbeiter gefährlich werden.

Bürokratie
: Das Unternehmen beschäftigt sich den Großteil der Zeit mit der Verwaltung seiner selbst.

Mangelhafte Qualität
: Das hergestellte Produkt hat eine niedrige Qualität. Die Fähigkeiten der Entwickler sind deutlich höher, als das schlechte Produkt meinen lässt. Das Selbstwertgefühl der Entwickler leidet.

Nicht haltbare Termine
: Der Chef gibt einen Termin (eine Deadline) vor, der lächerlich unrealistisch ist. Er wendet Druck an, um die Mitarbeiter anzuhalten, noch schneller zu arbeiten und den Termin einzuhalten.

Einzelne zum König machen
: Einzelne Personen werden herausgegriffen und bevorzugt. Andere werden vor dem Team bloßgestellt.

Überstunden
: Überstunden durch schlechte Planung und falsche Termine schweißen das Team nicht zusammen. Ein Entwickler wird über kurz oder lang die Kollegen dafür verantwortlich machen, dass er weniger Zeit mit seiner Familie und seinen Hobbys verbringt, als er eigentlich möchte. Macht das ganze Team »freiwillige« Überstunden, nur ein Mitarbeiter nicht, so ist er schnell ein Außenseiter.

Die richtige Teamgröße

Das Team kann nur gut kommunizieren und agieren, wenn es die richtige Größe hat. Wenn wir davon ausgehen, dass im Team jeder mit jedem diskutieren soll, so haben Sie bei n Personen $n \times (n-1)$ Kommunikationspaare. Gewöhnlich sagt man, agile Teams verkraften eine Größe von zwölf Personen. Sie können das mit einer schnellen Eingreiftruppe vergleichen, die sehr flexibel agiert. Obwohl mir dieser Vergleich nicht besonders gefällt, ist er doch in einer gewissen Weise sehr treffend: Militärische Sonderkommandos sind klein, flexibel, haben alle ausgewogene Fähigkeiten und sind selbstorganisierend. Ihnen wurde die Verantwortung übertragen, autonom zu handeln. Jedes Teammitglied

kann autonom Entscheidungen fällen. Dennoch gibt es auch Führungspersönlichkeiten. Es hat eine interne Struktur. Anders ist es beim nicht-agilen Ansatz: Jedes Teammitglied hat eine spezielle Funktion, ähnlich wie bei einem chirurgischen Team: Dem Team steht ein Chefarzt vor, der eindeutige Kommandos gibt. Daneben gibt es diverse Ärzte und Helfer mit speziellen Funktionen.

Wissen

Was bedeutet Wissen? Wissen ist eine immens wichtige Ressource mit ganz unterschiedlichen Ausprägungen: Es fängt damit an zu verstehen, was der Kunde tatsächlich möchte. Dann müssen daraus Anforderungen formuliert werden, mit denen richtig umgegangen werden muss. Und schlussendlich geht es natürlich auch um die technische und fachliche Expertise. Doch wie generiert ein agiles Team Wissen und wie geht es damit um?

Mehrwert

Wissen bedeutet auch zu verstehen und zu identifizieren, was wertvoll ist und tatsächlich Wert schafft. Welche Aktivitäten sind überflüssig? Welche Produktfunktionalität wird nicht gebraucht? Welche Funktionalität hat einen größeren Wert als eine andere? Daneben stellt Wissen aber auch einen Wert, also auch einen Wettbewerbsvorteil dar.

Obwohl auch in agilen Teams häufig Führungspersönlichkeiten neues Wissen (Werte, Assets) generieren, wird doch das ganze Team in diesen stetigen Prozess eingebunden. Diese eigenen Werte grenzen das Team und das ganze Unternehmen von Wettbewerbern ab: Wären diese Werte am Markt frei verfügbar oder könnten günstig gekauft werden, würden sehr viele Unternehmen über das gleiche Wissen verfügen.

Wissen bedeutet ebenfalls zu verstehen, dass der gesamte agile Entwicklungsprozess, die ganze Software und das ganze Team aus einzelnen Teilen bestehen und der Wert aus den Beziehungen untereinander generiert wird. Notwendiges Selbstbewusstsein, Mut, Erfahrung, Fähigkeiten, die Zusammenarbeit mit anderen, dies alles ist aus reinen Geschäftszahlen nicht abzulesen. Agile Projekte kennen dennoch die Bedeutung dieser Werte und arbeiten kontinuierlich und nachhaltig an ihrer Vermehrung.

Anforderungen

Ein agiles Projekt muss ein intensives Verständnis für den Kunden und seine Bedürfnisse erwerben. Das gelingt nur mit interdisziplinären Teams, die Experten für verschiedene Disziplinen vereinen und auch fachliches Wissen aufbauen können. Das Team muss nicht alle Technologien direkt beherrschen. Es muss einen Rhythmus und eine Kultur aufbauen, die es ihm ermöglichen, unbekannte Technologien zu erlernen, die Fachlichkeit vom Kunden zu lernen und die Anforderungen zu verstehen und umzusetzen. Dieses Wissen muss in das entwickelte Produkt einfließen und seine Qualität maximieren.

Management nach Zahlen

In nicht-agilen Projekten ist häufig ein *Management nach Zahlen* zu beobachten. Es ist geprägt von einem übermäßigen Fokus auf quantitative Werte, z. B. aktuelle operative Zahlen, die Maximierung des Umsatzes oder eine strikte Reduktion der Kosten. Im schlimmsten Fall kann dies dazu führen, dass Manager, die für eine kurze Zeit die Führung einer Unternehmung haben, ausschließlich den *Shareholder Value* maximieren wollen. Dazu kann die Maximierung eines Börsenkurses gehören. Häufig haben die Manager hohe dynamische Gehaltsanteile, die direkt abhängig sind von einem Börsenkurs, der Stimmung der Gesellschafter oder dem kurzfristigen Gewinn.

Dabei wird häufig die Qualität und die Nachhaltigkeit vernachlässigt. Programmierer werden gern als austauschbares Gebrauchsgut betrachtet. Dieses Muster wird begleitet von einer kurzsichtigen Denkweise. Sie berücksichtigt beispielsweise nicht, dass fehlende Mitarbeitermotivation und/oder hohe Mitarbeiterfluktuation das Unternehmen mittel- bis langfristig wesentlich teurer kommen als eine kurzfristige Investition. In diesem Zusammenhang kann der Begriff »Softwarefabrik« angeführt werden, der Versuch, die Softwareentwicklung zu automatisieren und den Programmierer als reinen Produktionsfaktor zu betrachten.

Dieser Ansatz berücksichtigt allerdings nur unzureichend, dass die Softwareentwicklung zu einem großen Teil ein kreativer, künstlerischer Prozess ist, der Freiraum, große Entfaltungsmöglichkeiten und hohe (intrinsische) Motivation des Entwicklers voraussetzt. Ferner gilt es zu bedenken, dass Mitarbeiter über die Zeit viel Erfahrung bei der Arbeit an einem Produkt aufbauen, die dem Unternehmen zu einem großen Teil verloren geht, wenn die Person es verlässt.

In diesem Zusammenhang wird manchmal zwischen Management nach *englischer Theorie* und Management nach *spanischer Theorie* unterschieden. Beim »englischen Führungsstil« geht das Management davon aus, es gäbe nur eine feste Menge von Werten. Beim »spanischen Ansatz« geht man davon aus, dass Werte durch Genialität und Technologie geschaffen werden können. Diese beiden Wertetheorien beruhen auf Beobachtungen, wie die ehemaligen Kolonialmächte England und Spanien mit Ihren Kolonien umgingen, um maximalen Profit zu erlangen.

Einem Management nach Zahlen kann durch das Umschwenken auf agile Methodik entgegnet werden. In Unternehmen gibt es daneben seit längerem einen *Balanced Scorecard*-Ansatz. In der Unternehmensführung und Projektplanung werden dabei nicht mehr ausschließlich operative Zahlen zugrunde gelegt. Vielmehr richtet sich das Unternehmen an strategischen Zielen aus und misst diese in regelmäßigen Abständen. Darunter fallen Merkmale wie zufriedene Mitarbeiter und Kunden.

Um zu verstehen, was der Kunde möchte, muss die Software sehr schnell nutzbar sein. Je früher Feedback vom Kunden kommt, desto besser kann das Team verstehen, was der Kunde möchte. Erst wenn der Kunde konkret mit einer ersten Version arbeitet, beginnt der Lernprozess des Teams.

Der Umgang mit Anforderungen umfasst ihre systematische Erhebung, Analyse, Spezifikation, Validierung, Verwaltung, Umsetzung und Verfolgung. Je schneller Anforderungen umgesetzt werden, desto schneller wird zugängliches Wissen generiert.

Das richtige Medium

In konventionellen Projekten werden Anforderungen gewöhnlich mit Microsoft Word in einer mehr oder weniger vorgegebenen Struktur erfasst und verwaltet. Das ist bequem, da jeder Mitarbeiter Word kennt und es leicht zu bedienen ist. In Großkonzernen werden zur Verwaltung von Anforderungen häufig Werkzeuge wie *CaliberRM* oder *Doors* eingesetzt, insbesondere dort, wo gesetzliche Regelungen für die Anforderungsermittlung existieren.

Anforderungen werden in agilen Projekten als User Stories formuliert. User Stories sind eine schlanke Form von Anwendungsfällen (Use Cases) und beschreiben die Funktionsanforderungen an die Software aus Sicht eines Akteurs (also beispielsweise eines speziellen Nutzers). Eine User Story folgt dem Muster »Als xxx kann ich xxx, um xxx.« Das ganze Team ist bei der Erstellung der Stories beteiligt, wobei der Kunde sie optimalerweise selbst erstellen sollte. Die User Stories werden auf *Story Cards* festgehalten.

Priorisierung

Anforderungen unterscheiden sich unter anderem in ihrem Wert, ihrem Nutzen und der somit entstehenden Gewichtung zueinander. Wie kann jedoch der *Wert* einer Anforderung gemessen werden, und für welche Umsetzung soll sich der Kunde entscheiden? Professor Kano stellte 1984 einen Ansatz vor, der unter dem Begriff *Kano-Modell* bekannt geworden ist (siehe Abbildung 4-5).

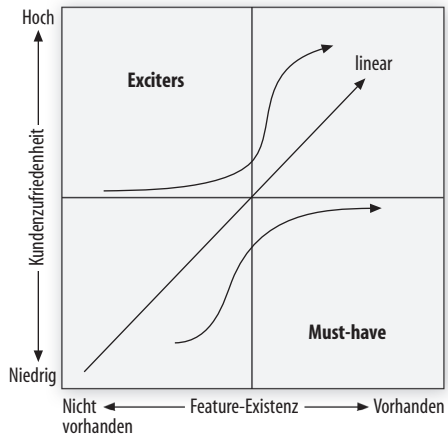


Abbildung 4-5: Das Kano-Modell

In Befragungen mit echten Kunden oder Kunden-Proxys (Stellvertreter von Kunden) werden Fragen in *funktionaler* und in *dysfunktionaler* Form gestellt. Die Kunden sollen bewerten, wie wichtig es ihnen ist, dass das Produktmerkmal vorliegt (funktionale Herangehensweise), oder wie sehr die Zufriedenheit leiden würde, wenn das Merkmal nicht vorläge (dysfunktionaler Ansatz). Auf diesem Weg werden die Anforderungen gewichtet.

Anschließend lassen sich die User Stories bestimmen, die *unbedingt* fertiggestellt werden müssen (*must-haves*), und welche von ihnen *linearer* Natur sind (je mehr desto besser, Kundenzufriedenheit steigt linear) und welche *Exciters* (*Begeisterer)* sind. Exciter sind Funktionalitäten, mit denen der Kunde gar nicht rechnet. Er würde auch ohne diese Funktionalität mit dem Produkt arbeiten und sie gar nicht vermissen. Ihre Existenz allerdings ist ein positiver *Eyecatcher*, der die Kundenzufriedenheit überproportional erhöht. Der Kunde ist dadurch auch bereit, mehr Geld für das Produkt auszugeben.

Den einzelnen User Stories werden Prioritätswerte zugeordnet. Dazu müssen Kunde und Entwicklungsteam Klarheit gewinnen, welche User Stories das höchste Risiko bezüglich Zeitplan, Kosten oder Fachlichkeit (»fachliches Risiko«) besitzen und welche den höchsten bzw. niedrigsten Mehrwert bieten. Dabei wird gewöhnlich ein Risikodiagramm erstellt (siehe Abbildung 4-6).

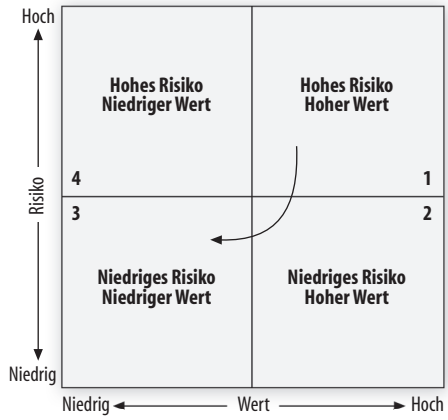


Abbildung 4-6: Ein schematisches Risikodiagramm

Das Release sollte mit denjenigen User Stories begonnen werden, die das höchste Risiko und den höchsten Nutzen mit sich bringen. Danach sind diejenigen Stories zu verwirklichen, die geringes Risiko, aber hohen Nutzen haben. Anschließend geht das Team die User Stories an, die geringes Risiko und geringen Nutzen haben. Die Fertigstellung von User Stories mit geringem Nutzen bei hohem Risiko ist zu vermeiden.

Wissen weitergeben

Erneutes Lernen ist Zeitverschwendung, deshalb muss Wissen weitergeben werden. In agilen Teams geschieht das hauptsächlich durch direkte Kommunikation. Die unterstützenden Maßnahmen haben wir bereits kennengelernt. Dazu gehören die räumliche Nähe der Teammitglieder zueinander, tägliche Stand-up Meetings und direkte Gespräche. In Pair-Programming-Sessions generieren zwei Entwickler zusammen Wissen, was sie in Design und Code ausdrücken und weitergeben. Nicht nur der Code, sondern der gesamte Entwicklungsprozess ist Wissen: Wissen über die Anforderungen und über die Fachlichkeit und technisches Wissen, also wie neue Technologien erfolgreich und optimal einzusetzen sind. Auch in agilen Teams spielen Dokumentationen eine Rolle, insbesondere vor dem Hintergrund der Weitergabe von Wissen.

Dokumentation

Dokumentation ist der ständige, transparente Umgang mit dem Projektstatus und den Anforderungen. Der Status ist an öffentlichen *Burn-Down-Charts* jederzeit ablesbar. Befassen wir uns mit dem Vorgehen eines agilen Projekts, werden wir uns mit diesen Medien weiter befassen. Auch die Anforderungen werden in Form von User Stories *dokumentiert*.

Häufig wird behauptet, agile Projekte dokumentierten gar nicht. Das ist falsch. Sie dokumentieren sehr wohl. Was genau sie dokumentieren, hängt von den Umständen im Projekt ab. Auch hier ist kein Dogmatismus zu verfolgen, sondern nüchtern die individuelle Situation zu prüfen. Gewöhnlich stellt sich ein agiles Team zunächst die Frage, was generiert werden soll, und für wen. Es müssen also das Zielpublikum identifiziert, der Zweck bestimmt und der Kontext geklärt werden.

Teaminternes Dokumentieren

Wenn für das eigene Team (die Entwicklungsabteilung) dokumentiert werden soll, handelt es sich dabei häufig um wichtige Designentscheidungen, die eine lange Lebenszeit haben. Es empfiehlt sich, das strategische Design einer Anwendung zu dokumentieren. Entscheidungen, die morgen wieder hinfällig werden, brauchen nicht dokumentiert zu werden. Außerdem gibt es noch die How-tos. Diese technischen Anleitungen, wie beispielsweise etwas installiert werden muss, wie eine Datenbank eingerichtet werden muss usw., können zentral zugänglich gemacht werden. Diese kompakten Informationen haben einen großen Mehrwert. Eine Architektur kontinuierlich mündlich beschreiben oder How-tos jederzeit auswendig parat haben zu wollen, ist nicht pragmatisch. Neben den harten Fakten sollten auch die Designentscheidungen mit in die Dokumentation aufgenommen werden. »Warum genau wird etwas so gemacht, wie es gemacht wird und nicht anders?« Das ist für das Verständnis enorm wichtig.

Benutzerdokumentation

Eine andere Zielgruppe sind die Nutzer der Software. Hier bieten sich häufig User Guides an, die die Nutzung mit der Software erläutern. Hier sollten Tutorials genutzt werden, die den User anhand von konkreten Szenarien durch die Software führen. Häufig ist ein Glossar eine gute Wahl. Erklären Sie dort die zentralen Begriffe des Projekts. Oft steht ein Projekt vor der Herausforderung, eindeutige Begriffe zu verwenden. Insbesondere in der Schnittstelle zur Fachlichkeit des Kunden wird doch dasselbe in unterschiedlichen Worten beschrieben. Das führt zu Verwirrung und Missverständnissen.

Dokumentationsregeln

Dokumentation sollte möglichst einfach sein und nach klaren Regeln geschrieben werden. Nutzen Sie zum Beispiel nicht zu viele Fonts. Sie können der Dokumentation auch UML-Diagramme beifügen. Hier aber gilt die Regel, dass diese Diagramme nicht das ganze System abbilden sollten. Sie sollen nur auf einen bestimmten wichtigen Punkt aufmerksam machen (*Hotspots*). Diagramme sollten in Fließtext erläutert werden, der hilft, den Leser durch die Diagramme zu führen. Auch für das *Overall Picture* (beispielsweise die Architektur) ist eine grafische Abbildung hilfreich. Es ändert sich nicht so schnell und bietet einen sehr guten Einstiegspunkt für Diskussionen oder für neue Mitarbeiter.

Wer schreibt die Dokumentation?

Der Entwickler sollte die Dokumentation selbst schreiben. In nicht-agilen Teams werden häufig *Technical Writer* eingesetzt. In agilen Teams schreibt der Entwickler die Dokumentation, weil er am besten Bescheid weiß. Es findet also kein Informationsverlust statt.

Das richtige Medium

Dokumentation kann in Papierform erstellt werden, manchmal ist das sogar notwendig. Je höher die Nutzungshürde für ein Medium ist, desto weniger wird es genutzt. Zentrale Laufwerke mit Word-Dokumenten, wie sie häufig in nicht-agilen Teams anzutreffen sind, werden selten fortgeschrieben und oft nicht gelesen. Dies gilt auch für Papierdokumentation, die ganze Schränke füllt (»Schrankware«).

In agilen Projekten bietet sich häufig ein elektronisches Medium an, ein Wiki. Ein Wiki ist eine aus editierbaren, untereinander verlinkten Seiten bestehende Webanwendung. Hier kann jeder dezentral Wissen hinzufügen. Da Wissen sehr schnell veraltet, kann es im Wiki problemlos und schnell fortgeschrieben werden. Ein Wiki ist leicht erreichbar, ein Browser genügt. Es kann auch verteilt genutzt werden, wenn also die Entwickler doch einmal räumlich entfernt voneinander sitzen. Es gibt viele verschiedene Wikis. Nutzen Sie ein möglichst einfaches, das Ihren individuellen Anforderungen am besten entspricht.

Automatisch generierte Dokumente

Neben der direkt fortgeschriebenen Dokumentation existiert auch die generierte Dokumentation. Darunter fallen beispielsweise die JavaDocs, die speziellen Kommentarzeilen innerhalb des Quellcodes, anhand deren das API der Anwendung in Form von HTML dokumentiert werden kann. Ein weiteres Beispiel sind automatisch erstellte Testberichte. Tests werden automatisiert durchgeführt und ihre Ergebnisse zentral bereitgestellt.

Diese generierten Dokumentationen sind letztendlich immer nur ein Ansicht des eigentlichen Inhalts. JavaDocs und Testreporte sind beispielsweise nur eine (ganz spezielle) Sicht auf den Quellcode. Generieren Sie Dokumentation, wo immer es geht. Agile Projekte setzen den Schwerpunkt auf die fachlichen Klassen und Testklassen. Dies sind die wichtigsten Artefakte des Projekts. Wenn Sie daraus weitere Dokumentationen generieren können, brauchen sie die generierten Artefakte selbst nicht aktiv fortzuschreiben. Sie schreiben lediglich den Quellcode fort: Die *Sekundärartefakte* werden automatisch erzeugt. Erzeugen Sie nur Sekundärartefakte, die Ihrem Projekt tatsächlich Mehrwert bringen.

Andere Formen der Wissensverbreitung

Eine andere Form der Dokumentation sind E-Mails. Manche nicht-agile Projekte richten sich Mailinglisten ein, die ungemein stark genutzt werden. Auf diese Weise soll Wissen manifestiert und weitergegeben werden. Häufig ist dies aber ein Antimuster: Schicken die Entwickler jeden kleinsten Schritt über die Liste, wird die Liste letztendlich gar nicht mehr gelesen. Daneben sind Mails recht schwer archivier- und durchsuchbar. Nutzen Sie derartige Listen, so reduzieren sie unbedingt den Verkehr und geben Sie die Informationen nicht direkt, sondern beschränken Sie sich auf Hinweise *auf* Informationen. Teilen Sie also nur mit, was von allgemeinen Interesse ist, wer das Wissen gerade hat und ob es konserviert wird (also beispielsweise in einem Wiki). Beschränken Sie sich auf das Wesentliche. Die Details sollten komplementär in dem Wiki platziert werden.

Häufig ist die falsche Nutzung von Mails in nicht-agilen Projekten ein Indikator dafür, dass es im Team nicht stimmt oder der Prozess suboptimal ist. Herrscht beispielsweise großer Leistungs- und Konkurrenzdruck vor, den der Manager nicht harmonisieren kann, werden die Entwickler indirekt angehalten, auf Ihre Leistungen mit Mails aufmerksam zu machen.

Neben dem Wiki sind auch weitere Medien denkbar. Manche agile Teams nutzen Videobänder und MP3-Aufnahmen. Es kann durchaus hilfreich sein, eine technische Diskussion am Whiteboard auf Video aufzunehmen oder von Meetings Audiodateien zu erstellen. Überprüfen sie den Mehrwert dieses Vorgehens in Ihrem Projekt.

Dokumentation wird nur gelesen und fortgeschrieben, wenn sie einfach zu erreichen ist und für die Nutzergruppe einen klaren Mehrwert liefert. Sie sollte komplementär eingesetzt und am besten generiert werden, um neue Views auf die primären Artefakte (den Quellcode) bereitzustellen.

Zeit

Sie waren doch sicherlich auch schon mal in einem Projekt tätig, in dem es Stress gab. Der Endtermin rückt näher, die Fehler werden mehr und es kommen veränderte Anforderungen auf Sie zu, die Sie noch mit aufnehmen sollen. Dann muss auch schon mal länger gearbeitet werden, damit der Termin eingehalten werden kann. Wenn Ihnen das bekannt vorkommt, scheinen Sie bis dato in einem nicht-agilen Projekt zu arbeiten.

Tempo

Wir haben bereits unter dem Blickwinkel der Motivation diskutiert, dass bei Google, einem sehr guten Beispiel eines Unternehmens, das die agile Methodik lebt, die Mitarbeiter 20% ihrer Arbeitszeit individuell gestalten können. Sie können Aktivitäten nachgehen, von denen sie meinen, dass sie für Google gewinnbringend sind. Wie passt das aber zusammen: Auf der einen Seite wird bei vollem Pensum gearbeitet, und bei Google sind nur 80% des geregelten Arbeitstags für die »gewöhnliche« Projekttätigkeit reserviert? Trotz vermeintlichen Terminstresses?

Die Entscheidung von Google für dieses Vorgehen hat zwei Gründe, die gleichermaßen genial wie einfach sind. Zum einen ist der Mitarbeiter durch den Freiraum wesentlich höher motiviert. Wir haben das bereits erläutert. Er kann die verbleibende Zeit aktiv ausfüllen und ein Projekt verfolgen, das ihn interessiert. Er wird zum Unternehmer im Unternehmen.

Der andere Grund ist noch raffinierter und eine Kopie des technischen Ansatzes, gemäß dem Google seine Hardware konfiguriert und verbindet: Google nimmt nicht einige wenige große Server, die die komplette Last aufnehmen. Es nimmt sehr viele kleine Rechner, die verteilt arbeiten. Durch die Lastverteilung auf viele kleine Knoten merkt der Surfer nicht, wenn ein einzelner Google-Rechner ausfällt. In einem agilen Softwareprojekt ist es ähnlich: Durch viele kleine Iterationen wird die Anfälligkeit für Fehler in der Software und im Prozess minimiert. Aber es kommt noch etwas dazu, das jeder Administrator eines Servers bestätigen kann: Die Systeme werden nicht auf 100% Auslastung gefahren. Ist das nämlich dauerhaft der Fall, nimmt die Durchlaufzeit einer Anfrage drastisch und überproportional zu. Deswegen werden Rechner (Server) gewöhnlich bei einer Auslastung von unter 80% gehalten.

Das betrifft auch die Mitarbeiter: Wer möchte bezweifeln, dass ein Entwickler, der dauerhaft auf 100% Auslastung fährt, überhitzt, Fehler macht, ausbrennt und frustriert wird. Google fährt sein wertvollstes Gut, die Mitarbeiter, auf 80%. So bleiben die Entwickler flexibel und handlungsfähig.

Das agile Projekt nimmt am meisten Tempo auf, wenn es in viele kleine Schritte zerlegt ist. Nachgelagerte Arbeitsstationen holen sich dabei ihre Arbeit, wann sie sie brauchen. Wenn z. B. ein agiler Entwickler an einer Stelle nicht weiterkommt, oder bei der Realisierung seines Tasks auf ein Spezifikationsdefizit trifft, fragt er direkt den Kunden, wie der es sich vorstellt. Dies ist ein *Pull*-System. Dort, wo in agilen Projekten eine Information

fehlt, wird sie aktiv und situationsabhängig geholt. Es existieren keine Informationen auf Vorrat, keine Prozesse und Standards um der Prozesse und Standards willen.

Flexibilität

Eine weitere Ausprägung dieses agilen Prinzips ist der Umgang mit den Arbeitspaketen. Es ist zwar wünschenswert, dass zu Beginn die Anforderungen an das System festgehalten werden. Das geschieht allerdings zunächst sehr grob, da sie erst im Laufe der weiteren Beschäftigung mit der konkreten Anforderung besser verstanden und weiter vorangetrieben wird. Das ist auch der Grund dafür, dass User Stories keine volle Spezifikation darstellen sollen, sondern lediglich Erinnerungs- und Diskussionspunkte sind.

Geht es in eine konkrete Iteration, werden nicht alle möglichen Anforderungen mitgenommen, sondern es wird eine Auswahl getroffen, welche wichtigen Funktionen in der nächsten Iteration implementiert werden sollen. Auch die Iteration ist zeitlich kurz gehalten. Abhängig davon, wie stabil die Anforderungen sind (während einer Iteration sollen die Auswahl und der Umfang der zu entwickelnden Anforderungen stabil bleiben), kann die Iteration eine Länge von einer Woche aufwärts haben.

Auch der Umgang mit den Defekten spiegelt diesen Ansatz wieder. Erneut wird der Arbeitsfluss mit einem Pull-System verwaltet. Bei agilen Projekten werden Defekte, die während der Entwicklung auftreten, nicht selten mit normalen Anforderungen gleichgesetzt. Kritische Defekte sollen nach dem Motto *Stop-the-Line* direkt angegangen werden. Aber nicht alle Defekte sind wichtig.

Nicht-agile Projekte erfassen und verwalten alle Ideen, Anforderungen und Defekte in einer Tracking-Datenbank. Wenn deren Abarbeitung rein zeitlich (bzw. aufgrund mangelnder Ressourcen) gar nicht möglich ist, wo ist dann der Mehrwert?

In agilen Projekten ist der Ansatz ein anderer. Bei Scrum beispielsweise existiert das *Product Backlog*, das die Anforderungen (und Defekte, die mit Anforderungen gewöhnlich gleichgeschaltet werden) enthält. Wenn aber absehbar ist, dass niedrig priorisierte Einträge in absehbarer Zeit nicht umgesetzt werden, dann werden sie nicht weiter im Backlog fortgeschrieben. Es wird bewusst auf den Aufwand der Verwaltung all dieser Einträge verzichtet. Diese Verwaltung von nicht realisierbaren Einträgen bringt keinen Nutzen, sie ist Verschwendung. In agilen Projekten wird auf eine Verwaltung derartiger Anforderungen oder Defekte bewusst verzichtet.

Das Wechselspiel mit der Qualität

Um das ganze Projekt in einem Fluss mit möglichst großem Durchsatz. zu halten sind Verschwendungen zu eliminieren, kritische Defekte sofort anzugehen und die individuelle Auslastung von 100 auf 80% zu drosseln. Auf diese Weise wird nicht nur das Tempo des Gesamtsystems, sondern auch die Qualität der Software maximiert. Und nur wenn diese maximal ist, können neue Anforderungen schnell umgesetzt und Defekte im Prozess und in der Software schnell lokalisiert werden. Qualität und Tempo beeinflussen

sich gegenseitig. In nicht-agilen Projekten wird der Qualität weniger Bedeutung beigemessen und dadurch das Tempo mittel- und langfristig drastisch reduziert. In agilen Projekten hat die Qualität einen ausgesprochen hohen Stellenwert. Durch stetige Optimierung der Software und des Prozesses wird die hohe Qualität nicht nur gehalten, sondern noch weiter erhöht. Dadurch kann auch das Team weiter unter hohem Tempo arbeiten.

Das missbrauchte Tracking-System

Ich war einmal in einem Projekt tätig, in dem ich für ein Java-GUI-Framework zuständig war. Wer einmal mit Java-Benutzerschnittstellen entwickelt hat, der weiß, dass das eine sehr komplexe Angelegenheit sein kann: Das resultiert einerseits aus der komplexen Implementierung einzelner Features und andererseits aus umfangreichen Wechselwirkungen zwischen den einzelnen visuellen Komponenten.

Nachdem nun eine Vielzahl von Endnutzern auf das System aufgeschaltet worden war, wurden immer mehr Rückmeldungen in das Tracking System eingetragen. Jeder konnte Einträge vornehmen, technische Anforderungen und fachliche. Die Tracking-Datenbank füllte sich enorm und konnte nicht im gleichen Tempo abgearbeitet werden.

Es war sogar so, dass viele Anforderungen sich gegenseitig widersprachen oder an einem Tag etwas umgesetzt werden sollte, das bereits am Tag davor ganz anders realisiert worden war. Viele Anforderungen wurden gar nicht verstanden, bei anderen wurden der konkrete Nutzen und insbesondere auch der Mehrwert für den Nutzer außer Acht gelassen.

Es existierte streckenweise keine vernünftige Release-Planung mehr. Das Tracking System uferte aus zu einem riesigen Moloch an Einträgen, die plötzlich alle mehr oder weniger wichtig waren. Einige Entscheidungsträger schlossen von der großen Anzahl der Einträge auf eine schlechte Qualität der Software. Sie hätten aber auf eine schlechte Qualität des Entwicklungsprozesses schließen sollen: Auf Ursachenanalysen wurde verzichtet, eine strategische Release-Planung und ein offener, konsistenter Umgang mit den Anforderungen fand nicht statt.

Ein agiler Umgang mit den Anforderungen mit einem Blick aufs Ganze hätte das Projekt noch erfolgreicher gemacht als es trotz dieses Missstandes war.

Entscheidungen

Entscheidungen in agilen Projekten zeichnen sich dadurch aus, dass sie bis zum spätestmöglichen Augenblick hinausgezögert werden. Wird die Entscheidung ohne Notwendigkeit früher gefällt, stehen möglicherweise noch nicht alle Informationen zur Verfügung. Zu dem späteren Zeitpunkt, zu dem die Entscheidung wirklich Auswirkungen hat, können auch andere Ereignisse hinzugekommen sein, die die frühe Entscheidung hinfällig machen. Dies steht aber nicht im Widerspruch dazu, so früh es geht eine Software bereitzustellen. Das Feedback des Kunden ist elementar und beeinflusst die zukünftigen Entscheidungen direkt. Entscheidungen werden aber auch Just-in-Time gefällt, genauso wie

Software Just-in-Time entwickelt und bereitgestellt wird. Eine Software, die erst in Wochen gebraucht wird, muss heute noch nicht produktiv gehen. Es mag andere Funktionen geben, die viel dringender sind und bereits heute genutzt werden müssen. Werden nicht diese implementiert, sondern die zukünftigen Anforderungen, dann ist das Verschwendung und zeugt von einer falschen Wertanalyse.

Der agile Prozess ist dadurch gekennzeichnet, dass die Anforderungen des Kunden genau identifiziert werden, ausschließlich Aktivitäten durchgeführt werden, die Wert schaffen, und Überflüssiges in der Software und vor allem im Entwicklungsprozess vermieden wird, damit schnell und mit hoher Qualität geliefert werden kann. Es sind jederzeit Entscheidungen notwendig. Jeder Entwickler fällt in kürzesten Abständen Entscheidungen, wie er etwas umsetzt. Die Entscheidung muss zum richtigen Zeitpunkt gefällt werden, also nicht zu früh und nicht zu spät.

Die größten Verschwender

Wir haben die fünf wesentlichen Bausteine erfolgreicher agiler Entwicklung gesehen. Werfen wir nun noch einmal einen Blick auf die häufigsten Fallen der Ressourcenverschwendung.

Die größten Verschwendungen können in folgende Bereiche unterteilt werden:

Unfertige Arbeit
: Unfertige Arbeit kann die Form von Dokumentationen, Spezifikationen und anderen Artefakten haben, zu denen es keinen Quellcode gibt. Nur der Quellcode schafft tatsächlich Wert! Auch unsynchronisierter Code hat keinen großen Wert: Verschiedene Entwicklungszweige (Branches) werden parallel entwickelt, oder der Entwickler führt seine Änderungen nicht regelmäßig dem zentralen Repository zu. Je länger der Code nicht synchronisiert wird, desto schwieriger ist die Synchronisierung. Weiterhin ist nicht getesteter Code überflüssig: Nur getesteter und integrierter Code steigert den Wert. Code soll direkt bei der Programmierung dokumentiert werden, nicht in einem späteren Schritt. Schließlich ist Code, der nicht schnell bzw. rechtzeitig dem Kunden bereitgestellt wird, wertloser Code.

Zusätzliche Features
: Agile Projekte zeichnen sich dadurch aus, dass nur die Funktionalität implementiert wird, die zu einem bestimmten Zeitpunkt benötigt wird und Wert schafft. Funktionalität auf Vorrat schafft keinen Wert, sondern steigert nur die Komplexität und Fehleranfälligkeit.

Erneutes Lernen
: Sich erneut in Situationen hineinzuversetzen und sich Erkenntnisse erneut anzueignen, ist eine große Verschwendung. Das betrifft den Kunden, der eine Entscheidung möglicherweise bereits gefällt hat und nun nochmal vor ihr steht. Es betrifft genauso interne Entscheider und faktisch jeden Entwickler, der sich etwas aneignen muss oder sich in einen Sachverhalt neu einarbeiten muss. Verschwenderisch ist es

auch, wenn Aktivitäten den falschen Leuten aufgetragen oder qualifizierte Personen nicht gefragt und gehört werden. Agile Projekte verhindern das durch ihren gemeinschaftlichen vorausschauenden Umgang mit den Aufgaben.

Übergaben
: Unnötige Übergaben sind Verschwendung von Wissen. Die Person, an die etwas übergeben wird, wird durch die Übergabe nie die Erkenntnisse und Erfahrungen des Übergebenden erhalten. Nur Entwickler, die wirklich an einem speziellen Codeabschnitt gearbeitet haben, besitzen das hundertprozentige Wissen über diese Passage. Geben Sie das Wissen mündlich an jemanden weiter, der wiederum selbst das Wissen weitergibt, ist das Wissen beim endgültigen Empfänger faktisch auf null reduziert.

Aktivitätenwechsel
: Agile Projekte konzentrieren sich auf eine Aktivität zu einer Zeit. Es wird nicht gleichzeitig an mehreren Tasks oder gar mehreren Projekten gearbeitet. Der Entwickler arbeitet immer nur an einem Task auf einmal. Eine Abarbeitung von mehreren Tasks gleichzeitig dauert durch die Kontextwechsel länger als eine lineare Abarbeitung. Eine Ausnahme kann sein, wenn es bei der Abarbeitung eines Tasks zu Wartezeiten kommt und der Entwickler auf den zweiten Task wechseln kann. In der agilen Praxis zeugt das allerdings davon, dass der Task (der die Wartezeit verursacht) zu groß ist. Er könnte in mehrere kleinere Tasks aufgeteilt werden.

Verzögerungen
: Jede Art von Verzögerung mindert den Wert und ist Verschwendung. Ein agiles Team arbeitet räumlich eng zusammen und tauscht sich kontinuierlich aus. Auf diese Weise werden permanent Entscheidungen gefällt, ohne dass es zu Verzögerungen kommt. Jeder im Team kann Entscheidungen fällen. Innerhalb des Teams existiert kein Flaschenhals, der das Tempo vermindert (beispielsweise ein disziplinarischer Vorgesetzter, der bei allen Entscheidungen des Teams mitreden will). Wissen ist genau dann gefragt, wenn es gebraucht wird – nicht zu früh, denn dann könnte es noch einmal geändert werden müssen, und nicht zu spät, dann könnte es nicht mehr aktuell sein.

Defekte
: Defekte werden nicht nur gefunden, sie werden durch umfangreiches Testen verhindert. Unit Tests und Akzeptanztests sind Spezifikationen und treiben die eigentliche Entwicklung vor sich her. Mit einem Test-First-Ansatz und häufigen Integrationen werden gegen Ende des Projekts keine Fehler mehr gefunden. Traditionelle Projekte haben gegen Ende die meisten Probleme und Fehler, agile Projekte haben gegen Ende ein lauffähiges, fehlerfreies System.

Nun sollten die Stützpfeiler agiler Entwicklung und ihre natürlichen Feinde einigermaßen klar geworden sein. Um die allgemeinen Ausführungen dieses Kapitels mit Leben zu füllen, werden wir im folgenden Kapitel sehen, wie agile Projekte konkret durchgeführt werden.

KAPITEL 5

Die Durchführung

In den letzten Kapiteln haben wir uns ausführlich damit beschäftigt, was agile Entwicklung ausmacht. Aufbauend darauf können wir nun das agile Vorgehen exemplarisch untersuchen: In diesem Kapitel gehen wir die Schritte eines Projekts chronologisch durch. Es beginnt mit der Aufbauorganisation und geht über das eigentliche Release, dessen Iterationen und die jeweiligen technischen Tasks bis hin zu einer Besprechung der Möglichkeiten, ein agiles Projekt zu verfolgen (»zu tracken«). Dabei wird Ihnen vieles aus den letzten Kapiteln bekannt sein und sich Ihr Bild vom konkreten agilen Vorgehen festigen. Nach dem Lesen dieses Kapitels haben Sie ein fundiertes Bild davon, wie Sie in Ihrem Projekt ein Release agil durchführen können. Nehmen Sie das hier vorgestellte Vorgehen als Schablone und passen Sie es individuell an Ihr Projekt an.

Aufbauorganisation

Zu Beginn setzen Sie das Projekt so auf, dass die wesentlichen Rollen abgebildet sind. Verankern Sie ein Verständnis dieser Rollen im ganzen Team. Das Team wird vom Coach geschult, der ebenfalls beim Aufbau der Struktur behilflich ist. Folgende Rollen gibt es:

Coach
: Ein Experte der agilen Softwareentwicklung, der über einen definierten Zeitraum das agile Team aufsetzt, die agilen Werte vermittelt und ein Projekt begleitet. Er greift steuernd ein, wenn das Team vom Kurs abkommt oder vor Schwierigkeiten steht. Situationsabhängig entwickelt er auch aktiv an der Software mit, um beispielsweise die korrekte Nutzung von Unit-Tests praktisch zu demonstrieren. Der Coach kann der Architekt/Lead Developer oder der Projektmanager sein, ist aber häufig eine externe, am freien Markt erworbene, temporäre Unterstützung.

Product Owner
: Der Product Owner hat die Verantwortlichkeit, den Wert des entwickelten Produkts zu maximieren. Er schreibt die User Stories und entscheidet, welche Stories in welcher Iteration implementiert werden. Er muss den Nutzen der Stories bestimmen und die Prioritäten verteilen. Der Product Owner ist gewöhnlich der Kunde.

Entwickler

Der Entwickler (auch Programmierer genannt) ist für die Umsetzung der User Stories verantwortlich. Er unterstützt gemeinschaftlich mit den anderen Entwicklern den Product Owner beim Verfassen der User Stories, schätzt die Aufwände der Realisierung ab, schreibt die Unit-Tests und hilft dem Product Owner beim Verfassen der Akzeptanztests. Ein Entwickler kann ein Datenbankexperte sein, ein Testexperte, ein Architekt, oder er kann andere Rollen übernehmen, die bei traditionellen Projekten explizit als solche aufgeführt werden.

Architekt/Lead Developer

Der Architekt ist häufig ein Senior-Entwickler und insbesondere Ansprechpartner zu allen Fragen rund um die Architektur der Software für das Management und/oder den Product Owner. Er hat immer den Überblick über die Gesamtarchitektur der Software. Diese Person ist entweder bereits in Ihrer Organisation vorhanden (als ausgewiesener Architekt im Team) oder bildet sich aus dem Team heraus. Die geeignete Person ist häufig eine Führungspersönlichkeit und wird sich ganz von allein herauskristallisieren. Mag es in kleinen Teams von bis zu zwölf Personen noch ohne diesen Architekten funktionieren, werden Sie in größeren Projekten auf seinen Einsatz nicht verzichten können. Der Architekt entwickelt auch selbst.

Boss

Der disziplinarische Vorgesetzte der Entwickler hält sie bei Laune. Er sorgt für gute Rahmenbedingungen und hält den Beteiligten organisatorisch den Rücken frei.

Build Manager

Der Build Manager sorgt für den Auf- und Ausbau der Infrastruktur, um die Software kontinuierlich zu erstellen und zu testen und am Ende bereitstellen zu können. Er ist ein Entwickler, aber nicht immer derselbe: In vorbestimmten Intervallen wechselt die Aufgabe, und ein anderer Entwickler kümmert sich um die Builds. Wechseln Sie nicht, entsteht eventuell eine Wissensinsel!

Projektmanager

Er konzentriert sich auf die Einhaltung der agilen Prinzipien. Obwohl die Aufgaben des Projektmanagements in agilen Projekten von allen Beteiligten zugleich wahrgenommen werden, hat dieser eine Mitarbeiter doch die besondere Pflicht, den Fortschritt des Projekts kontinuierlich zu verfolgen. So aktualisiert er täglich gut sichtbare Grafiken, die über den aktuellen Fortschritt und die noch offene Arbeit an einer Iteration informieren. Falls ein Ticketing-System eingesetzt wird, verwaltet der Projektmanager es. Falls Sie ein Wiki haben, wird auch es vom Projektmanager administriert (er verwaltet förmlich das Wissen). Es ist gewöhnlich ein *technischer* Projektmanager oder Teilprojektleiter, der einem übergeordneten Projektleiter berichtet.

Nutzer

Der Nutzer der Software. Das kann, muss aber nicht, der Product Owner sein.

Team

Das Team besteht aus allen oben angeführten Rollen.

Das Release

Das Release ist die größte Einheit eines Projekts. Es umfasst eine Summe von Softwarefunktionen (Features) und die damit verbundenen Arbeitspakete und Aktivitäten.

Die Release-Planung

Nach Abschluss eines Releases soll ein zu entwickelndes Produkt einen gewissen Stand von Geschlossenheit in sich erreicht haben. Es hat die Funktionalität, die es als Produkt komplett wirken lässt. Der Product Owner ist maßgeblich für die Release-Größe verantwortlich und wird bei der Bestimmung der richtigen Größe vom ganzen Team unterstützt.

Ein Release umfasst normalerweise einen Zeitraum von etwa drei bis sechs Monaten. Neben der Bereitstellung eines Produkts in einer vollständigen Funktionalität wird ein Release auch als strategische Planungseinheit herangezogen. Organisationen können so sagen: »Die Entwicklung des Produkts ABC dauert X Monate. Dann ist das Produkt (Release) fertig.« Nehmen wir an, unser Produkt soll in drei Monaten abgeschlossen sein. Diese Laufzeit ist dann eine fest vorgegebene Größe.

Stories

Innerhalb dieser Laufzeit werden die Features umgesetzt. Diese werden vom Product Owner in Form von User Stories geschrieben und haben die in Abbildung 5-1 gezeigte Form. Eine Story beschreibt kurz, *wer* etwas mit der Software macht, um *was* zu erreichen.

Die dargestellte Tabelle ist als Schablone zu sehen, die bei Ihrem Projekt durchaus abweichen kann. Neben einer Durchnummerierung in der ersten Spalte umfasst die Übersicht gewöhnlich die User Stories in kompakter Formulierung und ihre Aufwandsabschätzung in Story Points. Ich gehe etwas später noch genauer darauf ein.

Story No.	Story	Abschätzung (Story Points)
1	Als Arzt kann ich alle Patienten sehen, die ich am Tag habe.	3
2	Als Arzt kann ich über die Gesundheitsgeschichte meiner Patienten Auskunft geben.	5
3	Als Assistent kann ich einem Patienten einen Termin geben.	2
4	Als Assistent kann ich einem Patienten eine Verschreibung ausdrucken.	1

Abbildung 5-1: User Stories mit Aufwandsabschätzung in Story Points

Zu Beginn sind die User Stories noch recht grobkörnig und behandeln keine Details. Sie beschreiben, wer etwas mit der Software wie machen kann, um etwas zu erreichen. Es sind keine umfassenden Spezifikationen, schon gar nicht zu Beginn des Releases, es sind Erinnerungskrücken. Wenn Sie im Laufe der Iterationen tatsächlich umgesetzt werden, werden daraus einzelne technische Tasks abgeleitet. Um die Abschätzung kümmern wir uns ein wenig später.

Iterationen

Iterationen sind die nach dem Release nächstkleinere Planungsgröße. Das Projekt schreitet das Release nicht im Ganzen ab, dafür ist die Zeitspanne zu groß. Da wir kleinere Aktivitäten- bzw. Funktionspakete haben wollen, um dem Kunden früher Feedback zu geben und durch die schnellere Lieferung die Qualität zu erhöhen und Kosten zu senken, teilen wir das Release in Iterationen auf. Die Wahl einer Länge für die Iterationen hängt von vielen Faktoren ab:

1. Wie lang ist das Release?
2. Wie groß ist die Unsicherheit (technisch, fachlich, organisatorisch)?
3. Wie leicht kann Feedback vom Kunden eingeholt werden?
4. Wie lange bleiben Prioritäten unverändert?
5. Wie lange möchte man ohne Kunden-Feedback leben?
6. Wie hoch ist der Overhead einer Iteration?

Generell lässt sich sagen: Je kürzer das Release, desto kürzer auch die Iterationen. Da die Software am Ende einer Iteration einen konsistenten, in sich geschlossenen Stand haben soll, der dem Kunden vorgestellt werden kann, bestimmt sich die Wahl der Länge der Iteration unter anderem durch die Frage, wie häufig der Kunde einen Stand sehen kann oder will. Je kürzer der Abstand desto besser. Kurze Intervalle sind allerdings nicht immer möglich.

Nehmen wir zum Beispiel ein Release, das zwei Monate dauert. Würden wir hier eine Iterationslänge von einem Monat wählen, hätten wir nur einmal im Lauf des Releases die Gelegenheit, einen stabilen Zwischenstand zu präsentieren. Das wäre viel zu wenig. Hier wären vier oder mehr Iterationen optimal.

Ein weiterer wichtiger Punkt ist, wie lange die Stories und die Prioritäten stabil bleiben. Während einer Iteration sollen ausgewählte Stories umgesetzt werden. Die Iteration wird dabei inhaltlich nicht geändert, es werden beispielsweise keine neuen Stories aufgenommen. Ist die Iterationslänge zu lang gewählt, so dass sich regelmäßig in der Mitte der Iteration die Prioritäten ändern, muss also über die Iterationslänge nachgedacht und sie verkürzt werden.

Software soll ja in der Regel unter Einhaltung einer hohen Qualität schnellstmöglich ausgeliefert werden. Je mehr Unwägbarkeiten ein Projekt hat, und je unklarer die Anforderungen oder die technische Umsetzung, desto mehr Iterationen sind anzustreben. Viele kurze Iterationen bieten die Chance, häufig steuernd einzugreifen und die Planung zu verfeinern.

Auch die Messung des Fortschritts und der Team-Performance ist am besten zum Ende einer Iteration möglich. Ebenso können Retrospektiven am Ende einer Iteration gehalten werden, um die Arbeit des Teams im Lauf einer Iteration zu durchleuchten und für spätere Iterationen Verbesserungsschritte aufzusetzen.

Wir wählen im vorliegenden Fall eine Länge von zwei Wochen pro Iteration. Wir haben also sechs Iterationen. Nun müssen wir uns überlegen, wie viel Funktionalität das Team in den sechs Iterationen umsetzen kann. Dazu betrachten wir die *Velocity*.

Velocity

Als Velocity wird der Durchsatz des Teams bezeichnet, also wie viele Story Points es in einer Iteration abarbeiten kann. Ist das Team einmal in Schwung, ist die Velocity recht konstant, Schwankungen sind aber auch ganz normal. Mit Hilfe der Velocity bestimmen Sie bei der Release- und Iterationsplanung die Anzahl der Stories, die Sie umsetzen wollen. Doch wie kommen Sie zu dieser Velocity? Wie hoch ist sie? Generell haben Sie drei Möglichkeiten, die Velocity zu bestimmen:

Nutzen Sie entweder die Möglichkeit, frühzeitig Iterationen zu durchlaufen, um aussagekräftige Werte zu bekommen. Wenn das nicht möglich ist oder Ihr Team bereits einige Iterationen unter gleichen Umständen durchlaufen hat, dann ziehen Sie Werte aus der Vergangenheit heran. Wenn auch das nicht möglich ist, schätzen Sie und wählen Sie dabei ein Intervall, das im Lauf der Iterationen immer kleiner wird. Auf jeden Fall sollten Sie frühzeitig mit Iterationen beginnen und Erfahrungen sammeln, anstatt umfangreiche Dokumente verfassen, die eine Sicherheit in der Unsicherheit vorgaukeln und das ganze Release vorhersehen wollen. Die Velocity finden Sie nur durch praktische Iterationen heraus! Die drei Ansätze, die Velocity zu bestimmen, sehen wir uns im Folgenden kurz an.

Aus der Erfahrung mit durchgeführten Iterationen

Wenn Sie bereits einige Iterationen durchlaufen haben und sich wenig an den Rahmenbedingungen geändert hat, können Sie auf Ihre bisherige Erfahrung zurückblicken. Sie nehmen dabei, bildlich gesprochen, das Wetter von gestern als Grundlage für die *Vorhersage* des Wetters von morgen. Sobald sich allerdings die Teamzusammensetzung ändert, Anforderungen oder technische Lösungen unklar werden, neue Werkzeuge zum Einsatz kommen oder der Kunde wechselt, nimmt die Aussagekraft des alten Werts ab. Dennoch ist er ein Anhaltspunkt. Je größer die Unsicherheit, desto höher wählen Sie die Zahl der Iterationen. Je mehr Iterationen Sie durchlaufen, desto aussagekräftiger sind die Werte, die Sie ermitteln können.

Eine Iteration durchführen

Die zweite Möglichkeit ist, eine (oder mehrere) Iterationen zu durchlaufen, um Ihre Velocity zu messen. Häufig beginnen Projekte damit, die Anforderungen umfangreich zu spezifizieren, ein Design zu entwerfen, Dokumente zu erstellen usw. Nutzen Sie diese Zeit, um bereits eine oder mehrere konkrete Iterationen zu durchlaufen, an deren Ende ein vorzeigbares Ergebnis steht.

Wählen Sie die priorisierten Stories eine nach der anderen aus und arbeiten Sie so viele Stories in der ersten Iteration ab, wie sie schaffen. Bereits mit der zweiten Iteration haben Sie so einen recht guten Referenzwert. Manchmal liegen noch keine User Stories vor oder Sie müssen noch die technische Infrastruktur aufbauen. Häufig bietet sich dann die *Iteration 0* an: Bevor die eigentlichen Umsetzungen beginnen, führen Sie grundlegende technische Arbeiten durch – sie legen eine Projektstruktur im Versionskontrollsystem an, erstellen für das neue Projekt ein Wiki und kümmern sich um andere Punkte, die wir im weiteren Verlauf noch genau beschreiben werden.

Eine Vorhersage machen

Manchmal haben Sie nicht die Gelegenheit, erste Iterationen durchzuführen. Vielleicht möchte der Kunde sofort einen Vertrag unterschrieben haben, oder das Projekt beginnt erst in der Zukunft. Velocity vorherzusagen ist sicherlich die Option, die Sie als letzte wählen sollten, da sie mit vielen Unsicherheiten verbunden ist. Sie können eine Vorhersage machen, indem Sie die Stories sehr klein halten und frühzeitig in feinkörnige, technische Tasks aufteilen. Das machen wir gewöhnlich erst später, wenn wir konkrete Stories in der Iteration umsetzen. Wenn Sie einzelne Tasks in kleine technische Aktivitäten zerlegt und für diese konkrete Abschätzungen in realen Stunden und Tagen durchgeführt haben, können Sie das wieder auf Story Points der Stories zurückrechnen.

Abschätzung aller Stories

Stories werden gewöhnlich in Story Points abgeschätzt. Das ist ein immanentes Merkmal der agilen Entwicklung. Es handelt sich dabei um relative Abschätzungen, deren absolute Werte keine direkte Bedeutung haben. Es geht hier ausschließlich um die Gewichtung untereinander, so könnte man auch den Begriff Story Points durch »*Gummibärchen*« ersetzen. Es geht nur darum auszudrücken, ob eine bestimmte Story größer oder kleiner ist als eine andere (also ihr Implementierungsaufwand größer oder kleiner ist als bei einer anderen). Wichtig ist dabei auch, dass der konkrete Wert, also die Skalierung, nicht entscheidend ist. Eine User Story mit zehn Story Points sollte doppelt so groß sein wie eine Story mit fünf Punkten. Für die Umsetzung sind dann Stories mit Abschätzungen von einem bis zehn Punkten brauchbar. Größere Stories müssen Sie also mit der Zeit in kleinere aufsplitten. Doch wie wählen Sie die Story Points für die erste Story, wenn Sie mit den Abschätzungen starten? Dafür haben sich zwei Möglichkeiten etabliert.

Mit der kleinsten Story beginnen

Die erste Möglichkeit, um die User Stories abzuschätzen, ist mit der kleinsten Story zu beginnen. Sie suchen sich die Story heraus, von der Sie sicher sind, dass ihr Umfang der kleinste ist. Dieser Story geben sie einen Punkt. Alle weiteren setzen Sie zu ihr und dann untereinander in Beziehung.

Mit einer mittelgroßen Story beginnen

Sie wählen eine Story, von der Sie meinen, dass sie mittelgroß ist. Alle weiteren Stories setzen Sie dann zu ihr in Beziehung. Auch hier müssen Sie keine Angst haben: Sie können jederzeit Abschätzungen revidieren, falls Sie neue Erkenntnisse zu dieser Story gesammelt haben.

Das Planungsspiel

Wie kommt das Team nun zu den Abschätzungen? Es wird ein Planungsspiel, häufig auch als »Planning Poker« bezeichnet, durchgeführt, an dem die Entwickler und der Kunde teilnehmen. Das Vorgehen ist teilstrukturiert: Es folgt zwar klaren Regeln, ist aber dennoch recht locker aufgebaut. Der Kunde kommt zusammen mit dem Team *spielerisch* zu den Abschätzungen. Auf diese Weise wird etwas Druck und Anspannung herausgenommen, die gewöhnlich mit Aufwandsabschätzungen verbunden sind, und eine emotionale Distanz erreicht. Das ganze Team ist gefordert: Neben dem Kunden sind alle Entwickler beteiligt. Der Projektmanager moderiert und koordiniert das Spiel.

Das Planungsspiel kann entweder auf *Story Driven Commitment* oder *Date Driven Commitment* basieren.

Story Driven Commitment
: Der Kunde legt die für das nächste Release angestrebten Stories vor. Jede Story wird von ihm vorgestellt und anschließend von den Entwicklern, wie im Folgenden beschrieben, abgeschätzt. Sobald eine Anzahl Stories erreicht ist, die in diesem Umfang und der Kombination mit den Abschätzungen ein Release rechtfertigen, wird der Prozess gestoppt, und das Release ist geplant.

Date Driven Commitment
: Der Kunde wählt ein Release-Datum. Dies ist nun fix. Die Entwickler errechnen die Velocity für das Release, und schätzen, zusammen mit dem Kunden, die Stories ab. Der Kunde wählt für das Release die Stories aus, die für ihn den größten Wert darstellen, aufbauend auf den Abschätzungen für die einzelnen Stories. Dabei haben die Stories unterschiedliche Prioritäten. Der Kunde platziert also so viele Stories in das Release, wie es an Story Points aufnehmen kann. Dies ist das bevorzugte Vorgehen, untertreicht es doch das so wichtige *Time-Boxing*.

  Zu Beginn erklärt der Kunde zunächst die Stories einzeln und schreibt sie auch einzeln auf Karten, die *Story Cards*. Jede Story wird mit dem Team diskutiert, so dass sich ein gemeinsames Verständnis der anstehenden Aufgaben entwickelt. Die für die Realisierung der Story notwendigen Arbeiten werden grob besprochen. Anschließend schätzt jeder Entwickler die Story ab (den Aufwand zur Realisierung). Dazu gibt es vorgefertigte Karten, die der Entwickler verdeckt vor sich hält. Auf einer Karte steht jeweils eine Anzahl Story Points. Bei der recht groben Release-Planung können Story Cards mit 1, 2, 3, 4, 5, 8, 10, 15, 30 und 50 Story Points bereitgestellt werden. Mit zunehmendem Wert wird der Abstand zwischen den Points größer. Je größer das Feature, desto mehr Unwägbarkeiten sind damit verbunden, und desto

schlechter sind Abschätzungen möglich. Können Sie einzelne Stories nicht abschätzen oder sind die Stories zu groß, teilen Sie sie in mehrere kleinere auf. Ist das Feature zunächst nicht wirklich ausgestaltet oder wollen Sie einen Überblick gewinnen, so belassen Sie es zunächst bei den großen Stories.

Nachdem sich jeder Entwickler für eine Story-Abschätzung entschieden hat, drehen alle ihre Karten um. Auch hier gibt es nun verschiedene Möglichkeiten, wie Sie fortfahren können. Bei manchen Projekten ist es sehr effektiv, die Abschätzungen offen zu diskutieren und dann direkt zu einer abschließenden gemeinsamen Abschätzung zu kommen. Andere Projekte bevorzugen es, die beiden am stärksten nach oben und unten abweichenden Abschätzungen herauszunehmen (minimaler und maximaler Wert) und die beiden Entwickler zu fragen, was zu diesem Wert geführt hat. Im Rahmen dieser »Rechtfertigungen« (sehen Sie es spielerisch, diskutieren sie sachlich darüber) bekommen die beiden »Ausweichler« oder auch andere Entwickler neue Erkenntnisse. Typische Aussagen wären »Ich habe so viel geschätzt, weil wir hier noch xxx machen müssen« oder »Ich habe so wenig geschätzt, weil wir eine ähnliche Story bereits umgesetzt haben. Komponente XXX können wir für diese Story wiederverwenden.« Sie können nun in der offenen Diskussion zu einer gemeinsamen Abschätzung kommen und den Kunden jederzeit fragen, wenn etwas unklar ist. Wenn Sie ein, zwei Entwickler haben, die gern das Heft an sich reißen und dominant in der Diskussion ihre Werte durchsetzen, dann führen Sie, aufbauend auf der Diskussion, eine zweite Pokerrunde durch. Sie werden sehen, dass sich nun die Werte angeglichen haben und ein Konsens schnell erreicht werden kann. Wie auch immer Sie vorgehen, am Ende muss eine Abschätzung stehen, die von allen getragen wird.

Ist die vorläufige Abschätzung (vorläufig, weil sie sich eventuell noch ändern wird) gemeinschaftlich verabschiedet, identifizieren sich alle Beteiligten mit diesem Wert. Alle Beteiligten haben sich schließlich bei der Findung aktiv eingebracht. Auf der Story Card wird nun die Abschätzung eingetragen und die Karte an eine Wand gehängt. Dies machen Sie mit allen Stories so.

Der Kunde weiß von den Entwicklern, wie viele Story Points sie in dem Release abarbeiten können (die Velocity ist bestimmt worden). Entsprechend wählt er sich die Stories aus, die für ihn besonders wichtig sind. Generell müssen alle Stories einen Wert mit in das Produkt bringen, wobei dieser Wert unterschiedlich sein kann und das damit verbundene Risiko nicht unberücksichtigt bleiben darf. »Risiko« kann hier beispielsweise ein betriebswirtschaftliches Risiko sein: Die Story muss schnellstmöglich implementiert werden, um einen Wettbewerbsvorteil halten zu können. Beim technischen Risiko kann es anders aussehen: Umfangreiche technische Aktivitäten, Betreten von technischem Neuland oder eine Machbarkeitsstudie sollten schnellstmöglich angegangen werden, um zeitnah Feedback (von der Software selbst) erhalten zu können. Die ausgewählten Stories landen im *Product Backlog*, dem Themenspeicher (Feature-Speicher) für das Release.

Die einem Release zugewiesenen Stories sollten fix bleiben. Das Team konzentriert sich darauf, in der festgeschriebenen Zeit ein Produkt bereitzustellen, das all die für das Release vorgesehenen Stories erfüllt.

Während des Release oder im Anschluss ergeben sich natürlich schon neue Stories. Diese werden aber in einem zweiten Speicher gehalten und erst nach Abschluss des Release bei der nächsten Release-Planung Berücksichtigung finden. Doch auch hier gibt es Ereignisse, die ein Abweichen begründen. Beispielsweise können schon frühzeitig bereitgestellte Stories einen immensen Wettbewerbsvorteil mit sich bringen. Haben Sie wirklich einen signifikanten Marktvorteil, wenn Sie schnell die Stories auswählen und umsetzen können? Wir haben schon besprochen, dass bei Lean Development Geschwindigkeit ein Schlüsselfaktor ist. Ist das in Ihrem Projekt von drastischer Bedeutung, könnten Sie auch darüber nachdenken, die Release-Planung komplett wegzulassen und direkt mit den Iterationen zu beginnen. Vor jeder kurzen Iteration wählen Sie die dann die Stories, die den höchsten Wert bringen, aus und setzen sie um. Die Regeln sind die gleichen: Während der Iteration wird der Inhalt nicht geändert.

Haben Sie allerdings festgestellt, dass für Sie kein engerer, sondern ein weiterer Planungshorizont besser ist, dann nutzen Sie Releases. Das ist der weit häufigere Fall bei Software, die einen größeren Umfang hat. Auch aus strategischen Gründen könnte eher ein längeres Intervall genommen werden, oder Sie möchten die sich in kurzen Zeitspannen ändernden Anforderungen stabilisieren, um von einer chaotischen Realisierung auf Zuruf wegzukommen.

Abschätzung ist inhärente Unsicherheit

Vergessen Sie nicht, dass eine Abschätzung im wahrsten Sinne des Wortes eine Abschätzung ist. So offensichtlich das ist, so wenig wird es verstanden. Niemand ist Hellseher und kann den genauen Aufwand bei der Erstellung einer komplexen Software mit vielen Beteiligten und unzähligen Wechselwirkungen vorhersagen.

Erst mit der Entwicklung entsteht das genaue Wissen über die Anwendung, das vorher noch nicht da war. Sie können Abschätzungen möglichst genau hinbekommen, indem Sie beispielsweise die abzuschätzenden Aktivitäten möglichst klein halten oder ein Planungsspiel durchführen. Auch generelle und ganz spezielle Erfahrung hat positiven Einfluss auf die Abschätzung. Ich habe einmal in einem Projekt gearbeitet, in dem der Entwickler förmlich an den Pranger gestellt wurde, wenn eine Abschätzung nicht korrekt war. Das hat teilweise zu richtigen Glaubenskriegen geführt.

Eine Diskussion ist sicherlich in Ordnung. Andere Vorstellungen von den Aufwandsabschätzungen emotional aufzunehmen, ist aber kein Zeichen von Vertrauen. Führen Sie ein Planungsspiel durch, um die Spannung herauszunehmen und die Abschätzungen über das Team zu harmonisieren. Hat das ganze Team sich auf eine Abschätzung geeinigt, kann ein einzelner Entwickler nicht mehr an den Pranger gestellt werden.

→

Abschätzungen hängen ab von

- der Größe der Stories,
- der Erfahrung bei der Abgabe von Abschätzungen,
- der technischen Erfahrung,
- der Erfahrung mit dem Planungsspiel,
- dem Kunden (wie er sich einbringt) und
- der Vollständigkeit und Verlässlichkeit der Informationen.

Abschätzungen werden über die Zeit immer genauer. Verzweifeln Sie nicht, wenn erste Abschätzungen das Ziel weit verfehlen. Es ist durchaus üblich, dass zu Beginn eine Abschätzung um den Faktor 5 daneben liegen kann.

Warum klappen Planungsspiele so gut? Ein Planungsspiel verknüpft viele unterschiedliche erfolgreiche Praktiken, um zu guten Aufwandsabschätzungen zu kommen. Dazu gehören vor allem Expertenmeinungen, Analogien und Disaggregation in multifunktionalen Teams.[1] Folgende Vorteile dieses Vorgehens lassen sich zusammentragen:

- Abschätzungsabweichungen werden identifiziert.
- Spezifikations- und Verständnislücken werden frühzeitig aufgedeckt.
- Entwickler können direkt den Kunden (und sich gegenseitig) fragen.
- Die Gruppe ist multifunktional, es wird also Wissen aus verschiedenen Disziplinen eingebracht.
- Das Vorgehen ist spielerisch, die »Last« der Abschätzung wird herausgenommen.
- Es werden mehr Informationen geteilt.
- Es sind mehr Entwickler an den Abschätzungen beteiligt.
- Es wird Team Work angeregt.
- Das Team fühlt »collective ownership«.
- Das Wissen verschiedener Quellen wird kombiniert.
- Es wird verhindert, dass Wissen nur von einem Senior kommt.
- Juniors werden geschult (indirekte Weiterbildung).
- Offensichtlich falsche Abschätzungen werden abgefangen.
- Alle Beteiligten werden in Bezug auf Annahmen und Aktivitäten synchronisiert.
- Jeder Einzelne hat bei der Abschätzung gewisse Unabhängigkeit.
- Es handelt sich um eine kombinierte Delphi-Methode.
- Designdiskussionen werden früh geführt.

1 Disaggregation ist das gezielte Auflösen von Allianzen und verhärteten Strukturen in kleinere Einheiten.

Nun haben wir eine Abschätzung aller User Stories. Mit einer beispielhaft abgeschätzten Velocity von acht Story Points pro Iteration kommen wir in unserem Projekt auf sechs Iterationen mal acht Story Points pro Iteration, also 48 Story Points im ganzen Release.

Auswahl der Stories für das Release

Nun wissen wir, dass wir in dem Release 48 Story Points umsetzen können, und wie viele Story Points die einzelnen Features ausmachen. Der Kunde kann nun aus allen Features diejenigen aussuchen, die in der Summe einen Umfang von 48 Points haben und für ihn den größten Wert ausmachen. Zusammen mit den Entwicklern stellt der Kunde also nun das Release zusammen. Die Entwickler geben jederzeit Rückmeldung, wenn sie meinen, dass eine Story von einer anderen abhängig ist, also vor oder zeitgleich mit einer anderen umgesetzt werden sollte. Wenn nun die Summe der Story Points um ein, zwei Einheiten über der Velocity liegt, können Sie dies so hinnehmen, wohl wissend, dass es sich sowieso nur um Abschätzungen handelt. Oder, und das ist meine Empfehlung, Sie nehmen eine Story heraus bzw. wechseln die Stories, so dass sie definitiv unterhalb des theoretischen Werts rauskommen.

Der Release-Plan wird regelmäßig aktualisiert. Wenn Sie bereits mehrere agile Projekte erfolgreich zu Ende gebracht haben, wird sich der anfängliche Release-Plan im Laufe der Iterationen vielleicht nicht ändern. Es ist aber auch durchaus normal, wenn Sie nach jeder Iteration den Release Plan überarbeiten und anpassen.

Die Iteration

Die Iteration ist die nächstkleinere Einheit nach dem Release. Genauso wie ein Release beginnt die Iteration mit einer Planungsrunde. Eine Iteration besteht aus in dieser Iteration zu entwickelnden User Stories, die wiederum aus einzelnen technischen Arbeitspaketen (Tasks) bestehen.

Iterationsplanung

Zusammen mit dem Kunden und allen Entwicklern werden also ein Iterationsziel festgelegt und mit seiner Hilfe die umzusetzenden Stories ausgewählt. Dabei soll jede Iteration einen lieferbaren, in sich konsistenten Zwischenstand bereitstellen! Die Iterationsplanung ist mit der Release-Planung vergleichbar, allerdings sind der Funktionsumfang und der zeitliche Rahmen kleiner.

Tasks

Aufbauend auf der Auswahl der Stories werden einzelne technische Tasks identifiziert. Die Entwickler schätzen diese Tasks gemeinschaftlich ab. Das geschieht unabhängig davon, wer einen spezifischen Task umsetzen wird, was zu diesem Zeitpunkt schließlich auch noch nicht feststeht. Entsprechend werden alle Teilnehmer in die Abschätzung mit einbezogen. Hier kann erneut ein Planungsspiel durchgeführt werden.

Tasks werden in sogenannter *Idealzeit* geschätzt, also in echten Stunden. Diese Schätzung wird immer wieder mit der tatsächlich gemessenen *Realzeit* abgeglichen, so dass die Planung genauer wird. Für jede Abschätzung ist eine Größenordnung von wenigen Stunden anzustreben. Ist die Abschätzung für einen Task zu groß, zum Beispiel größer als ein Tag, wird der Task in kleinere aufgeteilt. Die Tasks werden auf separaten *Task Cards* erfasst. Die Entwickler arbeiten zwar weiter mit den zu Beginn erstellten Story Cards, die aber lediglich als Erinnerung und Planungsinstrument dienen. Der Aufwand in Stunden wird auf den jeweiligen Task Cards hinzugefügt. Auch sie sollen nicht die kompletten, im Detail erläuterten Aktivitäten beinhalten.

Berücksichtigen Sie in den Aufwandsstunden ggf. Ihre Pair Programming-Sitzungen. Wollen zwei Entwickler vier Stunden gemeinsam eine Aufgabe lösen, so beträgt der Aufwand für den Task acht Stunden!

Die Story Cards und die dazugehörigen Task Cards werden an eine für alle gut sichtbare Wand geheftet. Dabei werden die Karten gruppiert. Hier hat sich eine Unterteilung zwischen offenen, in Arbeit befindlichen und abgeschlossen Tasks bewährt (siehe Abbildung 5-2). Dabei werden vertikal die Stories (links) und die dazugehörigen Tasks (rechts) mit ihren Aufwandsabschätzungen und horizontal der Bearbeitungsstatus abgebildet. Beachten Sie, dass die Zahlen auf den Task Cards Arbeitsstunden in Idealzeit und nicht Story Points entsprechen.

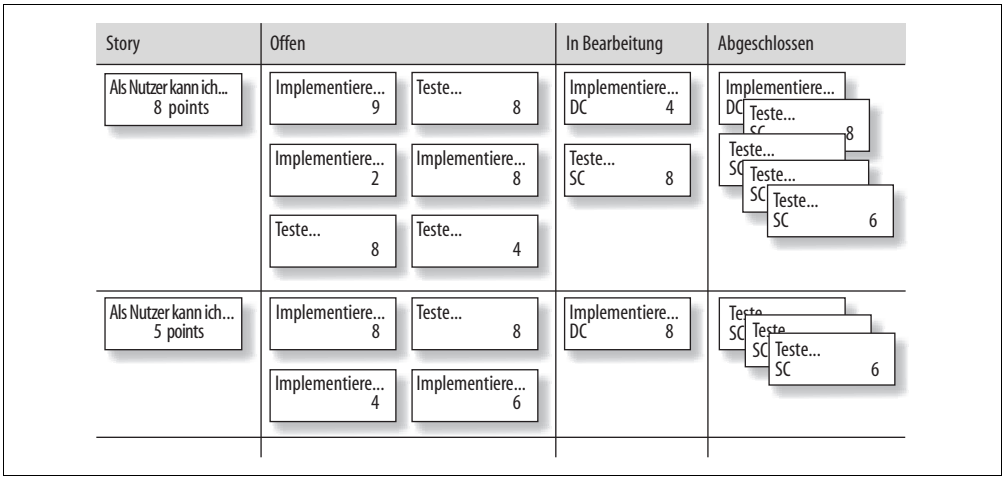


Abbildung 5-2: Gruppierung der Tasks nach Stories und Bearbeitungsstatus

Die Karten sind für alle gut sichtbar angebracht, beispielsweise auf dem zentralen Flur oder in einem zentralen Büro. Jeder kann den Status und den Fortschritt der Iteration verfolgen. Das ist viel greifbarer als Tracking über ein elektronisches Medium. Wenn Sie allerdings in einem räumlich verteilten Projekt arbeiten (also auch wenn innerhalb des Gebäudes die Büros nicht direkt nebeneinander liegen), kann eine elektronische Anwendung eingesetzt

werden. In dieser *Tracking-Datenbank* werden dann die einzelnen Tasks fortgeschrieben, die einzelnen Releases und Iterationen zugewiesen werden können. An der Abschätzung der Tasks und dem weiteren Umgang mit ihnen, beispielsweise der Auswahl durch einen Entwickler, ändert sich dadurch nichts. Dazu später im Kapitel mehr.

Die Umsetzung der Tasks

Die einzelnen, implementierbaren Tasks liegen nun vor und können von den Entwicklern umgesetzt werden. Während der Umsetzung gibt es tägliche *Stand-Up Meetings*. Die Entwicklung bedient sich eines *Konfigurationsmanagments* und geschieht testgetrieben (*Test-Driven Development*). Die Software wird kontinuierlich integriert (*Continuous Integration*) und unter Nutzung von *Standards* und in gemeinschaftlicher Verantwortung stetig verbessert (*Refactorings*). Ich stelle im technischen Teil dieses Buchs Werkzeuge vor, die die bei der Umsetzung notwendigen Schritte unterstützen. Dabei vertiefe ich dann situativ die hier vorgestellten Faktoren.

Entwicklung

Am Anfang oder sobald ein Entwickler einen Task abgeschlossen hat, nimmt er sich den nächsten. Falls mit Prioritäten gearbeitet wird, so berücksichtigt er die Priorität des Tasks und nimmt den Task mit der höchsten Priorität. Dabei entscheidet er selbst, was er als Nächstes bearbeiten wird. Der Task ist erst abgeschlossen, wenn die Tests fertig sind, der fachliche Code umgesetzt wurde, die Tests erfolgreich laufen und der neue Code (sowohl Tests als auch fachlicher Code) erfolgreich ins zentrale *Code Repository* eingespielt und mit dem existierenden Code integriert wurde. Ist der Code eingespielt, die automatischen Integrationstests offenbaren aber einen Defekt am neuen Code, ist der Code noch nicht abgeschlossen. Die Task Card bleibt dann weiterhin bei den offenen Punkten. Der Entwickler arbeitet ausschließlich an einem Task gleichzeitig, bis dieser abgeschlossen ist.

Pair Programming

Für die Erfüllung der Aufgabe kann es hilfreich sein, zusammen mit einem anderen Entwickler im Paar zu arbeiten. Was die konkrete Umsetzung angeht, existieren unterschiedliche Ansätze. Viele agile Projekte führen die Umsetzung ausschließlich in Paaren durch. Andere führen die Paare nur situationsabhängig zusammen. Probieren Sie es so, wie es für Sie am besten passt. Dazu kann es hilfreich sein, zunächst Paare zu bilden. Falls Sie damit mehrfach und nachhaltig Probleme haben, verzichten Sie darauf und beschränken Sie sich auf die natürliche Paarung, die agile Entwickler bei schwierigen Aufgabenstellungen automatisch eingehen. Viele Aufgaben lassen sich auch sehr gut allein abarbeiten. Lassen Sie aber niemals einen Entwickler allein von Anfang bis Ende an einem Task arbeiten und ihn abschließen, sondern sorgen Sie dafür, dass der Quellcode immer noch durch die Hände eines zweiten geht. Im Optimalfall ist das zweite Paar Hände während der gesamten Task-Entwicklung beteiligt. Man unterscheidet zwischen dem Fahrer, der gerade die Tastatur hat und codiert, und dem Beifahrer, der dem Fahrer

zuschaut und das Design im Auge behält. Die Rollen des Fahrers und des Beifahrers wechseln kontinuierlich und situationsabhängig.

Stand-up-Meetings

Das ganze Team synchronisiert sich täglich. Wir wir bereits diskutiert haben, berichtet reihum jeder Teilnehmer, was er seit gestern gemacht hat, welche Schwierigkeiten er dabei hatte, und was er bis morgen machen möchte. Folgende Antimuster sind nicht selten und müssen bei ihrem Auftreten erkannt und bearbeitet werden.

Der gerade Redende spricht zum Manager
: Anstatt zum ganzen Team zu reden, redet er zum Manager. Das ist zwar verständlich, aber nicht Sinn der Sache. Das ganze Team soll über den aktuellen Stand informiert sein.

Personen kommen zu spät
: Das Stand-up muss diszipliniert durchgezogen werden und nach 15 Minuten beendet sein. Deshalb beginnen Sie pünktlich. Warten Sie nicht auf Personen, die zu spät kommen, auch wenn es wichtige Personen sind.

Manchmal fällt das Meeting aus bzw. findet auf Zuruf statt
: Das Meeting findet zwar regelmäßig jeden Tag statt, der Manager verzögert es aber, wenn ihm andere Dinge dazwischenkommen. Die Entwickler warten, bis der Manager endlich lädt, und warten manchmal vergeblich. Sie arbeiten dabei zwar weiter, allerdings nicht in Ruhe.

Der Manager ist der Gastgeber
: Das ganze Team muss Gastgeber sein und sich somit für das Gelingen verantwortlich fühlen.

Personen unterbrechen
: Andere, die nicht Bestandteil des Teams sind, brechen hinein, und stören das Meeting.

Kaffeekränzchen
: Das Team tauscht sich nicht zielorientiert aus, sondern erzählt Persönliches.

Was war gestern?
: Es fehlt die klare Struktur. Was man seit gestern erreichen wollte, ist einem nicht mehr bewusst.

Die falschen Leute
: Zwingen sie niemanden zur regelmäßigen Teilnahme, der kein Interesse daran hat.

Problemlösungen
: Probleme werden nicht nur angesprochen, sondern auch gelöst. Das führt zu ausufernden Diskussionen. Im Stand-up sollen die Probleme nur angesprochen werden, so dass das Team sich ihrer bewusst ist. Nach der Problemschilderung wird jemand in der Runde oder ein Außenstehender gefunden, der im Anschluss bei der Problemlösung helfen kann.

Die eine Hälfte sitzt, die andere steht
: Alle müssen stehen!

Zu wenig Energie
: Wenn Sie merken, dass die Beteiligten einen müden Eindruck machen, denken Sie über einen anderen Zeitpunkt für das Meeting nach.

Probleme werden nicht angesprochen
: »Das schaffen wir«-Mentalität. In agilen Teams müssen Probleme direkt offen und ehrlich angesprochen werden. Gelingt das nicht, installieren Sie einen »Doom Sayer«: Reihum (damit sich nicht ein Einzelner unbeliebt macht) nimmt jemand die Rolle ein, den Beitrag eines anderen Teilnehmers kritisch zu hinterfragen.

Finger Pointing
: Anstatt sachlich und konstruktiv eine Lösung zu finden, werden für Probleme Schuldige gesucht.

Probleme nur im Stand-up
: Probleme werden nur im Stand-up angesprochen, nicht aber während des gesamten Arbeitstags. Probleme müssen direkt angesprochen werden.

Geschwafel
: Der Redende kommt nicht zum Punkt. Er soll einen schnellen und groben Überblick über seine Aktivitäten geben. Er soll auf Details verzichten und seine Aktivität zusammenfassend vortragen. Dabei ist entscheidend *was* er gemacht hat und nicht *wie* er es gemacht hat.

Konfigurationsmanagement

Konfigurationsmanagment (*Software Configuration Management/SCM*, siehe Kapitel Versionskontrolle mit Subversion) ist nicht nur in agilen Projekten eine zentrale Disziplin. In einer Umgebung, in der viele verschiedene Entwickler auf einer gemeinsamen Codebasis arbeiten und wo es unterschiedliche Quellcodezweige gibt (beispielsweise Produktlinien, oder auch während Release-Arbeiten), benötigen Sie ein Konfigurationsmanagementsystem. Es hilft Ihnen, all Ihre Artefakte zu verwalten. Im Zentrum des SCM befindet sich das zentrale *Code Repository*, in dem sämtliche Artefakte wie fachliche Klassen, Tests und Dokumentationen verwaltet werden. Es können auch Testergebnisse versioniert und verwaltet werden, allerdings ist das nicht unbedingt notwendig, da Sie jeden Stand Ihrer Software jederzeit im Zugriff haben, die zugehörigen Tests erneut ausführen können und somit die Testergebnisse identisch herleiten können. Ich gebe Ihnen zwei Szenarien, die die Notwendigkeit von SCM exemplarisch verdeutlichen:

Gleichzeitige Entwicklung

Die Entwickler holen sich den aktuellen Code aus dem Repository in den eigenen *Workspace*. Das ist die Arbeitsumgebung des Entwicklers auf dem eigenen Arbeitsplatzrechner. Solange die Änderungen nur dort vorliegen, hat das keine Konsequenzen für das Repository oder die Workspaces anderer Entwickler. Die Entwickler machen Änderun-

gen und Erweiterungen an den Testklassen und am Code. Sie spielen regelmäßig den Code wieder in das Repository (»Einchecken«). Erinnern Sie sich? Je länger sie damit warten, desto größer wird die Diskrepanz zur zentralen Codebasis, und desto größer der Qualitätsverlust. Bevor Sie den Code wieder einchecken, aktualisieren sie ihn mit den Änderungen, die in der Zwischenzeit andere gemacht haben. Falls ein anderer Entwickler an der gleichen Klasse eine Änderung gemacht hat (durchaus möglich, da wir mit *Collective Code Ownership* arbeiten), müssen diese unterschiedlichen Änderungen zu einem neuen Softwarestand überführt werden (es wird ein *Merge* durchgeführt). Danach erstellen Sie die Anwendung auf Ihrem Arbeitsplatzrechner (Sie erstellen ein *Build*) und führen somit automatisch die Tests aus. Ist der Build gelungen, sind insbesondere auch die Tests erfolgreich durchgelaufen, so wird der Code eingespielt. Auch aus Code im Repository wird in regelmäßigen Abständen (oder nach bestimmten Ereignissen wie dem Einspielen von Änderungen am Code) ein Build erstellt. Das geschieht gewöhnlich auf einem Integrationssystem, einem Rechner, der der Produktivumgebung, auf der die Software schließlich laufen soll, entspricht.

Branching

Zu verschiedenen Zeitpunkten ist es hilfreich, den zentralen Entwicklungsstrang zu verlassen (siehe Abbildung 5-3). Das ist zum Beispiel der Fall, wenn ein Release keine neuen Funktionalitäten mehr aufnehmen soll. An dem Release werden nun nur noch Defekte behoben und letzter Feinschliff durchgeführt. Das Release wird für den Produktivgang (das *Going Live*) vorbereitet. Gleichzeitig sollen andere Entwickler dasselbe Produkt bereits weiterentwickeln (neue Stories für die nächste Iteration sollen bereits umgesetzt werden) und nicht auf den Abschluss der Release-Arbeiten warten müssen. Hier teilen Sie den Entwicklungsstrang auf, um später den Hauptzweig (den *Head*, auch *Trunk* genannt) mit dem *Branch* wieder zusammenzuführen (ein *Merge*). Dabei können dann Konflikte entstehen: Wurde für das bevorstehende Release noch ein Defekt in einer Klasse eingespielt und die Klasse in dem Branch gleichzeitig weiterentwickelt, müssen die unterschiedlichen Änderungen am gleichen Artefakt zusammengeführt werden. Hier ist es wichtig, die verschiedenen Zweige nicht zu lange aufzuhalten, um die Konflikte zu minimieren und die Qualität zu steigern. Branches können auch aufbauend auf anderen Branches erstellt werden.

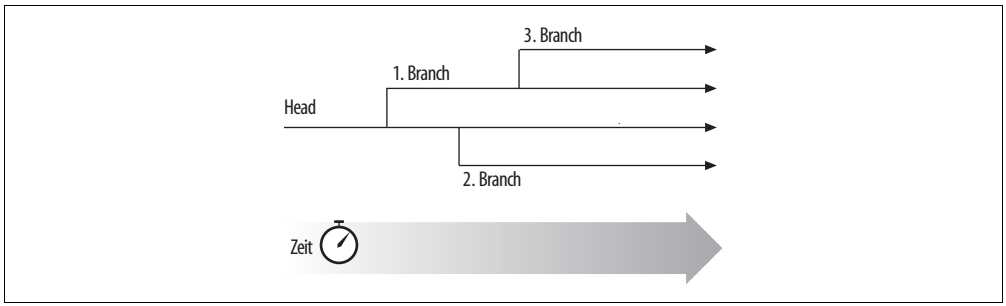


Abbildung 5-3: Von der einheitlichen Entwicklungslinie wird durch Erzeugung von Branches situationsabhängig abgewichen.

Test-Driven Development

Zu einer wirklich agilen Entwicklung gehört es ja, die Funktionalität *parallel* zu testen, siehe Kapitel 8 bis 17. Das heißt: Vor und während der Umsetzung der fachlichen Anforderung werden die *Story-Tests* (von den Kunden) und die *Unit-Tests* (von den Entwicklern) geschrieben, die ich im Anschluss detailliert vorstelle. Diese Tests werden im weiteren Entwicklungsverlauf kontinuierlich automatisiert ausgeführt. Dadurch dokumentieren Sie die Qualität der Software, ihr Design und den Entwicklungsstand. Die Tests sind dabei die Spezifikation, die die Entwicklung der eigentlichen Klassen vor sich her schiebt. Durch den *Test-First*-Ansatz, zuerst wird der Test geschrieben, dann der fachliche Code, *treiben* die Tests also die fachlichen Klassen vor sich her. Wichtig ist zu verstehen, dass die Tests nicht primär Defekte finden sollen, wie es in herkömmlicher Betrachtungsweise üblich ist. Das Testen ist, im Gegenteil, eine Maßnahme, die von Anfang an die Qualität sichert und Defekte *verhindert*.

Haben Sie dedizierte Tester in Ihrem Team, so lassen Sie sie nicht warten, bis der Code fertig ist. Beteiligen Sie sie aktiv an der Entwicklung. Die Tester können auch dem Kunden helfen, die Story-Tests zu schreiben.

Die Entwicklung der Tests und der fachlichen Klassen läuft verzahnt. Schreiben Sie nur dann neuen, fachlichen Code, wenn ein automatischer Test dafür vorliegt. Während der Entwicklung werden Sie kontinuierlich *Refactorings* machen (dazu später mehr).

Akzeptanztests/Story-Tests

Story-Tests sind fachlich motivierte, vom Kunden geschriebene Tests. Sie schlagen die Brücke zwischen den Anforderungen und der Anwendung und beschreiben, was das System funktional leisten soll. Sie manifestieren dabei die Kundenanforderungen und dokumentieren später, ob diese tatsächlich korrekt umgesetzt wurden. Nach Änderungen und Erweiterungen in späteren Arbeitsschritten kann jederzeit automatisiert die bereits bestehende Funktionalität verifiziert werden. Dieses Vorgehen ist als *Specification-By-Example* bekannt: Die Gesamtheit der Tests wird zu einer vollständigen, selbstverifizierenden Software-Spezifikation. Beim eigentlichen Durchlauf des Akzeptanztests wird zur Laufzeit das Ergebnis (das Verhalten der Anwendung) mit einem erwarteten Soll-Ergebnis verglichen. Der Lauf des Akzeptanztests geschieht automatisiert.

Obwohl dies auch eine Lösungsdimension ist, reduzieren Sie die Akzeptanztests keinesfalls ausschließlich auf ein *Capture & Replay*-Vorgehen, bei dem Sie auf der bereits bestehenden Funktionalität Arbeitsschritte durchführen, diese abspeichern, zu späteren Zeitpunkten erneut abspielen und das Ergebnis dieser Interaktionen mit den ursprünglichen vergleichen. Dieses nachgelagerte Vorgehen dokumentiert häufig nur den Status Quo und ist keine vorgelagerte Spezifikation. Die Story-Tests laufen gewöhnlich auf der Oberfläche ab, wobei sie die Existenz der zu testenden Oberfläche aber nicht voraussetzen (der Test kann geschrieben sein, ohne dass das fachliche Pendant dazu existiert).

Funktionale Tests werden gewöhnlich mit einem oder mehreren der Werkzeuge *WebTest* (siehe Projektkommunikation mit Trac), *Fit* (siehe Akzeptanztests für Webanwendungen mit WebTest) und *Selenium* (siehe Akzeptanztests mit Fit) geschrieben. Werden die funktionalen Tests vor der Anwendung (von dem Kunden) geschrieben bzw. schieben die funktionalen Tests die Entwicklung der Anwendung vor sich her, so spricht man auch von Akzeptanztests.

Unit-Tests

Komplementär zu den Story-Tests existieren die Unit-Tests, die von den Entwicklern geschrieben werden. Hier kommen gewöhnlich *JUnit* (siehe Kapitel 9, *Komponententests mit JUnit*) und/oder *TestNG* (siehe Kapitel 12, *Komponententests mit TestNG*) zum Einsatz. *Units* sind dabei einzelne Komponenten (oder Module), die separat betrachtet und getestet werden. Im Gegensatz zu den Stories sind Units technische Artefakte wie einzelne Klassen, müssen aber nicht auf solche beschränkt sein.

Unit-Tests sind ebenfalls automatisiert ablaufende Tests. Sie testen aber nicht den Geschäftszweck der Anwendung, sondern das Design und die Funktionalität des Codes. Sie sind also nicht fachlich, sondern technisch motiviert (obwohl sie die Funktionalität einer Klasse testen) und werden somit vom Entwickler geschrieben.

Der Entwickler fängt mit der Entwicklung eines Tests in einer Testklasse an. Anschließend entwickelt er den fachlichen Code in einer anderen, einer fachlichen Klasse, der dann den Test der Testklasse gerade besteht. Anschließend erweitert er den Test erneut und so geht es weiter. Er macht sich auf diesem Weg frühzeitig Gedanken über das Design der Software. Ganze Bücher befassen sich mit *Code Smells*, also der Beschreibung von wiederkehrenden Mustern für schlecht riechenden, also schlecht entworfenen Code, der dann systematisch verbessert werden kann. Dadurch verfügt der Code über eine implizite Qualität, und die einzelnen Komponenten sind möglichst lose gekoppelt. Die Tests werden somit zur eigentlichen Design-Spezifikation.

Unit-Tests testen nicht die Oberfläche (*UI*) einer Anwendung. Eine gute Software-Architektur sorgt dafür, dass die Oberfläche von der Logik darunter weitestgehend entkoppelt ist. Dazu müssen die Verantwortlichkeiten innerhalb der Anwendungen so getrennt werden, dass die Oberfläche nur noch eine ganz dünne Schicht auf der Anwendung selbst ist. Die eigentliche Logik befindet sich darunter, und kann auch ohne UI getestet werden. Das hat den Charme, dass Änderungen auf der Oberfläche die Logik darunter nicht beeinflussen. Auf diesem Wege kann der Großteil der Anwendung feingranular mit Unit-Tests getestet werden, ohne dass die Oberfläche einem Test unterzogen werden müsste. Berücksichtigen Sie hier ferner, dass visuelle Komponenten von Java Swing-Oberflächen auch über ein *Model* verfügen, in dem die Daten gehalten werden. Auch hier müssen Sie nicht zwingend auf die reine Oberfläche (das *UI* der Komponente). Die Oberfläche selbst (und die dahinter liegende Fachlichkeit) wird dann ausschließlich mit Story-Tests validiert. In vielen Fällen gelingt diese saubere Trennung. In den anderen Fällen müssen Sie

die Oberfläche explizit oder sogar manuell testen. Müssen Sie dennoch auf die Oberfläche zugreifen und Benutzer-Interaktionen steuern, so ist *Jemmy* (siehe Webanwendungen testen mit Selenium) ein Werkzeug dafür.

Die Unit-Tests werden automatisch und häufig durchlaufen. Units verrichten häufig keine autonomen Arbeiten, sondern greifen auf andere Klassen oder Datenbanken zu. Oft werden dadurch umfangreiche Prozesse angestoßen, die einerseits lange dauern, andererseits vor dem Hintergrund der Unit gar nicht getestet werden sollen. Um ein schnelles, isoliertes Test-Feedback zu erhalten, werden deswegen häufig *Mock-Objekte* eingesetzt. Diese Objekte simulieren die Funktionalität von Komponenten außerhalb des zu testenden Artefakts (so wird beispielsweise eine Datenbank »weggemockt«) und *isolieren* den Test. In verschiedenen Test-Modi werden die Tests mit Mock-Objekten (zum Beispiel im Workspace der Entwickler) und ohne Mock-Objekte durchlaufen (beispielsweise in der nächtlichen Integration, einem Integrationstest). Hier ist *EasyMock* ein bekanntes Werkzeug (siehe Komponententests mit TestNG).

Mit Hilfe von Unit-Tests können auch Performancetests durchgeführt werden. Zudem wird häufig nicht nur der Quellcode der Anwendung, sondern auch der Zustand der Datenbank in bestimmten Momenten getestet.

Manuelle Tests

Manuelle Tests ganz ersetzen zu wollen ist utopisch. Der Umfang dieser manuellen Tests sollte zwar auf ein Minimum reduziert werden, da sie fehleranfällig und zeitintensiv sind und häufig auch nur eine einfache Fleißarbeit darstellen, die wenig motiviert. Es ist allerdings trotz aller Automatisierung notwendig, dass ein konkreter Nutzer das System als Ganzes testet, sich die *Usability* anschaut und das System auf fachliche Konsistenz überprüft. Wie lässt sich damit arbeiten? Wurde die Identifikation einer Story vergessen? Hilfreich sind manuelle Tests auch im Kontext von *Grenztests*. Der Kunde oder ein explizit eingeschalteter Tester (der ein Entwickler sein kann) führt auf der Anwendung extreme Eingaben aus: Er gibt Daten ein, die maximale und minimale Grenzen des möglichen Wertebereichs darstellen, oder führt *unberechenbare* Interaktionen auf der Oberfläche aus.

Wenn Sie zweimal täglich oder nach jeder Integration zeitintensiv über die Oberfläche Ihrer Anwendung gehen um zu schauen, ob beim Betätigen eines Anlegen-Buttons etwas angelegt wird und beim Speichern-Button etwas gespeichert wird, dann machen Sie definitiv etwas falsch. In agilen Teams sind eingebaute Qualität und Zeit magische Kenngrößen. Rudimentäre Tests werden automatisiert durchgeführt und liefern bereits während des Vorgangs des *Continuous Integration* eine Rückkopplung zum Guten oder Schlechten.

Continuous Integration (CI)

Wir haben bereits verschiedene Szenarien gesehen, die Inkonsistenzen und Inkompatibilitäten zwischen lokalem Softwarestand auf dem Arbeitsplatzrechner der Entwickler und dem zentralen Repository zur Folge haben. Um diese Diskrepanzen schnellstmöglich zu

beseitigen und mögliche Defekte frühzeitig aufzudecken, muss der Code permanent zusammengeführt, also integriert werden. Dieses Vorgehen ist unter dem Schlagwort Continuous Integration (CI) bekannt.

CI durchzuführen ist schwieriger als es den Anschein hat. Gerade in nicht-agilen Projekten neigen die Entwickler dazu, den Code so lange im privaten Workspace zu haben, bis die Arbeiten daran abgeschlossen sind. Natürlich mag es auf den ersten Blick wie eine überflüssige Bremse wirken, den Code einzuchecken. Denn dafür ist immer erst ein Vergleich mit dem aktuellen Sourcestand, ein Build und die Tests notwendig. In agilen Projekten führen die Entwickler dennoch regelmäßig ihre Änderungen der zentralen Code-Basis zu, mehrfach täglich. Der Code sollte allerdings frei von Fehlern sein (sich kompilieren lassen), über möglichst keine Warnings verfügen und muss Tests auf dem lokalen Arbeitsplatz erfolgreich durchlaufen haben. Checken Sie nichts ein, von dem Sie nicht sicher sind, dass es – zumindest bei Ihnen auf dem Rechner – fehlerfrei läuft. Wird während der Erstellung eines Builds auf dem Integrationsrechner ein Fehler festgestellt, so checken Sie keine weiteren Änderungen und Erweiterungen ein, solange das ursprüngliche Problem für diesen Fehler nicht eliminiert ist.

Eine Hilfe, umfassende Builds kontinuierlich durchzuführen, ist eine mehrschichtige Systemumgebung. Bei diesem *Staging* läuft ein Teil der Tests auf der ersten Integrationsmaschine. Hier werden rudimentäre Tests durchgeführt, also ein *Smoke Test*, der wenige Minuten dauert. In regelmäßigen, aber größeren Abständen werden die Sourcen automatisch auf eine nachgelagerte Umgebung geschoben und installiert, auf der dann umfangreichere Tests laufen. Diese Tests laufen beispielsweise nachts oder am Wochenende und können mehrere Stunden dauern.

Mit der Erfahrung aus zahlreichen Projekten kann ich Ihnen nur ans Herz legen, besonderes Augenmerk auf CI zu legen, um qualitativ hochwertige Software schnell erstellen zu können und Fehler zu vermeiden bzw. sehr früh zu erkennen. Werkzeuge zur kontinuierlichen Integration sind *Ant* (siehe Kapitel 7, *Ant-Grundlagen* und Kapitel 18, *Ant – Teil 2: Praktische Einsatzszenarien*), *Maven* (siehe Kapitel 20, *Der Einsatz von Maven für Build-Prozesse*) und *CruiseControl* (siehe Kapitel 19, *Kontinuierliche Integration mit CruiseControl*).

Synchronisation

Die Entwicklung großer, komplexer Software mit mehreren Entwicklern, bei gleichzeitiger Arbeit an verschiedenen Softwareständen in möglicherweise räumlich verteilter Umgebung und Offenheit für Änderungen bei den Anforderungen des Kunden stellt hohe Ansprüche an den Entwicklungsprozess. Die Entwicklung droht stetig auseinanderzudriften.

In solch komplexen Projekten besteht die Gefahr, dass der Entwickler an der Anforderung vorbei entwickelt, die ursprünglichen Kundenanforderungen beim Abschluss keinen Mehrwert mehr generieren und der Code auf dem Arbeitsplatzrechner des Entwicklers schon längst einen veralteten Stand im Vergleich zum zentralen Repository hat.

Idiotensichere Software

Kennen Sie *Murphy's Law*? Alles, was schief gehen kann, geht auch schief. Das werden Sie sicherlich schon selbst bei Ihren Projekten festgestellt haben. Sie haben alles minutiös geplant und dann doch eine Kleinigkeit vergessen. Das überrascht nicht, da wir bei der Erstellung von Software tatsächlich eine Vielzahl von Wechselwirkungen berücksichtigen müssen. In nicht-agilen Projekten geht häufig genau das schief, was als garantiert fehlerfrei angesehen wurde. Nicht selten konzentrieren sich nicht-agile Projekte auf die Fehlerfreiheit von komplizierten Funktionen, um dann doch von einem offensichtlichen Fehler bei einer einfachen Komponente überrascht zu werden.

Agile Projekte wirken dem entgegen. Durch vollständige Automatisierung auch der einfachen Arbeitsschritte werden nicht nur die Entwickler von Routinetätigkeiten entlastet, sondern die Qualität des Entwicklungsprozesses und der Software deutlich erhöht: Fehler werden frühzeitig erkannt, die Arbeitsschritte sind wiederholbar und das Tempo ist hoch. Ziel muss es sein, den gesamten Prozess der Erstellung der Software (den Build) sowie die Software selbst *idiotensicher* zu machen. In agilen Projekten wird das durch folgende Faktoren ermöglicht:

- Build auf Knopfdruck (mit Werkzeugen wie Ant und Maven)
- Zeitlich gesteuerte Builds (Cruise Control)
- Sichtbarmachen der Ergebnisse der Builds
- Release auf Knopfdruck (inkl. Versionierung)
- DAU-sichere Installation: Nach dem Build wird die Software bereitgestellt oder direkt installiert. Muss die Software manuell installiert werden, ist diese Installation idiotensicher bzw. DAU-sicher (»DAU« = »dümmster anzunehmender User«).

Alles, was Sie mehr als einmal machen, sollten Sie automatisieren. Manuelle Arbeiten sind fehleranfällig und für einen hochqualifizierten Entwickler demotivierend.

Um die Vielzahl von latenten Unsicherheiten bei der Softwareerstellung in den Griff zu bekommen, führen agile Projekte durch ihr Vorgehen zahlreiche komplementäre *Synchronisationspunkte* ein. Diese sind in den Prozess eingebaut, greifen ineinander und kommen zeitnah zum Tragen. Dazu gehören folgende Punkte:

- Story-Tests synchronisieren Code und Anforderungen.
- Unit-Tests synchronisieren Code und erwartetes Verhalten.
- Continuous Integration synchronisiert den Code und all seine Komponenten.
- Stand-up Meetings synchronisieren das Team.
- Viele kleine Iterationen synchronisieren den Prozess, die Software, das Team, die Anforderungen und den Kunden.
- Jedes Deployment synchronisiert den Prozess, den Code, die Anforderungen, das Team und den Kunden.

- Die für alle sichtbaren Story Cards und das öffentliche Tracking des Projekts synchronisieren das Team.
- Durch Refactoring wird stetig das Code-Design überprüft.

Refactoring

Refactoring beschreibt den Prozess, die interne Struktur der Software zu verbessern, ohne das externe Verhalten zu ändern. Etwas technischer ausgedrückt sind das Maßnahmen, die zwar die öffentliche Schnittstelle (das öffentliche API) der Software unverändert lassen, die Software aber intern so ändern, dass sie besser wartbar und erweiterbar wird und Fehler verhindert bzw. Fehler schneller gefunden werden können.

Agile Projekte verzichten auf ein umfangreiches vorgelagertes Design der Software. Design und Implementierung werden iterativ und inkrementell durchgeführt, wobei immer eine lauffähige, schlanke Software im Mittelpunkt steht, die genau das leistet, was die Anforderung beschreibt. So kommt es, dass zu Beginn das Design der Software nicht optimal ist und im Laufe der Entwicklung stetig überprüft und immer wieder verbessert wird. Neben den eigentlichen fachlichen Klassen werden auch immer die Tests einem Refactoring unterzogen.

Defekte

Die Software und der Entwicklungsprozess können natürlich trotz aller Bemühungen Fehler aufweisen. Durch die beschriebenen Maßnahmen wie Automatisierung können Defekte in der Software oder auch im Prozess zwar sehr schnell identifiziert werden. Aber dann kommt es ganz darauf an, wie man mit den gewonnenen Erkenntnissen umgeht.

Identifizierte Defekte sind immer sofort zu lösen! Aus Defekten in der Software, deren Behebung verhältnismäßig lange dauern würde, werden neue User Stories erstellt. Die Grenze zwischen Anforderungen und Defekten ist in agilen Projekten fließend. Schließlich ist es unerheblich, ob eine User Story aus einem während eines manuellen Tests identifizierten Usability-Problem entsteht, einer falschen Beschreibung in einem Benutzerhandbuch oder einer geänderten oder gänzlich neuen Anforderung.

Standards

Agile Projekte besitzen auch deshalb eine hohe Qualität und ein hohes Tempo, weil Sie konsequent Standards nutzen. Diese Standards werden vom ganzen Team festgelegt, akzeptiert und getragen. Das Team entwickelt häufig eigene Standards, die zwar an bekannte Standards angelehnt sind, aber an die individuellen Gegebenheiten angepasst sind.

Dazu gehören

- Namenskonventionen,
- Coding Standards,

- Verzeichnisstrukturen,
- Konfigurationsmanagement-Praktiken,
- Werkzeuge,
- Error Logging und
- Testabdeckung.

Durch Namenskonventionen und Coding Standards können sich die Entwickler schnell überall einarbeiten. Der Code ist in allen Artefakten vergleichbar aufgebaut: Die Namen von Klassen, Methoden und Variablen folgen einer ähnlichen Struktur, die Kommentierungen sind identisch aufgebaut usw. Coding Standards legen beispielsweise fest, wie innerhalb des Codes Einrückungen zur besseren Lesbarkeit genutzt werden, also wie viele Zeichen in welcher Zeile eingerückt werden sollen, wie diese Zeichen eingefügt werden (per Leerzeichen oder Tabulator) usw.

Die Qualität des Codes entwickelt sich zwar kontinuierlich, sollte aber von Anfang an bezüglich der Namenskonventionen und Coding Standards optimal sein und dem Standard gehorchen. Um dies zu überprüfen, bietet sich eine Werkzeugunterstützung an. Mit *Checkstyle* beispielsweise können Ihre Anforderungen an den Code überprüft werden (siehe Kapitel 18, ab Seite 338). Hier können Sie frühzeitig erkennen, ob Code, Variablen oder Imports nicht genutzt werden oder mit JavaDoc die Dokumentation fehlerhaft ist. Beachten Sie auch von Checkstyle kontrollierte Komplexitäts-Metriken wie *Fan-Out Complexity* (Anzahl der Non-Java-Standard-Klassen, von der die zu untersuchende abhängig ist) oder *Cyclomatic Complexity* (Anzahl der Durchlaufpfade durch eine Klasse oder Methode). Zu hohe Werte dieser Größen weisen auf komplexe Klassen hin bzw. auf Klassen, die bald zu komplex werden und dann nur noch kostenintensiv erweitert und geändert werden können.

Setzen Sie hohe Standards und halten Sie diese ein. Befolgen Sie die Regel, dass Tests jederzeit hundertprozentig bestanden werden müssen! Das hat nicht nur eine direkte Auswirkung auf die Qualität, sondern auch eine psychologische. Die Metapher von der eingeworfenen Fensterscheibe zeigt: Ist das Projekt einmal auf einem chaotischen Weg, hat kein Entwickler ein Problem damit, etwas zu tun, was er in einem intakten Projekt nicht tun würde. Überprüfen Sie die Testabdeckung: Werden Ihre fachlichen Klassen wirklich alle getestet? Coverage-Werkzeuge wie *EMMA* (siehe Kapitel 19, *Testabdeckung mit EMMA und JUnit*) unterstützen Sie dabei festzustellen, welcher Code nicht durchlaufen oder nicht getestet wird. Was bringen zu 100% erfolgreich laufende Tests, wenn nur 60% des Codes getestet werden? Was ist mit den restlichen 40%? Verbessern Sie die Geschwindigkeit, mit der die Tests laufen. Je schneller und häufiger die Tests laufen, desto schneller bekommen Sie eine Rückmeldung über den Zustand der Software.

Haben Sie unterschiedliche Projekte, dann sollten diese alle über eine identische Verzeichnisstruktur verfügen. Auf diese Weise kann sich ein Entwickler in dem neuen Projekt schneller zurechtfinden, wenn er in einem anderen Projekt nicht mehr benötigt wird (weil dort nach einigen Iterationen die Produktivität sehr hoch geworden ist). Diese Verzeichnisstrukturen spiegeln sich auch im Konfigurationsmanagement-System (SCM)

wider, in dem ebenfalls projektübergreifend von einheitlichen Praktiken profitiert wird. Ein Beispiel dafür ist, dass die Namen der Tags immer demselben Muster folgen (Details dazu folgen im Kapitel Versionskontrolle mit Subversion).

Auch bezüglich des Designs und der Architektur bemühen sich agile Entwickler um die Einhaltung bewährter Richtlinien und guter Entwurfstipps: Entkoppeln Sie z. B. die Komponenten voneinander. Dazu gehört auch, das UI von der Logik und dem Datenbanksystem zu trennen. Obere Schichten in Ihrer *Architektur* kennen zudem untere Schichten, aber nicht umgekehrt. Ein Beispiel für ein gutes, standardisiertes *Design* ist der Umgang mit dem (Error-)Logging und dem Exception Handling in Ihren Java-Klassen. Logging und insbesondere die Ausnahmebehandlung sollten vor Beginn der Entwicklung der Funktionalität im Team diskutiert und einheitlich umgesetzt werden.

Reviews

Bei der Einhaltung und Durchführung von Standards sollten Sie auf Werkzeuge vertrauen, die entweder automatisch bei jedem Einspielen von Änderungen an der Software oder zu bestimmten Zeiten die Qualität des Codes überprüfen. Formale traditionelle Code-Reviews, wie sie in nicht-agilen Projekten häufig genutzt werden, verursachen öfter soziale Probleme, als dass sie wirklich helfen. Da in diesen nicht-agilen Projekten gewöhnlich ein erfahrener Entwickler den Code eines weniger erfahrenen Entwicklers überprüft, entstehen schnell Spannungen: Der Junior muss sich (oder denkt das zumindest) für seine Arbeit rechtfertigen. In Wahrheit ist aber das Artefakt (beispielsweise die fachliche Klasse) das Objekt der Prüfung. Was macht der Prüfer, wenn ihm etwas auffällt? Was vergleicht er eigentlich womit? Einhaltung von Konventionen und struktureller Aufbau lassen sich viel besser automatisiert testen. Einen umfassenderen Fehler in der Architektur oder im Design in einem dedizierten Review post mortem zu identifizieren, wenn das Kind also bereits in den Brunnen gefallen ist, ist viel zu spät. Was passiert, wenn der traditionelle Reviewer feststellt, dass der Code nicht dem Einsatzzweck dient?

In agilen Projekten wird der Code nicht nur automatisch umfassend überprüft (wie zum Beispiel mit Checkstyle): Daneben gibt es das Programmieren in Paaren, um direkt zeitnah zusammen das Design zu entwerfen und kontinuierlich zu reviewen. Durch kollektive Verantwortung hat jeder kontinuierlich mit jedem Artefakt zu tun und unterzieht es einem Review. Durch die kontinuierlich fortgeschriebenen Spezifikationen in Form von Unit- und Story-Tests werden die Fehler nicht nur viel schneller und besser identifiziert, die Tests treiben sogar die eigentliche Entwicklung und verhindern Fehler, bevor sie entstehen.

Nichtsdestoweniger kann es auch in agilen Teams für Juniors spannend und lehrreich sein, wenn sie Seniors bei Diskussionen hören und mit ihnen arbeiten und diskutieren. Durch interaktive Arbeit im Team existiert ein stetiger Austausch. Pair Programming-Sitzungen sind ein Geben und Nehmen. Selbst wenn ein erfahrener mit einem unerfahrenen Entwickler zusammenarbeitet, haben doch beide jederzeit die Möglichkeit, sich offen auszutauschen. Auch der erfahrenste Senior kann häufig noch etwas dazulernen und

neue Impulse bekommen. Gerade Neuzugänge haben eine ganz andere Sicht eines Artefakts, und ihnen fallen Sachen auf, für die andere vielleicht schon betriebsblind sind.

All das klappt nur in einem agilen Team, in dem das Management den Entwicklern den Rücken freihält. Dazu gehört auch der Verzicht auf individuelle Ziele, die für die Arbeit im Team kontraproduktiv sind. Das Management verzichtet auf die Zuweisung von individuellen Leistungszielen zur Beurteilung und zielabhängige Entlohnung der Mitarbeiter.

Wo traditionelle Projekte Code Reviews, Walkthroughs und Ähnliches praktizieren, verzichten agile Projekte bewusst darauf. Sie legen stattdessen einen Schwerpunkt auf die Förderung der ständigen Kommunikation und den Austausch.

Tracking

Als Tracking bezeichnet man die Verfolgung des Arbeits-/Projektfortschritts, insbesondere auch um korrigierend eingreifen zu können. Nicht-agile Projekte praktizieren häufig schwergewichtige Ansätze: Pflege- und zeitintensiv werden einzelne Aufgaben elektronisch auf einzelne Ressourcen verteilt und der Projektfortschritt errechnet. Diese Programme sind häufig kommerziell, und ihre Bedienung muss erlernt werden. Häufig kommen spezielle Charts zum Einsatz, beispielsweise Gantt-Charts. Erstellte Diagramme werden entweder ausschließlich elektronisch vorgehalten oder ausgedruckt und dem Team bereitgestellt. Das ist ein werkzeuglastiges Vorgehen.

Agile Projekte gehen einen anderen Weg: Sie zeichnen sich dadurch aus, dass sie leichtgewichtig den Projektfortschritt begleiten. Es wird dafür kein besonderes Programm eingesetzt. Die Grafiken können mit jedem Malprogramm oder auch manuell an einem Whiteboard erstellt werden. Sie sind recht einfach gehalten und somit schnell zu verstehen, aber dennoch sehr informativ. Die Grafiken und der Umgang mit ihnen unterstreichen das grundlegende agile Vorgehen, nicht ein Werkzeug in den Mittelpunkt zu stellen, wenn es auch ohne geht.

Burndown-Charts

Eine bewährte Grafik ist ein *Burndown-Chart*. Generell setzt die Grafik die Story Points mit der Zeit in Beziehung. Zu Beginn gibt es eine feste Anzahl von Story Points, die es in einer Iteration abzuarbeiten gilt. Jeden Tag werden die noch offenen Story Points weniger. Auf der x-Achse wird die Zeit abgetragen, auf der y-Achse finden sich die noch offenen Story Points wieder (siehe Abbildung 5-4).

Gewöhnlich werden die Story Points verbunden, so dass sich eine abfallende Kurve ergibt. Ist die Zeitachse am Enddatum angekommen, beträgt die Anzahl der noch offenen Story Points im Optimalfall null. Subtrahieren wir zwei benachbarte Story Point-Werte voneinander, haben wir die Leistung des Teams im zugrundeliegenden Zeitabschnitt. Nehmen wir als Zeitabschnitt eine Iteration, ist die Differenz der Story Points die *Velocity*.

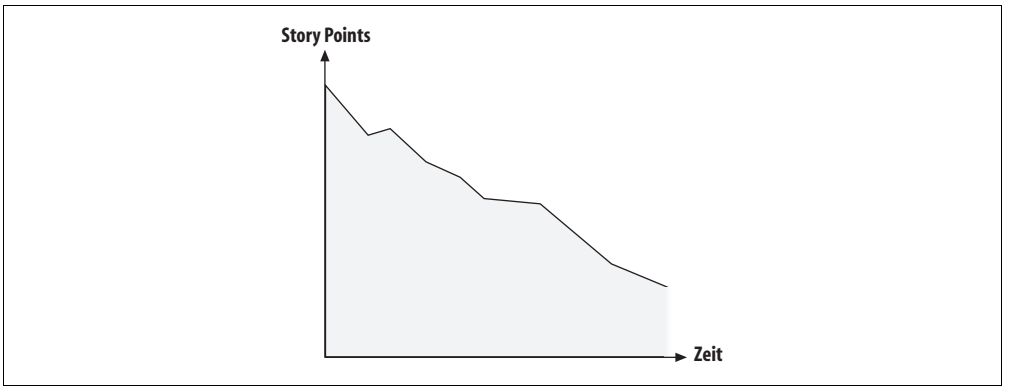


Abbildung 5-4: Story Points werden abgearbeitet

Sie können ein Burndown-Chart auch für ein ganzes Release pflegen. In Abbildung 5-5 wurde dem Release zu Beginn ein Umfang von 200 Story Points zugewiesen. Das Release wurde in elf Iterationen aufgegliedert. Die Zeitachse ist bei diesem Chart also grobkörniger, spiegelt sie doch nicht mehr die Zeit direkt wieder, sondern die Iterationen. Im diesem Beispiel nehmen die Story Points tatsächlich kontinuierlich ab, bis sie mit der letzten Iteration null erreichen. Wir sehen hier nun auch eine ganz normale Eigenschaft: Burndown-Charts zeigen nicht immer nur nach unten! Da zu Beginn die Anforderungen nicht umfassend verstanden werden, kann es immer wieder vorkommen, dass während der Iteration oder des Release neue Aktivitäten identifiziert werden, die zeitnah umgesetzt werden sollen, oder dass einzelne Stories sich als deutlich umfangreicher herauskristallisieren als vermutet. Ein anderes Beispiel ist, wenn während einer Iteration umfangreiche Refactorings durchgeführt wurden und somit kaum Story Points abgearbeitet wurden (dafür aber in nachgelagerten Iterationen umso mehr, da das Design der Software nun deutlich besser ist und Erweiterungen schneller umgesetzt werden können). Nur wenn diese Schwankungen zu groß sind oder die Kurve nicht mittelfristig nach unten zeigt, sollten Sie stutzig werden. Insbesondere werden Sie dann am Ende des Release nicht das erwartete Ergebnis liefern können.

Eine weitere bekannte Variante bei der Nutzung von Burndown-Charts ist es, die Änderungen im Umfang (*Scope*) sichtbar mit aufzunehmen. Wir haben im letzten Beispiel gesehen, dass eine Aufnahme von neuen Stories in den herkömmlichen Burndown-Charts einen Ausschlag nach oben verursacht. Andererseits könnten gleichzeitig andere Stories aus einer Iteration herausgenommen worden sein, weil sie nicht mehr wichtig sind und keinen Mehrwert für die Software generieren. Das ist aus dem herkömmlichen Burndown-Chart nicht ersichtlich. Der Leser erkennt zwar, dass die Anzahl der noch offenen Story Points kleiner ist, kann aber nicht unterscheiden, ob die Stories abgearbeitet oder entfernt wurden. Der Chart in Abbildung 5-6 berücksichtigt diese zusätzlichen Informationen. Hier werden neue Anforderungen unterhalb der x-Achse dargestellt. Zunächst eingeplante, dann aber entfernte User Stories werden durch ein Absetzen des Balkens oberhalb der x-Achse dargestellt.

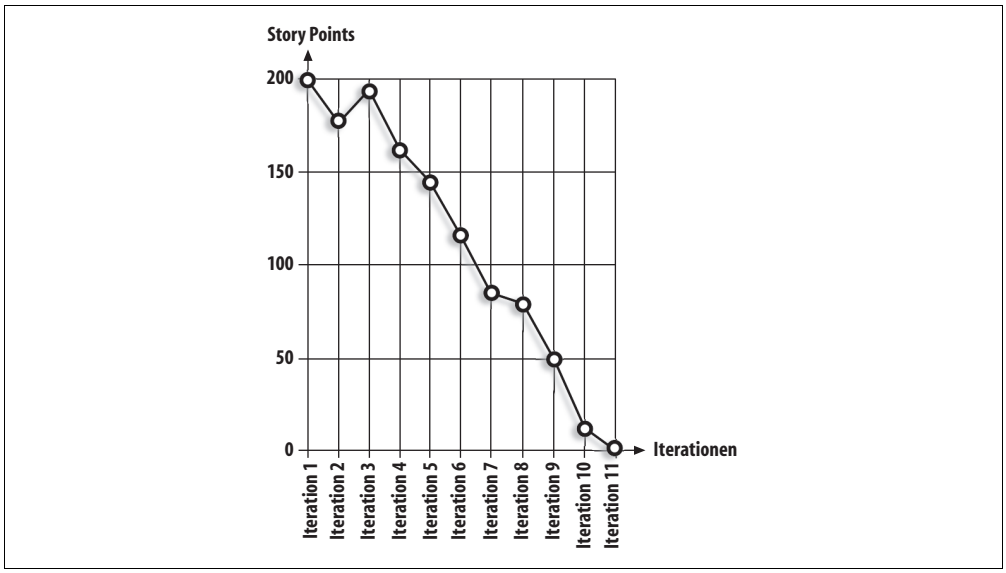


Abbildung 5-5: Burndown-Chart für ganzes Release

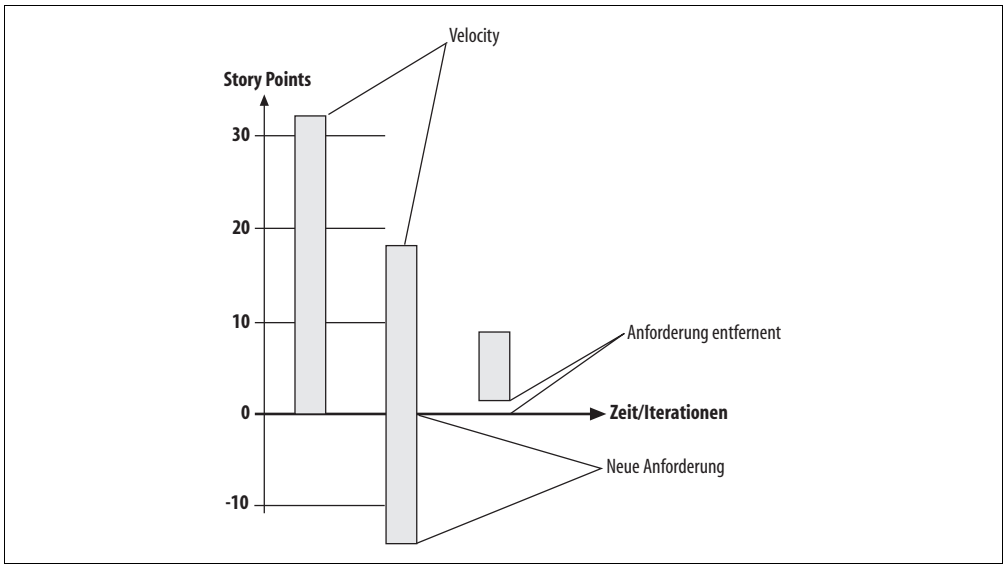


Abbildung 5-6: Story Points, differenziert nach neuen und entfernten Story Points: Von ursprünglich 32 Story Points wurden 14 abgearbeitet. Es kamen aber 14 neue Points hinzu. In der nächsten Iteration verbleiben noch 7 offene Points, da von den 32 Points 21 abgearbeitet wurden und 2 Points aus der Iteration bewusst herausgenommen wurden.

Eine Variante ist es, die Anforderungen oberhalb der x-Achse zu platzieren und wie gewohnt fortzuschreiben, und unterhalb der x-Achse die Defekte und geänderten User Stories zu platzieren (siehe Abbildung 5-7). Auf diese Weise unterscheiden Sie genauer, was exakt die Story ist, also eine Neuanforderung, eine Änderung oder ein Defekt.

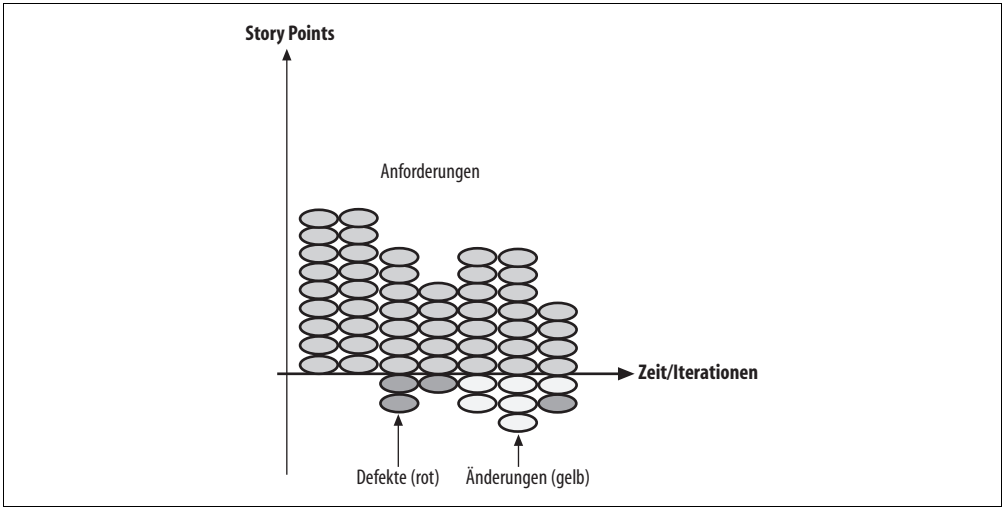


Abbildung 5-7: Story Points, differenziert nach Defekten, Änderungen und Anforderungen: Unterhalb der x-Achse werden Defekte (Bugs, Fehler) sowie Änderungen an den Anforderungen mit einer jeweils anderen Darstellung, zum Beispiel abweichender Farbe, gezeigt. Oberhalb der x-Achse befinden sich die noch offenen Story Points, die aus Neuanforderungen resultieren.

Sie sehen: Obwohl ein Burndown-Chart grundsätzlich immer den gleichen zeitlichen Verlauf hat, sind doch viele Varianten denkbar. Nutzen Sie genau die individuelle Anpassung, die für Ihr Projekt den meisten Nutzen hat.

Wollen oder müssen Sie andere Statistiken oder Charts erstellen, zum Beispiel für das höhere Management, dann greifen Sie die Informationen aus den Burndown-Charts auf und lassen Sie sie dort einfließen.

Burndown-Charts stoßen an Ihre Grenzen, falls in verteilten Teams gearbeitet wird. Haben nicht alle Entwickler Zugang zu den Charts, muss mit elektronischen Medien gearbeitet werden. Sie haben hier die Wahl, die Charts in einem Wiki zu integrieren oder ein Tracking-/Ticketing-System einzusetzen. Im Werkzeug-Teil dieses Buchs stelle ich mit Trac ein Wiki vor, mit dessen Hilfe Sie auch Ihre Aktivitäten verwalten können (siehe Kapitel 17, *Projektkommunikation mit Trac*). Weitere Werkzeuge sind das kommerzielle JIRA und das frei verfügbare Bugzilla.

Release- und Iterationsplanung pragmatisch

Einzelne Projekte haben ihren optimalen Weg im Umgang mit Anforderungen darin gefunden, diese zu sammeln und direkt im Anschluss die damit verbundenen Tasks zu identifizieren und in die *Tracking-Datenbank* einzuspeisen. Dabei können auch während der Iteration neue Tasks erstellt werden, die nach erfolgter Abschätzung und Priorisierung andere Tasks mit geringerer Priorität verdrängen können. Der Aufwand aller oder ein Teil der eingetragenen Tasks werden von den Entwicklern geschätzt. Die Schätzungen werden direkt zu den jeweiligen Anfoderungen in der Datenbank hinzugefügt. Aufbauend auf den Schätzungen priorisiert der Kunde den Task. Es wird also bewusst auf das Schreiben von Karten verzichet. Die Grenze zwischen Feature und Task weicht auf. Faktisch befinden sich ausschließlich Tasks im Tracking-System, die eine Größenordnung von einem halben bis wenige Tage haben und als Summe eine Iteration bzw. Release ausmachen. Das Wissen, welche Tasks Bestandteil eines Features (einer Story) sind, befindet sich in den Köpfen der Beteiligten bzw. wird vom Kunden in der Tracking-Datenbank während der Priorisierung eingefügt. Die Tracking-Datenbank fungiert damit als Product Backlog. Bekannte Tracking-Datenbanken sind *Jira*, *Bugzilla* oder *Trac*.

Abschluss

Jede Iteration und das Release selbst sind *time-boxed*, das zeitliche Ende steht also fest. Auf diese Weise haben alle Beteiligten ein klares Ziel vor Augen und die Teamleistung in dem Intervall kann gemessen werden. Messung bedeutet auch festzustellen, ob das Team produktiver arbeitet als in einer früheren Iteration.

Die Abnahme durch den Kunden

Mit dem Abschluss der Iteration und des Release wird die Software dem Kunden vorgestellt. Er kann damit arbeiten und ggf. wertvolles Feedback für weitere Arbeitsschritte geben. Die Software wird abgenommen: Der zu Beginn spezifizierte Leistungsumfang wird in einer *Abnahme* vom Kunden überprüft. Da der Kunde während der Entwicklungsarbeiten kontinuierlich über den Entwicklungsfortschritt informiert war und ihn sogar aktiv mitgestaltet hat, wird es bei der Abnahme keine Überraschungen mehr geben. Das gilt im Besonderen für das endgültige Release, dessen Iterationen der Kunde aktiv alle einzeln verfolgt hat.

Unvollständige Stories

Wenn nun das Iterationsende gekommen ist, so wird das Team hoffentlich die geplanten Stories alle umgesetzt haben. Haben Sie es nicht geschafft, eine Story bis zum Ende komplett umzusetzen, beenden Sie die Iteration und nehmen Sie die Story aus dieser Iteration heraus. Möglicherweise können Sie die Story auch teilen und einen nicht abgeschlossenen Teil mit in die nächste Iteration nehmen oder nach neuer Priorisierung zunächst mal

in das Product Backlog legen. Zu den Stories müssen selbstverständlich Tests vorliegen. Nur mit einer solch klaren Politik ist die Qualität gewährleistet und sichergestellt, dass alle den wahren Status des Projekts kennen.

Geben Sie niemals fehlerhafte oder sogar fehlende Funktionen als abgeschlossen aus! Das fällt zu einem späteren Zeitpunkt auf, so dass die Vertrauensbasis von Kunde und Entwicklungsmannschaft nachhaltig gestört ist. Wenn Fehler oder Funktionen erst nach dem offiziellen Abschluss behoben bzw. umgesetzt werden, startet das Team nicht mit einer lauffähigen Software von null in die nächste Iteration. Die Performance des Teams, also was sie tatsächlich in der Iteration geschafft hat, kann nicht gemessen werden oder wird falsch dargestellt. Davon und von der heimlichen Verschleppung von Aktivitäten in nachgelagerte Iterationen hat keiner etwas. In agilen Projekten muss eine Story mit den Tests im Konfigurationsmanagement verfügbar und dort fehlerfrei verarbeitbar sein.

Applaus

Nicht-agile Projekte verzichten auf Applaus und schütten den Entwickler nach einer Iteration (wenn es überhaupt welche gibt) nur zu gern mit den nächsten Arbeitspaketen zu. In agilen Projekten wird der Abschluss jeder Iteration ein wenig »gefeiert«. Die Iteration, oder später auch das ganze Release, wird offen dargestellt. Richten Sie ein Treffen ein, in dem die neuen Features mit den jeweiligen Entwicklern, die sich hier hauptsächlich eingebracht haben, in Verbindung gebracht werden. Dadurch wird die Leistung honoriert und im Unternehmen sichtbar gemacht.

Retrospektive

Bestandsaufnahmen werden nicht nur während der Iterationen durchgeführt. Wichtig sind diese Veranstaltungen auch nach der Iteration und nach dem Release, um gemeinsam zu identifizieren, wo es Probleme gegeben hat. Das darf kein Fingerzeigen werden, sondern es sollte eine gemeinschaftliche Analyse mit allen Beteiligten sein, also dem Team und dem Kunden, was optimiert werden kann. Wie häufig hört man »Dies und das brauchen wir nicht« oder »Wir können das schon« oder »Das machen wir!«. Ich werde immer stutzig, wenn ich die Behauptung höre, dass ein Team alles schon perfekt kann. Das ist für mich ein Indikator, dass das Team große Probleme haben könnte und nach außen den Eindruck von perfekter Arbeit erwecken möchte. Es gibt immer etwas zu verbessern!

Agile Teams befinden sich in einem stetigen Lernprozess. Die Erkenntnisse, die ein Team in einer Iteration gewonnen hat, helfen bei nachfolgenden Iterationen. Jeder macht Fehler. Und die sollen in Retrospektiven nicht thematisiert werden. Es geht um strukturelle und systematische Fehler, Prozessfehler, die immer wieder ähnliche negative Konsequenzen haben. Diese gilt es zu identifizieren und zu beheben. Ein einzelner Entwickler macht sicherlich Fehler, jeder ist nur Mensch, und bei der komplexen Herausforderung der Erstellung von Software keine Fehler zu machen, wäre nicht normal. Systematische Fehler mit schwerwiegenden Konsequenzen wird ein Entwickler aber sicher selten machen.

In solchen Fällen ist häufig der Prozess nicht in Ordnung. So etwas muss sachlich und offen im Team angesprochen werden. Haben Sie mehrere Probleme identifiziert, nehmen Sie sich eines Problems an und konzentrieren Sie sich in der nächsten Iteration auf seine Lösung. Bei der nächsten Retrospektive können Sie dann diskutieren, ob das Problem eliminiert wurde oder nicht. Genauso werden in einer Retrospektive aber auch gute Dinge hervorgehoben, z. B. Sachen, die sich bewährt haben!

KAPITEL 6

Agile Prozesse einführen

Für das Einführen von agilen Prozessen existieren verschiedene Ansätze. Wir haben bereits herausgearbeitet, dass kein Projekt, keine Unternehmensstruktur wie die andere ist. Es existiert somit keine *Silver Bullet*, also die *eine* Lösung, die schablonenhaft überall angewendet werden kann. Es sind je nach Ausgangssituation und Zielrichtung unterschiedliche Wege möglich und auch Herausforderungen denkbar, mit denen Sie bei der Einführung konfrontiert sein werden. Nachdem Sie dieses Kapitel gelesen haben, kennen Sie Strategien und mögliche Fallstricke bei der Einführung von agilen Prozessen. Sehen Sie die Strategien komplementär: Es sind verschiedene, sich gegenseitig ergänzende Ansätze.

Agil zu entwickeln bedeutet, während der Entwicklung dazuzulernen! Dies gilt insbesondere für den Prozess selbst. Auch die Einführung eines agilen Prozesses ist eine Art von Lernen. Sich selbst und seine Arbeit immer wieder in Frage zu stellen und die Arbeit zu optimieren und zu verbessern, ist elementarer Bestandteil agiler Prozesse. Bei Toyota hat man Jahrzehnte gebraucht, um den Produktionsprozess bis zu der Qualität hin zu optimieren, die er jetzt hat.

Wer überzeugt werden muss

Abhängig von den Rahmenbedingungen, die Sie vorfinden, muss die Einführung agiler Entwicklung mit mehr oder weniger viel Überzeugungsarbeit begonnen werden. Der Vorgesetzte muss das neue Vorgehen unterstützen, die Kollegen müssen sich auf die agile Entwicklung einstellen und der Kunde muss seine veränderte Rolle im Entwicklungsprozess verstehen und leben. Sie sollten allen beteiligten Personen den konkreten Nutzen agiler Entwicklung erläutern, siehe Kapitel Die Vorteile Agiler Entwicklung.

Vorgesetzte

Obwohl es denkbar ist, agile Ansätze im Verborgenen zu praktizieren, sollte das Management das Vorgehen im Optimalfall begreifen und aktiv fördern. Der Vorgesetzte ist dabei die Person, die gewöhnlich den Projekterfolg innerhalb des Unternehmens »nach oben«

vertreten muss, und umso interessierter ist er an der Art und Weise, wie die Software umgesetzt wird. Hat er den Nutzen begriffen und unterstützt er das Team bei der Durchführung, sind die wichtigsten Grundvoraussetzungen für eine erfolgreiche Durchführung gegeben. Berücksichtigen Sie aber, wie schwer es für einen Vorgesetzten oder Projektleiter sein kann, den Entwicklungsprozess umzustellen. Wurden bereits agile Methoden genutzt, kann in einem weiteren Projekt das Vorgehen mit neuen Praktiken angereichert werden, zum Beispiel mit intensivem »Test First«. Wird aber im ganzen Unternehmen mit Hilfe traditioneller Methoden entwickelt, begibt sich der Vorgesetzte ggf. auf dünnes Eis. Auch wenn ein Wasserfall-Prozess zu suboptimaler Software führt, kann sich der Vorgesetzte dahinter verstecken, dass das Unternehmen immer schon auf diese Weise Software erstellt hat. Er hat »alles versucht, damit das Projekt erfolgreich wird«. Führt er aber eine neue Entwicklungsweise ein und es kommt zu Rückschlägen oder einem Scheitern des Projekts, wird der Vorgesetzte mit dem Vorwurf konfrontiert, er hätte es besser wissen müssen und das alte Vorgehen beibehalten sollen. Ähnliche Abhängigkeiten sind häufig in großen Unternehmen anzutreffen, in denen per se eher auf Altbewährtes vertraut wird.

Under Cover

Sie haben auch die Möglichkeit, innerhalb eines schwergewichtigen Prozesses quasi verdeckt mit der agilen Entwicklung zu starten. Häufig kann man auch in einem nicht-agilen Umfeld relativ agil agieren, oder zumindest einige agile Praktiken einsetzen, nur eben *Under Cover*. Bei vielen Praktiken funktioniert das ausgezeichnet. Wenn Sie ein Wasserfall-Modell erstellen, können Sie beginnen, Unit-Tests zu schreiben, auch ohne dass das Management dies als Umstieg auf ein anderes Prozessmodell bewertet. Der Erfolg ist sicherlich mit Vorsicht zu bewerten. Haben Sie in Ihrem Unternehmen nicht die Möglichkeit, ein agiles Pilotprojekt zu durchlaufen, kann es tatsächlich eine erste Strategie sein, heimlich agile Methoden anzuwenden, um sie anschließend intern als erfolgreich zu vermarkten.

Kollegen

Die Kollegen machen das Team aus und führen das Projekt (hoffentlich) zum Erfolg. Es gibt selbstverständlich unterschiedliche Charaktere im Team, und jeder geht mit Änderungen anders um. Es ist normal, dass der Mensch Bekanntes nicht gern aufgibt. Nehmen Sie den Kollegen die Scheu und fördern Sie an erster Stelle die Werte. Sind die Personen mutig und haben Respekt voreinander, gehen sie offen mit den Schwierigkeiten des neuen Prozesses um. Und wenn ein Team ganz neu zusammengestellt wurde, muss es sich erst finden. Sie werden bei manchen Personen mehr Schwierigkeiten haben als bei anderen. Ganz sicher werden einzelne Kollegen das Pair Programming als fantastisches Neuland genießen. Andere werden es blockieren und sich sträuben, weil ihnen suspekt ist, dass ihnen ein anderer während der Arbeit über die Schultern guckt oder nach der

Arbeit die Komponente anfasst und weiterentwickelt. Hier sind häufig Ängste im Spiel, der Kollege könnte die Arbeit nicht gut finden und an den Fähigkeiten des Kollegen zweifeln. Treten Sie Ängsten bewusst entgegen und fördern Sie die Grundwerte der agilen Entwicklung, indem Sie ihren Nutzen betonen.

Kunde

Es ist sicher möglich, eine Software agil zu erstellen, ohne dass sich der Kunde »agil« einbringt. Möchte der Kunde beispielsweise nicht regelmäßig Feedback geben, können Sie den Kunden simulieren. Ein Vertreter des Team fungiert als Kunde und simuliert dessen Wünsche. Viel besser ist es natürlich, wenn der Kunde die agile Entwicklung und seine Rolle darin versteht. Machen Sie den Kunden mit dem Nutzen vertraut und machen Sie ihm die aktive begleitende Mitarbeit an der Entwicklung »seiner« Software schmackhaft. Es kann tatsächlich Spaß machen, sich nicht nur hinter Spezifikationsdokumenten zu verstecken, sondern aktiv Einfluss auf die Software zu nehmen!

Nach zwei, drei sehr produktiven Iterationen folgt vielleicht eine Iteration, die kaum neue Funktionalität bereitstellt. Engagement des Kunden bedeutet auch Vertrauen. Sehen die Kunden, wie Sie regelmäßig Software bereitstellen und die Software kontinuierlich ausgebaut wird, wird man es Ihnen auch verzeihen, wenn eine Iteration vielleicht eher aus internen Refactorings besteht als aus neuen Funktionen.

Einführungsstrategien

Seien Sie vorsichtig mit dem Kopieren von Prozessen, die in anderen Projekten funktioniert haben. Sie müssen einen für *Ihr* Projekt optimalen Weg finden.

Pilotprojekt

Eine exzellente Strategie ist der Start mit einem Pilotprojekt (einem Release). Ein Pilotprojekt kann ein Projekt sein, das für das Unternehmen nicht lebenswichtig ist und an dem sich der neue Prozess üben lässt. Auch kleinere, wenig komplexe Projekte ohne viele Wechselwirkungen mit anderen Abteilungen oder unterschiedlichen Kunden sind dazu prädestiniert, ein Pilotprojekt zu sein. Sie können aber auch ein ganz normales Projekt als Pilotprojekt angehen und daran ausprobierend das neue Vorgehen anwenden.

Für ein Pilotprojekt brauchen Sie die Unterstützung des Managements. Sie führen das Pilotprojekt komplett nach agilen Prinzipien durch, wie wir es uns angeschaut haben. Dazu ist es hilfreich, einen erfahrenen externen Coach ins Haus zu holen, der das gesamte Projekt von Anfang bis Ende begleitet, siehe unten. Dieser Coach gibt Anweisungen, erklärt und führt. Nach dem Abschluss des Piloten haben das Team und das Management wertvolle Erfahrung mit dem Prozess gesammelt, die in den nächsten Releases vertieft wird. Wenn es Ihre Aufbauorganisation unterstützt, bilden Sie ein Team für die agile Entwicklung aus, um anschließend das Wissen intern innerhalb des Unternehmens weiterzugeben.

Die Teammitglieder des Pilotprojekts können im weiteren Verlauf in anderen Projekten eingesetzt werden und dort als Coaches dienen. Eher unglücklich ist es, wenn Sie mehrere Projekte parallel auf agile Entwicklung umstellen – so lernen die Beteiligten nicht voneinander. In der Praxis wird dieser Fall aber ohnehin eher unwahrscheinlich sein, da das Management einen neuen Entwicklungsprozess zunächst an einem Projekt in Aktion sehen will (eine umfassende gleichzeitige Umstellung ist für das Management mit einem zu großen Risiko verbunden).

Als Ganzes einführen vs. in Teilen einführen

Sie fangen mit einem Pilotprojekt an. Nun stehen Sie vor der Frage, welche Komponenten aus dem gesamten Vorrat an agilen Praktiken Sie anwenden sollen. Erstellen Sie beispielsweise für Ihren Code JUnit-Tests. Führen Sie Time-Boxing durch. Ein Code ist erst fertig, wenn der fachliche Code und die Tests geschrieben wurden. Oder etablieren Sie kurze Feedback-Schleifen.

Viele Verfechter des Extreme Programming vertreten die Meinung, diese punktuelle Einführung könne nicht von Erfolg gekrönt sein, und nur im Komplettpaket sei agile Entwicklung erfolgreich. Das sehe ich anders. Gerade zu Beginn ist Ihr Team unerfahren in der agilen Entwicklung, und Sie können darüber nachdenken, zunächst mit grundlegenden Dingen wie Testautomatisierung oder der Entwicklung von Unit Tests anzufangen. Behalten Sie aber im Hinterkopf, dass das nur die ersten Schritte sein können, sich der agilen Entwicklung zu nähern. Verkaufen Sie dies auch entsprechend dem Management: Falls eine einzelne Methode bei Ihnen nicht erfolgreich einsetzbar ist, heißt das nicht, dass die ganze agile Einführung misslungen wäre. Rückschläge gehören zur Einführung dazu. Funktioniert eine Methode wie Pair Programming gar nicht, dann nutzen Sie sie zukünftig nicht mehr.

Was die Einführung nicht nur erleichtert, sondern deutlich mehr Erfolg verspricht, ist der Einsatz eines Coaches.

Coach

Wie Sie sich im Detail auch entscheiden, es sollte zunächst immer ein Experte der agilen Softwareentwicklung das Projekt begleiten. Es reicht nicht aus, dass sich die Beteiligten selbst mit der Materie beschäftigt haben. Sie können Antimuster allein nicht richtig identifizieren. Nur ein erfahrener Coach nimmt Fehler im Entwicklungsprozess wahr. Profitieren Sie von seinen Erfahrungen.

Doch wie kommen Sie an einen guten Coach? Wenn es diese Person in Ihrer Unternehmung nicht gibt, dann machen Sie es so, als wenn Sie anderes Wissen in Ihre Firma holen wollen: Engagieren Sie einen externen Experten. Das sollte ein erfahrener, agiler Praktiker sein oder eine ganze Unternehmensberatung, die mit agiler Entwicklung vertraut ist und Ihnen zur Seite stehen kann. Neben individuellen Experten gibt es immer mehr Firmen, die agile Entwicklung in Ihrem Portfolio haben. Obwohl das ein Indiz für Kennt-

nisse der Prozesse sein kann, sollten Sie mit Zertifizierungen aufpassen. Ein »Scrum Master« hat in einem zweitägigen Kurs die Grundbegriffe der agilen Entwicklung gelernt, aber ggf. noch nie ein agiles Projekt durchgeführt. Erfahrene Praktiker sind immer vorzuziehen.

Nach Abschluss eines Projekts können die eben noch selbst »gecoachten« Teammitglieder ihr Wissen innerhalb der Unternehmung weitergeben und wiederum selbst als Coach agieren.

Werte einführen

Abhängig davon, wo Sie derzeit mit Ihrem Projekt stehen, kann agile Entwicklung mit einem umfassenden Umdenken verbunden sein. Dieses Umdenken bezieht sich auf die grundlegenden Werte des Miteinander und nicht auf die konkreten Aktivitäten. Es müssen Werte wie Respekt, Mut und Offenheit in den Vordergrund gestellt werden, die mit konkreten Techniken zunächst nichts zu tun haben. Sie können Ihr Projekt zu einer agilen Entwicklung hinüberführen, indem Sie sich zunächst ausschließlich auf die Förderung der Werte konzentrieren:

- Stellen Sie sich die Frage, wie Sie die Teamarbeit verbessern können. Haben Sie leistungsabhängige Vergütung? Überprüfen Sie, ob sie wirklich fair ist und für die Teamarbeit nicht kontraproduktiv ist.
- Fangen Sie bei sich selbst an! Falls Sie Entscheidungsträger sind, haben Sie eine Vorbildfunktion. Aber auch als Mitarbeiter gilt: Gehen Sie respektvoll mit den Kollegen um, tun sie das auch mit Ihnen. Und wenn es interne Schwierigkeiten im Team gibt, dann versuchen Sie, den Grund dafür zu identifizieren. Hier muss der Manager helfen, vergangene Enttäuschungen oder andere Gründe für Missmut zu kompensieren. Binden Sie das Management immer wieder in das Projekt mit ein. Genug Synchronisierungspunkte und -möglichkeiten haben wir in den vorherigen Kapiteln vorgestellt.
- Fördern Sie Mut und Offenheit bei den Mitarbeitern, so dass sie direkt Probleme ansprechen. Um die Offenheit und den Mut zu fördern, kann bewusst ein *Doomsayer* zur offenen, zeitnahen Aussprache von schlechten Nachrichten oder möglichen Schwierigkeiten eingesetzt werden. Ein Doomsayer in Ihrem Projekt wäre beispielsweise eine Person, die jederzeit das Negative im Prozess sucht und darstellt. Die Person, die dieses Amt ausübt, wechselt. Jeder weiß, wer gerade Doomsayer ist, so dass Kritik von der Person nicht persönlich genommen wird, da sie zu ihren Aufgaben gehört. In dem Zusammenhang ist auch ein *Advocatus Diaboli* hilfreich, eine Person, die bewusst die Argumente der Gegenseite vertritt, ohne zwangsläufig selbst an sie zu glauben.
- Fördern Sie Entscheidungen Ihrer Mitarbeiter. Lassen Sie Freiraum zu. Jeder Mensch macht Fehler. Wenn ein Mitarbeiter mehrfach denselben Fehler macht oder ähnliche Fehler wiederholt auftreten, führen Sie Gespräche. Diese Gespräche sollten unbedingt unter vier Augen stattfinden.

Agile Planspiele

Es existieren verschiedene Möglichkeiten, wie Sie dem Team die agile Entwicklung nahebringen können. Eine Möglichkeit ist das Planspiel, das ursprünglich aus der Extreme Programming-Schiene hervorging: Die Mannschaft (mit Kunde oder ohne) durchläuft spielerisch ein agiles Projekt mit all seinen Einzelschritten. Besteht das Team zunächst nur aus Entwicklern, dann teilen Sie die Gruppe auf. Teilen Sie einigen Entwicklern die Entwicklerrolle zu und anderen die Kundenrolle. Ein Coach sollte immer dabei sein. An einem exemplarischen Projekt werden alle Beteiligten geschult.

Eine interessante Variante, agile Entwicklung einzuführen, kann auch anders aussehen: Betrachten Sie die Einführung als ein Projekt. Organisieren Sie sich einen erfahrenen Experten für agile Entwicklung. Setzen Sie sich mit dem Team zusammen und reden Sie über die Schwierigkeiten, die in den Projekten auftreten. Schlagen Sie agile Lösungsansätze vor, die genau diese Schwierigkeiten lösen. Notieren Sie diese Ansätze als User Stories auf Karten, die sie für alle gut sichtbar aufhängen. Nachdem alle gemeinsam alle Probleme formuliert haben, lassen Sie das Team die agilen Karten nach Priorität ordnen. Fangen Sie mit der Umsetzung der am höchsten eingestuften Karte an. Die anderen Karten merken Sie sich in einem Backlog. Auf diesem Weg umgehen Sie einerseits das Risiko, zu viel auf einmal ändern zu wollen. Ferner konzentrieren Sie sich bei den ersten Änderungen auf die wirklich wichtigen Dinge, die das Team selbst als solche herausgearbeitet hat. Das Team wird spielerisch an das agile Vorgehen herangeführt und baut auch direkt die Verantwortung auf, da es sich bei der Auswahl der ersten Schritte aktiv einbringt. Es ist sein eigener Kunde.

Fallstricke

Folgende Fallstricke sind bei der Einführung von agilen Projekten häufig zu beobachten:

Mitarbeiter werden »überfahren«
: Der Mensch braucht Sicherheit. Er ändert sich und seine Arbeit nicht gern. Verstehen Sie diese natürliche Einstellung und führen Sie die Leute behutsam zur agilen Herangehensweise. Manche Unternehmen gehen den radikalen Schritt und setzen Personen frei, die sich nicht ändern wollen – das ist meistens völlig überflüssig.

Zu hohe Geschwindigkeit bei der Einführung
: Falls es bei der Einführung Schwierigkeiten gibt, geben Sie den Personen maximale Unterstützung. Sie müssen alle Beteiligten »mitnehmen«.

Zu langsame Geschwindigkeit bei der Einführung
: Es werden nicht alle agile Werte oder Praktiken gleichzeitig eingeführt. Das kann den Erfolg minimieren und ein falsches Licht auf die neue Herangehensweise werfen (»Jetzt haben wir damit angefangen, und es bringt nichts.«).

Ernüchterung nach Rückschlägen
: Es wird auch Rückschläge geben. Werfen Sie die Flinte nicht direkt ins Korn!

Starke Opposition im mittleren Management

Das mittlere Management ist für den Erfolg maßgeblich verantwortlich. Fällt das höhere Management beispielsweise die Entscheidung, strategisch agil Software zu entwickeln, dann muss das mittlere Management diese Entscheidung mittragen und ausführen.

Führung, Zusammenarbeit und das Team

Ich hatte mich im Rahmen eines Projekts mit der Bewertung verschiedener Werkzeuge im Bereich J2EE beschäftigt und war bei der Benutzung dieser Werkzeuge mittlerweile ein ausgewiesener Experte. Da hörte ich eher durch Zufall, dass sich eine andere Person nun ebenfalls und ganz allein mit der Evaluierung beschäftigen sollte. Er bekam den exklusiven Auftrag vom Vorgesetzten, obwohl kurz zuvor ich derjenige war, der sich mit der Materie befassen sollte. Ich wandte mich direkt an den Entscheidungsträger mit dem Hinweis, dass ich mich genau mit dieser Evaluierung beschäftigte. Er nahm mich ins Schlepptau und wir gingen zu dem besagten Spezialisten. Im Raum saßen noch einige andere Leute, die unser folgendes Gespräch aufmerksam verfolgten. Der Entscheidungsträger sagte zu dem Kollegen »Der Herr Hüttermann sagte mir gerade, Sie könnten mit dem Werkzeug nicht umgehen.« Dieser eine Satz illustriert nur zu gut, wie wenig der Entscheidungsträger den agilen Entwicklungsprozess verinnerlicht hatte.

Machtverlust

Immer wenn Sie einen internen Prozess ändern, wird sich auch beim Aufgabenbereich und den Verantwortlichkeiten der Beteiligten etwas ändern. Das ist eventuell mit Machtverlust verbunden. Machtverlust kann ein Verhalten zur Folge haben, das den Werten agiler Entwicklung entgegensteht. Gerade im agilen Vorgehen geht mehr Macht auf das gesamte Team über. Dies kann das Management irritieren, da es Macht an die Entwickler abgeben muss. Auch hier hilft nur offenen Kommunikation und stetige Überzeugungsarbeit. Schwören Sie alle auf den Teamerfolg ein. Appellieren Sie an den Pioniergeist und die Innovationsfreude Ihrer Manager!

Der Kunde lehnt sich zurück und arbeitet nicht mit

Überzeugen Sie ihn von den Vorteilen und konfrontieren Sie ihn mit den Konsequenzen seiner Weigerung zur Mitarbeit.

Skepsis des Entscheiders bei der Einführung

Dem Entscheider wurde Skepsis antrainiert. Im Laufe der Jahre haben regelmäßig Berater Versprechungen gemacht. Er ist vom Nutzen erst überzeugt, wenn ein Pilotprojekt erfolgreich verlaufen ist.

Praktiken sind wichtiger als Werte

Die Werte werden wenig oder gar nicht verfolgt. Alle konzentrieren sich nur auf die Praktiken. Diese ohne Beachtung der Werte anzuwenden, bringt keinen Erfolg.

Mangelhaftes Verständnis, mangelnde Disziplin/Kompetenz (insbesondere der Entwickler)
Die Entwickler verstehen und leben die agilen Werte nicht. Praktiken wie »Continuous Integration« und »Test-Driven Development« erfordern große Disziplin und Kompetenz. Achten Sie auf diese vereinbarten Gepflogenheiten!

Pathologische Politik

Sie werden daran scheitern, agile Werte in einem Unternehmen einzuführen, das unter pathologischer Politik leidet. Durch sachlich nicht nachvollziehbare, nicht konstruktive Maßnahmen versucht ein Einzelner, seine Macht auszubauen bzw. zu erhalten. Dies kann sich beispielsweise durch ständiges Beschuldigen anderer manifestieren. Hier werden gern gerade diejenigen kritisiert, die nicht mehr für das Unternehmen arbeiten. Die Erscheinungsformen von pathologischer Politik, z. B. Diskreditierungen, Mobbing und anderes, stehen den Unternehmenszielen entgegen und erschweren es, die richtige Software mit hoher Qualität effizient bereitzustellen. Pathologische Politik kennzeichnet sich auch dadurch aus, dass Personen es vermeiden, Verantwortung zu übernehmen und schriftliches Festhalten ihrer Entscheidungen zu verhindern wissen. Somit müssen sie sich daran nicht messen lassen, was es auch einfacher macht, bei Misslingen eines Projekts die Verantwortung auf jemand anderen abzuwälzen. Dafür wird dann gern jemand genommen, der faktisch nur Entscheidungen *umgesetzt* hat. Der Vorgesetzte möchte gar nicht im Team Entscheidungen fällen lassen. Es ist für ihn viel angenehmer, wenn die Leute nicht alle Zugang zu allen Informationen haben und nach dem Scheitern eines Projekts ein einzelner an den Pranger gestellt wird. Häufig ist die pathologische Politik nicht direkt sichtbar, sondern nur die Reaktionen anderer auf diese Politik. Umso wichtiger sind Ursachenanalysen, um immer die wahren Gründe für Reaktionen Einzelner herauszufinden.

Iterieren ohne zu messen
Das Projekt ist timeboxed und Sie arbeiten iterativ und inkrementell, aber Sie messen nicht den Projektfortschritt. Ein Vorteil der strikten Durchführung von zeitlich abgeschlossenen Iterationen ist die Möglichkeit, die Team-Performance und den Projektfortschritt zu messen. Nutzen sie ihn!

Agil, aber keine Auslieferung
Sie arbeiten zwar agil in Iterationen, versäumen es aber, Ihre Inkremente auch tatsächlich auszuliefern und dem Kunden bereitzustellen.

Mehrwert und Nutzen unklar
Der Kunde hat Schwierigkeiten damit, den Geschäftswert einzelner Features zu bestimmen. Bieten Sie die Hilfe Ihres Coaches an, fordern Sie aber Input und aktive Mitarbeit ein.

Feedback wird nicht aufgenommen
Zu Beginn wurde noch agil entwickelt, es wird aber versäumt, die stetigen Rückkopplungen in die Iterationen mit aufzunehmen, um den Prozess fortlaufend zu verbessern. Bleiben Sie am Ball und investieren Sie in die Zukunft.

Keine in sich abgeschlossene Iterationen

Sie planen die Iterationen und starten sie, ihr Inhalt ist aber nicht fix. Der Kunde ändert kontinuierlich den Inhalt der Iteration. Hier müssen das Management, der Coach oder der Projektmanager die Funktion einer Firewall übernehmen, die die Iteration und die Entwickler vor äußeren Einflüssen schützt. Stetiger Austausch mit dem Kunden ist zwar wichtig, Diskussionen über zukünftige Features, die sich nicht in der aktuellen Iteration befinden, Gespräche über Aufwandsabschätzungen usw. dürfen aber nicht Bestandteil des kontinuierlichen Austauschs während einer Iteration sein.

Es passieren die gleichen Fehler immer wieder

Die gleichen Fehler werden immer wieder neu gemacht. Führen Sie Retrospektiven durch. Wenn Sie merken, dass die Leute in den Retrospektiven gelangweilt sind und sich die Probleme nicht ändern, fassen Sie den Umfang der identifizierten und für die nächste Iteration anvisierten Änderungen knapper: Nehmen Sie sich nur zwei oder drei Dinge vor, die Sie anders machen wollen.

Integration wird unterschätzt

Der eigentliche Integrationsprozess wird unterschätzt. Wollen Sie, dass Ihnen irgendwann alles um die Ohren fliegt? Nein? Dann verpflichten Sie alle zur kontinuierlichen Integration!

Zeitverschwendung

Die Teammitglieder sind mit zu vielen Aktivitäten konfrontiert, die keinen Mehrwert liefern, sondern nur den schnellen Projektfortschritt blockieren. Ein Beispiel dafür sind unnötige oder zu lange Meetings. Fragen sie das Team nach solchen Zeitgräbern.

Wissen wird vorausgesetzt

Es existiert keine Kultur des Lernens. Führungskräfte gehen davon aus, dass das Team jederzeit alles wissen muss. Appellieren Sie wieder an die Weitsicht!

Falsche Beratung

Der bei der Einführung unterstützende Coach versteht »Agilität« nicht und liefert falsche Beratung. Geben Sie der Methodik eine zweite Chance, indem Sie sich nochmals um einen geeigneten Coach bemühen. Hören Sie sich um und achten Sie auf Referenzen.

Besondere Situationen

In Ihrem Unternehmen können besondere Rahmenbedingungen vorhanden sein, die die agile Entwicklung erschweren oder besondere Anpassungen notwendig machen. Ich möchte hiermit die am weitesten verbreiteten Konstellationen kurz vorstellen. Diese wurden über die letzten Jahre intensiv diskutiert und finden sich in vielen Büchern wieder. Ich gehe daher nur kurz auf diese Punkte ein. Konsultieren Sie die am Ende des Buchs aufgeführten Bücher, um weitere Details zu erfahren, oder fragen Sie den Coach Ihres Vertrauens nach individuellen Einführungsstrategien.

Verteilte Teams

Eine besondere Situation ist die Entwicklung von Software in *verteilten Teams*. Gerade in Zeiten von *Offshoring* (Entwicklung zum Teil oder in ihrer Gesamtheit geografisch weit entfernt) und *Nearshoring* (Entwicklung zum Teil oder ganz im benachbarten Ausland) ist es häufig der Fall, dass Software räumlich verteilt entwickelt wird. In dieser Projektstruktur sollten Sie dennoch regelmäßigen Austausch pflegen. Im Stand-up-Meeting können per Telefon Mitarbeiter dazugeschaltet werden, die sich an einem anderen Ort befinden. Nutzen Sie hier einfache Lösungen: Videokonferenzen sind manchmal ganz schön, *Telefonkonferenzen* erfüllen den Zweck aber auch und sind deutlich leichter durchzuführen. Legen Sie einen Fokus darauf, dass die Entwickler sich persönlich sehen. Mischen Sie in regelmäßigen Abständen die Entwickler durch und lassen Sie sie jeweils für ein paar Tage einen anderen Entwicklungsstandort besuchen und dort arbeiten. Die Personen pflegen ein ganz anderes Miteinander, wenn Sie sich persönlich kennengelernt haben und sich regelmäßig sehen. Ist der Kunde nicht vor Ort, führen Sie einen Kunden-*Proxy* ein. Das ist eine Person aus dem lokalen Team, die einerseits mit dem Kunden im stetigen Austausch ist, andererseits am Entwicklungsstandort die Rolle des Kunden einnimmt und ihn dort vertritt.

Fixpreisprojekte

Eine weitere besondere Konstellation ist der Umgang mit *Fixpreisprojekten*. Diese stehen grundsätzlich im Widerspruch zum agilen Vorgehen, denn hier investiert der Kunde vorweg eine bestimmte Geldsumme, um einen zuvor spezifizierten Funktionsumfang zu erhalten. Agile Projekte brauchen natürlich auch eine vertragliche Grundlage, streben aber eine größere Flexibilität an. Denken Sie daher z. B. darüber nach, ob Sie den Kunden das fertige Produkt nicht pro Feature bezahlen lassen. Denkbar wäre auch, dass Sie mit dem Kunden vertraglich die Umsetzung einer bestimmten Anzahl von Story Points fixieren, die er während der Iterationen aber frei verteilen darf.

Zertifizierungen

Eine häufig diskutierte Fragestellung ist, wie sich agile Prozesse mit *Zertifizierungsprogrammen* wie CMMI vereinbaren lassen. Zertifizierungsprogramme setzen gewöhnlich eine gewisse Planbarkeit voraus: Zu gegebenen Zeitpunkten müssen bestimmte Artefakte vorliegen. Es wird gewöhnlich ein dokumentenlastiges Vorgehen gefördert, das nicht im Einklang mit agiler Entwicklung steht.

Auf der anderen Seite bauen Zertifizierungsprogramme auch auf einen wiederholbaren Prozess mit kontinuierlichen Rückkopplungen. Und sie analysieren, wie der Prozess mit Änderungswünschen bei den Anforderungen an die Software umgeht. In diesem Sinne sind sie dann doch wieder recht agil bzw. unterstützen agile Prozesse solche Normierungsbemühungen. Beispielsweise sagt man, dass Scrum *CMMI-kompatibel* ist.

Große Teams

Ein weiterer Diskussionspunkt ist die *Skalierung*: Was passiert, wenn das Team sehr groß ist (über zwölf bis mehrere hundert Mitglieder)? Auch hier gibt es zahlreiche Ansätze, wie agile Prozesse darauf abgestimmt werden können. Häufig stellt sich z. B. die Frage, wie ein tägliches Stand-up-Meeting mit sehr vielen Personen durchgeführt werden kann. Hier hat sich *Staging* bewährt, was auch als *Scrum-of-Scrum* bekannt geworden ist: Kleine, autonome Teams arbeiten an ihrem Teilgebiet und entsenden eine Person zum täglichen Stand-up-Meeting. Dort synchronisiert sich der Vertreter mit denen der anderen Teams. Anschließend führt das einzelne Team ein Stand-up-Meeting durch, in dem der Entsendete die Neuigkeiten aus dem übergeordneten Meeting mitteilt.

Keine oder viele Kunden

Häufig fehlt in Projekten der Kunde, oder es gibt mehrere. Keinen Kunden haben Sie beispielsweise bei einem *Open Source*-Projekt. Wenn Sie ein Produkt für einen breiten Markt entwickeln, haben Sie auch keinen direkten Kunden. Auch dort fehlt der eine direkte Auftraggeber. Hier hilft es, ein Mitglied des Teams als Kundenvertreter (Proxy) zu bestimmen, der die Kundenrolle einnimmt. Das Gleiche gilt, wenn Sie mehrere Kunden haben. Bestimmen Sie eine Person, die zum Team hin als der eine Kunde auftritt, also auch ein Kunden-Proxy ist, auf der anderen Seite sich jedoch stetig mit den verschiedenen Kunden in Kontakt steht und die unterschiedlichen Anforderungen zusammenbringt.

Legacy-Systeme

Schließlich ist es eine besondere Aufgabe, wenn Sie in einem agilen Projekt eine bestehende Altanwendung (*Legacy-System*) ablösen wollen. Hier ist es häufig schwierig, in kleinen Iterationen vorzugehen, da das Altsystem nur als Ganzes in einem Schritt ablösbar ist. Versuchen Sie dennoch, kleine Arbeitspakete zu identifizieren, und wählen Sie die Release-Länge entsprechend lang. Finden Sie im Rahmen der Ablösung zunächst die zu migrierende Funktionalität des Altsystems heraus. Das klingt erstmal absurd, in der Praxis ist es aber tatsächlich so, dass nach Jahren oder Jahrzehnten nicht mehr klar ist, was die Software eigentlich genau leistet. Da es häufig keine funktionalen Tests für diese Systeme gibt, ist der erste Schritt das Abbilden der Funktionen auf funktionale Tests. Sie spezifizieren somit, welche Eingaben in das System welche Ausgaben zu Folge haben. Die neue Anwendung muss dann diese Spezifikation ebenfalls erfüllen.

TEIL II

Eine agile technische Infrastruktur

Im zweiten Teil dieses Buchs beschreibe ich eine umfassende technische Infrastruktur, auf die agile Projekte aufsetzen können. Neben den Werkzeugempfehlungen erhalten Sie Hinweise zu Einsatz und Integrationsmöglichkeiten sowie Tipps, wie Sie sie agil nutzen können. Die zahlreichen Tipps und Nutzungsszenarien erheben dabei keinen Anspruch, eine umfassende Entscheidungsmatrix darzustellen, denn jedes Projekt ist anders. Der Werkzeugteil soll Ihnen die einzelnen Werkzeuge näher bringen und ihre Nutzung verdeutlichen. Ist Ihr Interesse an einem Werkzeug geweckt und denken Sie, dass das Werkzeug in Ihrem Projekt sinnvoll eingesetzt werden könnte, dann schauen Sie sich für weitere Informationen die zu dem Werkzeug verfügbare Literatur bzw. seine Dokumentation an.

Die Werkzeuge decken die in Kapitel 5, *Die Durchführung*, eingeführten *Building Blocks* der agilen Entwicklung ab. Dazu gehört das Versionierungssystem *Subversion,* mit dem Sie Ihre Artefakte (fachliche Klassen, Testklassen, Testergebnisse) verwalten können und das das Rückgrat Ihrer Entwicklung darstellen kann.

Ein weiterer sehr wichtiger Themenkomplex ist das *Testen*. Dabei geht es einerseits um funktionale Tests bzw. Akzeptanztests, andererseits um Komponententests. Akzeptanztests lassen sich mit *Fit* (siehe Kapitel 15, *Akzeptanztests mit Fit*), *WebTest* (siehe Kapitel 16, *Akzeptanztests für Webanwendungen mit WebTest*) und *Selenium* (siehe Kapitel 14, *Webanwendungen testen mit Selenium*) umsetzen. Mit Fit können Sie funktionale Tests auf Java Swing- oder Webanwendungen ausführen. Es lässt sich mit anderen Werkzeugen wie Selenium oder JUnit integrieren. WebTest basiert auf HTTPUnit und erlaubt die deklarative Beschreibung von funktionalen Tests auf Webanwendungen. Es ist eine Abstraktion: Der Anwender muss nicht mehr mit der eigentlichen zugrundeliegenden Test-Engine HTTPUnit arbeiten. Selenium geht einen anderen Weg: Hier nehmen Sie gewöhnlich die Schritte einmalig auf, um sie dann erneut abspielen zu können. Selenium ist insbesondere für Ajax-Anwendungen geeignet. Um die Kundenanforderungen mit der Anwendung zu synchronisieren und ihre Funktionalität zu testen, sollten Sie auf zumindest eins dieser Werkzeuge aufsetzen.

Neben den funktionalen Tests existieren Komponententests, die mit *JUnit* (siehe Kapitel 9, *Komponententests mit JUnit*) und/oder *TestNG* (siehe Kapitel 12, *Komponententests mit TestNG*) durchgeführt werden. Komponententests sind bei agilen Projekten ein absolutes Muss. Die von Entwicklern geschriebenen Tests überprüfen dabei, ob die Klassen Funktionalität richtig implementieren. Sie werden komplementär zu den funktionalen Tests geschrieben, die die fachliche Funktionalität der Anwendung testen und im Fall von Akzeptanztests das erwartete Verhalten vor dem Schreiben der Anwendung spezifizieren.

TestNG ist eine Lösung, die in mancher Hinsicht mächtiger als JUnit ist, wenn es z. B. um Testgruppierung und nebenläufige Tests geht. TestNG und JUnit können Sie mit Selenium integrieren. Zu testende Komponenten werden von lange laufenden Komponenten mithilfe von *EasyMock* (siehe Kapitel 11, *Der Einsatz von Mock-Objekten mit EasyMock*) isoliert. *Jemmy* (siehe Kapitel 13, *GUIs testen mit Jemmy*) hilft dabei, Komponenten auf der Oberfläche zu finden bzw. zu bedienen, und kann im Rahmen von Komponententests, aber auch zusammen mit Fit eingesetzt werden. Da die Entwicklung von Tests die Entwicklung der fachlichen Klassen führt und ihre Spezifikation darstellt, spricht man von T*est-Driven Development (TDD)*. Wie das generell aussehen kann, beschreibe ich für JUnit in Kapitel Komponententests mit JUnit.

Ergänzend dazu werden Coding-Standards überprüft, um die Qualität hoch zu halten und Defekte frühzeitig zu identifizieren und zu beheben. Mit *Checkstyle* (siehe Kapitel 18 ab Seite 338) werden Coding-Konventionen, mit *EMMA* (siehe Kapitel 10, *Testabdeckung mit EMMA und JUnit*) die Testabdeckung überprüft.

Da Sie die einzelnen Schritte (Testen, Standards überprüfen, SCM-Aktivitäten wie Tags erstellen) permanent begleitend und automatisiert durchführen sollten, setzen Sie nicht nur auf einen strikten Prozess der *kontinuierlichen Integration*, sie setzen auch Werkzeuge ein, die dieses Vorgehen unterstützen. Hauptwerkzeug ist dabei *Ant* (siehe Kapitel 7, *Ant-Grundlagen* und Kapitel 18, *Ant – Teil 2: Praktische Einsatzszenarien*), das als Schweizer Taschenmesser alle Werkzeuge automatisiert bedienen kann und dem Team die Ergebnisse, die die Werkzeuge liefern, bereitstellt. Ant ist das Bindeglied zwischen allen Einzelwerkzeugen. Ant stoßen Sie aber auch nicht manuell an. *CruiseControl* (siehe Kapitel 19, *Kontinuierliche Integration mit CruiseControl*) hilft dabei, die Skripten zu bestimmten Zeitpunkten anzustoßen und die Ergebnisse bereitzustellen. *Maven* (siehe Kapitel 20, *Der Einsatz von Maven für Build-Prozesse*) ist ein Werkzeug, das Abhängigkeiten zwischen Ihren Softwarebibliotheken verwaltet und den Build-Prozess unterstützen kann. Es steht dabei in Konkurrenz zu Ant, kann aber auch gemeinsam mit ihm eingesetzt werden.

Schließlich führt das Kapitel über *Trac* (siehe Kapitel 17, *Projektkommunikation mit Trac*) in das gleichnamige Wiki ein. Ein Wiki sollte zu jedem agilen Projekt gehören, um Wissen zentral zugänglich zu machen.

Die hier vorgestellten Werkzeuge sind Empfehlungen, die Ihnen dabei helfen können, schnelle Rückkopplungen zu erhalten, Qualität und Zufriedenheit hoch und das Risiko niedrig zu halten.

KAPITEL 7

Ant-Grundlagen

Ant ist ein in Java geschriebenes Werkzeug, das hauptsächlich als Build-Tool weite Verbreitung hat. Es ist ein von Apache entwickeltes Open Source-Projekt. Der Begriff Ant ist ein Akronym für *Another Neat Tool* (»noch ein hübsches Werkzeug«). Es kann trotz seiner Kompaktheit wertvolle Dienste leisten (wie Ameisen trotz ihrer überschaubaren Größe viel leisten können). Es entstand als Teil des Apache Tomcat-Projekts. Im Jahr 2000 wurde es aus der Tomcat-Codebasis extrahiert und wird seitdem separat fortgeschrieben.

Ant ist ein *Schweizer Taschenmesser*. Es ist sehr mächtig und erweiterbar und unterstützt den Entwickler dabei, viele einzelne Schritte zu automatisieren. Es ist keine Programmiersprache, sondern ein Rahmenwerk, das via Kommandozeile genauso genutzt werden kann wie über eine IDE-Integration. Ant kann in vielen Anwendungsbereichen genutzt werden, wie wir im weiteren Verlauf sehen werden. Wurden früher ausführbare Programme aus existierendem Quelltext und notwendigen Bibliotheken mit dem hauptsächlich im Unix genutzten *make* erstellt, so bietet Ant durch seine Plattformunabhängigkeit die Möglichkeit der Portierung in Verbindung mit einer für Menschen besser lesbaren Form der Skripten. Die Skripten basieren nicht mehr auf Shellscript-ähnlichen Kommandos, sondern werden in einer für den Menschen lesbaren XML-Syntax bereitgestellt.

Da Ant gewöhnlich im Rahmen eines Build-Prozesses eingesetzt wird, spricht man bei den Skripten auch von »Build-Dateien«. Dabei wird Ant von diesen Build-Dateien gesteuert, die in einer spezifischen XML-Syntax (siehe unten) geschrieben sind und in der Regel *build.xml* heißen. Die Begriffe »Skripte« und »Build-Dateien« werden synonym verwendet.

Leerzeichen und Tabulatoren sind in klassischen *make*-Dateien optisch nicht unterscheidbar, resultieren aber in einem unterschiedlichen Verhalten, was häufig zu Problemen führte. Ferner gibt es das eher kryptische *make* in unterschiedlichen Versionen. Dafür bietet Ant ebenso Lösungen wie bei der Integration vieler öffentlicher Erweiterungen. Ant-Skripten können in allen gängigen Java IDEs entwickelt, ausgeführt und gedebuggt werden. Weitere Vorteile sind die Erweiterbarkeit und die Möglichkeit, betriebssystemspezifische Befehle aufzurufen.

In diesem Kapitel gebe ich erst einmal nur Hinweise zur Installation und Nutzung von Ant. Ich stelle Terminologie und grundlegende Konzepte dar. In späteren Kapiteln werden wir dann kontinuierlich die jeweiligen Werkzeuge auch aus Ant heraus nutzen und somit immer wieder zu dem Werkzeug zurückkehren, es besser kennenlernen und komplexere Anwendungsszenarien illustrieren. Im zweiten Ant-Kapitel (siehe Kapitel 18, *Ant – Teil 2: Praktische Einsatzszenarien*) werden wir mithilfe von Ant Entwicklungsprozesse automatisieren.

Installationshinweise

Die letzte stabile Ant-Version ist unter *http://ant.apache.org/* verfügbar. Sie benötigen eine Java-JDK 1.2 oder höher (Sie sollten die JDK nehmen, nicht nur die JRE, um die komplette Funktionalität von Ant nutzen zu können). Um Ant zu installieren, entpacken Sie die heruntergeladene Datei in einem Verzeichnis Ihrer Wahl. Sie sollten dann auf Ihrem System folgende Umgebungsvariablen exportieren, was sie am besten automatisiert in Ihrem *.profile* tun.

```
export ANT_HOME=/usr/local/app/my/path/to/ant
export JAVA_HOME=/usr/local/java/my/path/to/java/jdk1.6.0
export PATH=${PATH}:${ANT_HOME}/bin
```

Mit diesen Einstellungen können Sie, auch nach einem Neustart Ihres Rechners, von der Kommandozeile aus Ant nutzen. Davon unabhängig ist die Nutzung aus IDEs heraus. Alle gängigen IDEs wie Eclipse oder IntelliJ IDEA verfügen von Hause aus über eine Ant-Unterstützung. Ihre Skripte können Sie also auch über eine IDE editieren und ausführen. Häufig werden Ant-Skripte auch ohne IDEs genutzt, zum Beispiel im Rahmen eines »Continuous Integration«-Prozesses.

Terminologie und Konzept

Ein überschaubares Ant-Skript zeigt uns das grundlegende Konzept und dient uns als Einstieg in die Terminologie.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- author Michael Huettermann -->
<project name="Agile" default="dist">
   <target name="dist" depends="prepare">
      <jar jarfile="lib/app.jar"
        basedir="classes"
        includes="com/huettermann/**"
      />
   </target>
   <target name="prepare">
      <mkdir dir="lib"/>
   </target>
</project>
```

Die Build-Datei ist ein *gültiges* XML-Dokument, dessen Wurzelelement *project* sein muss und das mit dem Befehl *ant* oder in Ihrer IDE angewendet werden kann. Das Skript ist immer ähnlich strukturiert: Es besteht aus verschiedenen *target*-Elementen. Diese kennzeichnen Sprungziele in das Skript, die separat angesprochen und ausgeführt werden können. Von außen wird ein Ant-Skript gewöhnlich direkt über ein *target* aufgerufen. Häufig wird ein standardmäßiges Default-Target innerhalb der Projektdefinition (im Wurzelelement *project*) angegeben. Dieses wird ausgeführt, wenn das Skript gestartet wird, ohne dass ein explizites Target zur Ausführung angegeben ist.

Ein Aufruf unseres Beispielskripts oben auf der Kommandozeile kann also folgendermaßen aussehen:

1. `ant prepare`: Das Target namens »prepare« wird direkt angesprochen und erstellt ein Verzeichnis.
2. `ant dist`: Target namens »dist« wird angesprochen und erstellt eine *.jar*-Datei. Aufgrund des »depends«-Attributs wird vorher allerdings noch das Target »prepare« durchlaufen.
3. `ant`: Das Default-Target wird angesprochen, in diesem Fall »dist«.

Neben den frei wählbaren strukturierenden »Targets« stellt Ant feste *Tasks* zur Verfügung. Tasks sind kleine, ausführbare Elemente, die eine definierte Ant-Funktionalität bzw. die Integration mit einem anderen Werkzeug anbieten. Sie stellen das eigentliche Herzstück der Skripten dar. Tasks wie die im Beispiel genutzten `jar` oder `mkdir` werden durch ihre Attribute und verschachtelte Elemente konfiguriert. Es gibt im Standardumfang von Ant integrierte Tasks und optionale Tasks, die noch explizit eingerichtet werden müssen. Tasks lassen sich in folgende Kategorien unterteilen: Es gibt Tasks zur Archivierung, zum Auditing (Protokollierung von Prozessen), zur Kompilierung, zum Deployment, zur Dokumentation, zur Ausführung, zum Arbeiten mit Dateien, für Zugriffe auf fremde Rechner, zum Testen und für viele andere Hilfsaufgaben. Mit Java lassen sich ferner leicht eigene Tasks entwickeln.

Die externe Bibliothek *ant-contrib* erweitert Ant um zusätzliche Tasks, beispielsweise zur Realisierung von Schleifen und Alternationen. Werkzeuge bieten häufig eigene Ant-Tasks an, um ihre Nutzung aus Ant heraus zu ermöglichen.

Im Beispielskript sehen Sie das Target `dist`, um Klassen im *classes*-Verzeichnis in das Java-Archiv *app.jar* im zu Beginn erstellten *lib*-Verzeichnis zu packen. Es werden dabei alle Klassen der Java-Pakete *com.huettermann* und untergeordnete berücksichtigt.

Das hier Gesagte sollte für einen ersten Start reichen. Weitere Details und Nutzungsbeispiele werden in den weiteren Kapiteln des Buchs beschrieben.

KAPITEL 8

Versionskontrolle mit Subversion

Die Versionskontrolle ist das Herzstück eines jeden agilen Softwareprojekts. In diesem Kapitel stelle ich mit Subversion ein kostenlos erhältliches Werkzeug vor, mit dessen Hilfe Sie Ihr agiles Projekt unter Versionskontrolle bringen können. Neben den grundlegenden Konzepten der Versionskontrolle, in diesem Zusammenhang auch als *Software Configuration Management (SCM)* bekannt, und Subversion selbst, illustriere ich insbesondere die *agile*, automatische Nutzung des Werkzeugs aus Ant heraus. Abschließend stelle ich drei Werkzeuge vor, die den täglichen Umgang mit Subversion erleichtern.

Versionskontrolle

Ein Versionskontrollsystem (VCS) ist die Basis eines jeden agilen Softwareentwicklungsprozesses. Sein Kern bildet dabei ein *Repository*, eine zentrale Stelle, an der die Projektartefakte verwaltet und versioniert werden.

Haben Sie schon einmal mit OpenOffice oder einem anderen Werkzeug ein Textdokument auf einem zentralen Laufwerk fortgeschrieben? Vielleicht sogar mit mehreren Personen gleichzeitig? Dann wissen Sie die Vorteile eines Versionskontrollsystems zu schätzen. Arbeiten Sie allein an einem Softwareprodukt, mag es für Sie noch möglich sein, ohne Versionskontrollsystem auszukommen. Aber bereits da stellt sich häufig die Frage »Wie sah meine Software zum Zeitpunkt X aus?«

Die Artefakte unter Versionskontrolle zu haben bedeutet, dass Personen gleichzeitig mit derselben Datei arbeiten können. Daneben kann jeder zurückliegende Stand der Software beliebig wiederhergestellt werden. Möchten Sie also wissen, wie die Software am Tag X oder im Zustand Y aussah, dann können Sie diesen Stand aus dem zentralen Repository herausholen.

Aber nicht nur sämtliche historische Stände sind zugreifbar, auch der aktuelle Entwicklungsstand kann jederzeit eingesehen und weiterverarbeitet werden. Das betrifft nicht nur die Software als Ganzes, sondern auch ihre Einzelteile, also jede einzelne Datei. Ein weiterer Vorteil eines Versionskontrollsystems ist, dass durch Änderungen an einer Datei der alte Zustand nicht unwiderruflich überschrieben wird: Änderungen können zurück-

genommen werden. Und es ist komfortabel möglich, zwei Versionen mehrerer Artefakte von unterschiedlichen Zeitpunkten gegenüberzustellen.

SCM-Terminologie

Die wesentlichen Begriffe beim SCM sind

- *Tag* oder *Label*: Eine eingefrorene Momentaufnahme der Artefakte, der ein symbolischer Name zugeordnet wird.
- *Repository*: Der zentrale Ort, an dem die Artefakte aufbewahrt werden.
- *Workspace* oder *Sandbox*: Der Entwickler holt sich eine Arbeitskopie auf den Rechner, um die Artefakte fortzuschreiben. Lokale Änderungen im Workspace sind für andere Entwickler zunächst nicht sichtbar.
- *Head* oder *Trunk*: Der Hauptentwicklungszweig. Hier wird das Produkt gewöhnlich weiterentwickelt. Änderungen der Entwickler in ihren Workspaces werden erst durch das Einspielen in den Head dem zentralen Codestand zugeführt und für andere Entwickler sichtbar.
- *Branch*: Ein paralleler, unabhängiger Entwicklungszweig, der vom Trunk oder von einem Branch ausgeht, einen symbolischen Namen besitzt und gewöhnlich wieder zurück in den Ausgangsstrang überführt wird.
- *Merging*: Zusammenführen von zwei Entwicklungszweigen, aber auch das Zusammenführen einer Arbeitskopie mit dem Repository.
- *Diffs*: Zwei Versionen einer Datei, eines Verzeichnisses oder eines ganzen Projekts.
- *Changeset*: Eine Gruppe miteinander in Beziehung stehender Änderungen.
- *Change Log*: Eine chronologische Liste von Änderungen an Artefakten.

In agilen Softwareprojekten arbeiten gleichzeitig mehrere Leute am Quellcode. Es herrscht *Collective-Code Ownership*: Jeder Entwickler arbeitet potenziell an jeder Komponente. Die Änderungen am gemeinsamen Quellcodebestand müssen dann werkzeugunterstützt synchronisiert werden. Im Rahmen jeder Integration, insbesondere auch am Ende einer Iteration, frieren Sie mit Subversion den Entwicklungsstand ein. Bei Bedarf können Sie von der Hauptentwicklung abzweigen und unterschiedliche parallele Entwicklungsströme haben. Subversion unterstützt Sie dabei.

Subversion

Es existieren zahlreiche Versionskontrollsysteme, kommerzielle (wie ClearCase) und kostenlose. Subversion ist frei und verfügbar unter *http://subversion.tigris.org*. Subversion ist faktisch der Nachfolger von *CVS* (Concurrent Versions System). CVS ist zwar immer noch weit verbreitet, die Verbreitung von Subversion nimmt jedoch kontinuierlich zu. Die Nutzung von Subversion erinnert im Wesentlichen an die von CVS, so dass es CVS-

Umsteiger besonders leicht haben. Dennoch gibt es konzeptionelle Unterschiede. Und was noch wichtiger ist: Subversion ist deutlich leistungsfähiger und bietet Lösungen für Probleme an, die von CVS unbeantwortet blieben.

Im Jahr 2000 suchte *CollabNet* Entwickler, die helfen sollten, CVS zu ersetzen. Subversion wurde zwar komplett neu geschrieben, konzeptionell sollte das Bewährte von CVS aber nicht gänzlich über den Haufen geworfen werden. Stattdessen sollten die Nachteile von CVS beseitigt und die Nutzung weitestgehend gleich bleiben, um den De-facto-Standard CVS leichter ersetzen zu können.

Ende August 2001, nach über einem Jahr Entwicklungszeit, trug Subversion sich selbst: Die Software wurde über sich selbst versioniert und nicht mehr über CVS, wie es vorher der Fall gewesen war. Das war ein Meilenstein im Rahmen der Entwicklung von Subversion und der Beweis dafür, dass es für die Arbeit mit großen Projekten alltagstauglich war.

Subversion zeichnet sich insbesondere aus durch

Verzeichnisversionierung
: Wo CVS noch einzelne Dateien versionierte, versioniert Subversion ergänzend Verzeichnisse.

Versionsgeschichte
: Die mangelhafte Versionierung bei Kopien und Umbenennungen von Dateien durch CVS wurde verbessert. Mit Subversion können Dateien und Verzeichnisse beliebig hinzugefügt, gelöscht, kopiert und umbenannt werden.

Atomare Commits
: Verschiedene Änderungen werden entweder ganz oder gar nicht auf das Repository angewendet. Abbrüche bei langen Verarbeitungen, die zu einem korrupten Datenbestand führten, gehören nun der Vergangenheit an.

Versionierung von Metadaten
: Dateien und Verzeichnisse haben eigene, frei wählbare Eigenschaften, die ebenfalls versionierbar sind.

Netzwerkkommunikation
: Der Repository-Zugriff ist abstrahiert. Es sind zahlreiche Verbindungsmöglichkeiten möglich. Subversion kommt mit einem eigenen, leichtgewichtigen Serverprozess daher. Obwohl dessen Skalierbarkeit und Autorisierungs- und Authentifizierungsmöglichkeiten häufig ausreichen, lässt eine einfache Nutzung über Apache keine noch so hohen Anforderungen unerfüllt.

Konsistente Datenhaltung
: Unterschiede im Datenbestand werden von Subversion über einen Binäralgorithmus errechnet. Dies funktioniert sowohl bei herkömmlichen Text als auch bei Binärdateien. Versionierung von Binärdateien wie Word-Dokumenten oder Bildern wird dadurch einfacher.

Effizientes Branching und Tagging
: Subversion sieht Branches und Tags als reine Kopien an. Branches und Tags werden also durch Konventionen zu dem, was sie darstellen *sollen*. Durch die Abkehr vom schwergewichtigen, expliziten Umgang mit ihnen, den CVS pflegte, verlaufen diese Operationen jetzt in kurzer Zeit.

Architektur und Design
: Ein verbreitetes, lauffähiges Softwareprogramm (CVS) wurde komplett neu geschrieben und von seinen historisch gewachsenen Eigenheiten befreit. Subversion besteht aus C-Bibliotheken, die laut CollabNet sehr einfach zu nutzen und leicht wartbar sind.

Installation und Konfiguration

Wir besorgen uns *subversion-1.4.2.tar.gz* sowie *subversion-deps-1.4.2.tar.gz*. Dieses sogenannte Dependency-Package enthält die Apache Portable Runtime (APR), Neon für die optionale WebDAV-Unterstützung und Zlib. Dieses Package benötigen Sie, wenn Sie von Ihrem Rechner auf ein entferntes Repository zugreifen wollen. In unserem Fall sind Client und Server auf einer Maschine.

Die beiden Dateien laden Sie von der Downloadseite herunter und packen Sie in ein Verzeichnis Ihrer Wahl mit

```
tar xvzf subversion-1.4.2.tar.gz
tar xvzf subversion-deps-1.4.2.tar.gz
./configure
make
make install
```

Durch standardmäßige Konfiguration nutzen Sie FSFS, die Subversion File System-Implementierung, als Medium für die Datenhaltung: Das Repository wird im Verzeichnissystem des Servers gehalten. Optional ist es auch möglich, eine Berkeley DB (BDB) einzusetzen. Einmal installiert, können Sie den Subversion-eigenen Server starten.

In */etc/profile* fügen Sie den Pfad zu den ausführbaren SVN-Werkzeugen Ihrem Pfad hinzu. Das ist das Subversion-Unterverzeichnis Ihrer Installation unterhalb des Wurzelelements, unter das sie die Distribution entpackt haben.

Die Personen, die auf das Repository zugreifen und über verschiedene Rechner verteilt sind, können entweder den hier vorgestellten Client nutzen oder andere Anwendungen heranziehen, wie beispielsweise *Tortoise* oder *IDEA*.

Im Subversion-Verzeichnis stehen folgende Programme zur Verfügung:

svn
: Kommandozeilen-Client-Programm, um mit dem Server zu kommunizieren.

svnversion
: Zeigt Statusinformationen der Arbeitskopien an.

svnlook
: Inspektion eines Repository.

svndumpfilter
: Filtert Repository-Dump-Ströme.

mod_dav_svn
: Plug-in-Modul für Apache-HTTP-Server. Strategie, um ein Repository für andere in einem Netzwerk verfügbar zu machen.

svnadmin
: Administrationswerkzeug, um Repositories zu erstellen, zu verwalten oder zu reparieren.

svnserve
: Eigener Server, der als Dämon oder via SSH laufen kann. Strategie, um ein Repository für andere in einem Netzwerk verfügbar zu machen.

Erstellung und Konfiguration eines Repository

Ein Repository wird auf dem Rechner eingerichtet, auf dem der Subversion-Server läuft. Selbstverständlich kann das auch derselbe PC sein. Wenn Sie mehrere Repositories für verschiedene Projekte anlegen müssen, legen Sie sie alle unter demselben Wurzelverzeichnis an. Dieses Wurzelverzeichnis ist hier *repositories*, das Sie vor der Erstellung des ersten Repository manuell anlegen müssen. Wir legen dann ein Repository namens »agile« an:

```
svnadmin create \home\michael\subversion\repositories\agile
```

Nachdem wir das Repository angelegt haben, enthält das Verzeichnis die folgende Struktur:

```
conf (beinhaltet Repository-Konfigurationsdateien)
dav  (Verzeichnis für Apache und mod_dav_svn)
db (beinhaltet die versionierten Daten, hier also unsere FSFS-Umgebung mit Revision-
Dateien)
format (eine einzelne Zahl, die die Versionsnummer des Repository-Layouts darstellt)
hooks (hook-Skripten: Skripten, die über bestimmte Prozesse informieren)
locks (Locking-Informationen, verwaltet Zugriffe auf Repository)
```

Wir werden nun einen manuellen Eingriff in die Repository-Installation vornehmen, um die Zugriffsberechtigungen zu konfigurieren. Wir möchten bestimmen, welcher Benutzer welche Rolle spielt (mehrere Benutzer können eine Rolle haben, es können beispielsweise mehrere Personen der Kategorie *Entwickler* zugeordnet sein), und welche Zugriffsmöglichkeiten auf Subversion er dadurch besitzt (Authentifizierung und Authentisierung).

Authentifizierung und Authentisierung

Authentifizierung bezeichnet den Vorgang der Überprüfung der Identität eines Beteiligten (Person oder Softwaresystem), beispielsweise durch ein Kennwort. *Authentisierung* bezeichnet den Vorgang des Nachweises der eigenen Identität, also beispielsweise durch Eingabe seines Benutzernamens. Ist der Nutzer schließlich authentisiert, so ist er *autorisiert*, eine Funktion oder ein Programm zu nutzen.

→

Im Englischen wird der Begriff *Authentification* für beide Vorgänge verwendet. Die ursprünglich in Nuancen verschiedenen Begriffe »Authentifizierung« (eigentlich: Bezeugen der Echtheit) und »Authentisierung« (eigentlich: Beglaubigung, Rechtsgültigmachung) werden heute häufig bedeutungsgleich nebeneinander gebraucht. Gewöhnlich melden Sie sich bei einem Softwaresystem an und bestätigen, dass Sie der sind, als der Sie sich ausgeben, indem Sie eine persönliche Kennung und ein nur Ihnen bekanntes Kennwort eingeben.

Dazu editieren wir die Datei *svnserve.conf* innerhalb des Verzeichnisses *conf*. Im Anfangszustand enthält die Datei Kommentare und mögliche Einträge, die alle auskommentiert sind. Wir kommentieren die folgenden Einträge ein:

```
[general]
anon-access=none
auth-access=write
password-db = passwd
authz-db=authz
realm=agile realm
```

authz ist nun die Datei, die die Zugriffsrechte je Anwender auf das Repository steuert. *passwd* ist die Datei, die die möglichen Nutzer des Repository und seine Passwörter festlegt. Bedenken Sie bitte, dass der Authentifizierungsmechanismus in diesem Modus recht einfach gehalten ist. Wenn Ihnen das nicht reicht, müssen sie den standardmäßigen SVN-Server durch einen Apache ersetzen.

Nun öffnen wir die mitgenerierte, standardmäßige *authz*-Datei und editieren sie wie folgt:

```
[groups]
admins=root
developers=michael
managers=peter
[/]
@admins=rw
@developers=rw
@managers=r
```

Was haben wir getan? Wir haben zunächst mittels *[groups]* drei Rollen eingeführt: die »admins«-Rolle, die »developers«-Rolle sowie die »managers«-Rolle. Den Rollen haben wir jeweils konkrete Benutzer zugewiesen (eine Rolle ist eine Gruppierung von Nutzern mit gleichen Rechten).

Unterhalb des Rollenblocks bestimmen wir die Zugriffsrechte auf die einzelne Bereiche des Repository. Hier können nun die Zugriffsrechte für einzelne Verzeichnisse einzeln gesetzt werden. Die Eingabe eines Slashs definiert das ganze Repository: Wir konfigurieren also den Zugriff auf das gesamte Repository unterhalb des Eintrags *[/]*. Wir definieren für unser ganzes Repository, dass die Gruppe »admins« (*@admins*) sowohl Schreib- als auch Lesezugriffsrechte besitzt (durch Angabe eines *r* und eines *w*), ebenso die Mitglieder

der »developers«-Gruppe. »managers« sollten lieber Managementaufgaben übernehmen. Wir verhindern, dass Mitglieder dieser Rolle schreibend auf das Repository zugreifen können, indem wir nur ein *r* zuweisen (Lesezugriff).

Nun müssen wir noch die einzelnen Nutzer mit ihrem jeweiligen Kennwort in der Datei *passwd* hinterlegen:

```
[users]
root=root
michael=michael
peter=peter
```

Wie Sie sehen, wählen wir auch hier verständliche Einträge, wie Sie sie in dieser Form (insbesondere das einfache Kennwort) produktiv niemals verwenden werden.

Nachdem wir nun das Repository angelegt und rudimentäre Authentifizierungs- und Authentisierungsmaßnahmen ergriffen haben, können wir den Server als Betriebssystemprozess starten. Der Server ist ein Dämon-Prozess, das heißt er läuft im Hintergrund.

```
svnserve -d -r /home/michael/subversion/repositories
```

Der Server lauscht nun auf *localhost* unter dem Standard-Port 3690, was geändert werden kann.

Der mit dem Parameter spezifizierte Pfad schränkt die Möglichkeiten des Nutzers ein, einen beliebigen Pfad auf dem Zielrechner zu adressieren. Mit dieser Einstellung können nur unterhalb des Verzeichnisses *repositories* liegende Projekte angesprochen werden.

Verbindungsmöglichkeiten von Client und Server

Der Client hat unterschiedliche Möglichkeiten, mit dem Server zu kommunizieren:

- *file:///...* direkter Zugriff auf das Repository lokal
- *http://...* Zugriff via WebDAV-Protokoll zu einem Apache
- *https://...* Zugriff via WebDAV-Protokoll zu einem Apache, verschlüsselt
- *svn://...* Subversion-Protokoll
- *svn+ssh://...* Subversion-Protokoll, zusätzlich durch einen SSH-Tunnel

Nun möchten wir unser Projekt im Repository anlegen. Dazu nehmen wir einen bestehenden Softwarestand und führen ihn der Versionskontrolle zu, wir *importieren* diesen Stand in das Repository. Anschließend können die Entwickler diesen versionierten Stand aus dem Projektarchiv herausholen und in ihrer Sandbox lokal weiterentwickeln.

Um dies und weitere Arbeitsschritte zu verdeutlichen, gehen wir im Folgenden von einem Projekt namens »myproject« aus. Dieses Projekt legen wir in ein Arbeitsverzeichnis *ini* ab. Innerhalb von *myproject* erzeugen wir drei Unterverzeichnisse: *trunk*, *branches* und *tags*. Innerhalb von *trunk* erstellen wir nun eine erste Datei namens *README* (oder

kopieren entsprechend das ganze bestehende Projekt hinein, wenn wir eins haben). Das sieht dann so aus:

```
/subversion/ini/myproject/branches/
/subversion/ini/myproject/tags/
/subversion/ini/myproject/trunk/
/subversion/ini/myproject/trunk/README
```

Nun importieren wir diesen Stand (also die Verzeichnisse und die Datei) in das Repository. Wir befinden uns im Verzeichnis *ini* und geben ein:

```
michael@BlueGene:~/subversion/ini$ svn import . file:///home/michael/subversion/repositor
ies/agile --message "initial message"
Hinzufügen     myproject
Hinzufügen     myproject/trunk
Hinzufügen     myproject/trunk/README
Hinzufügen     myproject/branches
Hinzufügen     myproject/tags

Revision 1 übertragen.
```

Sie haben die Möglichkeit, den Subversion-Client zu lokalisieren, so dass er Ausgaben in der Sprache Ihrer Wahl durchführt. Dazu exportieren Sie die Umgebungsvariable LC_MESSAGES, für »Deutsch« geht das so: *export LC_MESSAGES=de_DE*.

Mit dem Subversion-Programm *svn* können Sie Befehle absetzen. Ein `import` importiert das Verzeichnis, das Sie im Befehl übergeben, und alle rekursiv darunter liegenden Verzeichnisse und Dateien. Bei Befehlen, die die Artefakte im Repository ändern oder fortschreiben, müssen Sie eine Änderungsmeldung angeben. Diese Meldung dokumentiert Ihre Änderung und ist für alle sichtbar. So ist es möglich, den Hintergrund der Änderung nachzuvollziehen, ohne in die Änderung selbst hineinschauen zu müssen. Darüber hinaus können hier Zusatzinformationen angegeben werden, die ein späteres Verständnis dieser Änderung erleichtern bzw. diese Änderung erläutern.

Übergeben Sie Änderungsmeldungen, die tatsächlichen Mehrwert bringen. Verzichten Sie also auf Texte wie »Änderung commitet« und beschreiben Sie, *was* Sie *warum* gemacht haben. Das müssen keine langen Romane sein. Es hat sich als hilfreich erwiesen, als Bestandteil des Texts die Nummer des Tickets im Ticketing-System zu hinterlegen. Wenn Sie beispielsweise Jira oder Trac nutzen, dann übergeben Sie Subversion die Ticketnummer, die diese Änderung motiviert hat.

Die Eingabe der Änderung geschieht durch Nutzung des Parameters `message`. Lassen Sie beim Befehl diesen Parameter weg, öffnet sich ein Texteditor, in dem Sie diese Pflichtangabe ebenfalls machen können.

Wunderbar! Wir sehen anhand der Meldungen, dass die einzelnen Elemente erfolgreich übertragen wurden. Unser Repository ist gefüllt. Wir überprüfen das mit

```
michael@BlueGene:~/subversion/ini$ svn list --
verbose file:///home/michael/subversion/repositories/agile/
      1 michael              Dez 23 19:46 myproject/
```

oder von einem beliebigen Rechner über das *svn*-Protokoll mit

```
michael@BlueGene:~/subversion/repositories$ svn list svn://localhost/agile --
username michael --password michael
myproject/
```

In dieser Form können beliebige Hierarchiestufen und Verzeichnisinhalte dargestellt, in das Repository übertragen und der Versionsverwaltung hinzugefügt werden.

Auch wenn Ihnen `svnserve` reicht, werden Sie sicherlich noch weitere Maßnahmen ergreifen wollen, um Ihr System abzusichern. Dazu gehört, so wenigen Nutzern wie möglich physischen Zugriff auf das Repository zu erteilen. Hier würden Sie einen eigenen Linux-User anlegen, der `svnserve` startet und das Repository besitzt. Erlauben Sie keinem anderen Nutzer den Zugriff auf das Repository via `file:///` und führen Sie Ihre administrativen Arbeiten (beispielsweise `svnadmin`) mit dem User durch, dem das Repository gehört.

Nutzung

Ist Subversion bzw. das Repository einmal aufgesetzt, sind die Aktivitäten der einzelnen Nutzer vergleichbar und wiederkehrend. Zu den mit Subversion gewöhnlich durchgeführten Arbeiten gehören hauptsächlich die folgenden Aktivitäten, die in der gegebenen Reihenfolge stattfinden.

Checkout

Sind Artefakte dem Versionierungssystem zugeführt worden oder beginnen Entwickler neu mit der Arbeit an einem Projekt, werden sie eine individuelle Arbeitskopie des Projekts (oder eines Teils davon) vom zentralen Repository zu sich auf den Arbeitsplatzrechner holen: Sie *checken* es *aus*. Der Softwarestand auf dem eigenen Rechner kann beliebig geändert oder auch gelöscht werden. Solange Sie über den Subversion-Client die Sourcen nicht wieder mit dem Repository synchronisieren, sind Ihre persönlichen Änderungen von anderen Änderungen nicht betroffen und andere Entwickler bekommen von Ihren Änderungen nichts mit. Ein Checkout wird mit dem Schlüsselbegriff `co` durchgeführt (Kurzform von »checkout«). Dabei befinden Sie sich in einem Arbeitsverzeichnis, in dem die ausgecheckten Artefakte platziert werden sollen.

```
michael@BlueGene:~/subversion/workspace$ svn co svn://localhost/agile/myproject/trunk --
username michael --password michael
A    trunk/README
Ausgecheckt, Revision 1.
```

Mit `--username` und `--password` übergeben Sie Ihre *Credentials*. Das ist notwendig, da wir den Zugriff auf das Repository eingeschränkt haben. Die Antworten von Subversion zei-

gen, dass im vorliegenden Fall Ihr Trunk erfolgreich ausgecheckt wurde. Lassen Sie immer nur soviel auschecken, wie benötigt wird. In unserem Fall also den Trunk, denn nur hier soll gearbeitet werden. Wird an einem Branch gearbeitet, so lassen Sie diesen Branch auschecken. Der Inhalt der *README*-Datei ist wie folgt:

```
README
This is the initial version of our project to be imported to subversion.
```

Änderungen machen

Selbstverständlich werden Sie auch Änderungen an den Artefakten durchführen. Änderungen können sich durch Editieren von Dateien ergeben, aber auch durch ihre Löschung, Hinzufügen neuer Artefakte, Kopieren und Verschieben.

Stellen wir uns nun vor, ein Kollege hat ebenfalls etwas ausgecheckt und arbeitet am Projekt weiter. Um diesen Schritt zu simulieren, haben wir eine zweite Sandbox durch Anlage eines weiteren Verzeichnisses namens *workspace2* vorbereitet. Hier haben wir wie oben den Trunk ausgecheckt, um an der *README*-Datei eine marginale Änderung vorzunehmen und eine neue Datei hinzuzufügen.

Das sieht bei uns so aus:

```
drwxr-xr-x 3 michael michael 4096 2006-12-24 14:55 .
drwxr-xr-x 3 michael michael 4096 2006-12-24 14:37 ..
-rw-r--r-- 1 michael michael    8 2006-12-24 14:55 index.html
-rw-r--r-- 1 michael michael   76 2006-12-24 14:39 README
drwxr-xr-x 6 michael michael 4096 2006-12-24 14:37 .svn
```

Dem Satz in der *README*-Datei stellen wir ein »1.« voran.

```
README
1. This is the initial version of our project to be imported to subversion.
```

Die neue Datei *index.html* muss hinzugefügt und Subversion bekanntgemacht werden. Das geschieht mit dem Schlüsselbegriff `add`. Subversion dokumentiert den Erfolg.

```
michael@BlueGene:~/subversion/workspace2/trunk$ svn add index.html
A         index.html
```

Zu diesem Zeitpunkt ist die Änderung (die neue Datei und die Änderung an der *README*) aber noch nicht im Repository. Dritte bekommen davon noch nichts mit.

Änderungen untersuchen

Sie werden regelmäßig testen müssen, wie der Stand Ihrer Arbeitskopie im Vergleich zum Repository ist. Was hat sich mittlerweile geändert? Darunter fallen die Subversion-Befehle `status`, `diff` und `revert`.

```
michael@BlueGene:~/subversion/workspace2/trunk$ svn diff
Index: index.html
===================================================================
--- index.html  (Revision 0)
+++ index.html  (Revision 0)
```

```
@@ -0,0 +1 @@
+<html/>
Index: README
===================================================================
--- README      (Revision 1)
+++ README      (Arbeitskopie)
@@ -1 +1 @@
-This is the initial version of our project to be imported to subversion.
+1. This is the initial version of our project to be imported to subversion.


michael@BlueGene:~/subversion/workspace2/trunk$ svn status
A      index.html
M      README
```

Das `diff` zeigt uns die Änderungen, die wir gemacht haben, also die Punkte, in denen sich unsere Arbeitskopie vom Repository unterscheidet. Änderungen in Dateien werden gegenübergestellt, neue Dateien aufgeführt. Auch eine Statusabfrage mit `status` liefert Änderungen in einer kompakten Art. Ein Buchstabe vor der Datei liefert uns Erkenntnisse über die Art der Änderung bzw. einen Hinweis, was noch zu tun ist. `A` bedeutet, die Datei wurde von uns hinzugefügt. `M` bedeutet, die Datei wurde modifiziert. `revert` würde die Änderungen in der Arbeitskopie wieder rückgängig machen und diese auf den gleichen Stand wie im Repository zurücksetzen.

Änderungen einchecken

Während Sie an Ihrem Sourcestand weiterarbeiten, müssen Sie regelmäßig die Aktualität Ihres Stands überprüfen, Ihre Arbeitskopie aktualisieren und Änderungen anderer Entwickler in Ihren Stand aufnehmen: Da auch andere Entwickler täglich Änderungen vornehmen, müssen Sie diesen Vorgang häufig durchführen. Agile Projekte leben von häufigen Synchronisierungspunkten. Führen Sie *mindestens* einmal täglich ein Update durch.

Um dies zu simulieren, haben wir zeitgleich im ersten Workspace eine Änderung an der *README* Datei gemacht. Diese ist unabhängig von der Änderung im zweiten Workspace. Wir ersetzen den Punkt am Ende des Satzes durch ein Ausrufezeichen.

```
README
This is the initial version of our project to be imported to subversion!
```

Wie Sie sehen, ist dies fast der Anfangszustand der Datei, also ohne die Nummerierung vorweg (die Änderung im zweiten Workspace), nur dass ein Ausrufezeichen am Ende des Satzes den Punkt ersetzt. Wir möchten diese Änderung mit dem Befehl `commit` einchecken. Zuvor geben wir uns detaillierte Informationen über unseren Workspace aus. Das Kommando `info` liefert uns Informationen wie die URL des Repository und den letzte Autor, der Änderungen gemacht hat. Interessant für uns ist hier die Revisionsnummer. Wie wir später noch genauer sehen werden, ist der Stand der Artefakte in unserem Workspace mit einer Revision verbunden. Nun ist die Revision »1«.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn info
Pfad: .
URL: svn://localhost/agile/myproject/trunk
Basis des Projektarchivs: svn://localhost/agile
UUID des Projektarchivs: 827090f0-4825-0410-9e4e-dcc7fa201106
Revision: 1
Knotentyp: Verzeichnis
Plan: normal
Letzter Autor: michael
Letzte geänderte Rev: 1
Letztes Änderungsdatum: 2006-12-23 19:46:55 +0100 (Sa, 23 Dez 2006)
```

Mit dem Befehl `update` synchronisieren wir das Repository mit unserem Workspace. Die Durchführung liefert uns die Revision, in der sich die Artefakte nun befinden. Es liegen keine Änderungen vor, die in die Arbeitskopie übertragen werden müssten, da wir im anderen Workspace die Änderungen noch nicht bekanntgemacht haben.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn update
Revision 1.
```

Nun spielen wir die lokalen Änderungen mit dem Kommando `commit` dem Repository zu. Wir sehen, dass wir als Erfolgsmeldung eine neue Revisionsnummer erhalten.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn commit -m "Interpunktion"
Sende          README
Übertrage Daten .
Revision 2 übertragen.
```

Wir können nun erneut ein `info` durchführen, um uns von dem neuen Stand im Repository zu überzeugen.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn info
Pfad: .
URL: svn://localhost/agile/myproject/trunk
Basis des Projektarchivs: svn://localhost/agile
UUID des Projektarchivs: 827090f0-4825-0410-9e4e-dcc7fa201106
Revision: 2
Knotentyp: Verzeichnis
Plan: normal
Letzter Autor: michael
Letzte geänderte Rev: 2
Letztes Änderungsdatum: 2006-12-24 15:20:14 +0100 (So, 24 Dez 2006)
```

Änderungen einchecken und Konflikte auflösen

Nun wollen wir unsere Änderungen in *workspace2* einchecken. Dafür machen wir zunächst auch hier wie gewohnt ein `update`. Wir sehen als Meldung ein `C` vor der *README*-Datei. Sie wurde also geändert (auf Revision 2), wobei ein Konflikt auftrat. Ein Konflikt kann immer entstehen, wenn zwei Personen an derselben Datei überlappende Änderungen vornehmen.

In der Praxis werden nur sehr selten Änderungen eine manuelle Konfliktlösung notwendig machen, da Entwickler gewöhnlich an unterschiedlichen Klassen oder zumindest an unterschiedlichen Segmenten einer Klasse arbeiten. Häufig kann dann Subversion selbst

die Dateien in den verschiedenen Versionen zu einer neuen zusammenmischen. Wir haben ein Szenario gewählt, in dem eine einzelne Zeile einer Datei unterschiedlich geändert wurde. Das ist ein Spezialfall, der manuelle Konfliktlösung notwendig macht: Subversion kann bei konkurrierenden Änderungen nicht entscheiden, welche Änderung die richtige bzw. aktuelle ist.

Eine unserer Änderungen steht im Widerspruch zu einer aktuellen Änderung, die wir vom Server erhalten haben. In diesem Zusammenhang sehen wir in unserem Verzeichnis neue Dateien. Es sind temporäre Hilfsdateien, um die Änderungen zu verfolgen und ggf. zu verwerfen, den Konflikt also manuell aufzulösen.

```
michael@BlueGene:~/subversion/workspace2/trunk$ svn update
C    README
Aktualisiert zu Revision 2.


drwxr-xr-x 3 michael michael 4096 2006-12-24 15:23 .
drwxr-xr-x 3 michael michael 4096 2006-12-24 14:37 ..
-rw-r--r-- 1 michael michael    8 2006-12-24 14:55 index.html
-rw-r--r-- 1 michael michael  183 2006-12-24 15:22 README
-rw-r--r-- 1 michael michael   76 2006-12-24 15:22 README.mine
-rw-r--r-- 1 michael michael   73 2006-12-24 15:22 README.r1
-rw-r--r-- 1 michael michael   73 2006-12-24 15:22 README.r2
drwxr-xr-x 6 michael michael 4096 2006-12-24 15:22 .svn
```

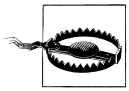
Die drei neuen Dateien haben folgende Funktionen:

README.mine
: Meine Datei, so wie sie in meiner Arbeitskopie vorlag, bevor sie aktualisiert wurde, also ohne Konfliktmarker.

README.r1
: Dies ist die Base-Revision, bevor meine Arbeitskopie aktualisiert wurde. Dies ist also die Datei, so wie ich sie augecheckt hatte, bevor ich meine neuesten Bearbeitungsschritte durchführte. »r1« ist die zugrundeliegende Revision.

README.r2
: Dies ist die Datei, so wie mein Subversion-Client sie gerade vom Server empfangen hat. Diese Datei ist der HEAD der Entwicklung. »r2« ist die zugrundeliegende Revision.

`diff` zeigt uns den aktuellen Stand, wobei wir auch hier den Konflikt innerhalb der *README*-Datei sehen. Wir sehen ferner Konfliktmarker, also besondere Textpassagen, die den Konflikt aus der überschneidenden Änderung genau eingrenzen und weitere Informationen geben.



Subversion ist nur ein Werkzeug, zwar ein mächtiges, aber dennoch kann es persönliche Kommunikation nicht ersetzen. Das kommt spätestens bei Konflikten zum Tragen. Wenn Ihre Datei mit Änderungen von Kollegen in Konflikt steht, müssen Sie das persönliche Gespräch suchen.

```
michael@BlueGene:~/subversion/workspace2/trunk$ svn diff
Index: index.html
===================================================================
--- index.html  (Revision 0)
+++ index.html  (Revision 0)
@@ -0,0 +1 @@
+<html/>
Index: README
===================================================================
--- README      (Revision 2)
+++ README      (Arbeitskopie)
@@ -1 +1,5 @@
+<<<<<<< .mine
+1. This is the initial version of our project to be imported to subversion.
+=======
 This is the initial version of our project to be imported to subversion!
+>>>>>>> .r2
```

Nun haben Sie drei Möglichkeiten, mit dem Konflikt umzugehen:

- Den Text manuell zusammenführen und den Konfliktmarker löschen (mit anschließendem `svn resolved`, siehe unten).
- Eine der temporären Dateien kopieren und als neue Arbeitsversion festsetzen (mit anschließendem `svn resolved`, siehe unten). So könnten Sie also beispielsweise Ihre Arbeitsversion nehmen und als Lösung melden.
- `svn revert` ***<dateiname>*** ausführen, um alle lokalen Änderungen zu verwerfen.

Wir ändern die Datei manuell zu

```
README
1. This is the initial version of our project to be imported to subversion!
```

Nun können wir mit dem Befehl `resolved` Subversion über die erfolgreiche Konfliktlösung informieren. Subversion wird dann die temporären Dateien aus dem Verzeichnis löschen. Solange die temporären Dateien vorliegen, ist ein `commit` nicht möglich.

```
michael@BlueGene:~/subversion/workspace2/trunk$ svn resolved README
Konflikt von »README« aufgelöst
```

Nun sind die temporären Dateien gelöscht und ein `diff` würde einen sauberen Stand ohne Konfliktmarker zeigen. Nun können wir den neuen Stand einchecken.

```
michael@BlueGene:~/subversion/workspace2/trunk$ svn commit -m "second iteration"
Sende          README
Hinzufügen     index.html
Übertrage Daten ..
Revision 3 übertragen.
```

Wir sehen, dass die Datei *README* (bzw. ihre geänderte Version) übertragen und die neue Datei *index.html* endgültig der Versionierung hinzugefügt wurde. Durch diese Änderung haben wir Revision 3 erreicht. Ein *diff* würde keine Ausgabe liefern, da unser Arbeitsbereich mit dem Repository synchron ist. Zurück im anderen Workspace müssen wir natürlich auch wieder ein `update` durchführen.

Um den chronologischen Ablauf aller Änderung zu verfolgen, bietet sich das Kommando `log` an. Genauso wie bei anderen Subversion-Befehlen lässt sich auch mittels `--revision` das `log` auf eine Revision beschränken. Durch `--verbose` erhalten wir ausführlichere Daten zur Revision.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn log
------------------------------------------------------------------------
r3 | michael | 2006-12-24 15:49:26 +0100 (So, 24 Dez 2006) | 1 line

second iteration
------------------------------------------------------------------------
r2 | michael | 2006-12-24 15:20:14 +0100 (So, 24 Dez 2006) | 1 line

Interpunktion
------------------------------------------------------------------------
r1 | michael | 2006-12-23 19:46:55 +0100 (Sa, 23 Dez 2006) | 1 line

initial message
------------------------------------------------------------------------
michael@BlueGene:~/subversion/workspace/trunk$ svn log --verbose --revision 3
------------------------------------------------------------------------
r3 | michael | 2006-12-24 15:49:26 +0100 (So, 24 Dez 2006) | 1 line
Geänderte Pfade:
   M /myproject/trunk/README
   A /myproject/trunk/index.html

second iteration
------------------------------------------------------------------------
```

Revisionen

Wenn Sie anfangs ein Repository anlegen, befindet sich das gesamte Repository in der Revision 0. Ein `import` und jedes folgende `commit` erhöhen die Revisionsnummer um 1. Wenn Sie ein Commit abgeschlossen haben, liefert Subversion die neue Revisionsnummer als Ausgabe zurück. Das mag auf den ersten Blick etwas verwirrend sein, insbesondere für CVS-Nutzer: Subversion unterscheidet nicht die Revisionen einzelner Dateien, für Subversion ist vielmehr das ganze Repository in der Revision X. Anders gesagt: Die Revision X einer Datei stellt die Datei dar, wie sie in der Revision X des Repository vorliegt. Dazu muss sich diese eine Datei in aufeinander folgenden Revisionen aber nicht zwangsläufig geändert haben!

Die Dateien in einer Arbeitskopie setzen sich aus unterschiedlichen Revisionen des Repository zusammen. Nehmen Sie beispielsweise drei Dateien, die Sie in der letzten Version 1 auschecken.

```
README:1
index.html:1
HelloWorld.java:1
```

Nun entspricht die Arbeitsversion exakt der Revision 1 des Repository. Wenn wir nun aber eine Änderung an der Datei *README* vornehmen und diese Änderung committen,

so wird unser Commit die Revision 2 des Repository erzeugen, und unsere Arbeitsversion sieht dann folgendermaßen aus (vorausgesetzt, es gab zwischenzeitlich keine anderen Commits):

```
README:2
index.html:1
HelloWorld.java:1
```

Nun modifiziert ein Kollege *index.html* und erstellt somit Revision 3 des Repository. Wenn wir nun ein `svn update` durchführen, wird unsere Arbeitsversion aktualisiert und sieht so aus:

```
README:3
index.html:3
HelloWorld.java:3
```

Versionierung (Tags)

Ein gängiges Versionskontrollkonzept ist das *Taggen*. Ein Tag ist eine eingefrorene Momentaufnahme der Artefakte, der ein symbolischer Namen zugeordnet wird. War es bei CVS noch die einzig zuverlässige Art, eine Momentaufnahme des ganzen Repository zu archivieren und jederzeit wieder darauf zuzugreifen, ist es bei Subversion etwas anders. Durch die Tatsache, dass das ganze Repository eine einheitliche Revisionsnummer pflegt, gibt es entsprechend auch zu jeder Nummer einen abrufbaren Stand. Während einer noch so langen Projektdauer kann jederzeit der Stand der Software nach einem bestimmten Commit abgerufen werden. In der Praxis ist es allerdings so, dass Sie einem Tag einen sprechenden Namen geben oder auch nur einen Teil des Repository mit einem Tag versehen, also *taggen* wollen.

Doch zu welchen Zeiten erstellen agile Projekte Tags? Agile Projekte ziehen dann ein Tag ein, wenn für sie ein Meilenstein erreicht ist, den es festzuhalten oder zu kommunizieren gilt. Am Ende einer Iteration, wenn die Arbeit der Iteration dem Kunden präsentiert wird, werden Sie vorher ein Tag erstellen wollen. So können Sie jederzeit nachhalten, dass genau diese Version dem Kunden für eine Begutachtung vorgelegt wurde. Oder gehen Sie einen Schritt weiter: Nehmen wir an, Sie führen nach jedem Commit einen Integrations-Build durch (später dazu mehr). Es ist denkbar, für jeden dieser Builds ein Tag einzuziehen. Das entspräche bei der Taktfrequenz allerdings der schon diskutierten Subversion-eigenen Versionierung über Revisionsnummern. Allzu viele Tags sind für die Übersichtlichkeit aber nicht förderlich. Hier müssen Sie ganz individuell eine adäquate Frequenz finden. Zumindest, wenn eine Version Ihres Produkts fachlich motiviert einen bestimmten Stand erreicht hat, also beispielsweise der beschriebene Abschluss einer Iteration gekommen ist, ist ein Tag mehr als gerechtfertigt.

Da wir in unserem agilen Projekt dauerhaft im Trunk die Software weiterentwickeln möchten, müssen wir auch nur diesen mit einem Tag versehen. Bei Subversion erstellen Sie das Tag, indem Sie eine Kopie des Trunk erstellen. Sie haben beim Aufbau der Projekt- bzw. Repository-Struktur bereits ein Verzeichnis *tags* neben dem Trunk-Verzeichnis angelegt. Sie können nun einen Stand von dem Trunk in das *tags*-Verzeichnis kopieren.

Wir erstellen ein Tag namens *release-1.0* durch Kopieren. Dabei nutzen wir das Kommando `copy`, das eine Quelle und ein Ziel erwartet, sowie die obligatorische Nachricht.

```
michael@BlueGene:~/subversion$ svn copy svn://localhost/agile/myproject/trunk svn://local
host/agile/myproject/tags/release-1.0 -m "Tagging Release 1.0"

Revision 4 übertragen.
```

Durch das Kopieren wird das Repository in eine neue Revision versetzt. Mittels `list` überprüfen wir das Verzeichnis *tags* und stellen fest, dass die Kopie angelegt wurde.

```
michael@BlueGene:~/subversion$ svn list svn://localhost/agile/myproject/tags
release-1.0/
```

Wenn wir in die Kopie hineinschauen, finden wir die Dateien auf ihrem aktuellen Stand (dem Stand des Trunk) wieder.

```
michael@BlueGene:~/subversion$ svn list svn://localhost/agile/myproject/tags/release-1.0
README
index.html
```

Wir sind bei der Erstellung von Tags aber nicht davon abhängig, ob der Entwicklungsstand jetzt gerade der ist, der kopiert werden soll. Das wäre sogar eine ganz unagile Vorgehensweise, denn wir müssten sicherstellen, dass niemand Änderungen auf dem Trunk macht, bis das Tag erstellt wurde. Bei der Erstellung von Tags können wir vielmehr auch vergangene Revisionen als Basis nehmen. Ein Tag können wir also auch jederzeit nachträglich erstellen.

Das ist möglich, da wir bei der Kopie eine Ausgangsrevisionsnummer angeben können. Im folgenden Beispiel erstellen wir vom Trunk des Repository mit der Revisionsnummer 1 eine Kopie und legen sie als Tag unter dem Namen *release-0.1* im Verzeichnis *tags* ab.

```
michael@BlueGene:~/subversion$ svn copy svn://localhost/agile/myproject/trunk svn:
//localhost/agile/myproject/tags/release-0.1 -m "Tagging Release 0.1" -r 1

Revision 5 übertragen.
```

Wir testen das durch eine Auflistung der Tags:

```
michael@BlueGene:~/subversion$ svn list svn://localhost/agile/myproject/tags
release-0.1/
release-1.0/
```

Tags sind Entwicklungszweige, die technisch gleichfalls ausgecheckt und weiterentwickelt werden können. Ein Tag sollte allerdings per Konvention nicht weiterentwickelt werden, da das nicht der Sinn der Erstellung der Kopie war. Sie möchten ja gerade einen festen Stand der Software vorhalten. Um Änderungen auf Tags zu verhindern, gibt es mehrere Möglichkeiten. Sie können beispielsweise Absprachen organisieren, an die sich jeder hält. Diese Absprachen werden im Team gelebt, jeder kennt sie und kann sie im zentralen Wiki nachlesen. Sie können aber auch die Rechte für den Zugriff auf diesen Teil des Repository (also die Tags) technisch einschränken. Dazu würden Sie das *tags*-Verzeichnis in der *svnserve.conf* mit eigenen, eingeschränkten Rechten versehen.

Separate Entwicklungszweige (Branches)

Gewöhnlich wird der Trunk von allen Entwicklern fortgeschrieben. Hier fließen Neuentwicklungen, Änderungen und Bugfixes ein. Während der Entwicklungsarbeit ergeben sich Situationen, in denen dieses Vorgehen nicht ganz ausreichend ist. Stellen Sie sich vor, Sie haben ein Tag erstellt, um eine Iteration kenntlich zu machen. Diese Iteration soll zum Kunden ausgeliefert werden. Nun möchten Sie aber einerseits an dieser Iteration keine beliebigen Änderungen mehr vornehmen, da der Kunde den Softwarestand in genau diesem Zustand abgenommen hat, andererseits wollen Sie natürlich auch nicht Ihre komplette Entwicklungsmannschaft in den Urlaub schicken. Die Entwicklung muss weitergehen. Mit dem Kunden zusammen haben Sie weitere Features aus dem Product Backlog genommen und setzen sie um. Doch wohin committen Sie die Erweiterungen und Änderungen?

Wenn Sie nun schon an Release 2.0 arbeiten, aber noch einen Bug in Release 1.0 fixen müssen, dann tun Sie das auf dem Release 1.0-Tag. Der Clou bei Subversion ist, dass ein Tag letztendlich das Gleiche ist wie ein Branch. Sowohl Tags als auch Branches sind technisch gesehen Entwicklungszweige. Sobald Sie ein Tag auschecken und daran Änderungen machen, wird es ein Branch. Wenn Sie das vorher schon wissen, oder wenn es sich so ergibt, dokumentieren Sie es durch eine Kopie im von uns zu Beginn angelegten *branches*-Zweig. Wir legen den Branch an:

```
michael@BlueGene:~/subversion$ svn copy svn://localhost/agile/myproject/tags/release-
1.0 svn://localhost/agile/myproject/branches/release-1.0 -m "Branching"

Revision 6 übertragen.
```

Nun checken wir den Branch aus in ein eigens dafür reserviertes Arbeitsverzeichnis auf unserem Arbeitsplatzrechner:

```
michael@BlueGene:~/subversion/workspace$ svn checkout svn://localhost/agile/myproject/bra
nches/release-1.0
A    release-1.0/index.html
A    release-1.0/README
Ausgecheckt, Revision 6.
```

Nun lassen wir uns die ausgecheckten Verzweigungen anzeigen. Dazu gehen wir in den Branch und checken via *info* die Information darüber, wo wir uns gerade befinden. Dazu verbinden wir das *info*-Kommando unter Linux mit einem *grep*, um genau diese URL-Information zu erhalten:

```
michael@BlueGene:~/subversion/workspace$ ls
release-1.0  trunk

michael@BlueGene:~/subversion/workspace$ cd release-1.0/
michael@BlueGene:~/subversion/workspace/release-1.0$ ls
index.html  README
michael@BlueGene:~/subversion/workspace/release-1.0$ svn info | grep URL
URL: svn://localhost/agile/myproject/branches/release-1.0
```

Sie können auch mit dem Kommando *switch* von Subversion den aktuellen Workspace auf einen anderen ändern. Ich habe allerdings die Erfahrung gemacht, dass es einfacher ist, den zweiten Entwicklungszweig separat auszuchecken. Einerseits werden Sie vielleicht den Trunk ebenfalls parallel benötigen, andererseits ist es transparenter, wenn Sie jederzeit wissen, auf welcher Basis Sie arbeiten, und das auch durch die unterschiedlichen Verzeichnisse dokumentiert ist.

Gehen wir nun davon aus, wir möchten im Branch eine Änderung vornehmen. Wir fügen unserer *README*-Datei eine zweite Zeile hinzu.

```
1. This is the initial version of our project to be imported to subversion!
2. This is a branch!
```

`diff` zeigt, dass sich unser Workspace im Vergleich zum Repository geändert hat.

```
michael@BlueGene:~/subversion/workspace/release-1.0$ svn diff
Index: README
===================================================================
--- README      (Revision 6)
+++ README      (Arbeitskopie)
@@ -1 +1,2 @@
 1. This is the initial version of our project to be imported to subversion!
+2. This is a branch!
```

Wir committen diese Änderung:

```
michael@BlueGene:~/subversion/workspace/release-1.0$ svn commit -m "commit to branch"
Sende          README
Übertrage Daten .
Revision 7 übertragen.
```

Nun ist die Änderung im Branch, aber nicht im Trunk. Das können wir testen, indem wir ins Trunk-Verzeichnis gehen und `update` durchführen.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn update
Revision 7.
```

In diesem Beispiel haben wir eine exemplarische Benennung der verschiedenen Versionen vorgenommen, die in vielen Projekten ausreichen wird. Ist das Projekt sehr umfangreich und Sie arbeiten permanent mit vielen Branches, oder wollen Sie die Aktivitäten noch genauer dokumentieren, dann nehmen Sie noch weitere Informationen mit in die Tags und Branches auf. Sie können die Tags frei benennen. Nennen Sie den Branch ruhig tatsächlich »BranchXYZ«.

Der Arbeitsbereich wird mit der neuesten Revision des Repository synchronisiert, allerdings ergeben sich dadurch keine Änderungen für den Trunk.

Die Änderungen sind bisher ausschließlich im Branch. Früher oder später müssen wir die Änderungen im Branch auch wieder in den Trunk zurückführen, da das unsere Hauptentwicklungslinie ist. Wie wir bereits festgestellt haben, ist es wichtig, das Repository

immer synchron zu halten. Das betrifft die täglichen Commits der Entwickler, aber auch die Entwicklungsverzweigungen. Je stärker die Branches vom Trunk abdriften, desto fehleranfälliger und aufwendiger ist das erneute Zusammenführen.

Wir führen nun Trunk und Branch explizit zusammen. Zunächst kopieren wir den Branch zurück in das Verzeichnis *tags* und wählen dazu einen Zielnamen, der unsere Arbeit kenntlich macht, hier *sp1* für »Server Pack 1«.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn copy svn://localhost/agile/myproject/b
ranches/release-1.0 svn://localhost/agile/myproject/tags/release-1.0-sp1 -m "sp1"

Revision 8 übertragen.
```

Nun führen wir dieses neue Tag mit dem Trunk zusammen. Diese Zusammenführung wird im aktuellen Arbeitsbereich aus Ihrem Trunk-Workspace durchgeführt. Sie geschieht über das Kommando `merge`. Auch hier werden ggf. Konflikte entstehen, die Sie wie gewohnt auflösen müssen.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn merge svn://localhost/agile/myproject/
trunk svn://localhost/agile/myproject/tags/release-1.0-sp1
U    README
```

`diff` zeigt: Die Änderungen aus dem Branch wurden in den Arbeitsbereich des Trunk übernommen.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn diff
Index: README
===================================================================
--- README      (Revision 8)
+++ README      (Arbeitskopie)
@@ -1 +1,2 @@
 1. This is the initial version of our project to be imported to subversion!
+2. This is a branch!
```

Schließlich committen Sie die Änderungen in den Trunk.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn commit -
m "merging branch and trunk into trunk"
Sende          README
Übertrage Daten .
Revision 9 übertragen.
```

Sie sollten nun aufräumen und den Branch mit `delete` aus dem *branches*-Verzeichnis löschen.

```
michael@BlueGene:~/subversion/workspace/trunk$ svn delete svn://localhost/agile/myproject
/branches/release-1.0 -m "deleting branch"

Revision 10 übertragen.
```

Sie müssen nicht unbedingt den Branch aus dem *branches*-Verzeichnis löschen. Es ist auch denkbar, dort immer einen Branch liegen zu lassen, der für alle Verzweigungen wiederverwendet wird.

Locking

Die standardmäßige Nutzung von Subversion geschieht nach dem *Copy-Edit-Merge-Modell*. Dabei checkt der Entwickler die Artefakte in seinen lokalen Arbeitsbereich, macht darauf Änderungen und Erweiterungen und führt einen Abgleich mit den zwischenzeitlich aufgelaufenen Änderungen im Repository durch, um letztendlich den neuen Stand in das Repository einzuspeisen. Es existieren allerdings auch Konstellationen, in denen Sie nicht auf diese Weise vorgehen wollen bzw. können, sondern mit *Locking* arbeiten. Dabei werden beim Auschecken aus dem Repository einzelne Dateien gesperrt und können in der Zeit bis zum Commit dieser Dateien von anderen nicht geändert (sondern nur gelesen) werden. Ein Einsatzszenario dafür ist beispielsweise das Editieren von binären Dateien wie Word-Dokumenten. Gleichzeitige Änderungen an binären Dateien können von Subversion nicht zusammengeführt werden. Um zu verhindern, dass gleichzeitig mehrere Personen an einer Binärdatei arbeiten, setzen Sie eine Sperre, ein Lock.

Optimistisches Locking

Optimistisches Locking bezeichnet den Vorgang, einen expliziten Lock auf eine Datei zu setzen, bevor Sie diese editieren können. Dabei sind Dateien standardmäßig nicht gelockt, die gemeinschaftliche Arbeit daran geschieht »optimistisch«. Subversion erstellt für den Lock-Eigentümer (den Lock-Ersteller) automatisch und für den Nutzer transparent ein »Lock Token«. Ein Lock (hier für die Datei *abc.doc*) wird auf der Konsole durchgeführt mit

```
svn lock abc.doc
```

und kann mit

```
svn status
```

abgefragt werden. Die gesperrte Datei ist in der Statusausgabe besonders gekennzeichnet. So signalisiert ein »K«, dass die Datei in der eigenen lokalen Arbeitskopie gesperrt ist. Ein »O« zeigt an, dass ein Lock in einer anderen, entfernten Arbeitskopie gehalten wird. Versuchen Sie, eine bereits gelockte Datei zu sperren, liefert Subversion die Meldung, dass die Datei bereits von Person XY gelockt ist.

Ein Lock wird vom Lock-Eigentümer beim Commit der Datei automatisch oder explizit mit einem

```
svn unlock
```

aufgelöst.

Locks können von anderen Entwicklern »gebrochen« und »gestohlen« werden. Das Locking-Konzept von Subversion ist als agiles Kommunikationsmittel zu verstehen, nicht als schwergewichtiges, starres Modell.

Das Vorgehen ist einfach und zweckmäßig, allerdings auch mit Nachteilen verbunden. Eine Herausforderung bei der Nutzung ist, dass Sie unter Umständen vergessen, dass Sie eine Datei gesperrt halten. Das kann problematisch werden, wenn Sie nicht verfügbar sind, beispielsweise aufgrund eines wohlverdienten Urlaubs. Für diese Fälle bietet Subversion die Möglichkeit, dass Sperren nicht fix und unwiderruflich sind, sondern auch von anderen Personen als nur dem Lock-Halter aufgelöst werden können. Eine weitere Rahmenbedingung ist, dass Sie die Konvention, Locks anzuwenden, als Prozessstandard etablieren müssen. Wenn einzelne Personen keine Sperren setzen, ist das ganze Vorgehen brüchig. Schließlich existiert in Subversion keine »Erinnerungsfunktion«, die den Lock-Halter an bestehende Locks erinnert.

Pessimistisches Locking

Die zweite Möglichkeit, mit Locks zu arbeiten, ist das *pessimistische Locking*. Dabei sind einzelne Artefakte standardmäßig gelockt bzw. nur lesbar. Möchte ein Entwickler eine solche Datei ändern, muss er handeln und einen Lock setzen. Dieser Ansatz unterscheidet sich also konzeptionell vom optimistischen Locking, bei dem das Artefakt standardmäßig editierbar ist.

Subversion bietet die Möglichkeit, im Rahmen von *Property Sets* bestimmte Dateien (oder das ganze Repository) mit Metainformationen zu versehen. Eine Metainformation (in Subversion als »Property« bezeichnet) kann Dateien und Verzeichnissen angehängt werden. Sie lässt sich hinzufügen, ändern und entfernen. Ein Beispiel für eine Metainformation ist das Kennzeichen, dass bestimmte Dateien als »zu locken« markiert werden. So markierte Dateien werden automatisch als »nur lesbar« ausgecheckt, es sei denn, es wird ein Lock erstellt.

Stellen Sie sich vor, Sie haben in einem Verzeichnis viele Word-Dokumente. Um dieses Verzeichnis mit Metainformation zu versehen, checken Sie es aus und setzen die entsprechende Eigenschaft für das Verzeichnis in der Form

```
svn propset -R svn:needs-lock yes verzeichnis
```

Dieser Befehl markiert die Dateien im Verzeichnis *verzeichnis* inklusive aller Unterverzeichnisse rekursiv (wegen des Kennzeichens *-R*) als pessimistisch gelockt. Dabei wird der Schlüsselbegriff `propset` mit dem Attribut `svn:needs-lock` genutzt (mit `propset` können Sie auch andere Metainformationen hinzufügen). Das `yes` ist der Wert der Markierung `svn:needs-lock`. Subversion überprüft lediglich die Existenz des Kennzeichens `svn:needs-lock` und nicht den Wert, so dass in diesem Fall `yes` ein beliebiger, frei gewählter Wert ist.

Im Anschluss müssen Sie *verzeichnis* in das Repository einspielen:

```
svn ci verzeichnis -m »nun müssen alle Dateien gelockt werden, um sie ändern zu können«
```

In agilen Teams können alle Entwickler jederzeit alle im zentralen Repository befindlichen Dateien editieren und fortschreiben. Das hält das Tempo der Entwicklung hoch. Locking sollten Sie spärlich und wohlüberlegt einsetzen. Es ist eine gute Praxis, Dateien, die nicht zusammengeführt werden können (wo *merge* nicht funktioniert), mit der Metainformation »*needs-lock*« zu versehen.

Der Locking-Modus wird auf die im Kontext befindlichen Artefakte angewendet, in unserem Beispiel alle Dateien des Verzeichnisses *verzeichnis*. Kommen neue Dateien hinzu, verfügen diese nicht über die Metainformation. Damit nicht für jede neu zum Repository hinzugefügte Datei die Eigenschaft `needs-lock` manuell gesetzt werden muss, gibt es eine weitere Möglichkeit: Die Konfiguration von Subversion kann geändert werden. Subversion hält auf dem Client der Nutzer eine *config*-Datei (unter Linux unter *.subversion* im Home-Verzeichnis des Anwenders), in der Sie die Eigenschaft zentral setzen können. Folgende dort vorgenommene Einträge markieren beispielsweise alle *.doc*- und *.jpg*-Dateien als »zu locken« bzw. fügen neuen Dateien dieses Typs diese Eigenschaft hinzu:

```
[miscellany]
enable-auto-props=yes
[auto-props]
*.doc=svn:needs-lock=*
*.jpg=svn:needs-lock=*
```

Automatisierte Nutzung mit Ant

Sie wollen nicht bei jeder Softwareerstellung die Prozedur der Erstellung von Tags manuell durchführen müssen. Stattdessen soll diese Aktivität Bestandteil des Build-Prozesses werden, also automatisch im Rahmen Ihrer anderen Aktivitäten (die wir später noch vorstellen werden) durchgeführt werden. Dazu nutzen Sie ein Ant-Skript (bzw. ergänzen ein bestehendes). Dieses Skript erstellt einen neuen Tag aufbauend auf einer bestimmten Revision im Repository.

```
<project name="subversion" basedir="." default="release">
<property name="svn.dir" value="svn://localhost/agile/myproject"/>
<property name="svn.trunk" value="${svn.dir}/trunk"/>
<property name="svn.tags" value="${svn.dir}/tags"/>

<property file="build.properties"/>

<target name="release" description="tagging">
<exec executable="svn" failonerror="true">
<arg value="copy"/>
<arg value="-r"/>
<arg value="${build.number}"/>
<arg value="${svn.trunk}"/>
<arg value="${svn.tags}/build-${build.number}"/>
<arg value="-m"/>
<arg value="tag for ${build.number}"/>
<arg value="--username"/>
```

```
<arg value="michael"/>
<arg value="--password"/>
<arg value="michael"/>
</exec>
<echo>Tag created: build-${build.number}</echo>
</target>
</project>
```

Wir definieren ein Skript, dessen standardmäßiges Ansprungsziel `release` ist. Bevor die Verarbeitung losgeht, setzen wir noch ein paar Properties: Wir extrahieren die Konfigurations- und Umgebungsdaten aus dem Target, um sie an zentraler Stelle jederzeit ändern zu können, ohne die Logik ändern zu müssen. Eine Property hat entweder einen Namen und einen Wert oder einen Namen und eine mit ihm verbundene Datei, in der dann die eigentlichen Schlüssel/Wert-Paare stehen. Einmal definiert, können Properties im Skript mit der Notation ${***Property-Name***} verwendet werden.

Zu Beginn definieren wir, wo das Repository liegt, und wo das Projekt im Repository zu finden ist. Anschließend definieren wir noch *Convenience Properties*, also eindeutige für sich sprechende Namen, die aufbauend auf dem Basiseintrag zu dem Trunk und den Tags verzweigen. Die Einführung dieser Properties ist nicht zwingend notwendig, kann doch bei Zugriffen auf Trunk und Tags *svn.dir* genutzt und jeweils ergänzt werden. Es ist im weiteren Verlauf bequemer, jeweils eigene Properties für die Ziele zu haben.

Die eigentliche Verarbeitung beschränkt sich nun auf das Kopieren des Trunk in einer spezifizierten Revision ins Tag-Verzeichnis. Den zugrundeliegenden Subversion-Befehl und seine Nutzung haben wir bereits kennengelernt. Hier sehen wir, wie er in ein Ant-Skript eingebettet werden kann. Wo der Subversion-Vorgänger CVS dedizierte Ant-Tasks besitzt (an entsprechenden Ant-Extensions für Subversion wird gearbeitet), müssen wir uns einstweilen mit dem Aufruf über Kommandozeile begnügen. Das tut der Verarbeitung keinen Abbruch.

```
<exec executable="svn">
<arg value="copy"/>
<arg value="-r"/>
<arg value="${build.number}"/>
<arg value="${svn.trunk}"/>
<arg value="${svn.tags}/build-${build.number}"/>
<arg value="-m"/>
<arg value="tag for ${build.number}"/>
<arg value="--username"/>
<arg value="michael"/>
<arg value="--password"/>
<arg value="michael"/>
</exec>
```

Wir spezifizieren über den Ant-Task `exec` die ausführbare Datei (das Executable). Diese muss natürlich nicht nur tatsächlich ausführbar sein, sie muss auch im Pfad liegen, damit sie von Ant gefunden und ausgeführt werden kann. Auf unserem System ist dies der Fall, wir haben zuvor *svn* direkt über die Konsole genutzt. Generell ist es immer ein guter Start, die ausführbare Anwendung über die Konsole zu testen.

Anschließend übergeben wir mit `arg`-Ausdrücken Kommandozeilenargumente für den Ausdruck. In zum `arg`-Task gehörende `value`-Attribute schreiben wir die Informationen, so wie wir sie auf der Konsole selbst eingeben würden. Aus diesen Angaben erzeugt Ant intern den Aufruf des Subversion-Clients.

Sie werden das Ant-Skript selbst ebenfalls mit Subversion versionieren wollen. Wenn Sie Ant nun auf Ihrem Arbeitsplatz oder einem Integrationsrechner automatisch ausführen (später dazu mehr), kann es sein, dass genau dieses Ant-Skript geändert wurde. Sie müssen also sicherstellen, dass vor dem Ant-Aufruf das Skript selbst aktualisiert wird. Die Aktualisierung der anderen Artefakte können Sie mit *update* auch aus Ant heraus bewerkstelligen. Wie machen Sie das? Sie erstellen einen Bootstrap-Runner. Dieser ist dem Ant-Skript vorgelagert, wird von Ihnen (oder automatisch) aufgerufen und ruft selbst das Ant-Skript auf. Dazu reicht ein einfaches explizites Skript, das einen Subversion-Kommandozeilenaufruf durchführt und dabei ein Update auf dem Ant-Skript durchführt. Anschließend ruft das Bootloader-Skript dann das eigentliche Ant-Skript auf.

Es fehlt noch die *build.number*, die wir in der Datei *build.properties* definieren und über

```
<property file="build.properties"/>
```

einbeziehen. Sie sieht so aus:

```
build.number=3
```

Wir geben die Build-Nummer im Ant-Skript in einem *echo* zu Informationszwecken aus. Ein Aufruf sieht nun folgendermaßen aus:

```
michael@BlueGene:~/subversion$ ant
Buildfile: build.xml

release:
     [exec]
     [exec] Revision 15 übertragen.
     [echo] Tag created: build-3

BUILD SUCCESSFUL
Total time: 0 seconds
```

Es hat alles geklappt, und sogar in einer gemessenen Zeit von 0 Sekunden. Können Sie auch manuell ein Tag so schnell anlegen?

Im vorliegenden Fall könnten Sie die Build-Nummer, wie jeden anderen Parameter, an das Ant-Skript beim Aufruf übergeben. Der Parameter wird mit seinem Wert einfach auf der Konsole angegeben und muss derselbe sein, wie er im Skript erwartet wird. Der Aufruf des Ant-Skripts sähe dann so aus:

```
ant -Dbuild.number=3
```

Das generelle Vorgehen ist so, dass Sie direkt vor das zu übergebende Schlüssel/Wert-Paar ein `-D` setzen.

Wir können nun noch das Ant-Skript optimieren und weniger fehleranfällig machen:

```
<project name="subversion" basedir="." default="release">
<property name="svn.dir" value="svn://localhost/agile/myproject"/>
<property name="svn.trunk" value="${svn.dir}/trunk"/>
<property name="svn.tags" value="${svn.dir}/tags"/>

<fail message="Please call script with build number!" unless="build.number"/>

<target name="release" description="tagging">
<exec executable="svn">
<arg value="copy"/>
<arg value="-r"/>
<arg value="${build.number}"/>
<arg value="${svn.trunk}"/>
<arg value="${svn.tags}/build-${build.number}"/>
<arg value="-m"/>
<arg value="tag for ${build.number}"/>
<arg value="--username"/>
<arg value="michael"/>
<arg value="--password"/>
<arg value="michael"/>
</exec>
<echo>Tag created: build-${build.number}</echo>
</target>
</project>
```

Was hat sich seit der ersten Version geändert? Wir verzichten auf das Einbeziehen von *build.properties*, da wir die Information daraus nun manuell übergeben. Ferner haben wir einen `fail`-Ausdruck ergänzt. Der ist hier nicht zwingend erforderlich, aber sehr hilfreich. Vergessen wir beim Aufruf des Skripts nämlich den Parameter, bricht es ab:

```
michael@BlueGene:~/subversion$ ant
Buildfile: build.xml

BUILD FAILED
/home/michael/subversion/build.xml:6: Please call script with build number!
```

Dieses Beispiel hat gezeigt, wie einfach Sie über Ant Subversion bedienen können. Auch diesen Ansatz möchte ich nicht als dogmatisch verstanden wissen: Schauen Sie, wo Sie Ihre Arbeit mit Subversion über Skripten vereinfachen können, und automatisieren Sie so viel wie möglich.

Wenn Sie nicht manuell die Revisionsnummer auswählen und an das Skript übergeben wollen, können Sie das auch automatisiert machen. Nutzen Sie dafür `svn info -xml`, um die Ausgabe in wohlgeformtem XML zu bekommen. Dieses XML kann dann von Ihrem Skript mit dem Ant-Task XmlProperty verarbeitet werden. Sie können sich so elegant die Revision aus dem XML heraussuchen und als Eingabe für Ihr Tagging nutzen.

Zusätzliche Werkzeuge

Neben Ant existieren zahlreiche weitere Werkzeuge, die Sie bei der Arbeit mit Subversion unterstützen. Möchten Sie nicht direkt mit der Kommandozeile arbeiten, haben Sie die Möglichkeit, aus Ihrer bevorzugten Entwicklungsumgebung (IDE) auf Subversion zuzugreifen. Daneben steht mit Kdiff3 ein Werkzeug zur Verfügung, das elegant beim Vergleich von Artefakten hilft.

IDE

Selbstverständlich haben alle gängigen Java-Entwicklungsumgebungen eine integrierte Subversion-Unterstützung, auch wenn diese eventuell nicht zum Basispaket gehört und nachgerüstet werden muss. Stellvertretend möchte ich an dieser Stelle auf IntelliJ IDEA eingehen.

Sehr komfortabel lassen sich mehrere Subversion-Repositories gleichzeitig über alle denkbaren Protokolle ansteuern. Der Zugriff gelingt nach erfolgreicher Autorisierung bzw. Authentifizierung. Im Subversion-Fenster sehen Sie dann das Repository mit dem Protokoll, das Sie zum Ansprechen benutzen (hier `svn`), und rekursiv alle Verzeichnisse und Dateien, die in dem Repository liegen (siehe Abbildung 8-1).

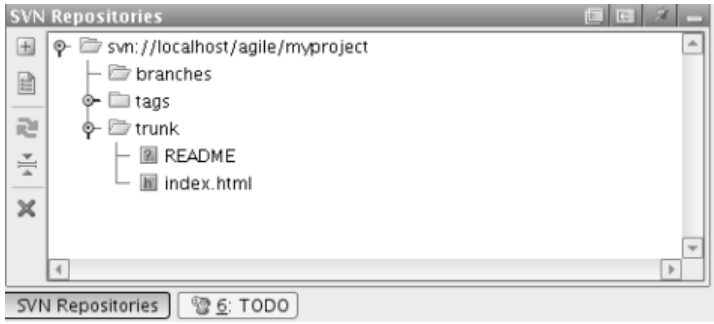


Abbildung 8-1 : Die Übersicht über das Repository von Subversion

Ein Klick auf ein Icon genügt, um sich weitere Informationen zu allen Artefakten im Repository darstellen zu lassen (siehe Abbildung 8-2). Hier sehen wir dann sofort die Revisionen mit dem jeweiligen Änderungsdatum. Alle denkbaren Funktionen werden unterstützt. So können wir in den Repositories browsen, einzelne Komponenten auschecken, ändern, wieder einchecken, kopieren, löschen und verschieben und Tags und Branches anlegen usw.

Ein genauerer Blick auf ein Artefakt zeigt die Änderungshistory: Wer hat wann was gemacht, welche Änderungsnachricht hat er dafür hinterlegt und welche Revision (in IDEA als Version bezeichnet) hatte dies zur Folge (siehe Abbildung 8-3).

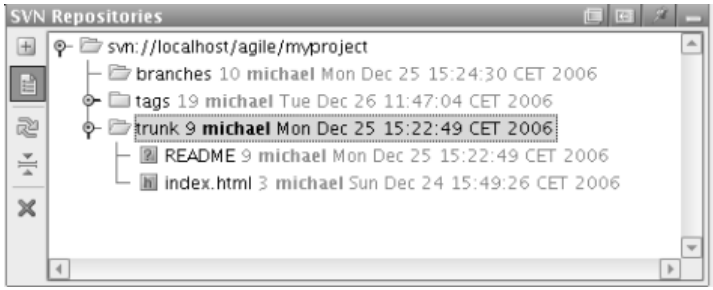


Abbildung 8-2: Details zu den Subversion-Artefakten

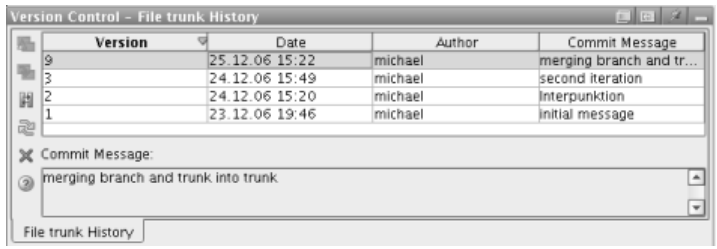


Abbildung 8-3: Die Änderungshistory zu einem Artefakt

KDiff3

Das für viele Plattformen (beispielsweise Windows oder Linux mit KDE3) verfügbare *Kdiff3* kann Ihnen helfen, mit möglichen Konflikten umzugehen. Es ist frei verfügbar unter *http://kdiff3.sourceforge.net/*. Es zeichnet sich durch folgende Eigenschaften aus:

- Möglichkeit zum Vergleichen und Zusammenführen von zwei oder drei Textdateien oder Verzeichnissen
- Zeigt die Unterschiede nicht nur auf Zeilenebene, sondern auch Buchstabenebene
- Integrierter Editor, um Konflikte zu lösen
- Ausdrucken von Unterschieden möglich

TortoiseSVN

Ein weiteres Werkzeug für Windows, um mit Subversion zu arbeiten, ist *TortoiseSVN*. Es ist verfügbar unter *http://tortoisesvn.tigris.org/*. Es ist ein Subversion-Client, der als Windows-Shell-Erweiterung implementiert ist: Es handelt sich nicht um eine IDE-Integration, sondern ermöglicht die Arbeit mit Dateien und ihren Versionen im Windows-Verzeichnis. Alle Protokolle werden komfortabel und mit großem Funktionsumfang unterstützt. Es lassen sich folgende Eigenschaften festhalten:

- Es ist einfach zu nutzen.
- Alle Kommandos sind über den Windows Explorer nutzbar (Einträge im Menü über rechte Maustaste, wie man es beispielsweise von *Kopieren* kennt).
- Die Befehle werden kontextsensitiv dargestellt: Nur diejenigen Kommandos werden angezeigt, die in einem bestimmten Zusammenhang Sinn machen.
- Der Status der Dateien ist direkt im Windows Explorer sichtbar.
- Das Verschieben von Dateien mit Drag-and-Drop wird unterstützt.

Subversion Pro, Polarion

Um unter *http://www.polarion.com/* Subversion Pro herunterzuladen, müssen Sie sich zunächst registrieren. Es handelt sich dabei um ein vom Unternehmen Polarion zusammengestelltes freies Paket, das aus unterschiedlichen Werkzeugen besteht. Dazu gehören *WebClient for SVN*, eine Weboberfläche zur Arbeit mit Subversion, *SVN Importer*, ein Werkzeug zur Datenmigration von anderen VCM-Systemen zu Subversion, *Subversive*, ein Subversion-Client für Eclipse, *Fast Track*, ein Tracker für Eclipse/Subversion, *SVK*, ein dezentrales VCM-System, das auf Subversion aufbaut, und *TortoiseSVN*.

Zusammenfassung

Ohne Versionierung haben Sie das Problem, dass Entwickler gegenseitig ihre Änderungen an den Artefakten überschreiben und nicht auf ältere Versionen von Dateien zurückgreifen können. Mit Subversion steht Ihnen ein mächtiges Werkzeug zur Verfügung, das Ihre Artefakte unter Versionskontrolle bringt. Subversion ist der kostenlose Nachfolger von CVS. In die Neuentwicklung sind viele konzeptionelle Ideen eingeflossen, deren Umsetzung die tägliche Arbeit noch einfacher macht, als es mit CVS bereits der Fall war. Subversion können Sie in Ihr agiles Projekt optimal einbinden. So lässt sich Subversion mit Ant integrieren und mit entsprechenden Werkzeugen wie jede gängige IDE bedienen. Subversion ist durch seine leichtgewichtige Funktionsweise, beispielsweise die Nutzung von Branches per Konvention, für agile Projekte prädestiniert.

KAPITEL 9

Komponententests mit JUnit

Entwickler (zumindest die meisten) haben schon immer ihre Java-Komponenten getestet. Das manuelle Testen und die Entwicklung von alleinstehenden Testklassen haben sich als recht ineffektiv herausgestellt, da es eine Herausforderung war, die Funktionalität der Klassen mit einem Soll-Verhalten zu vergleichen. Es haben sich daher technische Werkzeuge entwickelt, die bei der Erstellung von Tests unterstützen und erwartete Soll-Ergebnisse dem Ist-Zustand gegenüberstellen, sich automatisiert ausführen lassen und ihre Ergebnisse verständlich aufbereiten und bereitstellen. *JUnit* ist ein Werkzeug, das in diesem Kontext entwickelt wurde.

JUnit ist ein Framework, mit dem Entwickler Ihre Komponenten detailliert testen können. Mithilfe des Frameworks werden Komponenten durch in Java geschriebenen Testcode überprüft.

JUnit ist weit verbreitet und im Grunde Quasistandard bei Komponententests. Erst 2006 entwickelte sich das Werkzeug aus dem Versionszweig 3.x hin zu 4.x, was einen Quantensprung bedeutete. JUnit 4.x baut auf Java 5 auf, insbesondere auf seinem Annotationskonzept, das umfangreiche Verwendung findet. Es macht die Testklassen im Design sauberer, da das Verhalten nun durch Annotations gesteuert wird, nicht mehr durch Ableitungen und Namenskonventionen.

Dieses Kapitel behandelt JUnit 4.x-Testklassen, baut aber dennoch Brücken zu nach JUnit 3.x erstellten Testklassen. JUnit 3.x ist in vielen Projekten seit langer Zeit im Einsatz. Durch die Vermittlung des Verständnisses beider Konzepte wird eine Migrationshilfe angeboten, von 3.x- auf 4.x-Tests zu wechseln bzw. diese nebeneinander laufen zu lassen.

Ein weiteres Werkzeug für Komponententests neben JUnit ist TestNG, um das wir uns intensiv in Kapitel 12 kümmern werden.

Komponententests

Bei Komponententests (auch Modultests genannt) testen Sie einen Teil der gesamten Software. Dabei wird gewöhnlich das Verhalten einer Methode oder einer Klasse überprüft, indem bei der Ausführung einer Funktionalität der gelieferte Ist-Wert einem erwarteten Sollwert gegenübergestellt wird. Dabei steht die Frage »Mache ich das, was ich mache, richtig?« im Vordergrund. Komponententests sind gewöhnlich in derselben Sprache geschrieben wie die zu testenden Artefakte. Im Gegensatz dazu gibt es auch Akzeptanztests, die die zu implementierende Funktionalität definieren. Ein Anwender beschreibt dazu das Verhalten der Anwendung, bevor diese geschrieben ist. Dabei steht die Frage »Mache ich das Richtige?« im Vordergrund. Aber zurück zu den Komponententests.

JUnit

JUnit ist ein frei verfügbares Werkzeug, das anfänglich von Erich Gamma und Kent Beck, einem der Urväter des Extreme Programming, geschrieben wurde und über die Jahre eine breitere Gemeinde gefunden hat, die das Werkzeug weiterentwickelt. Es basiert auf Konzepten des Unit-Test-Frameworks für die Programmiersprache Smalltalk, das heute unter SUnit bekannt ist. Viele Sprachen besitzen Portierungen, die nach dem Schema *xUnit* benannt werden, für Java existiert z.B. die Portierung JUnit.

JUnit ist für seine Einfachheit bekannt: Das zugrundeliegende Konzept ist recht einfach umzusetzen. Es existieren zwar Fallstricke, die wir uns später anschauen, doch kommt der Entwickler schnell zu einem brauchbaren Ergebnis. Neben der einfachen Nutzung ist das Werkzeug auch durch die visuelle Darstellung von Fehlern und korrekten Durchläufen des Testcodes bekannt geworden. Wenn ein Test erfolgreich durchläuft, wird das mit einem grünen Balken symbolisiert, misslingt ein Test, ist der Balken rot.

Installationshinweise

Unter *http://junit.org* können Sie sich die aktuelle Version herunterladen. Die *.zip*-Datei mit der Version 4.x entpacken Sie an einen Ort Ihrer Wahl. Was Sie dabei auf jeden Fall benötigen, ist das *.jar*-File.

Die einzelnen Versionen im Versionszweig 4 unterscheiden sich nur in einzelnen Facetten. Im weiteren Verlauf wird JUnit 4.1 genutzt. Das gegen Ende der Bucherstellung neu erschienene JUnit 4.4 ergänzt das Werkzeug um neue Konzepte wie *Assumptions* (Annahmen über die Umgebung mithilfe des Schlüsselwortes `assumeThat`) und *Theories* (Zusammenführung von Testszenarien mit bestimmten *Assumptions* mithilfe der JUnit-Annotation `@Theory`) und der Möglichkeit, Assertions mit der lesbaren, kompakten Notation `assertThat` zu formulieren.

Bei Version 4.1 heißt dieses File *junit-4.1.jar*. Sie sollten es mit in den Klassenpfad aufnehmen.

```
export CLASSPATH=$CLASSPATH:/path/to/junit/junit-4.1.jar
```

Nach erfolgter Installation von JUnit können Sie Testklassen (Java-Klassen, die die JUnit-Klassen nutzen) erstellen und ausführen. Dazu ist ein *TestRunner* verfügbar, der über die Konsole aufrufbar ist (siehe gleich).

Die Beispiele basieren auf JUnit in der Version 4.1. Ausschlaggebend dabei ist, dass es sich um den Entwicklungszweig 4 handelt, da nur dort die hier vorgestellten Funktionen umfassend bereitstehen.

Ein erstes Beispiel

Um uns mit der Terminologie vertraut zu machen, starten wir direkt mit einem einfachen Beispiel. Nachdem das Konzept grundsätzlich verstanden ist, können beliebig komplexe Testfälle abgeleitet werden. Unser allererstes Beispiel (siehe Beispiel 9-1) testet die Java-Klasse *java.util.ArrayList*.

Erste Richtschnur kann sein, dass Sie zu einer funktionalen Klasse jeweils eine Testklasse haben. In der Praxis ist *ein* Test für *ein* Feature anzustreben, eine Funktionalität, die von einer Klasse angeboten wird. Eine Klasse kann auch mehrere Features anbieten, die dann einzeln getestet werden sollten. Unsere zu testende Klasse *java.util.ArrayList* hat die Eigenschaft, dass sie leer sein und neue Elemente aufnehmen kann. *ArrayList* kann die Anzahl der aufgenommenen Elemente zurückliefern. Wir möchten testen, ob eine neu initialisierte *ArrayList* wirklich leer ist und nach Hinzufügen eines neuen Elements dieses tatsächlich vorhanden ist. Berücksichtigen Sie hier, dass dieses Beispiel die generelle Nutzung von JUnit illustrieren soll. Dabei kommt bewusst ein einfaches Beispiel zum Einsatz. In der Praxis werden Sie für Java-Standardklassen keine Tests schreiben (siehe dazu Seite 176).

Die Testklasse *SampleTest* in Beispiel 9-1 ist eine gewöhnliche Java-Klasse. Sie müssen also beispielsweise nicht von einer JUnit-Basisklasse ableiten. Die von JUnit aufgerufenen Testmethoden sind mit der Annotation `@Test` gekennzeichnet. Diese Methoden prüfen gewöhnlich ein Feature der fachlichen Klasse durch Definition eines Soll- und Ist-Werts. Ein solcher Vergleich kann ein Vergleich eines aktuellen Wertes mit einem erwarteten Wert oder auch das Auftreten einer Exception sein.

Im Beispiel möchten wir zwei Tests durchführen, und zwar einerseits, ob die Collection zu Beginn tatsächlich leer ist. Dazu nutzen wir in der Methode `testEmptyCollection` das JUnit-Sprachkonstrukt `assertTrue()`. Dabei wird der Methode `assertTrue()` ein Parameter übergeben, dessen boolescher Ausdruck wahr sein muss, damit die Methode als korrekt gilt.

Der zweite Test `testOneItemCollection` fügt der Collection ein Element hinzu, um anschließend zu überprüfen, ob die Größe der Collection tatsächlich um 1 inkrementiert

wurde. Dazu nutzen wir die JUnit-Assertion `assertEquals`. Werden der `assertEquals`-Methode zwei Objekte übergeben, überprüft JUnit diese auf Gleichheit (ob die Objekte identisch sind und die Referenzen auf dasselbe Objekt zeigen).

Sie können mit `assertEquals` auch primitive Typen vergleichen. Dabei unterscheidet sich das JUnit-Vorgehen je nach Typ. Bei `double`-Werten wird beispielsweise deren tatsächlicher numerischer Wert verglichen. Bei `int`-Werten kommt Autoboxing zum Einsatz: Die `int`-Werte werden in Integer-Wrapperobjekte transformiert, die dann mittels `equals` verglichen werden.

Um die beiden Tests laufen zu lassen, benötigen wir jeweils eine instantiierte `ArrayList`. Das Anlegen des `ArrayList`-Objekts könnten wir jeweils in der Testmethode selbst durchführen. Da dies redundant wäre, ziehen wir den Schritt raus und legen ihn zentral in die Methode `setUp()`. Der Name der Methoden spielt dabei, ebenso wie die Namen der Klasse und der Testmethoden, keine Rolle.

Indem wir diese Methode mit `@Before` annotieren, informieren wir JUnit darüber, dass diese Methode vor jeder Testmethode ausgeführt werden soll. Dieser Initialisierungsteil einer Testklasse wird *Fixture* bezeichnet. Hier werden immer die konkrete Testfälle vorbereitet, eine Testumgebung aufgebaut und Variablen initialisiert.

Existieren in der Klasse Testmethoden, die dieses Fixture nicht benötigen, dann ist das ein Indikator dafür, dass man diese Testmethode in eine weitere Testklasse auslagern und von den anderen Testmethoden dieser Klasse separieren sollte.

Wir benötigen nun noch einen Mechanismus, um nach dem Test (genauer: nach Ausführung einer Testmethode) die Testumgebung wieder aufzuräumen. Das kann zum Beispiel bei für den Test geöffneten Datenbankverbindungen notwendig sein. JUnit bietet die Möglichkeit, Schritte nach der Ausführung einer Testmethode durchzuführen. Die mit `@After` annotierten Methoden werden nach jedem Test aufgerufen. In unserem Beispiel entfernen wir alle Elemente aus unserer Collection. `@AfterClass` und `@BeforeClass` kennzeichnen Methoden, die für alle Tests der Klasse nur einmal (ganz am Anfang vor den Tests, oder ganz am Ende nach allen Tests) ausgeführt werden.

Beispiel 9-1: Testklasse für java.util.ArrayList

```
package com.huettermann.junit;

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

import java.util.ArrayList;
import java.util.Collection;

public class SampleTest {
```

Beispiel 9-1: Testklasse für java.util.ArrayList (Fortsetzung)

```
    private Collection<String> collection;

    @BeforeClass
    public static void oneTimeSetUp() {
        // one-time initialization code
    }

    @AfterClass
    public static void oneTimeTearDown() {
        // one-time cleanup code
    }

    @Before
    public void setUp() {
        collection = new ArrayList<String>();
    }

    @After
    public void tearDown() {
        collection.clear();
    }

    @Test
    public void testEmptyCollection() {
        assertTrue(collection.isEmpty());
    }

    @Test
    public void testOneItemCollection() {
        collection.add("itemA");
        assertEquals(1, collection.size());
    }
}
```

Führen wir nun unsere Tests durch. Dazu kompilieren Sie zunächst die Klasse mit

```
javac -cp /path/to/junit/junit-4.1.jar SampleTest.java
```

um sie anschließend von dem JUnit-Runner aufrufen zu lassen, der die Tests durchführt. Der Junit-Runner ist ein Java-Programm, das Sie mit *java* starten. Beachten Sie, dass Sie dabei den Classpath richtig gesetzt haben müssen (das *junit.jar* muss genauso gefunden werden wie die zu testende Klasse) und der Aufruf die zu testende Klasse erwartet (ohne Extension!):

```
java -cp /path/to/junit/junit-
4.1.jar:/path/to/classes org.junit.runner.JUnitCore SampleTest
```

Der Aufruf liefert die Meldung, dass zwei Tests erfolgreich durchlaufen worden sind (siehe Abbildung 9-1).

```
JUnit version 4.1
..
Time: 0,029

OK (2 tests)

michael@BlueGene:~/tools/junit/com/huettermann/junit$
```

Abbildung 9-1: Unit Konsole – Zwei Tests erfolgreich durchlaufen

Die manuellen Schritte sind für das Verständnis von JUnit sehr hilfreich. Das manuelle Vorgehen, also auf der Konsole die Tests anzustoßen, wird in der Praxis nur in Einzelfällen praktikabel sein. Beim manuellen Ausführen müssen Sie beim Kompilieren und Ausführen immer den Classpath angeben oder sicherstellen, dass er im Profil des von Ihnen genutzten Benutzers hart verdrahtet ist. Diese verringerte Flexibilität in Kauf nehmend, haben wir ferner das Problem, dass wir nur wenige Zeilen Ausgabe auf der Konsole bekommen.

Tests, Zusicherungen und Exceptions

JUnit ermöglicht den Entwurf von Testklassen nach *Design-by-Contract*, der Definition von Verträgen bei der Interaktion von Klassen. Dabei wird mit Zusicherungen gearbeitet. Vorbedingungen sind Zusicherungen, die der *Aufrufer* einhalten muss, Nachbedingungen sind Zusicherungen, die der *Aufrufende* einhalten muss. Mit JUnit können Zusicherungen mit Assertions durchgeführt bzw. abgefragt werden, nachdem auf den zu testenden Klassen Aktivitäten durchgeführt wurden.

Die Zusicherungsmethoden JUnits gehorchen gewöhnlich der Form `assertXXX(...)`. Die Annotations, die für jede Instanz der Testklasse durchlaufen werden, setzen Methoden mit den Modifier `public void` voraus. Auf der anderen Seite existieren die Annotations des Klassen-Scope. Diese können ausschließlich `public static void` Methoden kennzeichnen. Sie können folgende Assertions nutzen:

- `@AssertEquals` (zwei Objekte sind gleich, Ausnahmen bei primitiven Typen)
- `@AssertTrue` (eine Bedingung ist wahr)
- `@AssertFalse` (eine Bedingung ist falsch)
- `@AssertNull` (ein Objekt ist Null)
- `@AssertNotNull` (ein Objekt ist nicht Null)
- `@AssertSame` (zwei Objekte referenzieren dasselbe Objekt)
- `@AssertNotSame` (zwei Objekte referenzieren nicht dasselbe Objekt)

→

Die eigentlichen Tests (die Assertions) liegen in mit `@Test` annotierten Methoden innerhalb der Testklassen. Neben `@Test` existieren noch weitere Annotations, die den Aufruf der Methode in Ihrer Testklasse steuern:

- `@After` (wird nach jeder mit `@Test` markierten Methode ausgeführt)
- `@AfterClass` (wird einmalig nach allen Tests ausgeführt)
- `@Before` (wird vor jeder mit `@Test` markierten Methode ausgeführt)
- `@BeforeClass` (wird einmalig vor allen Tests ausgeführt)
- `@Ignore` (markiert eine Methode, die von JUnit-Prozessierung ausgenommen werden soll)
- `@Test` (markiert eine Methode, die die Zusicherungen enthält und von JUnit ausgeführt wird)

Sie können in Ihren Tests auch Exceptions überprüfen. Dazu gibt es zwei Möglichkeiten. Die erste Möglichkeit ist, die erwartete Exception als Bedingung hinter der `@Test-Annotation` zu verwenden. Möchten wir also beispielsweise beim Zugriff auf eine leere Liste eine *IndexOutOfBoundsException* bekommen, so sähe das so aus:

```
@Test(expected = IndexOutOfBoundsException.class)
  public void testForException() {
      Object o = emptyList.get(0);
  }
```

Wenn Sie allerdings eine Methode haben, deren Test beim Auftreten einer unerwarteten Exception nicht erfolgreich sein soll, so arbeiten Sie mit `throws`.

```
  @Test
public void testForException() throws IndexOutOfBoundsException{
      Object o = emptyList.get(0);
  }
```

Da Sie gewöhnlich eine große Anzahl von Tests haben und diese, genau wie die fachlichen Klassen, in der von Ihnen bevorzugten IDE schreiben, möchten Sie die Tests selbstverständlich auch aus der IDE starten (dazu kommen die IDE-unabhängigen Skripten Ihres kontinuierlichen Integrationsprozesses).

Alle bekannten IDEs wie Eclipse, NetBeans oder IntelliJ IDEA bringen eine JUnit-Integration mit. Diese ist intuitiv nutzbar und von den IDE-Herstellern ausführlich dokumentiert. So ist es beispielsweise möglich, auf Knopfdruck Tests zu starten. IDEA beispielsweise erkennt beim Betätigen eines *RUN*-Buttons direkt, dass es sich um eine JUnit-Testklasse handelt.

Sie sollten Ihre Tests nicht von einer IDE abhängig machen. Einerseits haben Ihre Entwickler ggf. unterschiedliche IDEs, andererseits ist es wichtig, die Tests automatisiert während der kontinuierlichen Integration (unabhängig von IDEs) von Ant oder Maven gestartet laufen zu lassen.

> Das Starten von Tests in der IDE durch dort verfügbare Knöpfe sollte nur komplementär durchgeführt werden. Stattdessen sollten Sie es so einrichten, dass das Projekt über Ant-Skripten verfügt, die auch aus der IDE heraus ausgeführt werden können.

Auch hier ist die Voraussetzung, die *.jar*-Datei von JUnit in den Klassenpfad Ihres Java-Projekts zu legen, was Sie nur einmalig und visuell unterstützt komfortabel erledigen können. Das Ergebnis des Testlaufs wird dann direkt auf dem UI eingebunden (siehe Abbildung 9-2).

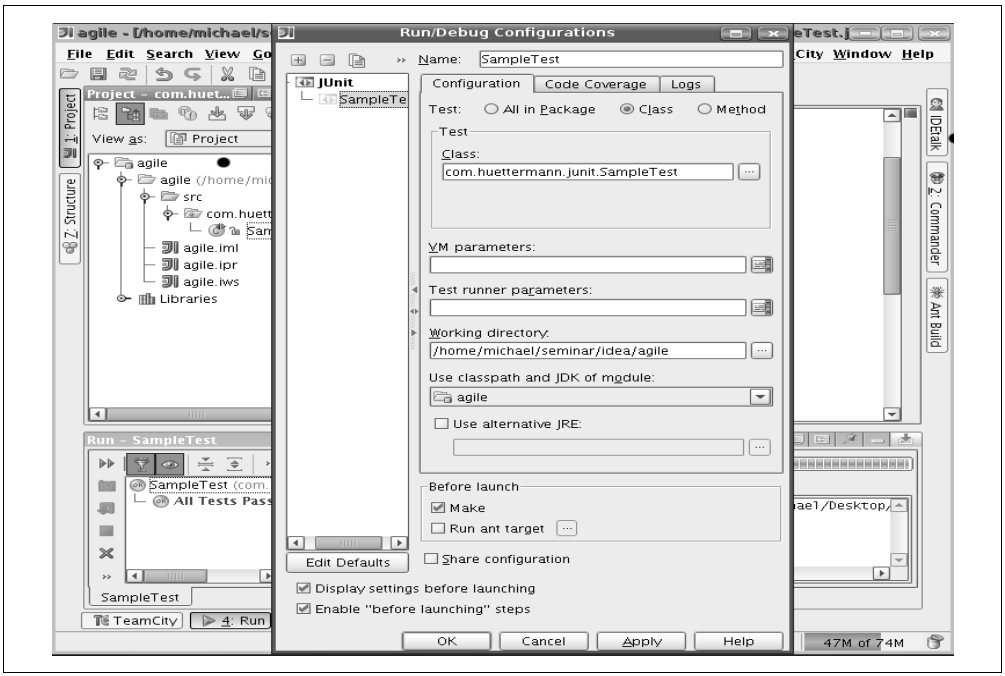


Abbildung 9-2: JUnit mit IDEA

Bei Eclipse reicht ein Mausklick auf ein Projekt, um alle dort befindlichen Unit-Tests starten zu lassen.

JUnit 3.x vs. JUnit 4.x

Die traditionellen Konzepte aus JUnit 3.x lassen sich mit neuen aus Version 4.x kombinieren. Das erlaubt eine sukzessive Umstellung bestehender alter 3.x-Testklassen bzw. die Erstellung neuer 4.x-Testklassen, die mit den alten 3.x-Testklassen zusammen ausgeführt werden.

Vom Design her sind die Konzepte aus 3.x und 4.x leider nicht sauber getrennt. Trotz aller Neuerungen und Mächtigkeit bleibt manchmal das Gefühl, dass mit JUnit 4.x schnell etwas bereitgestellt werden sollte, nur um mit der Konkurrenz von TestNG gleichzuziehen. Die wesentlichen Unterschiede zwischen JUnit 3.x und JUnit 4.x sind in der Tabelle gegenübergestellt.

Tabelle 9-1: JUnit 3.x vs. JUnit 4.x

Merkmal	JUnit 3.x	JUnit 4.x
Package	*Junit.framework*	*Org.junit*
Testklasse	Leitet von `TestCase` ab	-
Testmethode	Namenspräfixtest	`@Test`
assertXXX	geerbt	Statischer Import
Testmengen	`static Test suite()`	IDE oder *ant*
Vor jeder Testmethode	`setUp()`	`@Before`
Nach jeder Testmethode	`tearDown()`	`@After`
Vor allen Testmethoden	Durch `TestSetup` möglich	`@BeforeClass`
Nach allen Testmethoden	`TestSetup`	`@AfterClass`
Exception E prüfen	`fail()`	`@Test(expected=E.class)`

JUnit 3.x hatte noch ein eigenes GUI. JUnit 4.x kommt nur noch mit dem textuellen TestRunner daher. Auf ein eigenes GUI wird nun verzichtet. Das trägt der Entwicklung Rechnung, dass mittlerweile alle führenden IDEs JUnit-Unterstützung bieten.

Interessant ist, dass Sie JUnit 4-Tests in einer JUnit 3.x-Umgebung laufen lassen können. Dazu fügen Sie lediglich die folgende Methode jeder Testklasse hinzu:

```
public static Test suite() {
   return new JUnit4TestAdapter(MyClass.class);
}
```

In einer Umgebung, in der Sie Tests nach JUnit 3.x und Tests nach JUnit 4.x (mit Annotations) vorliegen haben, müssen Sie besonders aufpassen. Es folgt ein Beispiel dafür, wo hier Fallstricke liegen können.

```
package com.huettermann.junit;

import org.junit.*;
import static org.junit.Assert.*;
import junit.framework.TestCase;

/**
 * @author Michael Hüttermann
 */
public class MyTest extends TestCase{
    private int count = 0;
```

```
    @Test
    public void test2() {
        count++;
        assertEquals(1, count);
    }

    @Test
    @Ignore
    public void test1() {
        count++;
        assertEquals(3, count);
    }
}
```

Der Durchlauf wird einen fehlgeschlagenen Test melden. Wir haben die `test1`-Methode zwar mit `@Ignore` ausgezeichnet und würden vermuten, dass diese Testmethode übergangen wird. Leider haben wir gleichzeitig von `TestCase` abgeleitet. Wir haben hier die konventionelle Herangehensweise (Ableitung von `TestCase`) mit der neuen verheiratet (annotierte Testmethoden). Die Testmethoden werden zwar richtig erkannt, das `@Ignore` wird allerdings ignoriert. Ist die Semantik von `@Ignore`, dass sich diese Annotation selbst ignoriert?

Schauen wir uns das Beispiel in etwas abgewandelter Form an.

```
package com.huettermann.junit;

import org.junit.*;
import static org.junit.Assert.*;
import junit.framework.TestCase;

/**
 * @author Michael Hüttermann
 */
public class MyTest {
    private int count = 0;


    @Test
    public void test2() {
        count++;
        assertEquals(1, count);
    }

    @Test
    @Ignore
    public void test1() {
        count++;
        assertEquals(3, count);
    }
}
```

Was haben wir geändert? Wir leiten unsere Testklasse nicht mehr von `TestCase` ab. Ein Start der Testklasse durch IDEA liefert tatsächlich das erwartete Ergebnis. Ein Test wird erfolgreich durchgeführt und einer übersprungen:

```
IDEA: Test 'com.huettermann.junit.MyTest.test1' ignored.
```

Sie können JUnit 3.x-Tests mit JUnit 4.x-Tests verheiraten. Sehen Sie die beiden JUnit-Versionen nicht als sich ausschließende Versionen, von der die neue ein vollständiger Nachfolger des Vorgängers ist. Abhängig davon, was Sie testen wollen, kann ein Einsatz von JUnit 3.x-Konzepten sinnvoll sein. Ein Beispiel dafür sind die *Suites* (siehe Seite 171), denn hier werden Testklassen als Parameter erwartet, die tatsächlich von `TestCase` ableiten müssen.

Testanatomie

Nun möchte ich Ihnen zwei einzelne Nutzungsmöglichkeiten vorstellen, die Sie bei der Erstellung von Testfällen nutzen können, um in Ihrem Projekt gute agile Tests bereitzustellen. Stellvertretend sind das Praktiken, um Methoden über ganze Suites hinweg einmal aufrufen zu lassen sowie die Parametrisierung von Testmethoden.

Zentrale Fixture und tearDown

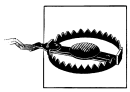
Eine interessante Variante bei der Erstellung von Suites und Aggregationen ist die zentrale Definition einer übergreifenden `setUp`-Methode und einer `tearDown`-Methode, um im Rahmen einer Suite übergreifend für alle Tests eine Methode einmalig zu Beginn oder am Ende auszuführen.

Wir haben diese beiden Methoden als Möglichkeit kennengelernt, vor bzw. nach dem Ausführen jeder Testmethode Vorbereitungen für den Test durchzuführen (Fixture), oder den Test sauber abzuräumen. Durch geschicktes Design können Sie eine Suite erstellen, die zentral für viele Testklassen und -methoden ein einheitliches Fixture oder entsprechend eine Abräumfunktionalität bietet.

Das Beispiel, das wir uns dazu anschauen, macht uns außerdem mit einem weiteren Konzept von JUnit vertraut. Es ist möglich, mit der Annotation `@RunWith` auf Testklassenebene eine Klasse zu definieren, mit der diese Suite ausgeführt werden soll. Es wird also bewusst der normale `junit.runner` durch einen eigenen ersetzt. `@RunWith` definiert dabei das Startverhalten. Die Klasse `AllTests` selbst leitet von keiner JUnit-Klasse ab.

```
package com.huettermann.junit;

import org.junit.runner.RunWith;
import org.junit.runner.JUnitCore;

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

```
/**
 * @author Michael Huettermann
 * Suite with central setup and teardown
 */
@RunWith(Suite.class)
@Suite.SuiteClasses({MyTestCase1.class, MyTestCase2.class})
public class AllTests {

    @BeforeClass
    public static void setUp() {
        System.err.println("setUp");
    }

    @AfterClass
    public static void tearDown() {
        System.err.println("tearDown");
    }

    /**
     * start if not via IDE
     * @param args start parameter
     */
    public static void main(String[] args) {
        JUnitCore.main(AllTests.class.getName());
    }
}
```



Da Sie Tests mithilfe von Ant beliebig gruppieren können und auch Ihre IDE das komfortable Starten von Tests unterstützt, ist die Anwendung des Suite-Konzepts nicht zwangsläufig notwendig. Sie können aber die Sprachmerkmale von JUnit 3 mit den neuen von JUnit 4 beliebig mischen. Es ist also denkbar, dass Sie Suites erstellen und diese per Ant einbinden und von JUnit 4 verarbeiten lassen.

Wir haben nun eine Klasse, die wir durch einen konfigurierten, vordefinierten Testrunner `Suite.class` als Suite starten. Dieser Suite fügen wir durch `@Suite.SuiteClasses` Testklassen hinzu (es werden also alle Testmethoden der übergebenen Testklassen hinzugefügt). Bevor all diese hier definierten Testklassen gestartet werden, wird die in dieser Klasse definierte `setUp()`-Methode ausgeführt. Hier können für alle Testfälle übergreifend Vorbedingungen vorbereitet werden. Diese setUp-Methode (der Name ist unerheblich) muss mit `@BeforeClass` annotiert werden. Das Gleiche gilt für das zentrale Abräumen der Testumgebung, nachdem alle Tests gelaufen sind. Dafür ist in der zentralen Klasse eine Methode mit `@AfterClass` zu kennzeichnen.

IDEA erkennt anhand `@RunWith` sofort, dass es sich um ein JUnit-Artefakt handelt. Die `main`-Methode ist hier gar nicht notwendig. Sie wird allerdings notwendig, wenn Sie `AllTests` von der Konsole starten.

Eine Methode parametrisieren

Sie können mit JUnit eine Testmethode parametrisieren. Das bedeutet, dass Sie eine Testmethode mehrfach, aber mit jeweils anderen Werten durchlaufen möchten. Dadurch haben Sie die Möglichkeit, das Verhalten der Testmethode (bzw. genauer: der zu testenden Klasse) mit unterschiedlichen außerhalb der Testklasse definierten Daten zu überprüfen.

Es sind verschiedene Möglichkeiten denkbar. Einerseits könnten Sie beispielsweise innerhalb der Testmethode externe Property-Dateien auslesen, und zwar mit jedem Durchlauf eine andere. Hier stelle ich Ihnen ein Beispiel vor, in dem die Testmethode die jeweils neuen herangezogenen »Daten« durch Parameter übergeben bekommt.

Als Beispiel fungiert eine Suite, die mehrere Male dieselbe Klasse instanziiert, allerdings jeweils einen anderen Wert übergibt. Es wird immer die gleiche Testmethode `testFoo()` ausgeführt. Wird die Klasse mit dem Parameter 5 erzeugt, scheitert der Test.

```
package com.huettermann.junit;

import junit.framework.*;

/**
 * @author Michael Hüttermann
 */
public class OneMethodMultipleTimes extends TestCase {

    int i;

    /**
     * Constructor taking the current value
     * @param i the value
     */
    public OneMethodMultipleTimes(int i) {
        super("testFoo");
        this.i = i;
    }

    /**
     * suite for test aggregation
     * @return Test
     */
    public static junit.framework.Test suite() {
        TestSuite result = new TestSuite();
        result.addTest(new OneMethodMultipleTimes(2));
        result.addTest(new OneMethodMultipleTimes(1));
        result.addTest(new OneMethodMultipleTimes(5));
        result.addTest(new OneMethodMultipleTimes(3));
        result.addTest(new OneMethodMultipleTimes(4));
        return result;
    }

    /**
```

```
     * My Test method.
     */
    public void testFoo() {
        assertTrue(i < 5);
    }
}
```

Instanzen und Abhängigkeiten

Die Ausführungsreihenfolge der Testmethoden ist nicht festgelegt. Für jede Testmethode wird eine neue Instanz der Testklasse instanziiert. Schauen wir uns dieses Beispiel an. Laufen die Tests erfolgreich durch?

```
package com.huettermann.junit;

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

/**
 * @author Michael Hüttermann
 */
public class MyTest {
    private int count = 0;

    @Test
    public void test1() {
        count++;
        assertEquals(1, count);
    }

    @Test
    public void test2() {
        count++;
        assertEquals(1, count);
    }
}
```

Beide Methoden laufen erfolgreich durch. Die Member-Variable `count` ist hier auf ganz traditionelle Weise als solche definiert und initialisiert, das heißt, ihr wird mit 0 ein Wert zugewiesen. Dies hat den gleichen Effekt, als ob wir der Variable den Wert 0 in einer mit `@Before` gekennzeichneten Methode zuweisen würden.

Möchten Sie viele Abhängigkeiten testen bzw. bauen Ihre Testszenarien auf eine Sequenz von verschiedenen nacheinander ausgeführten Testmethoden auf, dann erwägen Sie den Einsatz von TestNG.

Sowohl beim Durchlauf der Methode `test1()` als auch bei `test2()` ist `count` zu Beginn `0`. Die Reihenfolge, in der die beiden Tests laufen, ist nicht bestimmt. Abhängigkeiten und Testreihenfolgen lassen sich auf diese Weise nicht realisieren.

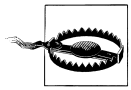
Testaggregation und Nebenläufigkeit

Gewöhnlich besitzen Sie viele Tests und viele Testklassen. Sie haben vielleicht sogar mehr Testklassen als Sie fachlichen Quelltext besitzen. Praktisch ist, die Testklassen weiter zu bündeln, also eine Gruppe von Klassen lose zusammenzufassen und diese zusammen auszuführen. So ist es möglich, die Artefakte effektiv, nämlich umfassend und schnell zu testen.

Die Nutzung von Testaggregation geschieht auch in JUnit 4.x nach wie vor mit dem Suite-Konzept. Generell wird davon ausgegangen, dass Sie Ihre Testgruppen mit Ihrer IDE formen (z.B. durch Start aller Tests zu einem Projekt in Eclipse) oder mit Ant oder Maven die Testklassen entsprechend zu Gruppen zusammenstellen.

Mittlerweile befassen sich die gängigen IDEs so intensiv mit JUnit-Integration, dass auch für diese Anforderung keine Wünsche offen bleiben. In IDEA zum Beispiel reicht ein Mausklick, um einen ganzen Satz von Tests anzustoßen. Auch mit Ant und Maven ist das möglich (dazu später mehr).

JUnit selbst bietet Einiges an Unterstützung bei der Aggregation von Tests. Hier kommen die *Suites* ins Spiel. Die Suites bieten eine weitere Abstraktionsebene oberhalb der Testklassen und sind ein Konzept aus JUnit 3.x, zu dem es in 4.x kein Gegenstück gibt. Suites ermöglichen, einzelne Testklassen zu Gruppen zusammenzufassen und die Tests in den Gruppen komplett in einem Schritt anzustoßen.



Verwechseln Sie die Gruppierung mit Suites nicht mit Gruppierung von einzelnen Testmethoden. TestNG ermöglicht, einzelne Testmethoden innerhalb einer Klasse zu Gruppen zusammenzufassen. Suites umfassen gewöhnlich ganze Klassen.

Wir schauen uns nun ein zusammenhängendes Beispiel an, das verschiedene Möglichkeiten der Testaggregation nutzt, siehe Beispiel 9-2. Dabei greifen wir auf die JUnit 3.x-Notation zurück. Die Klasse `MyTestCase1` verfügt über vier Testmethoden, deren Inhalt für unsere Zwecke irrelevant ist. Über die IDE gestartet, laufen alle vier Tests erfolgreich durch. Hier sehen wir das Vorgehen, wie es mit JUnit 3.x noch normal war. Die Testklasse leitet von `TestCase` ab, und die Testmethoden beginnen alle mit `test`. Ganz ähnlich sieht es mit der Klasse `MyTestCase2` aus. Hier existieren zwei Testmethoden, die ebenfalls erfolgreich durchlaufen. Ein öffentlicher Konstruktor ist bei diesem Vorgehen nach JUnit 3.x notwendig.

Testsuites zu erstellen ist eine statische Zusammenführung von Testklassen. Daneben ist es auch möglich, Testklassen dynamisch zusammenzusuchen und auszuführen. Erstellen Sie sich dabei ein Projekt, das die Testklassen im Classpath hat. Innerhalb des Projekts durchlaufen Sie den Classpath und filtern dynamisch die Klassen heraus, die Sie testen wollen,

und fügen Sie sie einer TestSuite hinzu. Bedenken Sie jederzeit, dass Ihr Vorgehen die Testgruppen zusammenzustellen auch im »Hintergrund« laufen muss: Tatsächlich ausschlaggebend ist, wie Ant die Tests während der kontinuierlichen Integration findet und starten kann.

Beispiel 9-2: Verschiedene Arten der Testaggregation

```
package com.huettermann.junit;

import junit.framework.Assert;
import junit.framework.TestCase;

/**
 * @author Michael Huettermann
 */
public class MyTestCase1 extends TestCase {

    /**
     * any test method
     */
    public MyTestCase1(String testName){
        super(testName);
    }

    /**
     * any test method
     */
    public void testSomeTest1(){
        String s1= "abc";
        String s2= "abc";
        Assert.assertSame(s1,s2);
    }

    /**
     * any test method
     */
    public void testSomeTest2(){
        String s1= "abc";
        String s2= "abc";
        Assert.assertEquals(s1,s2);
    }

    /**
     * any test method
     */
    public void testSomeTest3(){
        String s1= "abc";
        String s2= "abc";
        Assert.assertEquals(s1,s2);
    }

    /**
     * any test method
     */
```

Beispiel 9-2: Verschiedene Arten der Testaggregation (Fortsetzung)

```
    public void testSomeTest4(){
        String s1= "abc";
        String s2= "abc";
        Assert.assertEquals(s1,s2);
    }
}


package com.huettermann.junit;

import junit.framework.TestCase;
import junit.framework.Assert;
import org.junit.Test;

/**
 * @author Michael Huettermann
 */
public class MyTestCase2 extends TestCase {

    /**
     * constructor
     */
    public MyTestCase2(String testName){
        super(testName);
    }

    /**
     * any test method
     */
    public void testSomeTest5(){
        String s1= "abc";
        String s2= "abc";
        Assert.assertEquals(s1,s2);
    }


    /**
     * any test method
     */
    public void testSomeTest6(){
        String s1= "abc";
        String s2= "abc";
        Assert.assertEquals(s1,s2);
    }
}
```

`MyTestSuite` ist nun im Folgenden ein neues Konstrukt. Diese Klasse verfügt ausschließlich über eine statische Methode `suite` (siehe Beispiel 9-3), die von unserer IDE als JUnit Test Suite erkannt wird. Hier wird eine `ActiveTestSuite` erstellt. Das ist eine Suite, also ein Gruppierung von Testklassen, deren Testmethoden in separaten Threads durchgeführt werden. Auf diesem Weg kann Nebenläufigkeit getestet werden. Wir fügen dieser

Suite die Testklasse `MyTestCase1` hinzu. Wir möchten hier die Methode `testSomeTest1` der Testklasse testen, also übergeben wir genau diesen Namen der Testmethode bei der Erstellung des `TestCase`.

Beispiel 9-3: Unsere neue MyTestSuite

```
package com.huettermann.junit;

import junit.framework.Test;
import junit.extensions.ActiveTestSuite;
import junit.extensions.RepeatedTest;

/**
 * @author Michael Huettermann
 */
public class MyTestSuite  {

    /**
     * a suite
     * @return suite my suite
     */
    public static Test suite() {
        ActiveTestSuite suite = new ActiveTestSuite("Test suite");
        MyTestCase1 mtc = new MyTestCase1("testSomeTest1");
        suite.addTest(new RepeatedTest(mtc,5));
        return suite;
    }
}
```

Wir übergeben der Testsuite nicht nur die Testklasse, wir erstellen auch zunächst einen `RepeatedTest`. Ein RepeatedTest erlaubt das mehrmalige Ausführen eines Tests. `RepeatedTest` übergeben wir die Testklasse und teilen ferner mit, dass ihr Test fünfmal durchgeführt werden soll. Wir erreichen also durch diese Suite und die Dekoration mit ActiveTestSuite und RepeatedTest Folgendes: Wir stellen eine Suite bereit, die aus einer Klasse besteht, deren Testmethoden fünfmal und nebenläufig durchlaufen werden. Mit der Kombination `ActiveTestSuite` und `RepeatedTest` lassen sich also prima nebenläufige Konstellationen abbilden sowie Stress- oder Ausdauertests durchführen. Beide Artefakte befinden sich im *junit.extensions*-Package, das zur Standarddistribution gehört.

In TestNG gibt es das Konzept, einzelne Tests zu Gruppen zusammenzufassen und auszuführen. Diese Ausführungen können optional in separaten Threads angestoßen werden.

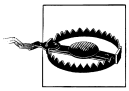
Lassen wir die Suite laufen, bekommen wir wieder die Information, dass alle Tests erfolgreich durchgeführt wurden. Wir verzichten hier auf *Logging*, das das nachweisen und die Testausgaben verfügbar machen würde.

Sie können bei Bedarf die einzelnen Testmethoden mit Logging-Ausgaben von Log4J oder der in Java 5 integrierten Logging-Funktionalität anreichern.

Sie können auch Suites zu Suites hinzufügen, Suites also beliebig schachteln.

Nun haben wir gesehen, wie wir unterschiedliche Testklassen einer Suite zuordnen können, um diese Tests in einem Schritt laufen zu lassen. Aber das ist noch nicht alles. Wir erstellen nun eine `MasterSuite`, und die hat es in sich! Die Klasse `MasterSuite` entspricht der Funktionsweise, die wir schon kennen: Sie implementiert eine Methode `suite()`, die eine `TestSuite` zurückliefert. Wir erstellen wieder eine `ActiveTestSuite`, der wir unterschiedliche Testfälle zuweisen. Diesmal erstellen wir Testfälle für die Methoden `testSomeTest2`, `testSomeTest3` und `testSomeTest4`. Daneben fügen wir nun auch noch die bereits bekannte TestSuite `MyTestSuite` hinzu. Dabei übergeben wir der Methode `addTest` den Verweis auf `MyTestSuite.suite()`. Sie sehen, dass man Suites schachteln kann.

Eine Suite kann andere Suites enthalten. Das lässt sich beliebig fortsetzen, so dass wir detailliert unterschiedliche Testszenarien zusammenstellen können. In diesem Fall werden alle der Suite hinzugefügten Testmethoden nebenläufig aufgerufen, der Test der eingebundenen Testsuite sogar mehrfach.



Herkömmliche TestSuites, die nicht mit einem `@RunWith(Suite.class)` ausgezeichnet sind, laufen nur mit Ant! Nutzen Sie sie auch per IDE, müssen Sie derzeit explizit den JUnit 3-Test Runner einsetzen.

Nun werden Sie sicherlich zu Recht anmerken, dass es doch recht umständlich sei, die jeweiligen Methoden einzeln der Suite hinzufügen zu müssen. Möchten wir eine Suite erstellen, die sich genau einer Untermenge aller Testmethoden bedient, ist diese Flexibilität aber hoch einzuschätzen. Es ist auch nicht nötig, wirklich eine Unmenge von Methoden einzeln explizit zu verankern. Unsere `MasterSuite` schließt mit einer weiteren Möglichkeit, der Suite Testfälle hinzuzufügen. Wir erstellen dabei on-the-fly eine weitere TestSuite. Übergeben wir `TestSuite` eine `Class`, werden alle Methoden der Klasse der Suite hinzugefügt. Diese frisch erstellte Suite fügen wir direkt `MasterSuite` hinzu. Auch die zwei Methoden von `MyTestCase2` werden nebenläufig zu sich selbst und zu den anderen Methoden der Suite ausgeführt. Die ganze Suite läuft erfolgreich durch.

```
package com.huettermann.junit;

import junit.extensions.ActiveTestSuite;
import junit.framework.Test;
import junit.framework.TestSuite;

/**
 * @author Michael Huettermann
 */
public class MasterSuite {
```

```
    //entry point, returns my suite
    public static Test suite() {

        //concurrent running suite
        TestSuite suite = new ActiveTestSuite("Test suite");
        //first test case
        MyTestCase1 mtc2 = new MyTestCase1("testSomeTest2");
        suite.addTest(mtc2);
        //next test case
        MyTestCase1 mtc3 = new MyTestCase1("testSomeTest3");
        suite.addTest(mtc3);
        //another test case
        MyTestCase1 mtc4 = new MyTestCase1("testSomeTest4");
        suite.addTest(mtc4);
        //now add a suite to the suite
        suite.addTest(MyTestSuite.suite());
        //now add all tests from one class to the suite
        suite.addTest(new TestSuite(MyTestCase2.class));

        return suite;
    }
}
```

Welche Artefakte sollte ich testen?

Eine für alle Situationen angepasste Anleitung, mit deren Hilfe Sie direkt entscheiden können, was Sie testen müssen, kann Ihnen niemand geben. Die »generischen« Anleitungen müssen mit den aktuellen, individuellen Projektanforderungen abgestimmt werden und sind von Fall zu Fall unterschiedlich. Jedes Projekt ist anders.

Es können einzelne generelle Empfehlungen gegeben werden: Testen Sie die Funktionalität Ihrer Klasse, also die fachlichen *Features* (ob es nun in einer Anwendung ist oder in einem von Ihnen bereitgestelltem Framework). Funktionen, die von externen Bibliotheken bereitgestellt werden (beispielsweise von Java selbst oder einem Framework), müssen die Entwickler nicht mehr selbst testen. Konzentrieren Sie Ihre Tests auf die von Ihnen bereitgestellte Logik.

Testen Sie keine atomaren Schritte. So sind *getter* und *setter* genauso Tabu wie einfache *Delegates*. Auf der anderen Seite können Testfälle durchaus Features testen, die sich über mehrere fachliche Klassen oder Packages erstrecken.

Auch sollten Sie die Tests auf *öffentliche* Features beschränken, also offen verfügbare Funktionalität (*public API*). Interne Funktionalität (*private API*) können Sie optional testen, das ist aber weniger wichtig.

Ein besonderer Anwendungsfall ist das Testen von nebenläufigen Abläufen. Gerade bei derartigen komplexen Szenarien ist es wichtig, die einzelnen Tests zu *isolieren*. Es ist häufig nicht oder nur sehr schwer möglich, ein korrektes nebenläufiges Verhalten zu über-

prüfen. Konzentrieren Sie sich daher auf einzelne *Synchronisationspunkte*, zentrale Stellen, an denen Sie genaue Bedingungen wissen, und isolieren Sie so die Tests.

Bei der Erstellung der Tests werden Sie auch auf die Fragestellung stoßen, ob Sie einen JUnit-Test in einer neuen Testmethode hinterlegen sollten oder nicht. Nehmen wir folgendes Beispiel:

```
public class MyTestCase {
    @Test
    public void testSomething() {
        // Set up for the test, manipulating local variables
        assertTrue(condition1);
        assertTrue(condition2);
        assertTrue(condition3);
    }
}
```

Das entspricht dem folgenden Beispiel:

```
public class MyTestCase {
    // Local variables become instance variables

    @Before
    public void setUp() {
        // Set up for the test, manipulating instance variables
    }

    @Test
    public void testCondition1() {
        assertTrue(condition1);
    }

    @Test
    public void testCondition2() {
        assertTrue(condition2);
    }

    @Test
    public void testCondition3() {
        assertTrue(condition3);
    }
}
```

Inhaltlich sind die beiden Testklassen identisch. Im ersten Fall testen Sie drei Bedingungen in einer Testmethode. In der zweiten Variante existiert für jeden Test eine Methode. Auch wenn es ein »generischer« Tipp ist: Entscheiden Sie selbst. Sie müssen auf keinen Fall für jeden Test eine neue Methode einführen. Gruppieren Sie stattdessen die Tests über die Methoden in einen fachlichen Kontext, der Ihnen einen pragmatischen Umgang mit den Tests ermöglicht und in Ihrem Team die größte Akzeptanz genießt.

Testklassen werden stetig weiterentwickelt. Indem Sie genauso versioniert werden wie fachliche Klassen, ist jederzeit der Zugriff auf einen alten Stand möglich. Ihr Integrationsprozess mit CruiseControl stellt immer neue Versionen Ihrer Software bereit, wobei Sie stets auf alte Stände zugreifen können. Die von Ant ermittelten und von CruiseControl bereitgestellten Testergebnisse bleiben verfügbar. Fügen Sie Testergebnisse nur dann der Versionierung zu, wenn sie nicht anderweitig bereitgestellt werden (zum Beispiel von CruiseControl) und sie darüber hinaus nicht trivial aus den bereits versionierten Artefakten neu generierten werden können. Tests auf demselben Softwarestand müssen schließlich immer dieselben Ergebnisse liefern.

Ein weiterer Punkt, den Sie sich vergegenwärtigen müssen, ist, dass wir hier nur JUnit in seiner originären Ausprägung diskutieren. Es existiert eine Vielzahl von JUnit-Erweiterungen: Frameworks, die auf JUnit aufsetzen und es für spezielle Einsatzgebiete konkretisieren. Zu den Beispielen dafür zählen HTTPUnit zum Testen von Web-Anwendungen, XMLUnit zum Testen von XML-Dokumenten und JUnitPerf für Performance-Tests.

Testautomatisierung und kontinuierliche Integration mit Ant

Sie wissen nun, wie man mit JUnit Komponententests realisieren kann. Nun möchten Sie aber nicht bei jeder Änderung am Quellcode alle Artefakte einzeln neu kompilieren, um sie dann alle Schritt für Schritt manuell erneut mit JUnit-Tests zu testen. Sie benötigen einen agilen Mechanismus, der Ihnen diese manuellen Schritte abnimmt, ein Werkzeug, das für Sie auf Knopfdruck diese Schritte durchführt und schnell Ergebnisse liefert, um direkte Feedbackschleifen zu ermöglichen.

Hier kommt *Ant* ins Spiel. Ant hilft dabei, die einzelnen Schritte zu automatisieren. Im folgenden Kapitel zeige ich Ihnen, wie Sie JUnit automatisiert und agil mit Hilfe eines Ant-Skripts bedienen können. Ant wird nicht nur die Klassen (Test- und fachliche Klassen) alle neu kompilieren (um einen wiederholbaren Ausgangsstand zu haben) und die Tests durchführen (also JUnit bedienen), es wird auch die von JUnit gelieferten Testergebnisse in eine für Menschen lesbare Form bringen und bereitstellen.

Wir gehen im weiteren Verlauf von folgender Verzeichnisstruktur aus: Wir haben zwei unterschiedliche Verzeichnisse unterhalb eines Wurzelverzeichnisses. Zum einen ist das ein Verzeichnis mit den eigentlichen fachlichen Klassen, zum anderen ein Verzeichnis *test*. Wir werden uns hier auf die JUnit-spezifischen Dinge kümmern, deswegen lassen wir die zu testende Fachlichkeit außen vor und konzentrieren uns auf den Testzweig.

Innerhalb des Testzweigs existieren die Verzeichnisse *src*, *classes* und *report* (siehe Abbildung 9-3). Im *src*-Verzeichnis liegen die Testklassen. Ins *classes*-Verzeichnis sollen die kompilierten Testklassen gelegt werden, im *report*-Verzeichnis möchten wir ein Reporting verankern. Es wäre auch denkbar, die Testklassen mit den fachlichen Klassen ins selbe Verzeichnis (also ins selbe Package) zu legen.

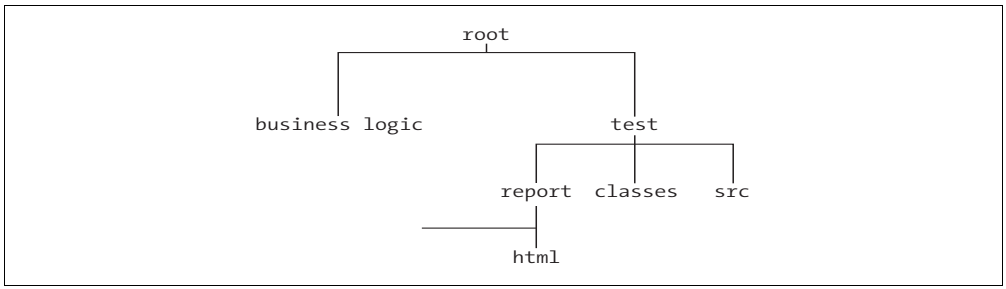


Abbildung 9-3: Das Projektverzeichnis

Der Einfachheit halber gehen wir außerdem davon aus, dass innerhalb des *root*-Verzeichnisses auf gleicher Höhe wie unser *test*-Verzeichnis unsere JUnit-Distribution in einem Verzeichnis *junit* sowie eine Ant-Build-Datei *build.xml* liegt. Im *src*-Verzeichnis liegen vier Testklassen im Package *com.huettermann.junit*, die wir im Lauf des Kapitels bereits kennengelernt haben. Es sind

```
MyTestCase1.java
MyTestCase2.java
MyTestSuite.java
SampleTest.java
```

Um das folgende Beispiel erfolgreich anwenden zu können, benötigen Sie Ant 1.7.0 oder höher. Der `<junit>`-Task von Ant unterstützt bei Versionen kleiner 1.7 nur JUnit kleiner Version 4. Möchten Sie also von den neuesten Features von JUnit profitieren, benötigen Sie eine Ant-Version aus dem 1.7.x-Zweig.

Diese verarbeiten wir mit dem Skript aus Beispiel 9-4.

Beispiel 9-4: Ant-Skript zur Steuerung von JUnit

```
<project name="Agile" default="all">

<property name="test.home" value="/home/michael/junit/test"/>

<target name="all" depends="checkout,clean,compile-src,compile-test,test,report,publish"/>

<target name="clean">
   <delete dir="${test.home}/classes"/>
   <delete dir="${test.home}/report"/>
   <mkdir dir="${test.home}/classes"/>
   <mkdir dir="${test.home}/report"/>
</target>

<target name="checkout"/>

<target name="compile-src"/>

<target name="publish"/>
```

Beispiel 9-4: Ant-Skript zur Steuerung von JUnit (Fortsetzung)

```
<target name="compile-test">
   <javac srcdir="${test.home}/src"
       destdir="${test.home}/classes"
       classpath="${test.home}/junit4.1/junit4.1.jar"
       >
      <include name="**/*.java"/>
   </javac>
</target>

<target name="test">
   <junit printsummary="yes" haltonfailure="no">
      <classpath>
         <pathelement path="${test.home}/classes"/>
      </classpath>

      <formatter type="xml"/>

      <batchtest fork="yes" todir="${test.home}/report">
          <fileset dir="${test.home}/classes">
             <include name="**/*.class"/>
          </fileset>
      </batchtest>
   </junit>
</target>

<target name="report">
   <junitreport todir="${test.home}/report">
     <fileset dir="${test.home}/report">
       <include name="TEST-*.xml"/>
     </fileset>
     <report format="frames" todir="${test.home}/report/html"/>
   </junitreport>
</target>
</project>
```

In dem Skript gehen wir sukzessive die Schritte `checkout`, `clean`, `compile-src`, `compile-test`, `test`, `report` und `publish` durch. Zunächst holen wir uns mit `checkout` die aktuellen Sourcen (fachliche und Testklassen) aus dem Repository. `clean` sichert uns zu, dass jeder Testlauf dieselben Vorbedingungen hat. Es löscht die Verzeichnisse, die während unserer Verarbeitung mit neuen Dateien gefüllt werden. Anschließend kompilieren wir die fachlichen Klassen. Auch das führe ich hier nur an, um einen kompletten Testzyklus zu illustrieren.

Nun kommen wir zum Kern des testspezifischen Teils. Wir kompilieren die Testklassen und legen die `class`-Dateien in das Verzeichnis *classes* in unserem Wurzelverzeichnis. Beachten Sie den Classpath und den Hinweis darauf, dass wir alle Java-Klassen in allen Verzeichnissen (Packages) unterhalb des Ausgangsverzeichnisses *src* kompilieren wollen.

Nach der Ausführung legt Ant die Dateien

```
MyTestCase1.class
MyTestCase2.class
MyTestSuite.class
SampleTest.class
```

in das Verzeichnis

```
classes/com/huettermann/junit/
```

also in das im Ant-Skript spezifizierte *classes*-Verzeichnis unter Berücksichtigung der Package-Struktur der Java-Klassen.

Nun wird das Ant-Target namens `test` ausgeführt. Es besteht aus dem `junit`-Task. Hier ist es wichtig, dass wir das Attribut `printsummary` auf »`yes`« setzen, um anschließend von einem Report profitieren zu können. Wenn `haltonfailure` auf »`no`« gesetzt ist, laufen alle Tests bis zum Ende durch, auch wenn einer fehlschlägt.

```
<junit printsummary="yes" haltonfailure="no">
```

Wir setzen den Classpath für JUnit auf unser *classes*-Verzeichnis.

```
<classpath>
   <pathelement path="${test.home}/classes"/>
</classpath>
```

Anschließend definieren wir einen Formatter. Hier haben wir verschiedene Möglichkeiten und entscheiden uns für »xml«. Die Testergebnisse sollen also in Dateien im XML-Format ausgegeben werden.

```
<formatter type="xml"/>
```

Der letzte innerhalb des `junit`-Tasks geschachtelte Task ist `batchtest`. Wir können mit Ant einzelne Testklassen aufrufen oder auch einen ganzen Stapel Testklassen in einem Rutsch verarbeiten. Das ist gerade vor dem Hintergrund einer Automatisierung und vollständigen Ausführung aller Tests sehr hilfreich. `fork` führt dabei die Verarbeitung außerhalb des Ant-Prozesses durch. Die generierten Testergebnisse sollen in das *report*-Verzeichnis gelegt werden. Wir berücksichtigen beim Testlauf alle Dateien mit der Endung `.class` in allen Unterverzeichnissen rekursiv.

```
<batchtest fork="yes" todir="${test.home}/report">
    <fileset dir="${test.home}/classes">
       <include name="**/*.class"/>
    </fileset>
</batchtest>
```

Nach diesem Task liegen im Verzeichnis *report* die Dateien

```
TEST-com.huettermann.junit.MyTestCase1.xml
TEST-com.huettermann.junit.MyTestCase2.xml
TEST-com.huettermann.junit.MyTestSuite.xml
TEST-com.huettermann.junit.SampleTest.xml
TESTS-TestSuites.xml
```

also für jede Testklasse eine XML-Ergebnisdatei der Struktur `TEST-`***`package.Name`***`.xml`. Für `SampleTest` sieht es so aus, dass wir zunächst verschiedene Properties aufgelistet bekommen und anschließend die Testfälle. Ein Auszug des Listings zeigt Beispiel 9-5.

Beispiel 9-5: Die Ausgabe

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuite errors="0" failures="0" hostname="BlueGene" name="com.huettermann.junit.SampleTest
" tests="2" time="0.073" timestamp="2006-12-09T11:33:38">
  <properties>
    <property name="java.runtime.name" value="Java(TM) SE Runtime Environment" />
    ...
  </properties>
  <testcase classname="com.huettermann.junit.SampleTest" name="testEmptyCollection" time="0.0
050" />
  <testcase classname="com.huettermann.junit.SampleTest" name="testOneItemCollection" time="0
.0010" />
  <system-out><![CDATA[]]></system-out>
  <system-err><![CDATA[]]></system-err>
</testsuite>
```

Die Datei

```
TESTS-TestSuites.xml
```

fasst alle einzelnen Testklassen zusammen. Die Struktur sehen Sie im Beispiel 9-6.

Beispiel 9-6: Struktur von TestSuites.xml

```
TESTS-TestSuites.xml
<?xml version="1.0" encoding="UTF-8" ?>
<testsuites>
  <testsuite errors="0" failures="0" hostname="BlueGene" id="0" name="MyTestCase1" package=
"com.huettermann.junit" tests="4" time="0.034" timestamp="2006-12-09T11:33:37">
...
  </testsuite>
  <testsuite errors="0" failures="0" hostname="BlueGene" id="1" name="MyTestCase2" package=
"com.huettermann.junit" tests="2" time="0.077" timestamp="2006-12-09T11:33:37">
...
  </testsuite>
  <testsuite errors="0" failures="0" hostname="BlueGene" id="2" name="MyTestSuite" package=
"com.huettermann.junit" tests="5" time="0.044" timestamp="2006-12-09T11:33:37">
...
  </testsuite>
  <testsuite errors="0" failures="0" hostname="BlueGene" id="3" name="SampleTest" package=
"com.huettermann.junit" tests="2" time="0.073" timestamp="2006-12-09T11:33:38">
...
  </testsuite>
</testsuites>
```

Hier fällt insbesondere ins Auge, dass die einzelnen TestSuite-Einträge über Anzahl der Tests und Erfolg oder Misserfolg Aufschluss geben.

Das nächste Target in unserem Ant-Skript ist `report`. Hier erstellen wir mit Hilfe des Tasks `junitreport` einen Testreport im HTML-Format. Dazu nehmen wir alle generierten XML-Dateien aus dem Verzeichnis *report* als Grundlage. Der Report wird in Form von HTML-Dateien mit HTML-Frames dargestellt. Die Dateien werden ins *html*-Verzeichnis innerhalb von *report* gelegt.

```
<target name="report">
   <junitreport todir="${test.home}/report">
     <fileset dir="${test.home}/report">
       <include name="TEST-*.xml"/>
     </fileset>
     <report format="frames" todir="${test.home}/report/html"/>
   </junitreport>
</target>
</project>
```

Nun sind wir fertig. Ein abschließender Ant-Target könnte uns die Ergebnisse aus dem Arbeitsbereich wegkopieren oder in eine Projektseite schieben.

Starten wir nun auf gleicher Höhe des Verzeichnisses *test* unser *build*-Skript mit

```
ant
```

so ergibt sich folgende Konsolenausgabe:

```
michael@BlueGene:~/junit$ ant
Buildfile: build.xml

clean:
   [delete] Deleting directory /home/michael/junit/test/classes
   [delete] Deleting directory /home/michael/junit/test/report
    [mkdir] Created dir: /home/michael/junit/test/classes
    [mkdir] Created dir: /home/michael/junit/test/report

compile-src:

compile-test:
    [javac] Compiling 4 source files to /home/michael/junit/test/classes

test:
    [junit] Running com.huettermann.junit.MyTestCase1
    [junit] Tests run: 4, Failures: 0, Errors: 0, Time elapsed: 0,172 sec
    [junit] Running com.huettermann.junit.MyTestCase2
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,064 sec
    [junit] Running com.huettermann.junit.MyTestSuite
    [junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 0,062 sec
    [junit] Running com.huettermann.junit.SampleTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,055 sec

report:
[junitreport] Processing /home/michael/junit/test/report/TESTS-
TestSuites.xml to /tmp/null1704248722
[junitreport] Loading stylesheet jar:file:/home/michael/ant/apache-ant-1.7.0RC1/lib/ant-
junit.jar!/org/apache/tools/ant/taskdefs/optional/junit/xsl/junit-frames.xsl
[junitreport] Transform time: 1567ms
```

```
[junitreport] Deleting: /tmp/null1704248722

all:

BUILD SUCCESSFUL
Total time: 7 seconds
michael@BlueGene:~/junit$
```

Wir sehen, wie die einzelnen Schritte erfolgreich durchlaufen werden. Insbesondere der Testbereich ist interessant: Alle Tests laufen erfolgreich durch. Während der Erstellung des Reports kommt ein XSL-Stylesheet zum Einsatz, das die XML-Testergebnisse in HTML transformiert.

Starten wir nun *index.html* innerhalb des Verzeichnisses *report/html*, sehen wir eine schöne visuelle Übersicht über die Testergebnisse. Als Einstiegsseite sehen wir im Frame links oben sämtliche Packages (der Testklassen), unten links die Klassen und rechts eine Zusammenfassung. In unserem Fall sind 13 Tests erfolgreich durchlaufen worden (siehe Abbildung 9-4).

Gehen wir nun auf eine einzelne Testklasse wie MYTESTCASE1, sehen wir weitere Details: Alle Testmethoden werden mit einer Information über Erfolg bzw. Misserfolg ausgegeben (siehe Abbildung 9-5). In dieser Übersicht sehen wir unten rechts einen Link auf PROPERTIES. Hier können wir weitere technische Ausführungsdetails erfahren, die wir zuvor schon innerhalb des XML-Dokuments gesehen haben. Ein Link auf *System.err*. zeigt uns die Ausgaben, die JUnit zum *System.err*.-Stream geschickt hat.

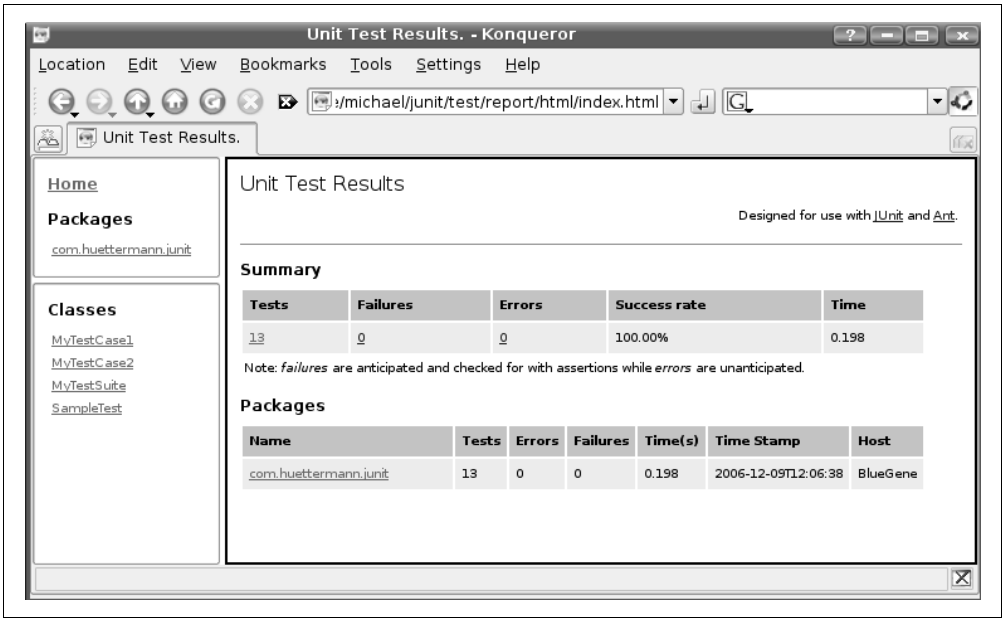


Abbildung 9-4: Testergebnisübersicht

Schließlich ist es sehr interessant, von der Startseite direkt auf die Zahl der erfolgreichen (oder nicht erfolgreichen) Tests zu klicken. Klicken wir hier auf die 13, wird eine Seite geöffnet, die alle Tests darstellt (siehe Abbildung 9-6).

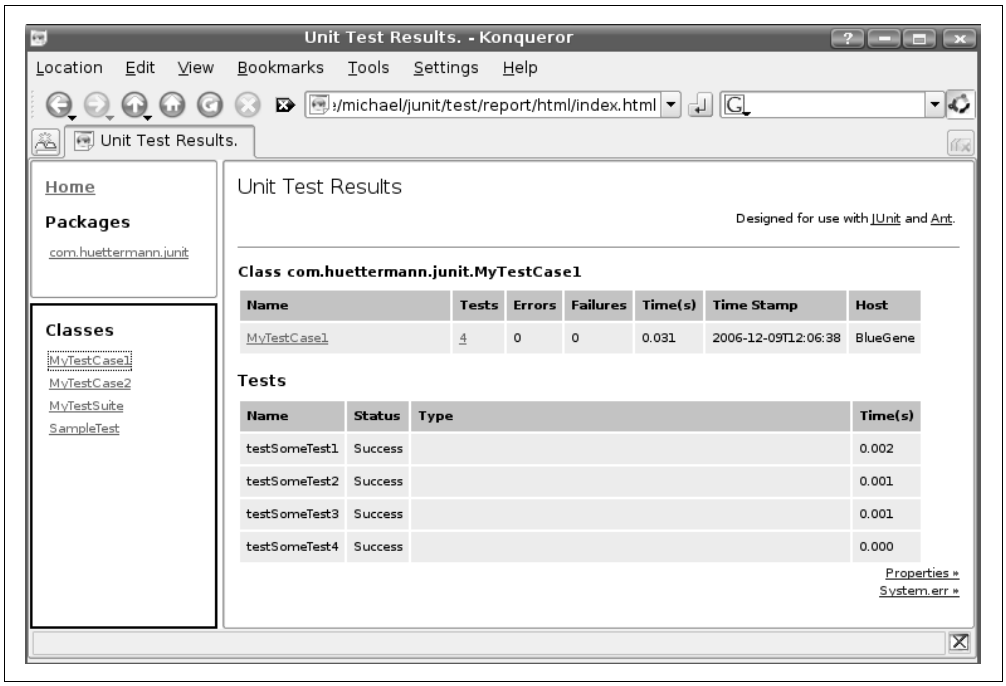


Abbildung 9-5: Testergebnis bezüglich der Methoden

Test-Driven Development mit JUnit

Test-driven Development hilft dabei, lose gekoppelte, aber geschlossene Systeme zu entwickeln, die wenige Defekte haben und über ein gutes Design verfügen. TDD verkürzt die Feedback-Schleifen bei der Entwicklung drastisch. Der Entwickler erhält konstant Rückkopplungen über Qualität und Funktionalität des Codes. Die Testklassen treiben die funktionalen Klassen vor sich her: Sie sind die Spezifikation, die dem Entwickler Sicherheit bietet. Sie fühlen sich sicher, dass das, was Sie heute entwickelt haben, nicht die Entwicklungen von gestern negativ beeinflusst. Das Design der Software wird evolutionär erstellt: Sie lernen während der Entwicklung hinzu, so dass Sie, um die Anforderung in einer einfachen Form umzusetzen, ihren Quellcode kontinuierlich refaktorisieren. Das können Sie aber nur guten Gewissens machen, wenn Sie zu all Ihren Klassen Unit-Tests besitzen, denn sonst können Sie nicht wissen, ob die neue Version der Software (nach der Refaktorisierung oder auch nach der Erweiterung an der Funktionalität) noch immer das Gewünschte leistet. Wir haben uns schon im ersten Teil des Buchs kurz mit der testgetriebenen Entwicklung beschäftigt, wollen aber nun im Zusammenhang mit Unit-Tests noch einmal eingehender die Vorgehensweise beleuchten.

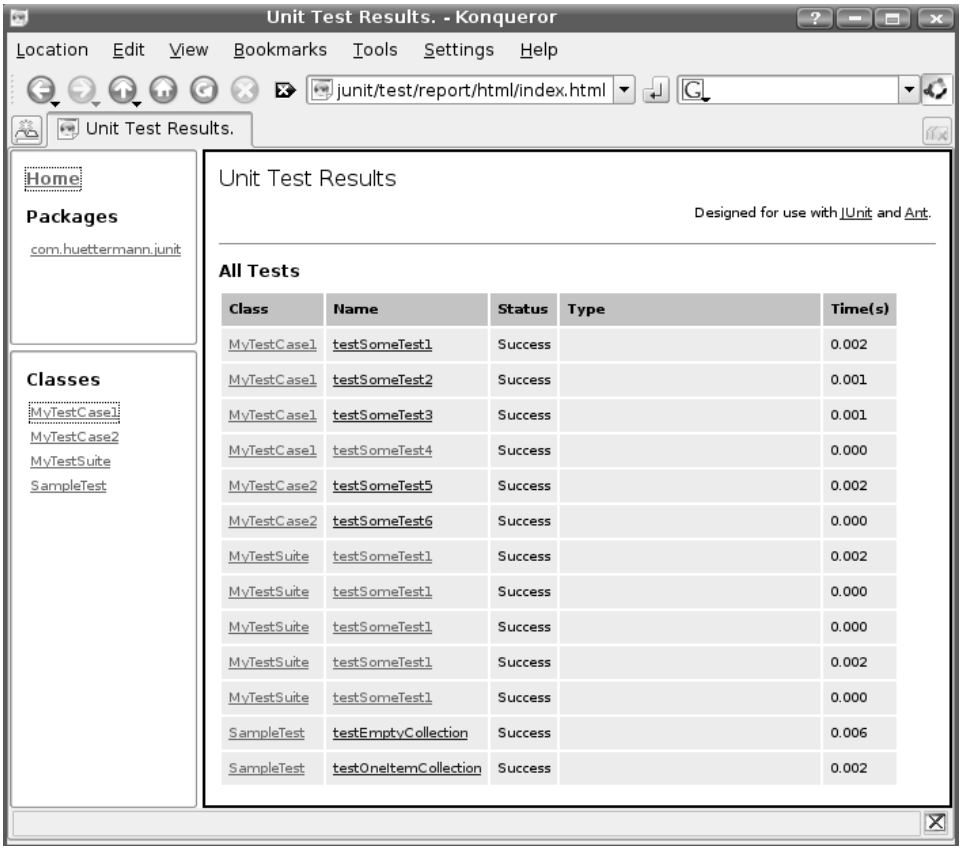
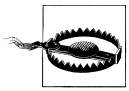


Abbildung 9-6: Testergebnis für alle Tests

Hohe Kohäsion bei niedriger Kopplung

Eine Klasse verfügt über eine hohe Kohäsion, wenn sie über eine funktionale Verantwortung in einem Bereich verfügt und mit anderen Klassen nur zusammenarbeitet, um ihre Aufgaben zu erfüllen. Die fachliche Funktionalität ist in der Klasse zentral angesiedelt, sie hat die Verantwortung für diese Funktionalität. Gleichzeitig ist sie von anderen Klassen entkoppelt: Änderungen in anderen Komponenten haben keinen Einfluss auf eine solche Klasse. Dadurch ist die entkoppelte Klasse isoliert betrachtbar und einfach zu verstehen und zu nutzen. Ein gutes Design zeichnet sich also durch hohe Kohäsion und niedrige Kopplung aus, und Sie sollten versuchen, beim Durchlaufen Ihres Entwicklungszyklus den Kohäsionsgrad immer weiter zu erhöhen.



Der Begriff »Test-driven Development« ist nicht eindeutig definiert. Gewöhnlich wird er mit »Test-First« gleichgesetzt bzw. setzt darauf auf. So umfasst TDD einerseits die Entwicklung von Tests als Spezifikation und ihre kontinuierliche Ausführung, andererseits die Realisierung von Tests vor der bzw. parallel zur Entwicklung von fachlichen Klassen (Test-First). Ein nachgelagertes Testen nach Test-Last kann die Entwicklung nicht treiben und das bestehende Design nur überprüfen und nicht nachhaltig prägen.

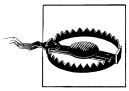
Zerlegen wir den Begriff TDD, kommen vor zu folgendem Ergebnis:

1. Development: Agile Entwicklung konzentriert die reine Entwicklung nicht auf die Implementierung. Das schwergewichtige Muster ist aufgebrochen. Während der Entwicklung analysieren Sie auch, designen, testen und integrieren.
2. Driven: Die Entwicklung wird von den Tests *geführt*, so dass der Begriff »Test-First Development« noch genauer wäre. Kent Beck fragt zu Recht, womit man die Entwicklung steuert, wenn nicht mit Tests.
3. Test: Automatisierte, auf Knopfdruck wiederholbare Tests. TDD ist keine Testing-Technik, es ist ein umfassendes Vorgehen, bei dem man analysiert, designt und alle Aktivitäten der Entwicklung strukturiert.

Beispielhaftes Vorgehen

TDD besteht aus folgenden Arbeitsschritten, die wir an einem Beispiel illustrieren:

1. Einen Test hinzufügen.
2. Alle Tests laufen lassen und überprüfen, dass der neue fehlschlägt. Der neue Test *muss* fehlschlagen, da Sie einen Test für eine Funktion geschrieben haben, die noch nicht existiert.
3. Eine kleine Änderung machen.
4. Alle Tests laufen lassen und überprüfen, ob nun alle erfolgreich durchlaufen.
5. Refactoring.
6. Alle Tests laufen lassen und überprüfen, ob alle erfolgreich durchlaufen.



Das Beispiel dient dem Verständnis von TDD und verzichtet auf unnötige fachliche Komplexität. In der Praxis ist der Test von reinen Getter-Methoden oder einfachen Konstruktoren nicht notwendig.

Einen Test hinzufügen

Wir möchten eine neue Funktionalität bereitstellen: Agile Ausprägungen (wie XP oder Scrum) sollen als Objekte angelegt und verwaltet werden. Wir erzeugen zunächst die Testklasse. Obwohl es nicht zwingend notwendig ist, ist es gut, die Testklassen mit »Test« enden zu lassen. So haben Sie direkt den Überblick, welche Klasse eine Testklasse ist und welche nicht.

```
package com.huettermann.junit;

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

/**
 * @author Michael Huettermann
 */
public class AgileTest {

    @Test
    public void agileName() {
        Agile agile = new Agile("XP");
        assertEquals("XP", agile.getName());
    }
}
```

Fangen Sie neu mit der Klasse an, existiert die Klasse `Agile` noch nicht, und Ihre IDE wird einen Fehler melden. Sie können die Klasse nicht kompilieren. Wir werden sie nun direkt in einer minimalen Form anlegen. IDEA bietet Ihnen ein *Quick Fix* an, eine Hilfe, mit der Sie direkt die Klasse erstellen können. Wir nennen die neue Klasse `Agile` und legen sie der Einfachheit halber ins selbe Verzeichnis.

```
package com.huettermann.junit;

/**
 * @author Michael Huettermann
 */
public class Agile {
}
```

Das reicht bereits, damit die Testklasse die zu testende Klasse erkennt. Nun erweitern wir diesen bloßen Klassenrumpf, damit die Testklasse kompiliert. Dazu ist ein Konstruktor notwendig und die Ergänzung der Methode `getName()`. Die rudimentäre Implementierung sieht wie folgt aus:

```
package com.huettermann.junit;
/**
 * @author Michael Huettermann
 */
public class Agile {

    /**
     * the name
     *
     * @param name der Name
     */
    public Agile(String name) {
    }

    /**
     * get name
     *
     * @return der Name
```

```
     */
    public String getName() {
        return null;
    }
}
```

Nun haben wir einen kompilierbaren Testfall angelegt, begleitet von der entsprechenden fachlichen Klasse in einer minimalen Form.

Alle Tests laufen lassen und überprüfen, dass der neue fehlschlägt

Nun werden wir die Tests laufen lassen. Wir haben derzeit nur einen, der bei Aufruf per IDE folgendes Ergebnis liefert:

```
java.lang.AssertionError: expected:<XP> but was:<null>
    at org.junit.Assert.fail(Assert.java:69)
    at org.junit.Assert.failNotEquals(Assert.java:314)
....
```

Dies verwundert nicht, da wir den Wert »XP« erwarten, aber null zurückbekommen.

Eine kleine Änderung machen

Der nächste Schritt hat den Zweck, die fachliche Klasse mit einer kleinen Änderung so bereitzustellen, dass der Test erfolgreich durchläuft.

```
package com.huettermann.junit;

/**
 * @author Michael Huettermann
 */
public class Agile {

    /**
     * the name
     *
     * @param name myName
     */
    public Agile(String name) {
    }

    /**
     * get name
     *
     * @return my name
     */
    public String getName() {
        return "XP";
    }
}
```

Alle Tests laufen lassen und überprüfen, ob alle erfolgreich durchlaufen

Durch ein bloßes Zurückgeben des Strings *XP* haben wir es geschafft: Der Test läuft erfolgreich durch. Damit haben wir die Anforderung umgesetzt. Sie sehen, dass das Design sehr schlank ist. Die Klasse leistet nur das, was Sie zur Erfüllung der Anforderung leisten soll. Aber sie leistet die Funktionalität hart verdrahtet, so dass wir noch nicht ganz fertig sind.

Refactoring

Die Klasse `Agile` soll natürlich nicht für jeden Fall den Wert `XP` als Namen zurückgeben. Der hart verdrahtete Wert muss refaktorisiert werden. Dazu bietet jede gängige IDE eine Hilfe an. Zunächst migrieren wir den harten Wert `XP` in eine lokale Variable.

```
package com.huettermann.junit;

/**
 * @author Michael Huettermann
 */
public class Agile {

    /**
     * the name
     *
     * @param name myName
     */
    public Agile(String name) {
    }

    /**
     * get name
     *
     * @return my name
     */
    public String getName() {
        String name = "XP";
        return name;
    }
}
```

Bei IDEA ist das mit *Introduce Variable* möglich (siehe Abbildung 9-7) – der String *XP* inklusive Anführungszeichen muss markiert sein, wenn die IDEA-Funktionalität aufgerufen wird. Im folgenden Dialog geben wir an, dass die neue Variable »name« heißen soll. Anschließend bietet ein Quick Fix (Assistent auf der linken Seite im Editor) die Möglichkeit, aus dem Parameter *name* des Konstruktors ein Attribut der Klasse zu machen und den übergebenen Wert dem Attribut zuzuweisen (siehe Abbildung 9-8). Ein Assistent fragt nach unserer Bestätigung, ob das zu erstellende Feld wirklich »name« heißen und final sein soll. Wir bestätigen.

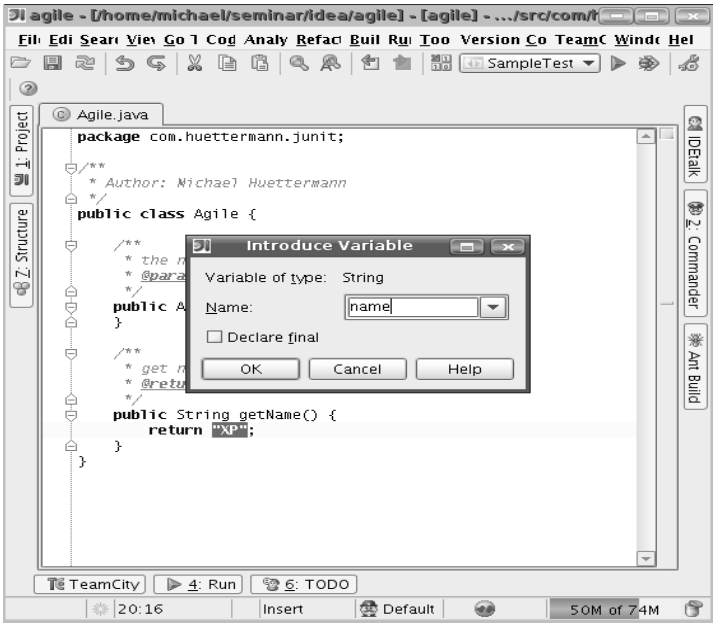


Abbildung 9-7: IDEA Refactoring: Introduce Variable

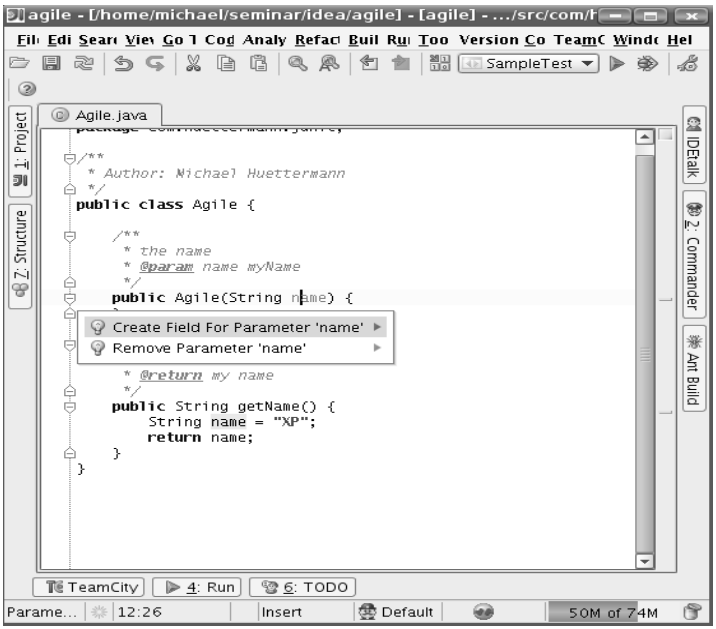


Abbildung 9-8: IDEA Refactoring: Aus einem Parameter ein Feld erstellen

```
package com.huettermann.junit;

/**
 * @author Michael Huettermann
 */
public class Agile {
    private final String name;

    /**
     * the name
     *
     * @param name myName
     */
    public Agile(String name) {
        this.name = name;
    }

    /**
     * get name
     *
     * @return my name
     */
    public String getName() {
        String name = "XP";
        return name;
    }
}
```

Anschließend entfernen wir einfach die Deklaration und Zuweisung der lokalen Variable `name`.

```
package com.huettermann.junit;

/**
 * @author Michael Huettermann
 */
public class Agile {
    private final String name;

    /**
     * the name
     *
     * @param name myName
     */
    public Agile(String name) {
        this.name = name;
    }

    /**
     * get name
     *
     * @return my name
     */
```

```
    public String getName() {
        return name;
    }
}
```

Nun haben wir in wenigen kleinen Schritten die Klasse so refaktorisiert, dass sie vom Design her ansprechend ist. Ziel ist es also, in diesem Schritt Redundanzen zu eliminieren und Abhängigkeiten aufzulösen. Redundanzen kommen direkt zu Stande, wenn hart verdrahtete Werte eingeführt werden. Abhängigkeiten zwischen Variablen, Methoden und Klassen sollen ebenfalls direkt aufgelöst werden. In unserem Fall gab es Abhängigkeiten zwischen den Namen.

Alle Tests laufen lassen und überprüfen, ob alle erfolgreich durchlaufen

Spätestens nach Abschluss unseres kleinen Refactorings lassen wir alle Tests nochmal laufen. Sie werden schnell ein eigenes Gefühl dafür gewinnen, wann genau Sie die Tests laufen lassen müssen. Es wird – anders als in unserem Beispiel – nicht nach jedem kleinen Refactoring notwendig sein, vor allem auch, weil die IDE ein klassenübergreifendes Refactoring gewährleistet, das die Klassen im neuen Zustand kompilieren lässt. Ist Ihnen hier ein Fehler unterlaufen, dann werden Sie dies anhand Ihrer Tests sehr schnell merken. Sie können die letzten Schritte, die Sie in der IDE durchgeführt haben, rückgängig machen.

Vorteile

TDD bietet folgende Vorteile:

Diskussionsgrundlage
: Testklassen bieten eine Diskussionsgrundlage, um sich mit einem Kollegen in einer Pair Programming-Sitzung über den Code zu unterhalten.

Isolierte Arbeit
: Durch detaillierte Aufteilung der Arbeit und ihren dedizierten Abschluss ist die Arbeit weniger stressig: Erstellen Sie einen Test, und die Funktionalität dazu läuft wie erwartet, dann können Sie Ihre Arbeit für den Tag mit diesem Meilenstein beenden und am nächsten Tag ausgeruht an gleicher Stelle fortführen.

Continuous Integration
: Erst durch TDD und die Testklassen haben Sie die Möglichkeit, kontinuierlich zu integrieren. Die Tests signalisieren hier, ob die Integration fehlgeschlagen ist oder nicht.

Schlankes Design
: Sie implementieren nur genau das, was Sie benötigen, um den Test erfolgreich zu bestehen. Auf diese Weise ist das Design immer so schlank wie möglich und dabei so komplex wie nötig, und es erfüllt genau die Anforderungen (»Just-in-Time-Design«).

Sicherheit
: Wenn Sie Änderungen am Code machen, geben Ihnen die Tests die Sicherheit, dass der Test unverändert das macht, was Sie erwarten. Das gibt die notwendige Sicherheit, um weitere Refactorings zu machen, um das Leben Ihrer Komponente zu verlängern.

Geschwindigkeit
: Wenn Tests und Refaktorisierungen Ihnen große Sicherheit geben, können Sie Code viel schneller in die Produktivumgebung verteilen und dem Kunden bereitstellen.

Testarten
: Es können mit TDD verschiedene Arten von Tests erstellt werden: Neben Komponenten- und Regressionstests spielen auch Tests im Sinn von »Learning Tests« (Testen, um die Fachlichkeit zu verstehen) und »Explanation Tests« (Testen, um Fachlichkeit zu erklären) eine große Rolle.

Zusammenfassung

JUnit ist das Standardwerkzeug bei der Durchführung von Komponententests. Die etablierten, mit JUnit 3.x eingeführten Methoden wurden in JUnit 4.x um neue Konzepte erweitert. Einzelne Tests lassen sich in TestSuites aggregieren und in Ihren Build-Prozess integrieren. Unit-Tests mit JUnit geben Ihnen direkt Feedback über Ihre Klassen. Denken Sie über »Corner Cases« nach, Grenzwerte, die Sie testen können und so Ihren Code robuster machen. Das Design der Klassen wird durch Ihre JUnit Tests nicht nur verbessert, Sie lernen auch während der Entwicklung mehr über das Verhalten, das Sie von der fachlichen Klasse erwarten. Darüber hinaus bekommen Sie bei umfassender Testabdeckung und erfolgreichen Tests ein gutes Gefühl für Änderungen und Erweiterungen. Sie wissen, dass der erfolgreich getestete Code nach Ihren Erwartungen funktioniert. Tests sind die verlässliche Dokumentation und Spezifikation Ihrer fachlichen Klassen.

Bei der testgetriebenen Entwicklung treiben die Tests die Entwicklung der fachlichen Klassen vor sich her. Änderungen am Test und an der fachlichen Klasse werden iterativ durchgeführt. Das führt zu einem guten, schlanken Design und der Gewissheit, dass Sie tatsächlich auch alles testen, was Sie entwickeln.

KAPITEL 10

Testabdeckung mit EMMA und JUnit

Sie haben Ihre Features realisiert und Tests dazu erstellt. Sie haben viele verschiedene Tests, die Ihre fachliche Funktionalität testen. Aber wie wollen Sie wirklich wissen, wie viel von Ihrer Fachlichkeit durch Tests abgedeckt ist?

Hier kommen sogenannte *Coverage*-Werkzeuge ins Spiel. Ein Werkzeug, das in diese Kategorie fällt, ist *EMMA*. Es kann sowohl nicht genutzten als auch nicht getesteten Code identifizieren. EMMA zeichnet sich aus durch

- umfangreiches Reporting ohne signifikanten Overhead bei der Ausführung,
- Integration in den Build-Prozess undUnterstützung des Testens sowohl von einzelnen Komponenten als auch von ganzen, umfangreichen Applikationen.

EMMA instrumentalisiert den Code (reichert den Bytecode mit eigenen Informationen an), analysiert ihn anschließend und erstellt einen Report. Die eigentliche Nutzung ist unterschiedlich möglich. Die EMMA-Instrumentalisierung kann einerseits »offline« geschehen, das heißt die Klassen werden von der Platte gelesen und anschließend instrumentalisiert auf Platte zurückgeschrieben. Der »On-the-fly«-Modus von EMMA andererseits spart sich den Umweg, die angereicherten Klassen auf Platte zu schreiben. Seine Nutzung und die Erstellung geschehen zur Laufzeit in einem Schritt innerhalb der Java Virtual Machine.

Im diesem Kapitel beschreiben wir die Nutzung von EMMA anhand eines kompakten Beispiels, das die grundlegenden Szenarien zeigt und auf mögliche Herausforderungen eingeht. Ferner werden wir EMMA in Ant integrieren.

Installation und Beispiel

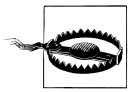
Wir laden uns die frei verfügbare EMMA-Distribution von *http://emma.sourceforge.net/* herunter und entpacken sie in ein Verzeichnis, das wir in unserem Fall *emma* nennen.

Bytecode-Instrumentalisierung

Bei der Bytecode-Instrumentalisierung nimmt EMMA die bestehenden Klassen (also die *class*-Dateien), fügt ihnen eigene Metainformationen hinzu und generiert anschließend neue *class*-Dateien. EMMA ist ein reines Java-basiertes Coverage-Werkzeug. Das heißt, EMMA setzt keine Java Virtual Machine-Einstellungen voraus. Nachdem Sie EMMA als Laufzeitabhängigkeit Ihren Sourcen hinzugefügt haben, können Sie diese auf herkömmlichem Weg laufen lassen. Insbesondere werden dabei die zugrundeliegenden Java-Quelldateien nicht verändert (was beispielsweise das Werkzeug Clover macht).

Jede mit `@Test` annotierte Methode wird von JUnit durchgeführt. Sicher, wir könnten so die korrekte Arbeit von JUnit und die Testskripten selbst weiter überprüfen, deutlich interessanter ist es allerdings, die fachlichen Klassen zu betrachten. Welche Klassen und Methoden werden von den Tests angestoßen und durchlaufen? Was wird getestet?

Zur Illustration der EMMA-Funktionalität setzen wir ein einfaches Szenario auf, anhand dessen wir alle wesentlichen Punkte diskutieren und beliebig komplexere Beispiele herleiten können. Wir gehen von einer fachlichen Klasse `Agile.java` aus.



Ein reines Value-Objekt, insbesondere seine Getter-Methode, würden Sie in der Praxis nicht testen. Dieses Szenario ist ein kompaktes Beispiel, das in die wesentlichen Punkte der Codeabdeckung einführt und auf übliche Probleme aufmerksam macht. Dabei wird bewusst auf die Einführung von umfangreicher Geschäftslogik verzichtet, da diese am Konzept nichts ändert, sondern nur das Verständnis erschwert. Zur Implementierung von `equals`-Methoden, wenn Sie diese wirklich überschreiben müssen, bietet die IDE gewöhnlich Hilfen an. Aber: Unterziehen Sie Ihr Projekt einer Code-Coverage-Untersuchung, werden Sie auch Klassen wie die zwei hier vorgestellten verarbeitet bekommen und müssen das Ergebnis richtig zu interpretieren verstehen.

```
package com.huettermann.junit;

/**
 * @author Michael Huettermann
 **/
public class Agile {
   private final String name;

   /**
    * the name
    * @param name myName
    **/
   public Agile(String name) {
      this.name = name;
   }
```

```
    /**
     * get the name
     * @return name the name
     **/
    public String getName() {
       return name;
    }

    @Override
    public boolean equals(Object o) {
       if (this == o) return true;
       if (o == null || getClass() != o.getClass()) return false;
       Agile agile = (Agile) o;
       if (!name.equals(agile.name)) return false;
       return true;
    }

    @Override
    public int hashCode() {
       return name.hashCode();
    }
}
```

Wie Sie sehen, ist `Agile.java` im Wesentlichen ein *Value Object*. Es speichert einen im Konstruktor übergebenen Wert in einem Attribut. Wir möchten `equals` der Objektklasse überschreiben. Da es dann auch geboten ist, die `hashCode`-Methode zu überschreiben, tun wir das auch. Wir haben also mit einem einfachen Fall wesentliche Szenarien aufgesetzt: Einerseits haben wir eine (in der Praxis natürlich mehrere) Methoden, die wir ausimplementieren und mit Logik füllen. Hier kommen beispielsweise auch Alternationen vor (`if`-Statements). Andererseits gibt es die Konvention, eine Methode zu überschreiben (`hashCode`).

Nun haben wir eine Testklasse `AgileTest`, die exemplarisch genau diese von uns implementierte Methode testet (die `equals`-Methode). Daneben testen wir noch die Erstellung des Objekts. In echten Projekten wäre dies sicher grenzwertig. Haben Sie im Konstruktor Ihrer Klasse Logik hinterlegt, dann testen Sie, ob das Objekt »richtig« und unter den jeweiligen Umständen mit allen Abhängigkeiten instanziiert wurde.

```
package com.huettermann.junit;

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

/**
 * @author Michael Huettermann
 **/
public class AgileTest {

    @Test
    public void agileName() {
        Agile agile = new Agile("XP");
        assertEquals("XP", agile.getName());
    }
```

```
        @Test
        public void agileEquals() {
            Agile agile1 = new Agile("XP");
            Agile agile2 = new Agile("XP");
            assertTrue(agile1.equals(agile2));
        }
    }
```

Unsere beiden Java-Dateien liegen im Verzeichnis *src/com/huettermann/junit*.

Zunächst testen wir, ob das Value Object korrekt angelegt wurde. Wir legen es an, und überprüfen anschließend, ob sein `name`-Attribut korrekt gefüllt ist und die Methode `getName()` auch den richtigen Wert liefert. Der zweite Test überprüft, ob zwei unterschiedliche `Agile`-Instanzen, die den gleichen Wert aufgenommen haben, identisch (`equals`) sind. Wir haben die `equals`-Methode überschrieben, dies ist also ein Test dieser Methode (den wir auch mit `assertEquals` durchführen könnten).

Welche Ergebnisse erwarten wir?

Automatisierte Nutzung via Ant

Ein typisches Ant-Skript, das EMMA nutzt, ist in Beispiel 10-1 mit Ant dargestellt.

Beispiel 10-1: EMMA mit Ant

```
<project name="Agile" default="all">
<property name="root" value="/home/michael/junit"/>
<property name="test.home" value="${root}/test"/>

<property name="emma.dir" value="${root}/emma/lib"/>
<path id="emma.lib">
   <pathelement location="${emma.dir}/emma.jar"/>
   <pathelement location="${emma.dir}/emma_ant.jar"/>
</path>
<taskdef resource="emma_ant.properties" classpathref="emma.lib"/>

<target name="all" depends="clean,compile-src,compile-test,instr,test,report-instr,report"/>

<target name="clean">
   <delete dir="${test.home}/classes"/>
   <delete dir="${test.home}/report"/>
   <delete dir="${test.home}/coverage"/>
   <mkdir dir="${test.home}/classes"/>
   <mkdir dir="${test.home}/report"/>
   <mkdir dir="${test.home}/coverage"/>
</target>
<target name="compile-src"/>
<target name="compile-test">
<javac srcdir="${test.home}/src"
       destdir="${test.home}/classes"
       classpath="${test.home}/junit4.1/junit4.1.jar"
       debug="true"
       >
   <include name="**/*.java"/>
```

Beispiel 10-1: EMMA mit Ant (Fortsetzung)

```
</javac>
</target>

<target name="instr">
   <emma>
      <instr instrpath="${test.home}/classes"
          outdir="${test.home}/classes"
          metadatafile="${test.home}/coverage/metadata.emma"
          mode="overwrite" merge="true"/>
   </emma>
</target>

<target name="report-instr">
   <emma>
      <report sourcepath="${test.home}/src" verbosity="info" >
          <fileset dir="${test.home}/coverage">
             <include name="*.emma"/>

          </fileset>
          <html outfile="${test.home}/coverage/index.html"
            columns="name,class,method,line,block"/>
      </report>
   </emma>
</target>

<target name="test">
   <junit printsummary="yes" fork="yes" haltonfailure="no">
      <classpath path="${test.home}/classes">
         <pathelement path="${test.home}"/>
         <path refid="emma.lib"/>
      </classpath>

      <jvmarg value="-Demma.coverage.out.file=${test.home}/coverage/coverage.emma" />
      <jvmarg value="-Demma.coverage.out.merge=true" />

      <formatter type="xml"/>

      <batchtest fork="yes" todir="${test.home}/report">
          <fileset dir="${test.home}/classes">
                       <include name="**/*Test*.class"/>
          </fileset>
      </batchtest>
   </junit>
</target>
<target name="report">
   <junitreport todir="${test.home}/report">
     <fileset dir="${test.home}/report">
       <include name="TEST-*.xml"/>
     </fileset>
     <report format="frames" todir="${test.home}/report/html"/>
   </junitreport>
</target>
</project>
```

Zwei Schritte im Build-Prozess sind für EMMA ausschlaggebend. Das ist zum einen das `instr`-Target, zum anderen das `report-instr`-Target. Daneben fällt auf, dass wir ein Verzeichnis *coverage* einsetzen.

Bevor wir die JUnit Tests im Target `test` durchführen, müssen wir die Klassen instrumentalisieren. Das bedeutet, dass dem fachlichen Quellcode vom EMMA Metainformationen hinzugefügt werden.

```
<target name="instr">
   <emma>
      <instr instrpath="${test.home}/classes"
          outdir="${test.home}/classes"
          metadatafile="${test.home}/coverage/metadata.emma"
          mode="overwrite" merge="true"/>
   </emma>
</target>
```

Die Instrumentalisierung besteht aus dem `instr`-Task von EMMA. Der Pfad wird den zu instrumentalisierenden kompilierten Klassen übergeben. In das Ausgabeverzeichnis werden die neuen manipulierten Artefakte geschrieben. Eingabe- und Ausgabeverzeichnis sind identisch, was bedeutet, dass die ursprünglichen Klassen überschrieben werden. Würden Sie die Ausgangsklassen weiterverwenden wollen (z.B. verteilen), so wie sie kompiliert wurden, würden Sie dafür sicherlich ein neues Verzeichnis angeben.

Bevor wir die EMMA-Tasks nutzen, müssen wir sie allerdings bekannt machen. Dafür haben wir die beiden *.jar*-Dateien *emma.jar* und *emma-ant.jar* eingebunden.

```
<property name="emma.dir" value="${root}/emma/lib"/>
<path id="emma.lib">
   <pathelement location="${emma.dir}/emma.jar"/>
   <pathelement location="${emma.dir}/emma_ant.jar"/>
</path>
<taskdef resource="emma_ant.properties" classpathref="emma.lib"/>
```

Sind die Klassen instrumentalisiert, können wir die Tests laufen lassen. Währenddessen protokolliert EMMA, wie weit die Klassen durchgeführt werden. Dazu müssen wir *emma.lib* im Pfad bekannt machen.

```
<path refid="emma.lib"/>
```

Wichtig sind ferner die Übergabeparameter für JUnit. Einerseits übergeben wir Pfad und Namen der Datei *coverage.emma*. Hier speichert EMMA Verarbeitungsinformationen. Andererseits müssen wir `merge` auf `true` setzen. Vergessen Sie das, werden die Ergebnisse überschrieben und im späteren Report nur eine Klasse komplett dargestellt.

```
<jvmarg value="-Demma.coverage.out.file=${test.home}/coverage/coverage.emma" />
<jvmarg value="-Demma.coverage.out.merge=true" />
```

Nachdem die Unit-Tests gelaufen sind, können wir den Report erstellen.

```
<target name="report-instr">
   <emma>
      <report sourcepath="${test.home}/src" verbosity="info" >
          <fileset dir="${test.home}/coverage">
```

```
            <include name="*.emma"/>
          </fileset>
          <html outfile="${test.home}/coverage/index.html"
            columns="name,class,method,line,block"/>
      </report>
   </emma>
</target>
```

sourcepath gibt hier den Pfad zu den ursprünglichen Java-Dateien an. Das ermöglicht uns, im späteren Report ebenfalls den Quelltext zu sehen. Auch unsere *emma*-Datei im Verzeichnis *coverage* muss hier bekannt gemacht werden.

EMMA bietet verschiedene Ausgabeformate. Wir wählen HTML, lassen als Einstiegspunkt für die generierten Dateien eine *index.html* im *coverage*-Ordner erstellen und geben an, was der Report anzeigen soll. Hier haben wir uns dafür entschieden, in den EMMA-Reporttabellen jeweils Spalten für Name, Klasse, Methode, Zeilen und Blöcke anzulegen. EMMA bietet umfangreiche Konfigurationsmöglichkeiten bei Darstellung, Sortierung etc., was in der Onlinedokumenation ausführlich beschrieben ist.

Führen wir nun das neue Ant-Skript aus, ergibt sich folgende Ausgabe:

```
michael@BlueGene:~/junit$ ant
Buildfile: build.xml

clean:
   [delete] Deleting directory /home/michael/junit/test/classes
   [delete] Deleting directory /home/michael/junit/test/report
   [delete] Deleting directory /home/michael/junit/test/coverage
    [mkdir] Created dir: /home/michael/junit/test/classes
    [mkdir] Created dir: /home/michael/junit/test/report
    [mkdir] Created dir: /home/michael/junit/test/coverage

compile-src:

compile-test:
    [javac] Compiling 2 source files to /home/michael/junit/test/classes

instr:
    [instr] processing instrumentation path ...
    [instr] instrumentation path processed in 113 ms
    [instr] [2 class(es) instrumented, 0 resource(s) copied]
    [instr] metadata merged into [/home/michael/junit/test/coverage/metadata.emma]
            {in 6 ms}

test:
    [junit] Running com.huettermann.junit.AgileTest
    [junit] Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 0,038 sec

report-instr:
   [report] processing input files ...
   [report] 2 file(s) read and merged in 2 ms
   [report] writing [html] report to [/home/michael/junit/test/coverage/index.html] ...
```

```
report:
[junitreport] Processing /home/michael/junit/test/report/TESTS-
TestSuites.xml to /tmp/null930480196
[junitreport] Loading stylesheet jar:file:/home/michael/ant/apache-ant-1.7.0RC1/lib/ant-
junit.jar!/org/apache/tools/ant/taskdefs/optional/junit/xsl/junit-frames.xsl
[junitreport] Transform time: 1108ms
[junitreport] Deleting: /tmp/null930480196

all:

BUILD SUCCESSFUL
Total time: 3 seconds
michael@BlueGene:~/junit$
```

Bei der Ausgabe fallen die beiden EMMA-spezifischen Schritte auf: `instr` und `report-instr`. `instr` teilt uns mit, dass EMMA erfolgreich unsere beiden Klassen instrumentalisiert hat (alle in unserem Verzeichnis). `report-instr` dokumentiert interne Schritte für die Erstellung des Reports. Wenn Sie Interesse daran haben, näher in die interne Verarbeitung von EMMA zu blicken, kann ein `debug`-Eintrag im Ant-Skript ein erster Einstieg sein. Jeder EMMA-Task besitzt ein optionales Attribut `verbose`, dessen Wert die Gesprächigkeit der Ausgabe direkt beeinflusst.

Schauen wir uns nun das Ergebnis unseres Tests an. Ein kurzer Blick auf den JUnit-Report zeigt uns, dass beide Tests erfolgreich durchgelaufen sind (siehe Abbildung 10-1). Das freut uns sehr.

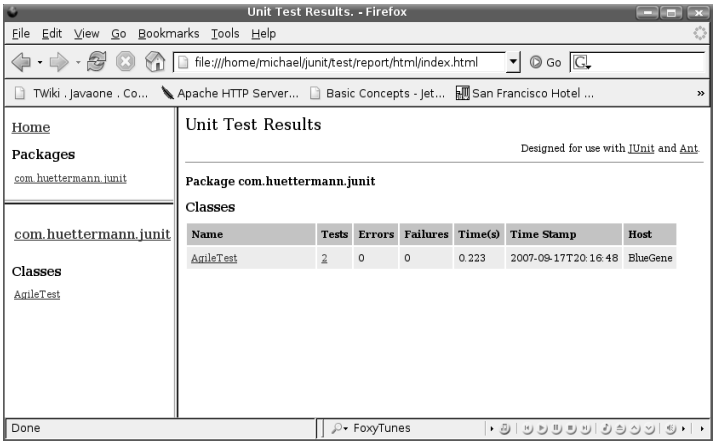


Abbildung 10-1: Testergebnis AgileTest

Testabdeckung

Unser Build-Skript (bzw. EMMA) hat innerhalb des Verzeichnisses *coverage* diverse HTML-Seiten hinterlegt, von denen *index.html* unser Einstiegspunkt ist. Wir öffnen diese Seite und sehen diverse Übersichtsangaben (siehe Abbildung 10-2). Das fängt im oberen

Bereich damit an, wie die Abdeckung der Klassen, Methoden, Zeilen und Blöcke ist. Das sind die von EMMA voreingestellten Standardspalten, die wir umkonfigurieren können. Darunter sehen wir direkt eine Übersicht aller Pakete und Klassen. Das wirkt direkt beruhigend auf uns. Beide unsere Klassen wurden von EMMA erkannt. Im unteren Bereich befindet sich eine Übersicht über die Pakete. Da wir nur ein Paket haben, ist hier eine Zeile abgebildet. Die Übersicht verrät uns auch, dass nur sechs von sieben Methoden durchlaufen wurden, und 90 % der Zeilen. Das macht uns neugierig auf mehr.

Wir klicken nun auf das Paket *com.huettermann.junit* um detaillierte Informationen zu bekommen und festzustellen, wo die 10 Prozentpunkte Abdeckung geblieben sind. Ein Klick auf den Link führt uns zur nächsten Seite, einer Übersicht über alle Klassen des Pakets. Hier sehen wir nun unsere beiden Klassen, die fachliche Klasse und die Testklasse (siehe Abbildung 10-3). Die Testklasse hat 100% Abdeckung, was uns mittlerweile nicht mehr wirklich überrascht. Unser Value Object hat eine Abdeckung von drei von vier Methoden. Der Wert »75%« bei der *Blockabdeckung* ist so schlecht, dass EMMA ihn rot einfärbt.

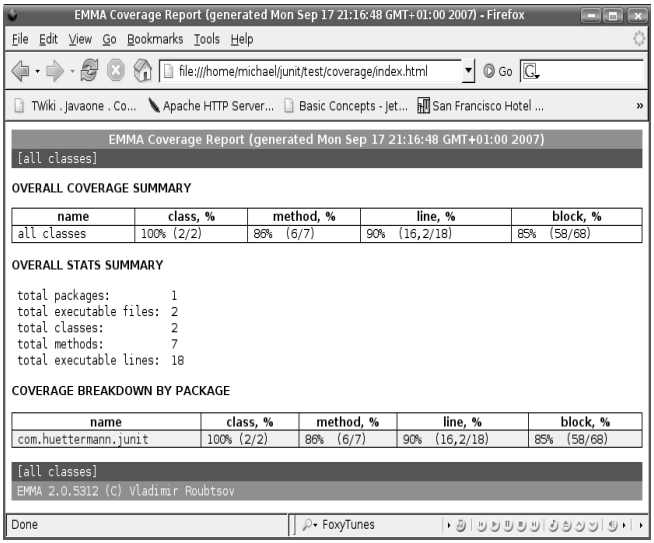


Abbildung 10-2: EMMA-Einstiegsseite

Block Coverage

EMMA nutzt als Hauptmetrik die »Blockabdeckung«. Blöcke sind dabei Sequenzen von Bytecode-Instruktionen, die als eine atomare Einheit ausgeführt werden. Die bei anderen Coverage-Werkzeugen wichtige Metrik der »Zeilenabdeckung« ist bei EMMA weniger wichtig. EMMA sieht hier die Bytecode-Blöcke als zentrale aussagekräftige Einheiten, da mehrere Zeilen im Quelltext in einem Block liegen können und Zeilen nicht zwingend Bytecode zur Folge haben (z.B. Kommentare).

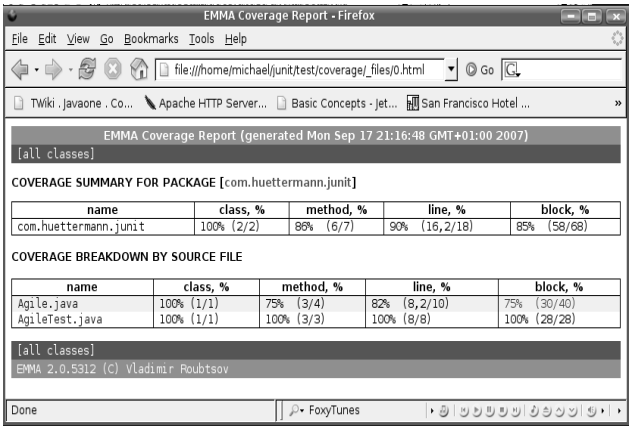


Abbildung 10-3: EMMA-Klassenübersicht

Gehen wir deshalb auf die Klasse `Agile.java` und klicken den Link an, um mehr zu erfahren. Eine neue Seite öffnet sich, die sich nur noch mit dieser Klasse beschäftigt (siehe Abbildung 10-4). Die allgemeinen Infos ganz oben kennen wir schon. Darunter sehen wir aber jetzt alle Methoden der Klasse, aufgelistet mit jeweiliger Abdeckung. Wir sehen: Der Konstruktor und der Getter werden wunderbar durchlaufen. Die `hashCode`-Methode wird aber leider gar nicht durchlaufen. Wir haben Sie überschrieben, testen sie aber nicht. Die `equals`-Methode wird durchlaufen, aber nur 84% ihrer Zeilen. Um dem Grund dafür weiter auf die Schliche zu kommen, bietet uns EMMA unterhalb der Statistiken sehr praktisch die Sourcen der Klasse an.

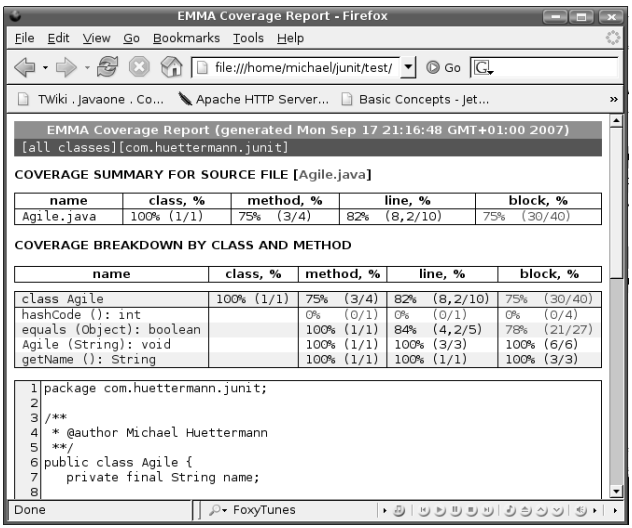


Abbildung 10-4: EMMA-Klassendetails

Wir scrollen etwas hinunter (siehe Abbildung 10-5) und sehen, dass EMMA die jeweiligen Stellen im Code farblich hinterlegt (wir können auch von der Übersicht oben mit Links auf die Methoden zur jeweiligen Stelle springen). Die durchgelaufenen Passagen, also der Konstruktor und der Getter, werden mit grün hinterlegt. Die Methode `hashCode` wird rot hinterlegt, sie wurde gar nicht tangiert. Die Methode `equals` ist teilweise grün, teilweise gelb. Wir kennen dieses Ampel-Schema bereits von JUnit. Hier bedeutet sie Folgendes: Grüne Farbe zeigt komplette Abdeckung an, und gelb sind Bereiche, die nur teilweise durchlaufen wurden. Um herauszufinden, was solche Bereiche sind, nehmen Sie eine `if`-Abfrage. Es kann pro Durchlauf nur ein `if`-Zweig durchlaufen werden. Das ist also vertretbar, sollte aber doch unsere Aufmerksamkeit haben. Schließlich könnten wir uns hier überlegen, andere Tests zu schreiben, die auch diese noch nicht durchlaufenen `if`-Zweige testen. Das können *Randwertebereichtests* sein (Tests mit Werten, die im Randbereich der zulässigen Wertemengen liegen) oder *Negativtests*, die absichtlich falsche, nicht valide Eingaben übergeben. Für die nicht durchlaufende Methode `hashCode` könnte ebenfalls ein Test nachgelegt werden.

Wie viel Testabdeckung ist nötig? Wie wir am konkreten Szenario gesehen haben, sind 100% Abdeckung aller Zeilen zwar erreichbar, aber nicht mehr effizient. Mit einer Abdeckung von 80% haben Sie bereits einen sehr guten Wert erreicht.

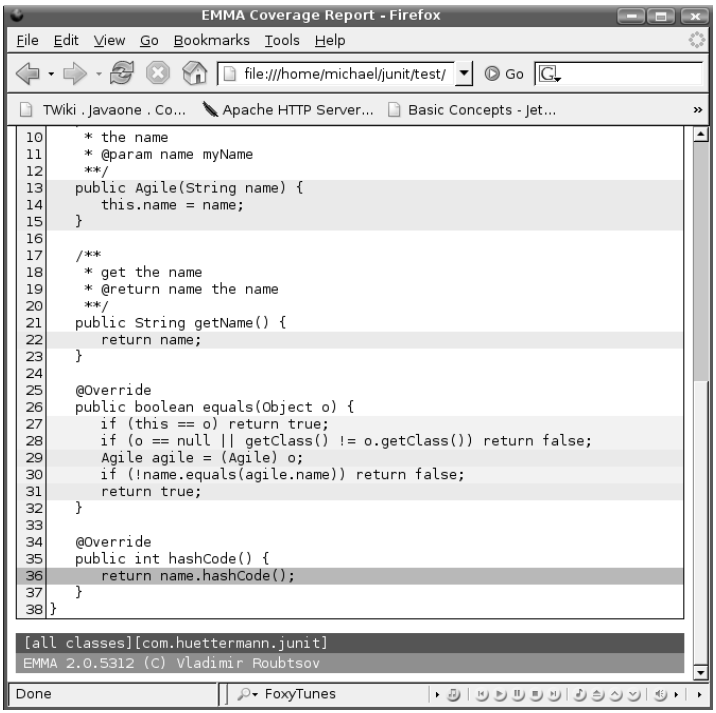


Abbildung 10-5: EMMA-Klassendetails, Teil 2

Zusammenfassung

Um festzustellen, wie viel Code Sie tatsächlich mit Ihren Tests abdecken, sollten Sie Code Coverage-Werkzeuge einsetzen. Es bringt nichts, wenn immer alle Tests grün sind, aber nur einen Bruchteil Ihrer fachlichen Klassen testen. Es bringt auch nichts, wenn Sie sehr viele Tests haben, die aber das Falsche testen.

EMMA ist ein frei verfügbares Werkzeug, das sich in Ihren Build-Prozess integrieren lässt und weitere Informationen über die Qualität der Software, insbesondere der Tests, liefert. Eine Testabdeckung von 100% ist utopisch. Konzentrieren Sie sich zumindest darauf, dass alle Klassen und deren Methoden zu 100% durchlaufen werden. Auch hier kann es in Einzelfällen schon zu Problemen führen, diese Messlatte einzuhalten: Stellen Sie sich vor, Sie überschreiben in einer Klasse die *equals*-Methode. Nun müssen Sie auch die *hashCode*-Methode überschreiben. Die wird bei Ihnen aber vielleicht gar nicht ausgeführt. Und schon liegt die Abdeckung unter 100%.

Haben Sie dennoch immer den Vorsatz, alles zu testen! Verzichten sie aber auf den Test von atomaren Statements, auf Getter und Setter, auf Delegates oder Tests von Fremdkomponenten (wie das Java-SDK). Auf Block- oder Zeilenebene erreichen Sie mit einer Abdeckung von 80% bereits einen guten Wert. Eine höhere Abdeckung zu erreichen ist möglich, allerdings mit einem exponentiell höheren Testaufwand verbunden, dessen praktischer Nutzen fraglich ist.

Sagt jemand zu Ihnen »Diese Klasse ist untestbar« oder »Diese Klasse kann ich nicht testen«, dann deutet das auf ein Designproblem in eben dieser Klasse hin (und darauf, dass die fachliche Klasse vor der Testklasse geschrieben wurde).

KAPITEL 11

Der Einsatz von Mock-Objekten mit EasyMock

Das Testen von Komponenten wird dadurch erschwert, dass sie häufig auf komplizierte oder nur kostspielig zu erstellende Ressourcen zurückgreifen. Ein Beispiel dafür sind umfangreiche Datenbankzugriffe, die Sie im Rahmen der Komponententests nicht immer durchführen wollen. Wenn Sie bei jedem Testlauf auf die Datenbank zugreifen, resultieren daraus deutlich längere Testintervalle und Sie bekommen erst spät eine Rückkopplung. Darüber hinaus wollen Sie auch nicht das Verhalten dieser Fremdkomponenten (wie der Datenbank) testen, sondern zunächst einmal lediglich das Verhalten Ihrer Klasse. Viel besser ist es daher, die Komponententests isoliert von den fremden Komponenten durchzuführen. Hier hat sich der Mock-Ansatz als hilfreiche Lösung durchgesetzt.

Von Mocks und Dummys

Es existiert eine Vielzahl von Ansätzen und unterschiedlichen Begriffen, die letztendlich alle in die gleiche Richtung gehen: das isolierte Testen einer eigenen Komponente gegen ein simuliertes Fremdsystem. Es haben sich unter anderen folgende Ansätze und Begriffe entwickelt:

Stub
: Ein Attrappe für noch nicht realisierte Funktionalität. Die richtige Implementierung wird später den Stub ersetzen. Es handelt sich um eine Imitation, die zweckmäßige Resultate liefert (meistens konstante Rückgaben).

Dummy
: Eine Attrappe, die die echte Implementierung ersetzt. Ein Dummy ist intelligenter als ein Stub, da er sich zur Laufzeit auswählen lässt und Objekte sich konfigurieren lassen.

Mocks
: Ein Mock ist ein Ersatz für eine Produktivumgebung, die das *Testen von innen* ermöglicht. Beim Testen von innen enthält das Mock-Objekt den Testcode. Mocks sind quasi Stubs mit eingebauter Testfunktionalität. Mock-Objekte kapseln das erwartete Verhalten in einer eigenen Klasse und sind die »intelligenteste« Version von Attrappen.

Der Einsatz von Mocks kann in folgenden Situationen hilfreich sein:

- Das reale Objekt hat ein nicht deterministisches Verhalten.
- Das reale Objekt ist schwierig zu erstellen.
- Das reale Objekt hat ein schwierig zu rekonstruierendes Verhalten.
- Das reale Objekt ist langsam.
- Das reale Objekt ist ein UI.
- Der Test muss das reale Objekt fragen, wie es aufgerufen wurde (zum Beispiel für Callbacks).
- Das reale Objekt existiert noch nicht.

Mit einem Stub wird häufig die Entwicklung einer eingebundenen Funktionalität begonnen. Durch Bereitstellung eines Stubs kann an der Klasse schon weitergearbeitet werden, ohne dass die eingebundene Funktionalität verfügbar ist. Ein Dummy implementiert den Stub aus und macht ihn konfigurierbar. Der Dummy liefert bereits intelligentere Rückgabewerte an die nutzende Klasse zurück. Eine komplexe Definition des Verhaltens der simulierten Komponente und das Schreiben von Tests sind weder bei einem Stub noch bei einem Dummy elegant möglich.

Mock-Objekte erlauben Komponententests (Unit-Testing) in vollkommener Isolation. Verteilte Software, die aus unterschiedlichen komplexen Subsystemen besteht, mit entfernten Servern kommuniziert und auf verteilte Komponenten zugreift, ist ohne Mock-Objekte nicht aussagekräftig zu testen. Mock-Objekte ersetzen dabei aus der Sicht der zu testenden Komponente Fremdkomponenten wie Datenbankverbindungen oder Nachrichtendienste. Der Testumfang und die Wechselwirkungen werden dadurch deutlich reduziert, so dass man sich auf das Verhalten einer einzelnen Komponente konzentrieren kann. Artefakte außerhalb des Prüflings, also der zu testenden Komponente, werden als Black Box angesehen, deren wahres Verhalten für den Test keine Rolle spielt.

Vorteile von Mock-Objekten

Der Entwickler profitiert von folgenden Vorteilen beim Einsatz von Mock-Objekten:

- Unit Tests sind lokalisiert und isoliert.
- Unit Tests lassen sich schneller durchführen und erlauben zügige Rückkopplungen.
- Eine Komponente kann entwickelt werden, obwohl eine andere, abhängige noch nicht vorliegt. Dadurch können auch kritische Entscheidungen hinausgeschoben werden.
- Sie ermöglichen strukturiertes Vorgehen zum Testen von Komponenten durch das Setzen des erwarteten Werts zu Beginn. Die Testlogik ist dabei im Mock-Objekt.

Nachteile von Mock-Objekten

Obwohl die Vorteile überwiegen, muss sich der Entwickler der Nachteile beim Einsatz von Mock Objekten bewusst sein.

- Es muss mehr Code geschrieben werden. Das Verhalten einer Komponente wird gewöhnlich durch die Implementierung eines Interface realisiert, so dass der Entwickler tatsächlich *alle* Methoden des Interface implementieren muss. Mehr Code bedeutet auch immer eine erhöhte Wahrscheinlichkeit von neuen Defekten.
- Die Komplexität wird erhöht. JUnit zeichnet sich durch seine Einfachheit aus. Hinzugefügte Mock-Objekte erhöhen die Komplexität deutlich.
- Mock-Objekte müssen, genauso wie jeder gewöhnliche JUnit-Testcode, mit dem fachlichen Code stetig synchronisiert werden: Bei Änderungen am fachlichen Code müssen ggf. die Mock-Objekte aktualisiert werden. Insbesondere wenn sich der fachliche Code häufig ändert, führt das zu großem Aufwand.
- Es kann aufwendig sein, für komplexe Komponenten tatsächlich Mock-Objekte zu erstellen. Häufig ist das Verhalten der zu emulierenden Komponenten nicht deterministisch oder wechselt regelmäßig.

Implementierung

Bevor wir uns ein konkretes Beispiel anschauen und dabei ein Framework einsetzen, das uns die Arbeit deutlich erleichtert, sollten wir uns die grundlegende Vorgehensweise bei der (manuellen) Erstellung von Mock-Objekten vergegenwärtigen. Gewöhnlich ist es so, dass wir mit einem Interface starten müssen. Dieses Interface definiert das Verhalten der fremden, zu simulierenden Komponente. Die fremde Komponente implementiert dieses Interface, genauso wie das Mock-Objekt, das durch seine Implementierung das Verhalten der Komponente annimmt.

Ein Mock-Objekt ist auch dadurch charmant in der Nutzung, dass es ein Verhalten und einen Zustand hat, einen *State*. Wir können also zu Beginn des Tests unserem Mock-Objekt ein bestimmtes Verhalten mitgeben, indem wir Parameter oder Attribute setzen (die vom Test genutzt werden können). Zu Beginn setzen wir eine Erwartung an das Mock-Objekt: Wir teilen ihm selbst mit, was die erwarteten Ausgaben sind, wie oft Methoden des Mocks aufgerufen werden und was sie für Resultate zurückliefern sollen.

Anschließend rufen wir den fachlichen Code auf und nutzen das Mock-Objekt als Parameter. Das geschieht häufig im Kontext von JUnit-Tests. Das Mock-Objekt wird jetzt aktiv im fachlichen Code genutzt. Es existiert gewöhnlich eine *verify*-Methode, die als letzter Schritt aufgerufen wird, um festzustellen, ob der reale Wert dem erwarteten Wert entspricht.

EasyMock installieren

Um die Mock-Objekte nicht von Hand erstellen zu müssen, haben sich Frameworks etabliert. *EasyMock* (*http://www.easymock.org/)* ist ein mächtiges, frei verfügbares Werkzeug, mit dessen Hilfe automatisch Mocks generiert werden können. Dabei wird der Java-Proxy-Mechanismus genutzt, um die Mocks tatsächlich on-the-fly erstellen zu kön-

nen. So erzeugte Mocks können komfortabel refaktorisiert werden, Rückgabewerte haben und Exceptions werfen. Auch die Reihenfolge von Methodenaufrufen wird gewährleistet.

Auch JMock ist ein bekanntes Framework zur Erstellung von Mock-Objekten. Es ist frei nutzbar und kann unter *http://www.jmock.org/* bezogen werden.

Die Installation geschieht denkbar einfach. Der Download des *.jar*-Files (im weiteren Verlauf wird mit der Version 2.2 gearbeitet) und das Hinterlegen in den Klassenpfad genügen. Es wird Java 5 vorausgesetzt. Um ein Mock-Objekt zu erstellen, gehen wir in drei Schritten vor:

1. Ein Mock-Objekt erstellen für ein Interface, das wir simulieren möchten
2. Das erwartete Verhalten aufnehmen
3. Dem Mock-Objekt mitteilen, dass es sein Verhalten abspielen soll

Erstes Szenario: »Wegmocken« von Callbacks

Das folgende Beispiel verdeutlicht das Vorgehen mit EasyMock.

Das Interface

Zunächst gehen wir davon aus, wir haben ein Interface, das wir simulieren möchten. Diese Schnittstelle wird von einer zu testenden Klasse verwendet. Die Testklasse selbst verwendet dieses Interface ebenfalls. Unser Interface ist in Beispiel 11-1 dargestellt.

Beispiel 11-1: Das zugrundeliegende Interface

```
package com.huettermann.easymock;

/**
 * @author Michael Huettermann
 */
public interface MyInterface {

    void recordCreated(int no, String name);

    void recordChanged(int no, String name);

    int recordAmount();

    void methodUnderConstruction() throws UnsupportedOperationException;
}
```

Unser Interface besitzt vier Methoden. Die ersten beiden erstellen einen Datensatz bzw. modifizieren ihn, wobei sie jeweils einen Schlüssel und einen Namen erwarten. Der Name ist ein beliebiger String, der als Inhalt unseres Datensatzes fungieren soll. *Record-*

Amount liefert die Anzahl der angelegten Datensätze zurück. Schließlich besitzt das Interface eine Methode, deren Implementierung noch nicht vorgesehen ist.

Der Prüfling

Es existiert eine Klasse, deren Verhalten getestet werden soll (siehe Beispiel 11-2).

Beispiel 11-2: Der Prüfling, also die zu testende Klasse

```
package com.huettermann.easymock;

import java.util.*;

/**
 * @author Michael Huettermann
 */
public class ClassUnderTest {

    private Set<MyInterface> listeners = new HashSet<MyInterface>();
    private Map<Integer, String> records = new HashMap<Integer, String>();

    /**
     * Add listeners.
     *
     * @param listener myListener
     */
    public void addListener(MyInterface listener) {
        listeners.add(listener);
    }

    /**
     * Create a new record.
     *
     * @param no   identifier
     * @param name name
     */
    public void createRecord(int no, String name) {
        if (records != null &&
                !records.containsKey(no)) {
            records.put(no, name);
            for (MyInterface listener : listeners) {
                listener.recordCreated(no, name);
            }
        }
    }

    /**
     * Change existing records.
     *
     * @param no   identifier
     * @param name name
     */
```

Beispiel 11-2: Der Prüfling, also die zu testende Klasse (Fortsetzung)

```
    public void changeRecord(int no, String name) {
        if (records != null &&
                records.containsKey(no) &&
                !records.get(no).equals(name)) {
            records.put(no, name);
            for (MyInterface listener : listeners) {
                listener.recordChanged(no, name);
            }
        }
    }

    /**
     * Get amount of records.
     *
     * @return record quantity
     */
    public int getRecordQuantity() {
        return records.size();
    }

    /**
     * A strange method which provokes an exception.
     */
    public void doSomeStrangeBadThings() {
        for (MyInterface listener : listeners) {
            try {
                listener.methodUnderConstruction();
            }
            catch (Throwable t) {/**/}
        }
    }
}
```

Die Klasse `ClassUnderTest` bietet funktionale, öffentliche Methoden. `CreateRecord` dient zum Anlegen eines Datensatzes. Bei einem Aufruf wird der übergebene String mit dem Identifier als Schlüssel in eine Map geschrieben. Das gelingt nur, wenn die Map noch keinen Schlüssel wie den übergebenen hat. Nach erfolgreichem Anlegen wird die Methode `recordCreated` aufgerufen, um interessierte Parteien über diesen Vorgang zu unterrichten. Das Beispiel verdeutlicht die Vorgehensweise einer Listener-Architektur. Die hier benachrichtigten Listener werden mit der Methode `addListener` der Klasse bekanntgegeben.

Die Methode `changeRecord` dient der Modifikation von bestehenden Einträgen. Auch hier werden bei Erfolg die Listener benachrichtigt und der Callback `recordChanged` aufgerufen. Schließlich existiert noch eine dritte Methode `getRecordQuantity`. Sie dient dazu, die Anzahl der aktuellen Einträge mitzuteilen. Die Methode `doSomeStrangeBadThings` ist eine Methode, die eine Exception werfen wird.

Die Testklasse

Schauen wir uns nun die korrespondierende Testklasse an (siehe Beispiel 11-3).

Beispiel 11-3: Die Testklasse

```
package com.huettermann.easymock;

import static org.easymock.EasyMock.*;
import org.junit.Before;
import org.junit.Test;
import org.junit.Assert;

public class ClassTest {

    private MyInterface mock;
    private ClassUnderTest classUnderTest;

    @Before
    public void setUp() {
        mock = createStrictMock(MyInterface.class);
        classUnderTest = new ClassUnderTest();
        classUnderTest.addListener(mock);
    }

    @Test
    public void createRecord() {
        mock.recordCreated(1, "Heartbreak");
        replay(mock);
        classUnderTest.createRecord(1, "Heartbreak");
        verify(mock);
    }

    @Test
    public void createAndChangeRecord() {
        mock.recordCreated(1, "Heartbrek");
        mock.recordChanged(1, "Heartbreak");
        replay(mock);
        classUnderTest.createRecord(1, "Heartbrek");
        classUnderTest.changeRecord(1, "Heartbreak");
        verify(mock);
    }

    @Test
    public void createAndChangeMultipleRecord() {
        mock.recordCreated(1, "Heartbrek");
        mock.recordChanged(1, "Heartbreak");
        expectLastCall().times(1);
        replay(mock);
        classUnderTest.createRecord(1, "Heartbrek");
        classUnderTest.changeRecord(1, "Heartbreak");
        classUnderTest.changeRecord(1, "Heartbreak");
        classUnderTest.changeRecord(1, "Heartbreak");
        verify(mock);
```

Beispiel 11-3: Die Testklasse (Fortsetzung)

```
    }

    @Test
    public void checkAmountRecord() {
        mock.recordCreated(1, "Heartbrek");
        mock.recordChanged(1, "Heartbreak");
        expectLastCall().times(1);
        mock.recordCreated(2, "Heartbreak2");
        replay(mock);
        classUnderTest.createRecord(1, "Heartbrek");
        classUnderTest.changeRecord(1, "Heartbreak");
        classUnderTest.changeRecord(1, "Heartbreak");
        classUnderTest.createRecord(2, "Heartbreak2");
        Assert.assertEquals(classUnderTest.getRecordQuantity(), 2);
        verify(mock);
    }

    @Test
    public void breakMe() {
        mock.methodUnderConstruction();
        expectLastCall().andThrow(new UnsupportedOperationException());
        replay(mock);
        classUnderTest.doSomeStrangeBadThings();
        verify(mock);
    }
}
```

Die Testklasse `ClassTest` nutzt Funktionen von JUnit und erweitert sie um Mock-Tests. Doch gehen wir einmal durch, was genau gemacht wird. Zunächst gibt es eine `setUp`-Methode, die vor jedem Test ausgeführt werden soll. Hier werden die Vorbereitungen getroffen: Zunächst wird das Mock-Objekt angelegt. Es ist vom Typ `MyInterface` und wird durch Übergabe von `Interface.class` an `createStrictMock` von EasyMock erzeugt. Dies generiert ein Mock-Objekt, das auch die Reihenfolge der Methoden, die auf ihm aufgerufen werden, kontrolliert (bei `createMock` ist die Reihenfolge der Methoden unerheblich). Danach wird die zu testende Klasse erstellt. Schließlich übergeben wir der Methode `addListener` unser Mock-Objekt.

```
@Before
public void setUp() {
    mock = createStrictMock(MyInterface.class);
    classUnderTest = new ClassUnderTest();
    classUnderTest.addListener(mock);
}
```

Die Testmethoden sehen nun genauso aus, wie wir es schon von den herkömmlichen JUnit-Tests kennen. Es gibt allerdings den Unterschied, dass wir hier auch auf das Mock-Objekt zugreifen.

Zu Beginn definieren wir das *Verhalten* des Mock-Objekts, also wie die Methoden des Interface später aufgerufen werden. Nachdem wir `replay` aufgerufen haben, ist `mock` ein Mock-Objekt für unser `MyInterface`: Es erwartet keine Aufrufe mehr. Anschließend füh-

ren wir Testaufrufe auf unserer zu testenden Klasse durch, die das zu Beginn spezifizierte Verhalten des Interface zur Folge haben sollen. Beim ersten Test `createRecord` bedeutet das, dass wir vom Prüfling einen Aufruf des Callbacks `createRecord` mit den Parametern erwarten, die wir an den Prüfling übergeben haben. `Verify` kontrolliert dabei Soll und Ist. Dieser Test läuft erfolgreich durch.

```
@Test
public void createRecord() {
    mock.recordCreated(1, "Heartbreak");
    replay(mock);
    classUnderTest.createRecord(1, "Heartbreak");
    verify(mock);
}
```

Wenn wir allerdings den Test ändern und z.B. einen falschen Parameter erwarten

```
@Test
public void createRecord() {
    mock.recordCreated(1, "Heartbreak");
    replay(mock);
    classUnderTest.createRecord(2, "Heartbreak");
    verify(mock);
}
```

dann ergibt ein Testdurchlauf

```
java.lang.AssertionError:
  Unexpected method call recordCreated(2, "Heartbreak"):
    recordCreated(1, "Heartbreak"): expected: 1, actual: 0
    at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
```

Schauen wir uns nun den zweiten Test an. Dieser Test soll die Änderung an einem bestehenden Satz testen. Es wird zunächst wieder ein Satz erstellt und dann geändert. Erinnern Sie sich? JUnit erstellt für jeden Test eine neue Instanz Ihrer Klasse. Sie müssen also den Satz erneut erstellen, den Sie anschließend ändern möchten.

```
@Test
public void createAndChangeRecord() {
    mock.recordCreated(1, "Heartbrek");
    mock.recordChanged(1, "Heartbreak");
    replay(mock);
    classUnderTest.createRecord(1, "Heartbrek");
    classUnderTest.changeRecord(1, "Heartbreak");
    verify(mock);
}
```

Ein Aufruf der Methoden `createRecord` und `changeRecord` auf dem Prüfling soll die Callbacks `recordCreated` und `recordChanged` in dieser Reihenfolge mit den spezifizierten Parametern zur Folge haben. Auch dieser Test läuft durch.

Wenn wir den Test ändern und die Reihenfolge der zu erwartenden Callbacks umdrehen

```
@Test
public void createAndChangeRecord() {
    mock.recordChanged(1, "Heartbreak");
    mock.recordCreated(1, "Heartbrek");
```

```
        replay(mock);
        classUnderTest.createRecord(1, "Heartbrek");
        classUnderTest.changeRecord(1, "Heartbreak");
        verify(mock);
    }
```

dann bekämen wir beim Durchlauf des Tests

```
java.lang.AssertionError:
  Unexpected method call recordCreated(1, "Heartbrek"):
    recordChanged(1, "Heartbreak"): expected: 1, actual: 0
    at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:29)
```

Was macht nun die dritte Testmethode namens `createAndChangeMultipleRecord`? Diese Methode simuliert den Versuch, einen bestehenden Satz mehrfach mit demselben Wert zu ändern. Das soll keinen Aufruf des Callbacks zur Folge haben, da sich der Satz nicht ändert. Wie testen wir das? Wir erwarten die beiden Callbacks `recordCreated` und `recordChanged` jeweils einmal. Die Kardinalität eines zu erwartenden Methodendurchlaufs können wir mit der Methode `expectLastCall().times()` definieren. Anschließend erstellen wir einen Satz und versuchen, ihn dreimal mit demselben Wert zu ändern. Der Test läuft durch, der Callback wird tatsächlich nur einmal ausgelöst.

```
 @Test
public void createAndChangeMultipleRecord() {
    mock.recordCreated(1, "Heartbrek");
    mock.recordChanged(1, "Heartbreak");
    expectLastCall().times(1);
    replay(mock);
    classUnderTest.createRecord(1, "Heartbrek");
    classUnderTest.changeRecord(1, "Heartbreak");
    classUnderTest.changeRecord(1, "Heartbreak");
    classUnderTest.changeRecord(1, "Heartbreak");
    verify(mock);
}
```

Wenn wir den Callback `recordChanged` zweimal erwarten, wird die Methode

```
@Test
public void createAndChangeMultipleRecord() {
    mock.recordCreated(1, "Heartbrek");
    mock.recordChanged(1, "Heartbreak");
    expectLastCall().times(2);
    replay(mock);
    classUnderTest.createRecord(1, "Heartbrek");
    classUnderTest.changeRecord(1, "Heartbreak");
    classUnderTest.changeRecord(1, "Heartbreak");
    classUnderTest.changeRecord(1, "Heartbreak");
    verify(mock);
}
```

folgendes Output liefern:

```
java.lang.AssertionError:
  Expectation failure on verify:
    recordChanged(1, "Heartbreak"): expected: 2, actual: 1
    at org.easymock.internal.MocksControl.verify(MocksControl.java:71)
```

Der Test würde fehlschlagen, da die Methode nur einmal ausgeführt wird, wir aber nun zwei Durchläufe erwarten. Die Methode `checkAmountRecord` führt einen weiteren Test von JUnit ein, um die Anzahl der gerade vorhandenen Sätze zu überprüfen. Wir erstellen einen Satz, ändern ihn und erstellen einen weiteren. Es sollten anschließend zwei Sätze vorliegen. Der Test läuft erfolgreich durch.

```
@Test
public void checkAmountRecord() {
    mock.recordCreated(1, "Heartbrek");
    mock.recordChanged(1, "Heartbreak");
    expectLastCall().times(1);
    mock.recordCreated(2, "Heartbreak2");
    replay(mock);
    classUnderTest.createRecord(1, "Heartbrek");
    classUnderTest.changeRecord(1, "Heartbreak");
    classUnderTest.changeRecord(1, "Heartbreak");
    classUnderTest.createRecord(2, "Heartbreak2");
    Assert.assertEquals(classUnderTest.getRecordQuantity(), 2);
    verify(mock);
}
```

Die letzte Methode unserer Testklasse `breakMe` soll das korrekte Werfen einer Exception überprüfen. Die Methode `methodUnderConstruction` liefert eine `UnsupportedOperationException`. Rufen wir `doSomeStrangeBadThings` auf dem Prüfling auf, wird tatsächlich die Exception geworfen. Der Erwartungswert wird hier mithilfe der Methode `expectLastCall` von EasyMock definiert. Zunächst wird der Callback erwartet und dann sein Output inhaltlich überprüft. `ExpectLastCall().andThrow()` überprüft, ob die zuvor beschriebene Callback-Methode genau diese Exception wirft. Und das geschieht tatsächlich: Der Test läuft durch.

```
@Test
public void breakMe() {
    mock.methodUnderConstruction();
    expectLastCall().andThrow(new UnsupportedOperationException());
    replay(mock);
    classUnderTest.doSomeStrangeBadThings();
    verify(mock);
}
```

Sie werden gewöhnlich in Ihrem Prüfling diese Exception abfangen. Tun Sie das nicht, wird natürlich tatsächlich die Exception geworfen.

Operationen auf dem Mock können nacheinander durchgeführt werden. Dabei wird auch eine Notation unterstützt, um dem Mock Operationen in einer Zeile verkettet mitzugeben. Dieses Vorgehen ist als *Fluent Interface* bekannt. Dabei lassen sich die Aufrufe in einem Fluss lesen.

Wir wir gesehen haben, ist eine Integration von EasyMock einfach möglich und ermöglicht Ihnen das isolierte Testen Ihrer Komponente.

Zweites Szenario: »Wegmocken« einer ressourcenintensiven DB-Schnittstelle

Im zweiten Beispiel möchten wir nun mit Hilfe von EasyMock eine Datenbankkommunikation durch einen Mock ersetzen, um die Funktionalität einer Klasse, die genau über diese Datenbankkommunikation verfügt, testen zu können.

Die Interfaces

Unser Objektgeflecht besitzt zwei Interfaces. Das erste Interface IService beschreibt fachliche Methoden, über die wir unseren Service nutzen möchten (siehe Beispiel 11-4).

Beispiel 11-4: Das Service-Interface

```
package com.huettermann.easymock.beispiel2;

/**
 * Definiert den Service.
 *
 * @author Michael Huettermann
 */
public interface IService {
   /** eine fachliche Methode */
   public String getSomething();
}
```

Das zweite Interface beschreibt die Kommunikation mit der Datenbank. Dabei wird mit *getDC* ein Datencontainer zurückgeliefert (siehe Beispiel 11-5).

Beispiel 11-5: Die Datenbankschnittstelle

```
package com.huettermann.easymock.beispiel2;

import java.util.Map;

/**
 * Definiert die DB/Fremdsystem-Schnittstelle.
 *
 * @author Michael Huettermann
 */
public interface IConnector {

   /** Liefert einen DC zurück */
   public Map<String,String> getDC ();
}
```

Der Datenbank-Connector

Der Datenbank-Connector implementiert dieses Interface. Er greift auf die Datenbank zu und liefert Ergebnisse zurück (siehe Beispiel 11-6).

Beispiel 11-6: Der Datenbank-Connector

```
package com.huettermann.easymock.beispiel2.connector;

import java.util.HashMap;
import java.util.Map;

import com.huettermann.easymock.beispiel2.IConnector;

/**
 * Der eigentliche Connector, der mit einer entfernten DB verbindet.
 *
 * @author Michael Huettermann
 */
public class Connector implements IConnector {

   /**
    * Greift auf die DB zu und bekommt einen Datencontainer
    * in Form einer Map zurück.
    */
   public Map<String,String> getDC() {

      // ...
      // ...
      // do some heavy DB stuff
      // ...
      // ...

      // Wir bekommen eine Map zurück.
      Map <String,String>map = new HashMap<String,String>();
      map.put("KEY", "EINS");

      System.err.println("DB CONNECT!!!");

      return map;
   }
}
```

Dieser Datenbank-Connector wird von unserer fachlichen Klasse `ServiceImpl` genutzt. `ServiceImpl` implementiert das fachliche Interface IService und greift innerhalb der implementierten Methode `doSomething()` auf den Connector zu, wertet Rückgabewerte aus und gibt Daten zurück. Dabei nutzt die Klasse eine `getConnector()` Methode, die über den Modifier `protected` verfügt und eine Instanz des Connectors erzeugt (»Lazy Initialization«) und zurückliefert. Beachten Sie dabei, dass das private Attribut `con` vom Typ Connector ist (siehe Beispiel 11-7).

Beispiel 11-7: Die Implementierung, die auf den Connector zugreift

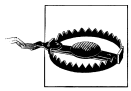
```
package com.huettermann.easymock.beispiel2.service;

import java.util.Map;

import com.huettermann.easymock.beispiel2.IConnector;
```

Beispiel 11-7: Die Implementierung, die auf den Connector zugreift (Fortsetzung)

```
import com.huettermann.easymock.beispiel2.IService;
import com.huettermann.easymock.beispiel2.connector.Connector;

/**
 * Implementiert die Fachlichkeit.
 *
 * @author Michael Huettermann
 */
public class ServiceImpl implements IService {

   /** der Connector zur DB */
   private IConnector con;

   /**
    * Erstellt Connector, liefert den Connector zurück.
    * @return IConnector
    */
   protected IConnector getConnector() {
      if(con==null) {
         con=new Connector();
      }
      return con;
   }

   /**
    * Exemplarische fachliche Methode, die aus der DB etwas holt
    * und den Rückgabewert verarbeitet.
    */
   public String getSomething() {

      // Greift auf den Connector/die DB zu.
      Map <String,String>dc = getConnector().getDC();

      // verarbeitet die Response
      if(dc.containsKey("KEY")) {
         return dc.get("KEY");
      } else {
         return "ERROR";
      }
   }
}
```

Die Testklasse

Kommen wir nun zur Testklasse. Wir möchten die Funktionalität der fachlichen Klasse `ServiceImpl` testen. Da diese über Datenbankkommunikation verfügt, mocken wir sie weg (siehe Beispiel 11-8).



Im vorliegenden Beispiel erstellen wir die Instanz des Connectors »lazy«, das heißt mit dem ersten Zugriff auf die `getConnector`-Methode. Wenn wir in der Basisklasse `ServiceImpl` die Instanziierung des Connectors direkt im Rahmen der Deklaration und Initialisierung durchführen, wird trotz Ableitung im Test immer eine Klasse vom eigentlichen Connector erzeugt.

Möchten Sie darüber hinaus mit einem Singleton arbeiten, soll es also nur eine Instanz Ihres Connectors geben, oder nutzen Sie Connection-Pooling, dann müssen Sie das Beispiel entsprechend anpassen. Das Design ist für die Erläuterung der Funktionsweise von EasyMock nicht entscheidend.

Beispiel 11-8: Testklasse mit Connector-Mock

```
package com.huettermann.easymock.beispiel2.tests;

import java.util.HashMap;
import java.util.Map;

import junit.framework.TestCase;

import org.easymock.EasyMock;

import com.huettermann.easymock.beispiel2.IConnector;
import com.huettermann.easymock.beispiel2.service.ServiceImpl;

/**
 * Ein Integrationstest
 *
 * @author Michael Huettermann
 */
public class TestThisThat_Integration extends TestCase {

   // der Mock
   private IConnector con;

   @Override
   public void setUp() throws Exception {
       con = EasyMock.createMock(IConnector.class);
   }

   /**
    * Testet, ob MyServiceImpl().getSomething() richtig arbeitet.
    * ServiceImpl() greift auf Umsysteme zu, so dass diese Zugriffe
    * simuliert werden müssen.
    */
   public void testMethode(){
      final Map <String,String>myMap = new HashMap<String,String>();
      myMap.put("KEY", "EINS");

      EasyMock.expect(con.getDC()).andReturn(myMap);
      EasyMock.replay(con);

      String value  = new MyServiceImpl().getSomething();
```

Beispiel 11-8: Testklasse mit Connector-Mock (Fortsetzung)

```
        assertTrue(value.equals("EINS"));

        EasyMock.verify(con);
    }

    /**
     * Eine Erweiterung unserer Service-Klasse ServiceImpl, die auf den Mock
     * aufsetzt.
     */
    private class MyServiceImpl extends ServiceImpl {
        @Override
        protected IConnector getConnector() {
            return con;
        }
    }
}
```

Die Testklasse erzeugt beim Start einen Mock für IConnector. Dieser Mock spielt in der inneren Klasse `MyServiceImpl` eine entscheidende Rolle. Dort leiten wir unsere Klasse `ServiceImpl` ab und überschreiben die `getConnector`-Methode. Diese liefert nun den Mock zurück, nicht mehr das ursprüngliche Connector-Objekt der Basisklasse.

Der Klassenname endet mit `_Integration`. Unterscheiden Sie so Tests, die übergreifend auf fremde Komponenten wie Datenbanken zugreifen von Tests, die nur einzelne Klassen testen.

In der Testmethode erstellen wir nun eine Map, die als Sollwert dient: Diese Map soll der Mock beim Aufruf der Methode `getDC()` zurückliefern. Nun rufen wir die fachliche zu testende Methode `doSomething()` auf und stellen fest, dass sie richtig funktioniert. Ausgehend von der vom Mock zurückgelieferten Map liefert die fachliche Klasse mit `EINS` den richtigen Wert zurück.

Zusammenfassung

Mocks sind dazu geeignet, langlaufende Tests zu beschleunigen, um schneller Feedback zu erhalten und den Entwicklungs- und Testprozess zu beschleunigen. Dabei können Sie komfortabel das Verhalten der Mocks beschreiben. Sie können Ihre Testklassen so konfigurieren, dass abhängig von einer Konfigurationseinstellung die Tests im Mock-Modus laufen oder gegen die wahre Datenbank aufgeführt werden. Nutzen Sie den Mock-Modus für schnell laufende Tests im Workspace und den »normalen, Mock-freien Modus« für Integrationstests, die die Datenbank (oder andere entfernte Services) tatsächlich ansprechen. Die Mock-Objekte werden in Unit-Tests eingebettet und genutzt. Die Erstellung von Mocks ist mit EasyMock komfortabel möglich.

KAPITEL 12

Komponententests mit TestNG

JUnit ist ein leistungsfähiges Werkzeug, das durch Einfachheit glänzt und lange Jahre das einzige Werkzeug und somit der unangefochtene Marktführer auf dem Gebiet der Komponententests war. Es hatte aber auch einige Nachteile. So setzte JUnit hauptsächlich auf Konventionen wie die Benennung von Testmethoden und war dadurch nicht sonderlich flexibel. Weiterhin mussten Testklassen von JUnit-Basisklassen ableiten. Daneben war JUnit für bestimmte Testszenarien einfach nicht mächtig genug, wie z.B. für das Testen von Nebenläufigkeit. In diesem Kapitel stelle ich Ihnen mit TestNG (Test*N*ew*G*eneration) eine mächtige Alternative zu JUnit vor. Das Kapitel führt anhand eines fortgeschrittenen Beispiels in TestNG ein, so dass Sie am Ende besser entscheiden können, ob und in welchen Fällen JUnit oder TestNG die für Sie bessere Lösung darstellt.

Steckbrief

Ab 2003 war es hauptsächlich Cedric Beust, der mit TestNG an einer leistungsfähigen Alternative zu JUnit arbeitete. Dabei wurden frühzeitig Java-Annotations genutzt und das Leistungsspektrum für Unit-Tests erweitert. Nicht zuletzt TestNG ist es zu verdanken, dass JUnit einen Entwicklungsschub erfuhr, weil es mit der Konkurrenz TestNG gleichziehen musste. Das Ergebnis war die Version 4.x von Junit.

TestNG hat im Vergleich zu JUnit folgende Vorteile:

- Tests lassen sich fein in sehr kleine Einheiten gruppieren.
- Sie können zwischen verschiedenen Gruppen von Tests umfangreiche Abhängigkeiten spezifizieren, die die Ausführungsreihenfolge festlegen.
- Tests können nicht nur mit Ant, sondern auch mit einem eigenen leichtgewichtigen XML-Dokument spezifiziert werden.
- Im Gegensatz zu JUnit arbeiten die Testmethoden auf denselben Instanzen, die Testmethoden erzeugen also nicht immer neue Instanzen.
- Nebenläufigkeitstests werden out-of-the-box unterstützt.
- Detailliertes Reporting: Der TestNG-Report gibt ausführlich über Ergebnisse und Testreihenfolgen Aufschluss.

TestNG hat im Vergleich zu JUnit folgende Nachteile:

- Tests können nicht als Plugin-Tests durchgeführt werden, was aber notwendig sein kann, wenn Sie Eclipse RCP- bzw. OSGi-Anwendungen schreiben.
- Die Tool-Unterstützung zur Nutzung aus einer IDE ist derzeit noch nicht ganz so gut wie bei JUnit. Obwohl es für TestNG Plugins zur Nutzung aus einer IDE gibt, ist die JUnit-Unterstützung häufig out-of-the-box vorhanden.

TestNG zeichnet sich außerdem durch folgende Merkmale aus:

- Testmethoden können mit Parametern aufgerufen werden und parametrisiert werden.
- TestNG kann JUnit-Tests laufen lassen.
- Fehlgeschlagene Tests können separat erneut gestartet werden.
- TestNG kann JUnit-konforme Reports erzeugen.

TestNG ist konzipiert für die einfache, schnelle Erstellung von Tests und hat eine ausgezeichnete Integration mit Ant. Es ist durch seine Flexibilität und Mächtigkeit primär dafür angelegt, High-Level-Tests durchzuführen, also Tests mit vielen großen Suites und vielen Abhängigkeiten, wobei ein fehlgeschlagener Test nicht den ganzen Ablauf beenden muss. Ganz im Gegenteil: Nicht erfolgreiche Tests können einzeln erneut durchgeführt werden. TestNG hat außerdem Vorteile beim Testen von nebenläufigen Anwendungen. So kann beispielsweise per Annotation konfiguriert werden, wie viele Threads wie oft eine Testmethode aufrufen sollen. Auch die Abhängigkeiten lassen sich, anders als bei JUnit, komfortabel definieren. Reihenfolgen von Tests und Testgruppen sind konfigurierbar.

Die Annotations sind vergleichbar mit denen von JUnit 4.x. Tabelle 12-1 erläutert die verfügbaren Annotations.

Tabelle 12-1: TestNG-Annotations

Annotation	Bedeutung
`@BeforeSuite`	Die annotierte Methode läuft, bevor alle Tests dieser Suite laufen.
`@AfterSuite`	Die annotierte Methode läuft, nachdem alle Tests dieser Suite laufen.
`@BeforeTest`	Die annotierte Methode läuft vor einem Test.
`@AfterTest:`	Die annotierte Methode läuft nach einem Test.
`@BeforeGroups:`	Die annotierte Methode läuft, bevor einer der Gruppentests läuft.
`@AfterGroups:`	Die annotierte Methode läuft, nachdem einer der Gruppentests läuft.
`@BeforeClass:`	Die annotierte Methode läuft, bevor alle Testmethoden dieser Testklasse laufen.
`@AfterClass:`	Die annotierte Methode läuft, nachdem alle Testmethoden dieser Testklasse laufen.
`@BeforeMethod`	Die annotierte Methode läuft, bevor eine Testmethode läuft.
`@AfterMethod`	Die annotierte Methode läuft, nachdem eine Testmethode läuft.
`@DataProvider`	Markiert eine Methode als Datenlieferant für eine Testmethode.
`@Factory`	Markiert eine Methode als Factory, die von TestNG als Testklasse genutzt wird.
`@Parameters`	Beschreibt, wie Parameter einer Testmethode übergeben werden.
`@Test`	Markiert eine Klasse oder eine Methode als Teil eines Tests.

Ein Beispiel: Agile, nebenläufige, gruppierte Verarbeitung

TestNG ist in der Anwendung gut vergleichbar mit JUnit 4.x, weswegen wir direkt mit einem umfangreichen Beispiel anfangen, wobei eine Haupteigenschaft von TestNG betrachten: die mächtige Steuerung von Ablauf und Abhängigkeiten.

Verzeichnisstruktur

Für den weiteren Verlauf setzen wir folgende Verzeichnisstruktur voraus:

```
michael@BlueGene:~/testng$ ls -la
insgesamt 36
drwxr-xr-x  8 michael michael 4096 2006-12-26 14:49 .
drwxr-xr-x 48 michael michael 4096 2006-12-26 14:49 ..
-rwxr--r--  1 michael michael 1825 2006-12-26 14:49 build.xml
drwxr-xr-x  3 michael michael 4096 2006-12-26 14:49 classes
drwxr-xr-x  2 michael michael 4096 2006-12-26 14:49 junit-report
drwxr-xr-x  2 michael michael 4096 2006-12-26 14:45 lib
drwxr-xr-x  2 michael michael 4096 2006-12-26 13:42 src
drwxr-xr-x  3 michael michael 4096 2006-12-26 14:35 test
drwxr-xr-x  3 michael michael 4096 2006-12-26 14:49 testng-report
```

Im Verzeichnis *lib* liegt das über *http://testng.org* herunterladbare *.jar*-File. In unserem Fall nutzen wir *testng-5.3.1-jdk15*. Sie benötigen TestNG-Version 5.3.1 oder höher, um von den hier vorgestellten Funktionen in den Bereichen Nebenläufigkeit und Reporting profitieren zu können. Das *classes*-Verzeichnis nimmt die Class-Files auf, die zu den Java-Quelldateien im *src*-Verzeichnis korrelieren. Das *test*-Verzeichnis enthält die Testklassen, in *testng-report* werden die Report-Seiten hinterlegt, die wir in JUnit-konforme Reports transformieren und ins Verzeichnis *junit-report* legen.

Die Klassen unter Test: Exceptions und BusinessObject

Im *src*-Verzeichnis liegen unsere Quelldateien, die wir mit TestNG testen wollen. Wir nutzen ein *BusinessObject*, um unveränderliche Daten zu kapseln (siehe Beispiel 12-1). Es enthält Attribute, die wir jederzeit abfragen, aber nicht mehr ändern können (»immutable object«).

Wir erstellen uns mehrere eigene Exceptions: *DuplicateKeyException* (siehe Beispiel 12-2), *RecordNotFoundException* (siehe Beispiel 12-3) und *SecurityException* (siehe Beispiel 12-4).

Beispiel 12-1: Ein BusinessObject

```
package com.huettermann.testng;

/**
 * immutable business object to be stored in the DB mock
 *
 * @author Michael Huettermann
 */
```

Beispiel 12-1: Ein BusinessObject (Fortsetzung)

```
public class BusinessObject {
    private long personnalNumber;
    private String firstName;
    private String lastName;

    public BusinessObject(long personnalNumber, String firstName, String lastName){
        this.personnalNumber=personnalNumber;
        this.firstName=firstName;
        this.lastName=lastName;
    }

    public String getFirstName(){
        return firstName;
    }

    public String getLastName(){
        return lastName;
    }
}
```

Beispiel 12-2: Unsere DuplicateKeyException

```
package com.huettermann.testng;

/**
 * Exception thrown for locking issues: a key already exists.
 *
 * @author Michael Huettermann
 */
public class DuplicateKeyException extends Exception {

    /**
     * Creates a default <code>SecurityException</code> instance.
     */
    public DuplicateKeyException() {
    }

    /**
     * Creates a <code>SecurityException</code> instance passing a text.
     * @param text the exception text
     */
    public DuplicateKeyException(String text) {
        super(text);
    }
}
```

Beispiel 12-3: Die RecordNotFoundException

```
package com.huettermann.testng;

/**
 * Exception thrown for locking issues: record not found
 *
```

Beispiel 12-3: Die RecordNotFoundException (Fortsetzung)

```
 * @author Michael Huettermann
 */
public class RecordNotFoundException extends Exception {

    /**
     * Creates a default <code>SecurityException</code> instance.
     */
    public RecordNotFoundException() {
    }

    /**
     * Creates a <code>SecurityException</code> instance passing a text.
     * @param text the exception text
     */
    public RecordNotFoundException(String text) {
        super(text);
    }
}
```

Beispiel 12-4: Die SecurityException

```
package com.huettermann.testng;

/**
 * Exception thrown for locking issues: security issue.
 *
 * @author Michael Huettermann
 */
public class SecurityException extends Exception {

    /**
     * Creates a default <code>SecurityException</code> instance.
     */
    public SecurityException() {
    }

    /**
     * Creates a <code>SecurityException</code> instance passing a text.
     * @param text the exception text
     */
    public SecurityException(String text) {
        super(text);
    }
}
```

Die Klassen unter Test: Der RessourceAdapter

Der *RessourceAdapter* (siehe Beispiel 12-5) simuliert ein Datenhaltungsmedium, z.B. eine Datenbank. Er verwaltet zentral den Zugriff auf die Datenhaltung in einer nebenläufigen Umgebung (viele Nutzer arbeiten gleichzeitig mit der Software und auf der Datenbank). In unserem Fall werden die abzuspeichernden Daten (die Business-Objekte) in einer Map abgelegt.

Da verschiedene Klienten gleichzeitig über den RessourceAdapter auf die Datenhaltung zugreifen können, müssen wir den Zugriff auf unsere Datenbank *thread-safe* machen. Es soll nur erlaubt sein, dass ein Klient zu einer bestimmten Zeit einen neuen Datensatz anlegt. Das gewährleisten wir durch die Nutzung eines *Semaphors*, ein Objekt, auf das wir synchronisieren.

Es können nur Datensätze angelegt werden, deren Schlüssel in der Datenbank noch nicht existieren. Die Methode `isCreated` kann die Existenz eines Satzes überprüfen. Beim Update, also einer Aktualisierung eines bestehenden Datensatzes, müssen wir zunächst den Zugriff auf genau diesen Datensatz (auf den Schlüssel) locken. Dazu nutzen wir die Methode `lockRecord`, die wartet, wenn ein Datensatz gerade noch gelockt ist (also von einem anderen Thread geändert wird). Als Lock-Tokens werden dabei mit `getLockKey` jeweils neue Lock-Schlüssel generiert.

Beim Update auf den Datensatz mit `updateRecord` werden entsprechende Exceptions geworfen, wenn der zu aktualisierende Satz nicht vorhanden, nicht gelockt oder von einem anderen Klienten für die Modifikation gelockt ist. Das Locking (Sperren) und Unlocking (Entsperren) von Sätzen muss der Einfachheit halber der Klient selbst durchführen (was im Produktiveinsatz durch eine Fassade weiter abstrahiert würde).

Beim Entsperren werden der von einem Klienten gehaltene Lock aufgehoben und wartende Threads darüber informiert. Die Methode `getRecord` liefert das Geschäftsobjekt aus der Datenbank zurück. Lesen soll immer erlaubt sein, weshalb keine weiteren Sperrvorkehrungen getroffen werden.

Beispiel 12-5: Der RessourceAdapter

```
package com.huettermann.testng;

import java.util.Map;
import java.util.HashMap;
import java.util.Collections;

/**
 * The adapter controlling the access on the remote ressource.
 *
 * @author Michael Huettermann
 */
public class RessourceAdapter {

    /** lock key */
    private static long lockKey=0;
    /** store the locked entries */
    private static final Map<Long, Long> reservedRecords = new HashMap<Long,Long>();
    /** mutex: only create one new entry at one time */
    private static final Object creationSemaphore = new Object();
    /** external ressource simulation for regulating accesses on */
    private static Map<Long, BusinessObject> database =
        Collections.synchronizedMap(new HashMap<Long, BusinessObject>());
```

Beispiel 12-5: Der RessourceAdapter (Fortsetzung)

```
    /**
     * Create a new entry.
     * @param personnalNumber personal number, unique id for a person
     * @param firstName first name
     * @param lastName last name
     * @throws com.huettermann.testng.DuplicateKeyException key already in use
     */
    public synchronized void createRecord(long personnalNumber, String firstName, String last
Name)
                throws DuplicateKeyException {
        //two distinct clients are not allowed to create a new entry the same time
        //wrapp exception in a business exception
        synchronized(creationSemaphore) {
            if(database.get(personnalNumber)!=null){
                throw new DuplicateKeyException();
            }
            database.put(personnalNumber, new BusinessObject(personnalNumber, firstName, last
Name));
        }
    }

    /**
     * Returns if number is created
     * @param personnalNumber no
     * @return <code>true</code> if created else <code>false</code>
     */
    public boolean isCreated(long personnalNumber){
        synchronized(creationSemaphore){
            return database.containsKey(personnalNumber);
        }
    }

    /**
     * update an entry.
     * @param personnalNumber personal number, unique id
     * @param firstName first name
     * @param lastName last name
     * @param lockKey lock
     * @throws  com.huettermann.testng.RecordNotFoundException record not found
     * @throws com.huettermann.testng.SecurityException security issue
     */
    public synchronized void updateRecord(long personnalNumber, String firstName, String last
Name, long lockKey)
            throws RecordNotFoundException, SecurityException {
        //record correct locked?
        synchronized(reservedRecords) {
            //the record does not exist yet
            if(database.get(personnalNumber)==null){
                throw new RecordNotFoundException();
            }
            //does the lock manager contains the entry?
            if(!reservedRecords.containsKey(personnalNumber)) {
```

Beispiel 12-5: Der RessourceAdapter (Fortsetzung)

```
                throw new SecurityException("unlock security");
            }
            long myLockKey = reservedRecords.get(personnalNumber);
            //is the lock key equal? only the client who locks the records is allowed to
unlock it
            if(myLockKey!=lockKey){
                throw new SecurityException("unlock security");
            }
            database.put(personnalNumber, new BusinessObject(personnalNumber, firstName, last
Name));
        }
    }

    /**
     * Returns if key is locked.
     * @param personnalNumber no
     * @return long lock key
     */
    public long getLockKey(long personnalNumber){
        synchronized(reservedRecords){
            if(reservedRecords.containsKey(personnalNumber)){
                return reservedRecords.get(personnalNumber);
            }else{
                return -1;
            }
        }
    }

    /**
     * lock a record
     * @param personnalNumber personal number id
     * @return LockKey
     * @throws  com.huettermann.testng.RecordNotFoundException record not found
     */
    public long lockRecord(long personnalNumber) throws RecordNotFoundException{
        synchronized(reservedRecords){
            if(database.get(personnalNumber)==null){
                throw new RecordNotFoundException();
            }
            //if already locked, wait
            try {
                while(reservedRecords.containsKey(personnalNumber)){
                      reservedRecords.wait();
                }
            }
            catch(InterruptedException ie){
                throw new RuntimeException(ie.getMessage());
            }
            //retrieve a lock key inside the monitor
            long lockKey = getLockKey();
            reservedRecords.put(personnalNumber,lockKey);
```

Beispiel 12-5: Der RessourceAdapter (Fortsetzung)

```
            //give all waiting threads the same chance
            reservedRecords.notifyAll();
            return lockKey;
        }
    }

    /**
     * unlock record
     * @param personnalNumber personal number
     * @param lockKey lock key
     * @throws  com.huettermann.testng.SecurityException security issue
     */
    public void unlockRecord(long personnalNumber, long lockKey) throws SecurityException {
        synchronized(reservedRecords){
            //does a lock on this record exist?
            if(!reservedRecords.containsKey(personnalNumber)){
                throw new SecurityException("unlock security");
            }
            long myLock = reservedRecords.get(personnalNumber);
            //is the lock the same? only then unlock.
            if(myLock!=lockKey){
                throw new SecurityException("unlock security");
            }
            reservedRecords.remove(personnalNumber);
            reservedRecords.notifyAll();
        }
    }

    /**
     * deliver a unique key
     * @return Lock key
     */
    private long getLockKey(){
        return lockKey++;
    }

    /**
     * returns a record or <code>null</code> if not available
     * @param personnalNumber Pesonalnummer
     * @return Business Object
     */
    public BusinessObject getRecord(long personnalNumber){
        return database.get(personnalNumber);
    }
}
```

Die Testklasse

Die TestNG-Testklasse befindet sich im *test*-Verzeichnis. Sie ist in Beispiel 12-6 abgebildet.

Beispiel 12-6: Die Testklasse

```
package com.huettermann.testng.test;

import com.huettermann.testng.RessourceAdapter;
import org.testng.annotations.Test;
import org.testng.annotations.BeforeSuite;
import org.testng.annotations.BeforeGroups;
import org.testng.Assert;

/**
 * @author Michael Huettermann
 */
public class Client {
    private static RessourceAdapter ra = new RessourceAdapter();

    public Client(){
    }

    @BeforeSuite(groups = {"update"})
    public void setUpSuite(){
        //
    }

    @BeforeGroups(value = {"update"}, groups = {"update"})
    public void setUpGroup(){
        //
    }

    @Test(groups={"create"},dependsOnMethods={"createOne"},
    threadPoolSize = 6, invocationCount = 6,  timeOut = 2000)
    public void createTwo() {
        try {
            ra.createRecord(1, "Christian", "Pander");
            Assert.fail("should throw an exception");
        }
        catch (Exception e){
           //e.printStackTrace();
        }
    }

    @Test(groups={"create"})
    public void createOne() {
        try {
            long myNo=1;
            Assert.assertTrue(!ra.isCreated(myNo));
            ra.createRecord(myNo, "Hans", "Wohlfahrt");
            Assert.assertEquals(ra.getRecord(myNo).getFirstName(),"Hans");
            Assert.assertEquals(ra.getRecord(myNo).getLastName(),"Wohlfahrt");
        }
        catch (Exception e){
            //e.printStackTrace();
            Assert.fail("should not throw an exception");
        }
```

Beispiel 12-6: Die Testklasse (Fortsetzung)

```
    }

    @Test(groups={"create"})
    public void createThree() {
       try {
           ra.createRecord(3, "Kurt", "Schmidt");
           Assert.assertEquals(ra.getRecord(3).getFirstName(),"Kurt");
           Assert.assertEquals(ra.getRecord(3).getLastName(),"Schmidt");
       }
       catch (Exception e){
           Assert.fail("should not throw an exception");
       }
    }

    @Test(groups={"update"},dependsOnMethods={"createOne","createTwo","createThree"},
    threadPoolSize = 2, invocationCount = 5,  timeOut = 2000)
    public void updateOne() {
        try {
             long myNo = 1;
             long lockKey=ra.lockRecord(myNo);
             Assert.assertTrue(((long)lockKey==(long)ra.getLockKey(myNo)));
             ra.updateRecord(myNo, "Hans-Dieter", "Meier", lockKey);
             ra.unlockRecord(myNo,lockKey);
             Assert.assertEquals(ra.getRecord(myNo).getFirstName(),"Hans-Dieter");
             Assert.assertEquals(ra.getRecord(myNo).getLastName(),"Meier");
             Assert.assertTrue(ra.getLockKey(myNo)==-1);
         }
         catch (Exception e){
             Assert.fail("should not throw an exception");
         }
    }
}
```

Wir erkennen direkt, dass die Testklasse von keiner TestNG-Basisklasse ableiten muss, sondern ihr Verhalten über Annotations deklariert wird. Interessant ist auch die Member-Variable `ra`, die an zentraler Stelle statisch erstellt wird. Auf diese eine Instanz werden alle Testmethoden zugreifen.

```
private static RessourceAdapter ra = new RessourceAdapter();
```

SetUps, Gruppierung und Testdurchführung

Die erste mit Annotations gekennzeichnete Methode ist `setUpSuite`. `@BeforeSuite` signalisiert, dass diese Methode einmalig zu Beginn der ganzen TestSuite, also zu Beginn aller Testmethoden durchlaufen wird. Der Inhalt der Methode ist leer, wir möchten hier lediglich den Einsatz der Methode illustrieren.

```
@BeforeSuite(groups = {"update"})
public void setUpSuite(){
    //
}
```

Die zweite Methode ist `setUpGroup`. Sie wird durch die Metainformation `@BeforeGroup` vor den dort spezifizierten Gruppen ausgeführt. Eine Suite kann aus verschiedenen Gruppen bestehen. Wir nutzen die Gruppen `create` und `update`, um Methoden fachlich zu gruppieren. Die Gruppennamen haben wir uns selbst überlegt. Da `@BeforeSuite` und `@BeforeGroups` mit Gruppen (jeweils »update«) eingeschränkt wurden, werden diese Methoden nur für diese Gruppen aufgerufen. Die Tests werden, wie wir später sehen werden, für bestimmte Gruppen gestartet.

```
@BeforeGroups(value = {"update"}, groups = {"update"})
public void setUpGroup(){
    //
}
```

Anschließend folgt mit `createTwo` die erste Testmethode. Wie der Name bereits suggeriert, soll hier ein Datensatz angelegt werden. Dazu wird die Methode `createRecord` des RessourceAdapter mit einem beliebigen Schlüssel (hier nehmen wir die *1*), einem Vor- und einem Nachnamen aufgerufen. Wir sehen, dass ein TestNG-Fehler geworfen wird (`Assert.fail` wird aufgerufen), obwohl das Anlegen erfolgreich ist. Warum soll das Anlegen scheitern?

Dazu müssen wir uns zunächst die weiteren Metainformationen dieser Methode anschauen. Das Attribut `dependsOnMethods` steuert die Ablaufreihenfolge und gibt vor, dass die Testmethode `createOne` vor der Methode `createTwo` ausgeführt werden soll. Wir schauen uns `createOne` gleich an. Welches Verhalten deklarieren wir noch? Wir lassen diese Testmethode von TestNG mehrfach aufrufen. Wir lassen dazu sechs Threads erstellen (`threadPoolSize=6`) und wollen mit diesen Threads die Methode insgesamt sechsmal aufrufen (`invocationCount=6`). Dabei nutzen wir ein Time-out von zwei Sekunden (`timeOut=2000`): Wenn diese Methode nicht nach zwei Sekunden erfolgreich endet, wertet TestNG ihre Ausführung als gescheitert.

```
@Test(groups={"create"},dependsOnMethods={"createOne"},
threadPoolSize = 6, invocationCount = 6,  timeOut = 2000)
public void createTwo() {
    try {
        ra.createRecord(1, "Christian", "Pander");
        Assert.fail("should throw an exception");
    }
    catch (Exception e){
       //e.printStackTrace();
    }
}
```

Schauen wir uns nun die Methode `createOne` an.

```
@Test(groups={"create"})
public void createOne() {
    try {
        long myNo=1;
        Assert.assertTrue(!ra.isCreated(myNo));
        ra.createRecord(myNo, "Hans", "Wohlfahrt");
        Assert.assertEquals(ra.getRecord(myNo).getFirstName(),"Hans");
```

```
            Assert.assertEquals(ra.getRecord(myNo).getLastName(),"Wohlfahrt");
        }
        catch (Exception e){
            //e.printStackTrace();
            Assert.fail("should not throw an exception");
        }
    }
```

Sie fällt in die Gruppe `create` und wird vor `createTwo` einmalig ausgeführt. Wir überprüfen zu Beginn mit `isCreated`, ob es schon einen Datensatz mit dem Schlüssel `1` in der Datenbank gibt. Das ist nicht der Fall. Anschließend legen wir einen Satz mit genau diesem Schlüssel an. Nun verstehen wir auch, warum bei der korrekten Ausführung der Methode `testTwo` eine Exception geworfen werden soll: Es existiert dann bereits ein Datensatz mit dem Schlüssel `1`. Der Test in `createTwo` läuft erfolgreich durch. Er ist letztendlich ein Test, ob der Datensatz in `createOne` richtig angelegt wurde und noch verfügbar ist und ob das Exception-Handling des RessourceAdapter korrekt funktioniert.

Das gilt auch für die Tests innerhalb der Testmethode `createOne`: Nach dem Anlegen des Datensatzes überprüfen wir die angelegten Werte aus der Datenbank. Läuft dabei etwas falsch, scheitert z.B. die Verarbeitung oder funktioniert unser Exception-Handling nicht richtig, dann wird der `catch`-Block der Methode `createOne` aufgerufen. Da eine Exception in diesem Fall, anders als bei der Testmethode `createTwo`, nicht richtig und unerwartet wäre, würden wir das Werfen aufnehmen und als inkorrekten Test protokollieren (`Assert.fail`). Bewusst fangen wir dabei alle Exceptions ab, die auftreten können, indem wir `Exception` catchen.

Was sollten wir aus dieser Methode noch mitnehmen? TestNG gelingt es, die Methoden der verschiedenen Gruppen (zunächst die Initialisierungsmethoden wie die mit `@BeforeGroups` gekennzeichneten, dann die Testmethoden) in der gewünschten Reihenfolge laufen zu lassen. Das gelingt aber nur, wenn diese von sich abhängigen Methoden sequenziell durchlaufen werden und nicht nebenläufig! Mit anderen Worten: Nicht voneinander abhängige Methoden werden nebenläufig durchlaufen. Bedenken Sie das, wenn Sie auf zentrale Member-Attribute zugreifen.

```
@Test(groups={"create"})
    public void createThree() {
       try {
           ra.createRecord(3, "Kurt", "Schmidt");
           Assert.assertEquals(ra.getRecord(3).getFirstName(),"Kurt");
           Assert.assertEquals(ra.getRecord(3).getLastName(),"Schmidt");
       }
       catch (Exception e){
           Assert.fail("should not throw an exception");
       }
    }
```

Die dritte Methode, die einen Datensatz anlegen soll, gehört ebenfalls wieder der Gruppe `create` an. In ihr legen wir einmalig einen neuen Datensatz mit dem Schlüssel `3` an. Ob das gelingt, überprüfen wir auf die gleiche Weise.

Abhängigkeiten und Nebenläufigkeit

Die letzte Methode der Testklasse ist `updateOne`. Sie befindet sich in einer anderen Gruppe als die zuvor betrachteten, nämlich in `update`. Würden wir also ausschließlich die `create`-Gruppe der Testklasse laufen lassen (den Start der Testklasse schauen wir uns etwas später an), würde diese Methode nicht berücksichtigt. Weiter fallen die Abhängigkeiten via `dependsOnMethods` auf: Hier geben wir vor, dass die Methode nach allen drei Methoden durchlaufen werden soll, die die Datensätze anlegen. Darüber hinaus sollen zwei voneinander unabhängige TestNG-Threads insgesamt fünfmal die Methode aufrufen, wobei auch ein Time-out von zwei Sekunden gilt. Wir simulieren damit verschiedene Klienten, die gleichzeitig mehrfach diese Methode aufrufen. Auf diese Weise können wir das Verhalten des RessourceAdapter testen, insbesondere sein Verhalten bei der Aktualisierung von Datensätzen und sein Locking-Verhalten.

```
@Test(groups={"update"},dependsOnMethods={"createOne","createTwo","createThree"},
threadPoolSize = 2, invocationCount = 5,  timeOut = 2000)
public void updateOne() {
    try {
         long myNo = 1;
         long lockKey=ra.lockRecord(myNo);
         Assert.assertTrue(((long)lockKey==(long)ra.getLockKey(myNo)));
         ra.updateRecord(myNo, "Hans-Dieter", "Meier", lockKey);
         ra.unlockRecord(myNo,lockKey);
         Assert.assertEquals(ra.getRecord(myNo).getFirstName(),"Hans-Dieter");
         Assert.assertEquals(ra.getRecord(myNo).getLastName(),"Meier");
         Assert.assertTrue(ra.getLockKey(myNo)==-1);
     }
     catch (Exception e){
         Assert.fail("should not throw an exception");
     }
}
```

Wie wir sehen, wollen wir den Datensatz mit dem Schlüssel `1` modifizieren. Dazu sperren wir ihn (die Sperrung kontrollieren wir!), ändern ihn, entsperren ihn wieder und überprüfen, ob der Datensatz nun die neuen Werte enthält. Schließlich testen wir noch, ob die Sperre wieder erfolgreich aufgehoben wurde. In diesem Fall wird mit dem von uns definierten Protokoll -1 vom RessourceAdapter zurückgeliefert. Diesen Rückgabewert überprüfen wir mit dem TestNG-Test `Assert.assertTrue`. Eine Exception sollte dabei nicht geworfen werden, weswegen wir dies mit `Assert.fail` überwachen.

Aufruf der Tests: Das Build-Skript

Wir haben die Möglichkeit, mit einem Ant-Skript komfortabel automatisiert die TestNG-Tests laufen zu lassen. Im Beispiel 12-7 nehmen wir das folgende Ant-Skript:

Beispiel 12-7: Unser Build-Skript

```
<project default="reports">
    <property name="root.dir" value="/home/michael/testng"/>
    <path id="cp">
```

Beispiel 12-7: Unser Build-Skript (Fortsetzung)

```
        <pathelement location="${root.dir}/lib/testng-5.3.1-jdk15.jar"/>
        <pathelement location="${root.dir}/classes"/>
        <pathelement location="${root.dir}/test"/>
    </path>

    <taskdef name="testng" classpathref="cp" classname="org.testng.TestNGAntTask"/>

    <target name="ini">
        <delete dir="${root.dir}/classes"/>
        <delete dir="${root.dir}/junit-report"/>
        <delete dir="${root.dir}/testng-report"/>
        <delete>
            <fileset dir="${root.dir}/test" includes="**/*.class"/>
        </delete>
        <mkdir dir="${root.dir}/classes"/>
        <mkdir dir="${root.dir}/junit-report"/>
        <mkdir dir="${root.dir}/testng-report"/>
    </target>

    <target name="create">
       <testng classpathref="cp" groups="create,update" haltOnFailure="true" verbose="2" suite
name="Agile Development" testname="Some agile tests" outputdir="${root.dir}/testng-report">
          <classfileset dir="${root.dir}/test" includes="**/*.class"/>
       </testng>
    </target>

    <target name="compile">
        <javac srcdir="${root.dir}/src" destdir="${root.dir}/classes" classpathref="cp">
           <include name="**/*.java"/>
        </javac>
        <javac srcdir="${root.dir}/test" destdir="${root.dir}/test" classpathref="cp">
           <include name="**/*.java"/>
        </javac>
    </target>

    <target name="reports" depends="ini,compile,create">
      <junitreport todir="${root.dir}/junit-report">
        <fileset dir="${root.dir}/testng-report/Agile Development">
          <include name="*.xml"/>
        </fileset>
        <report format="noframes"  todir="junit-report"/>
      </junitreport>
    </target>

</project>
```

Die Verzeichnisstruktur

Zunächst definieren wir einen grundlegenden absoluten Pfad namens *root.dir*. Von ihm aus adressieren wir die notwendigen Artefakte bzw. zeigen auf Zielverzeichnisse. Wir definieren uns zentral und für das Build-Skript übergreifend einen Klassenpfad namens

cp. Dieser durch diese Variable ausgedrückte Suchpfad setzt sich aus drei einzelnen Pfaden zusammen. Einerseits ist das unsere TestNG-*.jar*-Datei, die die im Ant-Skript genutzten Tasks enthält und mittels `taskdef` angemeldet wird. Daneben werden noch das *classes*-Verzeichnis (in dem alle fachlichen Class-Dateien liegen) und das *test*-Verzeichnis (in dem die Testklassen liegen) dem Klassenpfad hinzugefügt. Der Klassenpfad kommt direkt bei der Definition der TestNG-Tasks zum Einsatz.

```
<property name="root.dir" value="/home/michael/testng"/>
<path id="cp">
    <pathelement location="${root.dir}/lib/testng-5.3.1-jdk15.jar"/>
    <pathelement location="${root.dir}/classes"/>
    <pathelement location="${root.dir}/test"/>
</path>

<taskdef name="testng" classpathref="cp" classname="org.testng.TestNGAntTask"/>
```

Die Targets

Das Default-Target des Skripts ist `reports`, das von `ini`, `compile` und `create` abhängt. Wir schauen uns die Targets in der Reihenfolge ihrer Ausführung an.

Im `ini`-Target wird sichergestellt, dass jeder Testlauf die gleichen Vorbedingungen hat. Dazu werden das *classes*-Verzeichnis und die Reporting-Verzeichnisse komplett gelöscht, um sie anschließend direkt wieder neu anzulegen. Das gewährleistet zudem, dass jederzeit, insbesondere auch vor dem ersten Testlauf überhaupt, die Verzeichnisse angelegt werden. Wir löschen außerdem die Class-Dateien im *test*-Verzeichnis. Da hier auch der Quellcode der Testklasse liegt, löschen wir alle Dateien im Verzeichnis (und rekursiv in seinen Unterverzeichnissen), die über die Endung *.class* verfügen. Gehen Sie immer so vor, um sicherzustellen, dass in den jeweiligen Verzeichnissen auch tatsächlich aktuell generierte Artefakte liegen.

```
<target name="ini">
    <delete dir="${root.dir}/classes"/>
    <delete dir="${root.dir}/junit-report"/>
    <delete dir="${root.dir}/testng-report"/>
    <delete>
        <fileset dir="${root.dir}/test" includes="**/*.class"/>
    </delete>
    <mkdir dir="${root.dir}/classes"/>
    <mkdir dir="${root.dir}/junit-report"/>
    <mkdir dir="${root.dir}/testng-report"/>
</target>
```

Nach der Initialisierung durchläuft das Skript das Target `compile`. Hier werden bei jedem Testdurchlauf alle fachlichen Klassen im *src*-Verzeichnis mithilfe des `javac`-Tasks kompiliert und die Class-Dateien in das *classes*-Verzeichnis gelegt. Das Gleiche gilt für die Testklasse: Diese kompilieren wir neu, wobei das Kompilat im selben Verzeichnis, also in *test*, platziert wird. Wir kompilieren jeweils alle Java-Dateien in sämtlichen Unterverzeichnissen unterhalb der angegebenen.

```
<target name="compile">
    <javac srcdir="${root.dir}/src" destdir="${root.dir}/classes" classpathref="cp">
       <include name="**/*.java"/>
    </javac>
    <javac srcdir="${root.dir}/test" destdir="${root.dir}/test" classpathref="cp">
       <include name="**/*.java"/>
    </javac>
</target>
```

Das nächste Target in der Ablaufssequenz ist `create`. Hier werden nun der eigentliche Test durchgeführt und TestNG genutzt, d.h. wir erstellen die Tests und die zugrundeliegenden Testartefakte. Der TestNG-Task erwartet dabei den von uns zu Beginn zusammengestellten Klassenpfad. Ferner definieren wir, welche Gruppen wir ansprechen möchten. Hier finden wir die beiden Gruppen `create` und `update` aus der Testklasse wieder. Das Attribut `haltOnFailure` lässt das Testskript abbrechen, falls die Tests auf einen Fehler laufen. `Verbose` reichert die Ausgaben von TestNG an, so dass wir über den Testlauf zusätzliche Ergebnisse erhalten. Wir könnten das Attribut auch ganz weglassen oder ihm den Wert *1* zuteilen, was dann weniger Zusatzinformationen ausgeben würde.

Wir weisen der TestSuite den Namen `Agile Development` und dem Test den Namen `Some agile tests` zu. Auch das sind optionale Werte, die von TestNG in seinem Reporting standardmäßig genutzte Felder ersetzen werden. Schließlich definieren wir noch mit `outputdir` das Verzeichnis, in dem der TestNG-Report hinterlegt werden soll. Bei der Verarbeitung beziehen wir uns auf die Klassen im *test*-Verzeichnis, so dass wir mittels `classfileset` das Verzeichnis bekanntgeben müssen.

```
<target name="create">
      <testng classpathref="cp" groups="create,update" haltOnFailure="true" verbose="2"
suitename="Agile Development" testname="Some agile tests" outputdir="${root.dir}/testng-
report">
         <classfileset dir="${root.dir}/test" includes="**/*.class"/>
      </testng>
   </target>
```

Das Reporting

Nun sind die als Vorbedingung deklarierten Targets durchlaufen, so dass endlich `reports` gestartet werden kann. Dieser Target nutzt den JUnit-Task `junitreport`, um aus den Testergebnissen einen grafischen Report zu generieren. TestNG legt seine Testergebnisse in einem standardisierten, von JUnit weiterverarbeitbaren Format ab. So ist es möglich, die TestNG-Testergebnisse nicht nur im TestNG-eigenen Protokoll vorliegen zu haben, sondern durch einen weiteren Verarbeitungsschritt in das bekannte JUnit-Reportformat zu transferieren. Das kann hilfreich sein, wenn Sie das Format nutzen möchten, weil es in Ihrer Abteilung bekannt ist oder Ihnen inhaltlich oder optisch mehr zusagt. Der JUnit-Report wird in das Verzeichnis *junit-report* gelegt.

```
<target name="reports" depends="ini,compile,create">
  <junitreport todir="${root.dir}/junit-report">
    <fileset dir="${root.dir}/testng-report/Agile Development">
```

```
        <include name="*.xml"/>
      </fileset>
      <report format="noframes"  todir="junit-report"/>
    </junitreport>
  </target>
```

Die Ausführung

Nun möchten wir das Ant-Skript ausführen. Die Eingabe eines einfachen `ant` im Verzeichnis, in dem das Build-Skript liegt, reicht. Was dann passiert, zeigt Beispiel 12-8.

Beispiel 12-8: Ausführung des Ant-Skripts

```
michael@BlueGene:~/testng$ ant
Buildfile: build.xml

ini:
   [delete] Deleting directory /home/michael/testng/classes
   [delete] Deleting directory /home/michael/testng/junit-report
   [delete] Deleting directory /home/michael/testng/testng-report
    [mkdir] Created dir: /home/michael/testng/classes
    [mkdir] Created dir: /home/michael/testng/junit-report
    [mkdir] Created dir: /home/michael/testng/testng-report

compile:
    [javac] Compiling 5 source files to /home/michael/testng/classes
    [javac] Compiling 1 source file to /home/michael/testng/test

create:
   [testng] [Parser] Running:
   [testng]   Agile Development
   [testng]
   [testng] PASSED: createThree
   [testng] PASSED: createOne
   [testng] PASSED: createTwo
   [testng] PASSED: createTwo
   [testng] PASSED: createTwo
   [testng] PASSED: createTwo
   [testng] PASSED: createTwo
   [testng] PASSED: createTwo
   [testng] PASSED: updateOne
   [testng] PASSED: updateOne
   [testng] PASSED: updateOne
   [testng] PASSED: updateOne
   [testng] PASSED: updateOne
   [testng]
   [testng] ===============================================
   [testng]     Ant test
   [testng]     Tests run: 13, Failures: 0, Skips: 0
   [testng] ===============================================
   [testng]
   [testng]
   [testng] ===============================================
```

Beispiel 12-8: Ausführung des Ant-Skripts (Fortsetzung)

```
   [testng] Agile Development
   [testng] Total tests run: 13, Failures: 0, Skips: 0
   [testng] ===============================================
   [testng]

reports:
[junitreport] Processing /home/michael/testng/junit-report/TESTS-
TestSuites.xml to /home/michael/testng/junit-report/junit-noframes.html
[junitreport] Loading stylesheet jar:file:/home/michael/ant/apache-ant-1.7.0/lib/ant-
junit.jar!/org/apache/tools/ant/taskdefs/optional/junit/xsl/junit-noframes.xsl
[junitreport] Transform time: 976ms

BUILD SUCCESSFUL
Total time: 3 seconds
```

Die Ausgabe auf der Konsole signalisiert uns einen erfolgreichen Testzyklus. Nach der Initialisierung wurden fünf Artefakte kompiliert, ehe der eigentliche Test durchgeführt wurde. 13 Testmethoden wurden erfolgreich aufgerufen, ohne dass ein Test fehlschlug. Die Zahl 13 resultiert aus der Anzahl der Testmethoden und der dort von uns jeweils spezifizierten Ausführungsanzahl. Um weitere Informationen über das Testergebnis zu erhalten, schauen wir uns den TestNG-Report an.

Darstellung der Ergebnisse

Im Verzeichnis *testng-report* finden wir zu unserer TestSuite eine generierte *index.html*-Startseite. Diese zeigt uns das Testergebnis auf einen Blick (siehe Abbildung 12-1). Wir haben eine TestSuite namens `Agile Development` gestartet, von der 13 Testmethoden erfolgreich durchlaufen wurden. Es gab keine fehlerhaften Tests und keine Testmethoden, die übersprungen wurden (hier würden zum Beispiel Methoden gezählt, die aufgrund ihrer spezifizierten Abhängigkeit nicht durchgeführt wurden, wenn ein Test in einer zuvor durchlaufenen Methode fehlschlug).

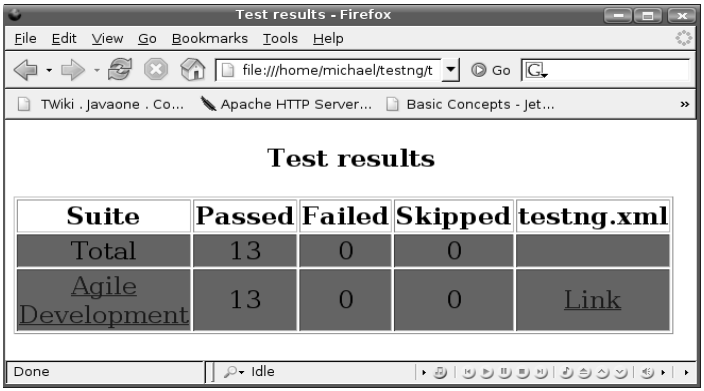


Abbildung 12-1: Ein TestNG-Report

Ferner wird ein Link auf eine Datei namens *testng.xml* angeboten. Diese wurde von TestNG für uns erstellt und enthält die Informationen, die wir TestNG über das Ant-Skript mitgeteilt haben, so zum Beispiel die Gruppen, den *verbose*-Grad, den Testnamen und die Testklasse (siehe Beispiel 12-9). Der Eintrag `annotations=»JDK«` besagt, dass wir tatsächlich Java 5-Annotations nutzen. Alternativ könnten wir auch Java 1.4 einsetzen und die TestNG-Metainformationen in eingebetteten JavaDoc-Kommentaren angeben. Das sähe dann zum Beispiel so aus:

```
/**
*@testng.test groups = "create"
*     dependsOnMethods = "createOne"
*/
```

Ebenfalls interessant der Eintrag `junit=»false«`. TestNG kann auch JUnit-Tests laufen lassen, was wir hier nicht wollen.

Beispiel 12-9: Das TextNG-XML-Skript

```
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite thread-count="5" verbose="2" name="Agile Development" annotations="JDK">
  <test name="Some agile tests" junit="false">
    <groups>
      <run>
        <include name="create"/>
        <include name="update"/>
      </run>
    </groups>
    <classes>
      <class name="com.huettermann.testng.test.Client"/>
    </classes>
  </test>
</suite>
```

Erinnern Sie sich: TestNG kann auch über Konsole mit Hilfe dieser *testng.xml*-Datei gestartet werden. Die Generierung dient nur unserer Information. Wir müssten diese Datei erstellen, wenn wir TestNG nicht aus Ant heraus nutzen würden (was ich aber deutlich komfortabler finde).

Der Name der Suite verweist auf eine weitere Seite mit ausführlichen Informationen, die wir uns nun anschauen (siehe Abbildung 12-2). Auf dieser Seite sehen wir nun links oben eine Zusammenfassung der durchgeführten Tests: Wir haben eine Testklasse, die aus vier Testmethoden besteht. Daneben wird protokolliert, dass wir zwei Gruppen genutzt haben, die wir uns über Hyperlink anschauen können.

Darunter wird unten links das Testergebnis kompakt dargestellt. Wir haben 13 Läufe von Testmethoden, alle waren erfolgreich, keiner war fehlerhaft und alle wurden tatsächlich durchgeführt (hier würden zum Beispiel Methoden gezählt, die aufgrund ihrer spezifizierten Abhängigkeit nicht durchgeführt wurden, wenn ein Test in einer zuvor durchlaufenen Methode fehlgeschlagen ist). Auf der rechten Seite sehen wir nun die Details zum Testlauf. Die einzelnen Zeilen der Tabelle, für jeden Methodendurchlauf eine, sind grün hinterlegt, da die Tests erfolgreich durchlaufen wurden.

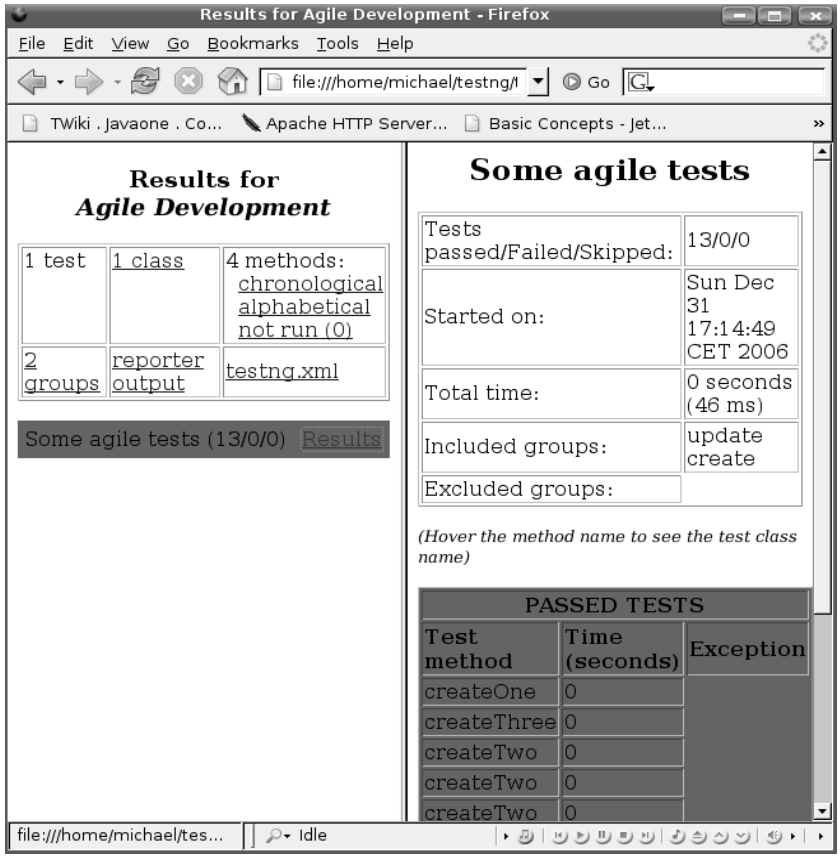


Abbildung 12-2: Der TestNG-Detailreport

Spannend ist ein Blick auf den chronologischen Ablaufbericht. Diesen erhalten wir durch Klick oben links (*4 methods: chronological*, siehe Abbildung 12-3). Dadurch ändert sich der Inhalt des rechten Teils der Seite, und wir sehen alle TestNG-Methodenaufrufe, sowohl die der Testmethoden als auch die der Initialisierungsmethoden. Auf diesem Weg können wir auch sehen, dass diese (wie zum Beispiel *setUpSuite*) tatsächlich zu Beginn einmalig gelaufen sind. Scrollen wir auf der Seite weiter nach unten und nach rechts, sehen wir auch, welche Testmethoden von welchen Threads aufgerufen werden. Auf diese Weise kann die korrekte nebenläufige Ablaufsequenz überprüft werden.

Die Werte unserer Testmethoden (Methodenname und Thread bzw. sein Name) sind Tabelle 12-2 zu entnehmen.

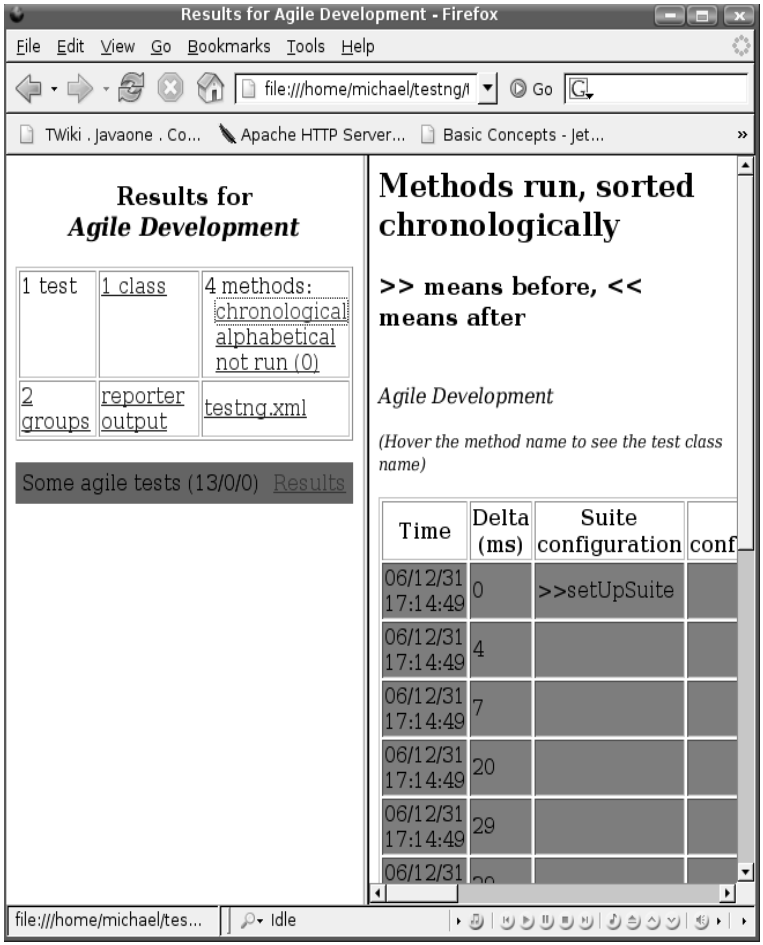


Abbildung 12-3: Chronologische Übersicht über die Aufrufe

Tabelle 12-2: Nebenläufigkeit durch separate Threads

Testmethode	Thread
`createThree`	`main@26399554`
`createOne`	`main@26399554`
`createTwo`	`pool-1-thread-1@15020296`
`createTwo`	`pool-1-thread-4@27189676`
`createTwo`	`pool-1-thread-3@16130931`
`createTwo`	`pool-1-thread-2@12773951`
`createTwo`	`pool-1-thread-6@12719253`
`createTwo`	`pool-1-thread-5@32635808`
`updateOne`	`pool-2-thread-2@21171036`

Tabelle 12-2: Nebenläufigkeit durch separate Threads (Fortsetzung)

Testmethode	Thread
updateOne	pool-2-thread-1@11626165
updateOne	pool-2-thread-2@21171036
updateOne	pool-2-thread-1@11626165
updateOne	pool-2-thread-2@21171036

Schließlich können wir noch einen Blick ins *junit-report* Verzeichnis wagen, in dem der JUnit-konforme Testreport liegt (siehe Abbildung 12-4).

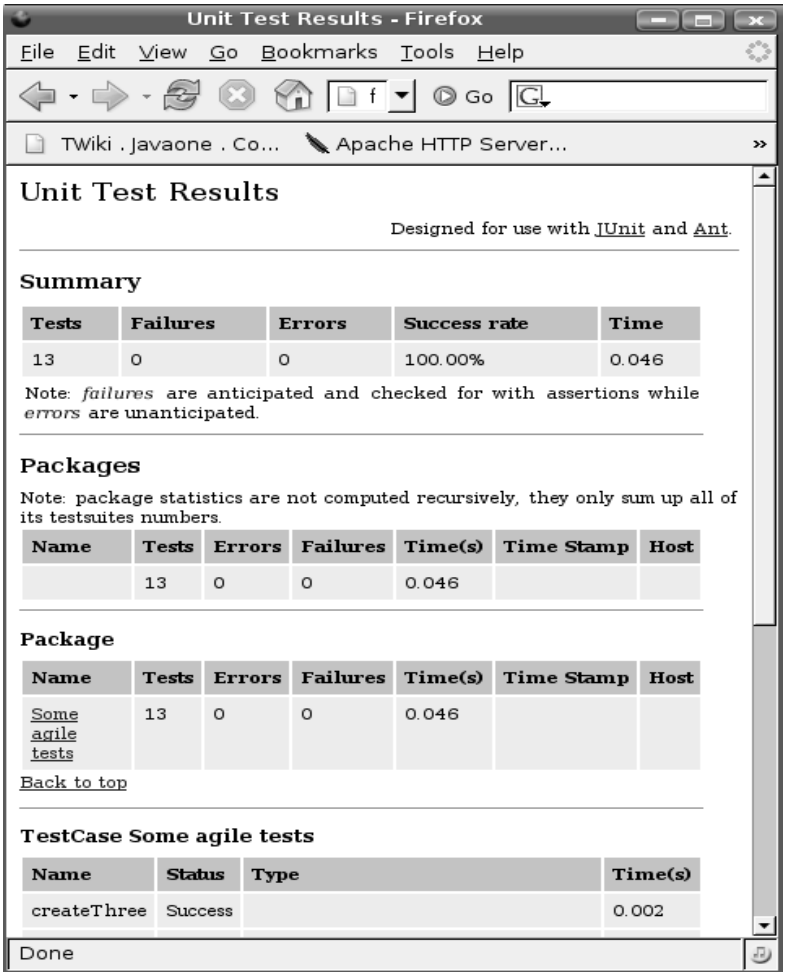


Abbildung 12-4: TestNG Report, nach JUnit portiert

Zusammenfassung

TestNG ist ein Framework, mit dessen Hilfe Sie Ihre Komponenten testen können. Es steht in Konkurrenz zu JUnit und ist in manchen Teilen mächtiger, z.B. im Bereich Nebenläufigkeit, Abhängigkeiten und Gruppierung von Tests. Sie müssen also auch bei fortgeschrittenen Anwendungsszenarien nicht auf das Testen Ihrer Komponenten verzichten.

Im Beispiel haben wir exemplarisch Eigenschaften von TestNG illustriert und erläutert, wie Sie derart fortgeschrittene Tests mit TestNG organisieren können. Davon unabhängig ist selbstverständlich auch immer das Design der Testklassen zu sehen. So sollten Sie beispielsweise beim Design von fachlichen Klassen, die nebenläufige Aktivitäten verfolgen, wohldefinierte zentrale Synchronisierungspunkte einführen, die Sie isoliert testen können.

TestNG ist frei verfügbar und lässt sich in Ihren Build-Prozess integrieren (Nutzung per Ant) und aus einer IDE nutzen (beispielsweise als Plugin für Eclipse oder als Basisfunktionalität in IntelliJ IDEA ab Version 7, Meilenstein 2). TestNG bietet sogar die Möglichkeit, JUnit Tests auszuführen. Testergebnisse werden ausführlich dokumentiert. Das TestNG-Reportdokument kann in ein JUnit-Reportdokument transferiert werden.

KAPITEL 13

GUIs testen mit Jemmy

Java-Client-(Standalone-)Applikationen haben eine große Verbreitung. Dabei spielt die Swing-Bibliothek eine große Rolle, um komplexe Oberflächen (*User Interface, UI*) flexibel erstellen zu können. Komponententests zu schreiben bedeutet prinzipiell, alles zu testen! Wir haben uns bisher auf Funktionalität konzentriert, die nicht über ein GUI verfügt. Doch was machen wir mit GUI-Anwendungen? Benutzerschnittstellen bzw. die Interaktionen darauf müssen auch getestet werden. Hier gibt es verschiedene Ansätze und Werkzeuge, von denen Jemmy eines ist. Es hat weite Akzeptanz gefunden, und zwar zu Recht, wie wir im Folgenden sehen werden. Bevor wir uns ausführlich mit Jemmy auseinandersetzen, gehen wir auf die grundlegenden Konzepte von Swing für Java und Herausforderungen beim Testen von Swing-Anwendungen ein.

Swing

Swing ist ein API zur Programmierung von grafischen Benutzeroberflächen. Die von Sun Microsystems entwickelte Bibliothek ist seit Java 1.2 Bestandteil seiner Standardedition. Swing gehört zu den Java Foundation Classes, den Bibliotheken zur Entwicklung von grafischen Benutzerschnittstellen, zu denen auch Java2D und das Accessibility-API gehören. Swing baut auf AWT auf, dem Abstract Windowing Toolkit, einer älteren, weniger mächtigen Bibliothek.

Swing zeichnet sich durch seine Flexibilität und Modularität aus. Es stellt unterschiedliche visuelle Komponenten zur Verfügung. Diese reichen von einfachen Labels für die Darstellung von Text bis hin zu komplexen Bäumen und Tabellen.

Swing zeichnet sich durch seine Plattformunabhängigkeit aus. Die Swing-Komponenten sind in Java geschrieben und entsprechend auf jede Plattform portierbar, für die es eine Java-Laufzeitumgebung gibt.

Swing und das AWT

Gewöhnlich sind Swing Komponenten von der Basisklasse JComponent abgeleitet, die wiederum von AWT Container ableitet. So kann Swing als Schicht (Layer) oberhalb des AWT begriffen werden. Swing-Komponenten beginnen mit einem *J* (wie JComponent), AWT-Komponenten dagegen kommen ohne ein *J* aus. Viele Swing-Komponenten haben AWT-Gegenstücke (beispielsweise entspricht JButton in Swing Button in AWT).

Swing-Komponenten werden als leichtgewichtig angesehen, wohingegen AWT-Komponenten als schwergewichtig gelten. Alle leichtgewichtigen Komponenten werden in einem schwergewichtigen Container aufbewahrt. Es werden heutzutage gewöhnlich nur noch Swing-Komponenten eingesetzt. AWT-Anwendungen sind nicht mehr zeitgemäß. Aufgrund der internen Struktur ist davon abzuraten, Swing und AWT-Komponenten gleichzeitig zu nutzen.

Sie sollten möglichst nur die Standardkomponenten nutzen. Durch die Flexibilität von Swing können Sie die Komponenten zwar beliebig erweitern und modifizieren bzw. auch ganze eigene neue schreiben (zum Beispiel durch Zuweisung von eigenen Oberflächen und Modellen), aber Sie sollten sich über die Konsequenzen im Klaren sein. Ein agiles Team hält sich lieber an Standards. Proprietäre Komponenten zu pflegen und weiterzuentwickeln ist nämlich zeitintensiv. Auch ist es denkbar, dass die Komponente durch individuelle Entwicklung nicht mehr zum eingestellten Look and Feel passt. Das kann zu Regressions- und Migrationsproblemen führen. Möchten Sie beispielsweise die Java-Laufzeitumgebung wechseln, benötigen Sie umfangreiche Test Suites, um das Verhalten vor der Umstellung mit dem Verhalten nach der Umstellung vergleichen zu können. Eigene Komponenten können außerdem dazu führen, dass sie, wenn sie unzureichend entwickelt sind, nicht von Werkzeugen genutzt werden können. Werkzeuge nutzen die JavaBeans-Eigenschaften. Swing-Komponenten sind JavaBeans. Falls Sie eigene Komponenten schreiben, müssen Sie darauf achten, dabei die JavaBeans-Spezifikationen zu erfüllen.

Die Mächtigkeit von Swing wird häufig unterschätzt. Allein durch den Umfang der Komponenten und ihre Mächtigkeit kommen umfangreiche Anwendungen zu Stande. Dazu kommen Konzepte wie die LayoutManager für die Positionierung der Komponenten auf dem UI und die Listener, um Abhängigkeiten und Interaktionen der Benutzer behandeln zu können.

Durch Implementierungen des *Observer Patterns* ist eine lose Kopplung der Komponenten anzustreben, was ebenfalls innerhalb von Swing selbst beherzigt wird. Auf dem Client (dem Arbeitsplatzrechner bzw. genauer: dem Teil der Java Anwendung, der darauf läuft) werden gewöhnlich mehrere Threads aktiv Arbeit verrichten. Das Öffnen von eigenen, neuen Threads innerhalb der Anwendung ist im EJB-Container nicht zu empfehlen und im Webcontainer höchstens geduldet bzw. gar nicht notwendig. In einer Standalone-Anwendung für Java auf dem Client ist dies gang und gäbe.

MVC

Model-Delegate ist die Variante des bekannten *MVC-Architekturmusters*, das in Swing Anwendung findet. In Swing werden visuelle Komponenten MVC-gehorchend in drei logische Teile unterteilt.

Model
: Das Model repräsentiert den *Zustand* der Komponente. Das ist beispielsweise der Status eines Buttons. Das Model kennt weder View noch Controller; es kommuniziert aber mit ihnen, indem es sogenannte Broadcasts bzw. Events aussendet.

View
: Der View umfasst das *Aussehen* der Komponente. Das kann ein gedrückter Button sein, eine Farbe oder ein Font. Der View ist dafür verantwortlich, die Darstellung der Komponente nach außen zum Anwender hin aktuell und zum Status der Anwendung und des Model synchron zu halten. So empfängt es Benachrichtigungen vom Model oder Controller über Zustandsänderungen.

Controller
: Der Controller bestimmt, welche Aktionen ausgeführt werden, wenn eine Komponente benutzt wird. Er kontrolliert die Ablauflogik der Swing-Anwendung. Wird beispielsweise ein Button gedrückt, benachrichtigt der Controller den View, dass sich das Aussehen des Buttons ändern soll.

Controller, Model und View sind rein logisch weitestgehend entkoppelt. In den Swing-Klassen selbst befinden sich View und Controller in einem Objekt, dem UI Delegate. So ist es eigentlich genauer, die Swing-Architektur als *Model-Delegate* zu bezeichnen.

Multi-Threading

Swing selbst erstellt »nach Belieben« eigene Threads, um Aktivitäten nebenläufig durchzuführen. Das bleibt dem Benutzer und Entwickler verborgen. Daneben ist es in vielen Konstellationen den Entwicklern dringend zu empfehlen, selbst Threads zu erstellen und zu nutzen. Werden zeitintensive Aktivitäten durchgeführt, beispielsweise eine Datenbank ausgelesen oder ein Server kontaktiert, dann sollte das unbedingt in einem separaten Thread geschehen. Auf diese Weise reagiert das UI weiter auf Interaktionen und ist für den Nutzer nicht geblockt.

Da der Nutzer also unabhängig von der Applikation mit dem UI interagieren soll, wird beim Starten der Applikation ein Thread erzeugt, der UI-Interaktionen überwacht und steuert. Dieser Thread überprüft kontinuierlich die *Event Queue*, die interne Warteschlange, auf neue Events wie Mouse Events (Maus wurde gedrückt) oder Key Events (Tastatur bedient) hin. Der AWT-Thread (*Event Dispatch Thread*) nimmt die auflaufenden Events aus der Queue und wendet sie auf die Anwendung an. Wird beispielsweise auf einen Button gedrückt, überträgt der AWT-Thread den Mausklick zum JButton. Alle Interessenten werden in diesem Fall über den Callback `actionPerformed` über diesen Vorgang benachrichtigt.

Sie können die Event Queue auch mit eigenen Events füttern, oder auch die ganze Event Queue durch eine eigene ersetzen. Wenn Sie so etwas machen, sollten Sie aber genau überlegen, ob der jeweilige Nutzen eines solchen Eingriffs den Aufwand und das Risiko rechtfertigt.

Der AWT-Thread ist auch für das *Repainting* des UI zuständig. Repainting bedeutet beispielsweise, dass sich die Oberfläche zeichnet, wenn Sie das UI in den sichtbaren Bereich auf Ihrem Monitor ziehen oder sich der Zustand der Komponente geändert hat. Repainting geschieht entweder (aus Sicht des Entwicklers) automatisch oder durch einen expliziten Aufruf der `repaint`-Methode. Alle Bestandteile der Oberfläche sollten im AWT-Thread laufen.

Swing-Komponenten sind, bis auf wenige Ausnahmen, nicht *thread-safe*. Das bedeutet, dass der Entwickler selbst dafür sorgen muss, dass der Zugriff auf die Komponenten entsprechend synchronisiert und für den Zugriff aus mehreren Threads heraus ermöglicht wird. Insbesondere heißt das auch, dass Änderungen an der Komponente, die den View betreffen, vom AWT-Thread durchgeführt werden müssen. Wenn derselbe Thread die Komponente aktualisiert und anzeigt, ist die Darstellung konsistent. Macht das ein anderer Thread ohne besondere Maßnahmen zu ergreifen, kann es zu Inkonsistenzen oder sogar zu Dead-Locks (Threads blockieren sich) kommen und einem Absturz bzw. Einfrieren der Anwendung.

Um dies zu verhindern, müssen Sie `SwingUtilities.invokeLater()` verwenden, um einen Thread in den Event-Dispatch-Thread zu »patchen«. Das sieht so aus:

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        // do some UI stuff
    }
});
```

Testen von Swing-Anwendungen

Was gibt es für Möglichkeiten zum Testen von Swing-Anwendungen? Nun, warum entwickeln wir sie nicht auch testgetrieben, genauso wie wir andere Java-Komponenten realisieren? Es spricht nichts dagegen.

Hier sollte nur von vornherein klar sein, was genau getestet werden soll. Häufig ist es gar nicht notwendig, konkrete Benutzerinteraktionen auf dem UI in einem Test nachzuvollziehen. Es ist ein weit verbreitetes Antimuster, zu denken, der Test müsse genau so aussehen wie eine Benutzerinteraktion. Es reicht häufig aus, die Komponenten umfangreichen Unit-Tests zu unterziehen. In dieser Form wird das UI als hauchdünne architektonische Schicht über den eigentlichen Klassen (den *Treibern*) begriffen (*Humble Dialog Box*). Diese Treiber sind umfangreiche Komponenten, die eine lose Kopplung an das UI aufweisen und somit weitestgehend genauso getestet werden können, wie wir das schon in den JUnit- und TestNG-Abschnitten gesehen haben.

Sie können Jemmy auch einsetzen, um im Rahmen von Akzeptanztests mit dem Werkzeug Fit Komponenten auf Ihrer Oberfläche zu bedienen. Die hier illustrierte Nutzung von Jemmy im Kontext von Unit-Tests ist ein Szenario von vielen.

Die konzeptionelle Entkopplung beim Testen der Darstellung und der Logik darunter setzt also beispielsweise voraus, dass ein gewisser Button auf dem UI korrekt gedrückt wurde. Dann wird die korrekte Verarbeitung *nach* dem Drücken getestet, also ob der hinter dem Button platzierte Code die richtige Aktion durchführt. Wir testen also Model und Controller, nicht den View.

Diese Tests stoßen aber an Grenzen: Beispielsweise ist das Testen von Usability problematisch. Es kann nicht automatisch durchgeführt werden, da keine Ist-Usability gegen eine Soll-Usability automatisiert verglichen werden kann. Weitere damit zusammenhängende Aspekte sind fachliche Konsistenz, UI-Vorgaben, das Layout usw.

Neben den reinen Unit-Tests und ergänzenden manuellen Akzeptanztests sind also in der Praxis immer Szenarien anzutreffen, die explizites Testen der Anwendung erfordern, automatisches Testen der Oberfläche *und* der Logik dahinter.

Capture Replay

Eine Möglichkeit, die Anwendung *als Ganzes* zu testen, ist das Aufnehmen von Interaktionen, um sie anschließend wieder abspielen zu können. Diesen Vorgang nennt man *Capture Replay* (CR). Bei Swing-Anwendungen ist hier der kommerzielle *Mercury WinRunner* eine Option (siehe *http://www.mercury.com*).

Dieses Abspielen kann mit Überprüfungen angereichert werden, um bestimmte Zustände zu konkreten Zeitpunkten zu überprüfen. Ich habe einmal ein derartiges CR-Werkzeug geschrieben, da es in einem konkreten Anwendungsfall durchaus hilfreich war. Bedenken Sie aber, dass die Aufnahmen recht fragil sind. Je nach Konzept des Test-Frameworks kann eine Änderung der Reihenfolge der Komponenten auf dem UI oder eine Änderung des Namens einer Komponente das aufgenommene Skript brechen.

Neben der Reihenfolge der Komponenten und ihrer eindeutigen Identifikation über den Namen sind noch weitere Verfahren denkbar, um die Komponenten, auf denen gerade eine Aktion durchgeführt wurde, zu identifizieren. Die Komponente wird dabei mit der Aktion abgespeichert und zu einem späteren Zeitpunkt in der gleichen Sequenz wie bei der Aufnahme erneut auf die Zielanwendung abgespielt.

Beim Abspielen kann direkt die Klasse `java.awt.Robot` zum Einsatz kommen. Sie bewegt den Cursor und führt tatsächliche Interaktionen auf der Anwendung aus, also zum Beispiel auch Eingaben von Buchstaben. Ein anderer Ansatz ist es, eigene Events zu erzeugen, die Sie dann in den Event Dispatch-Thread speisen.

Codezentrierte Tests

Eine andere Herangehensweise, um die Anwendung als Ganzes zu testen, ist es, die visuellen Komponenten aus einem Testcode heraus zu bedienen. Hier wird keine Interaktion aufgenommen, sondern eine Testklasse geschrieben. Jemmy verfolgt diesen Ansatz, den wir uns im weiteren Verlauf anschauen.

Einführung in Jemmy

Jemmy ist ein Testwerkzeug, das von Sun Microsystems entwickelt wurde und frei verfügbar ist. Es entstand im Rahmen des *NetBeans*-Projekts, ist aber nun unabhängig davon verfügbar. Mit Jemmy können Swing-Oberflächen umfassend getestet werden.

Installation

Die Installation ist denkbar einfach. Auf *http://jemmy.netbeans.org* kann die *.jar*-Datei heruntergeladen werden. Daneben gibt es noch eine *.zip*-Datei, die neben dem *.jar* auch die JavaDocs und den Quellcode enthält. Die *.jar*-Datei legen Sie in den Klassenpfad, und schon kann es losgehen.

Konzepte

Jemmy ist in der Lage, alle Swing-Komponenten zu finden. Dazu gehören Containerkomponenten wie Frames und Dialoge (die selbst wieder Komponenten enthalten), aber auch die einzelnen Komponenten wie Buttons und Eingabefelder selbst. Zur Identifikation der Komponenten, zum Finden der richtigen Instanz im gesamten Objektgeflecht, gibt Jemmy dem Entwickler unterschiedliche Möglichkeiten an die Hand. Hier können beispielsweise der Index der Komponente (die wievielte von einem bestimmten Typ auf der Oberfläche es ist), der Name (`name`-Attribut) der Komponente oder das Label (textuelle Beschreibung auf dem UI, wobei bei Lokalisierung der Oberfläche beispielsweise der Name vorzuziehen ist) genutzt werden. Beliebige eigene Algorithmen zum Auffinden von Komponenten können entwickelt werden. Es ist dazu lediglich ein sehr übersichtliches `ComponentChooser`-Interface zu implementieren.

Jemmy findet die Komponenten im Objektgeflecht selbstständig, wir müssen beim Design der funktionalen Klassen keine weiteren Vorkehrungen treffen. Gewöhnlich werden innerhalb der Testklassen keine Methoden auf den Komponenten direkt aufgerufen, sondern nur sogenannte *Jemmy-Delegates*, die Jemmy zur Verfügung stellt und die Aktionen auf die Swing-Komponenten delegieren. Es ist aber trotzdem möglich, auch direkt auf der Swing-Komponente zu arbeiten.

Im Rahmen der Tests können eigene Events in den Event Dispatch-Thread von AWT gelegt werden. Auf diese Weise lässt sich beispielsweise ein Drücken eines Buttons anstoßen. Der Umgang mit Komponenten kann *thread-safe* erfolgen.

Jemmy selbst muss mit einem Werkzeug für Komponententests kombiniert werden. Es ist unabhängig von einem konkreten Werkzeug. So kann es zum Beispiel mit JUnit oder TestNG (aber auch mit Fit) gleichermaßen genutzt werden. Komponenten können mit Jemmy gefunden und manipuliert werden. Dem eingesetzten Testwerkzeug wie JUnit ist es überlassen, die Komponenten nun tatsächlich zu testen, also einen Sollwert und einen Istwert gegenüberzustellen.

Jemmy ist ein Entwicklerwerkzeug. Der Entwickler schreibt vor und während der Realisierung der funktionalen UI Komponenten die Testklassen. Die Testklassen treiben auch hier die funktionalen Klassen, wie wir es schon bei den herkömmlichen Unit-Tests herausgearbeitet haben, und helfen mit, das Design zu verbessern.

Gegen Änderungen am funktionalen Code ist auch Jemmy nicht gefeit. Testcode und funktionaler Code müssen verzahnt zusammen fortgeschrieben werden. Soll Jemmy beispielsweise eine visuelle Komponente anhand ihres Labels oder Namens finden, führt eine Änderung dieses Werts unausweichlich zu einem negativen Ergebnis. Das Suchen über Index (Position der Komponente relativ zu den anderen) ist direkt gebrochen, wenn eine Komponente vom gleichen Typ im Objektbaum vor der zu suchenden eingefügt wird.

Ein ausführliches Beispiel: Testen der Filterung bei Tabellen

Wir schauen uns nun ein konkretes Beispiel an. Wir möchten die mit Java 6 neu bereitgestellte Funktionalität der Swing-Komponente `JTable` testen, Zeilen einer Tabelle nach einem regulären Ausdruck zu filtern. Auf diese Weise können einzelne Zeilen, die nicht dem Ausdruck genügen, ausgeblendet und vor dem Benutzer versteckt werden.

Da wir bereits ausführlich die manuelle Ausführung von Unit-Tests beschrieben haben, werden wir uns hier auf die Funktionalität von Jemmy konzentrieren und die eigentliche Ausführung der Tests in die Obhut einer IDE unseres Vertrauens geben. Wir wählen hierfür IntelliJ IDEA.

Wir erstellen zwei Klassen: `UITest` sowie `TableFilter`. Beide liegen im Verzeichnis *com.huettermann.jemmy*. `TableFilter` ist die funktionale, zu testende Klasse und `UITest` der Test dafür.

Die Ausgangsklasse

FilterTable ist direkt kompilierbar, da sie ausschließlich die Java 6-Distribution und keine weiteren Bibliotheken nutzt. Schauen wir uns zunächst diese Basisklasse `TableFilter` an (siehe Beispiel 13-1).

Beispiel 13-1: TableFilter, die zu testende Klasse

```
package com.huettermann.jemmy;

import javax.swing.*;
import javax.swing.event.DocumentListener;
```

Beispiel 13-1: TableFilter, die zu testende Klasse (Fortsetzung)

```
import javax.swing.event.DocumentEvent;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableModel;
import javax.swing.table.TableRowSorter;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

/**
 * @author Michael Huettermann
 */
public class TableFilter extends JFrame {
    private JTextField myFilter;
    private TableRowSorter<TableModel> sorter;

    //put all stuff straight into constructor
    TableFilter(String title) {

        //title and polishing
        super(title);
        setDefaultCloseOperation(EXIT_ON_CLOSE);

        //basic table model, no content
        TableModel tm = new DefaultTableModel(5, 2);
        JTable table = new JTable(tm);
        table.setName("myTable");
        table.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

        //create and install the row sorter
        sorter = new TableRowSorter<TableModel>(tm);
        table.setRowSorter(sorter);

        //put table into JScrollpane to allow scrolling
        JScrollPane sp = new JScrollPane(table);
        //polishing
        sp.setPreferredSize(new Dimension(400, 100));

        //put scrollpane into northern region
        getContentPane().add(sp, BorderLayout.NORTH);

        //a panel for the enter components
        JPanel myPanel = new JPanel();
        myPanel.add(new JLabel("Filter:"));
        myFilter = new JTextField(20);
        myPanel.add(myFilter);

        //add filter listener
        myFilter.getDocument().addDocumentListener(
                new DocumentListener() {
                    public void changedUpdate(DocumentEvent e) {
                        newFilter();
                    }
```

Beispiel 13-1: TableFilter, die zu testende Klasse (Fortsetzung)

```
                    public void insertUpdate(DocumentEvent e) {
                        newFilter();
                    }

                    public void removeUpdate(DocumentEvent e) {
                        newFilter();
                    }
                });

        //exit button
        JButton exitButton = new JButton("Exit");
        exitButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                //normal end
                dispose();
            }
        }
        );

        //add button to panel, we will not win an usability award for this
        myPanel.add(exitButton);

        //filter panel into the center region
        getContentPane().add(myPanel, BorderLayout.CENTER);

        // Resize and display
        pack();
        setVisible(true);
    }

    /**
     * Start example stand alone.
     *
     * @param args start
     */
    public static void main(String[] args) {
        new TableFilter("Agile Development with Jemmy");
    }

    /**
     * update the row filter regular expression according to text field
     */
    protected final void newFilter() {
        try {
            sorter.setRowFilter(RowFilter.regexFilter(myFilter.getText(), 0));
        } catch (java.util.regex.PatternSyntaxException e) {
            /* */
        }
    }
}
```

Führen wir Beispiel 13-1 über die IDE aus, dann erscheint eine leere Basistabelle mit zwei Spalten und fünf Zeilen (siehe Abbildung 13-1). Unterhalb der Tabelle befinden sich ein Eingabefeld und ein Button. Es handelt sich bei der Tabelle um eine Basistabelle, die keinen Inhalt hat. Wir können nun ganz einfach beliebigen Inhalt der Tabelle hinzufügen (siehe Abbildung 13-2).

Geben wir nun in dem Filterfeld einen Wert ein, werden direkt in Echtzeit ein Filter auf die Tabelle angewendet und nur noch die Zeilen dargestellt, deren erste Spalte mit dem eingegebenen Filter-String beginnt (siehe Abbildung 13-3). Im Beispiel haben wir fünf Zeilen eingefügt, wobei die dritte, vierte und fünfte Zeile jeweils mit »aaa« beginnen. Geben wir in das Filterfeld nun *aaa* ein, werden tatsächlich nur noch genau diese Zeilen dargestellt und die beiden ersten, die nur zweimal den Buchstaben *a* besitzen, ausgeblendet.

Abbildung 13-1: Die Basistabelle

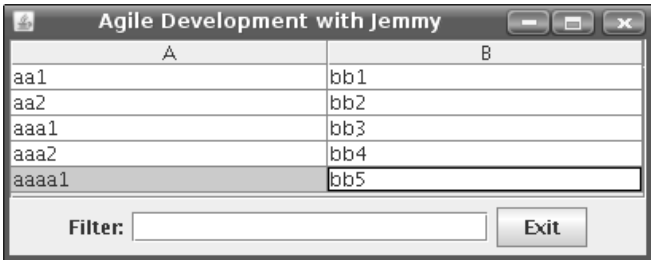


Abbildung 13-2: Werte hinzufügen

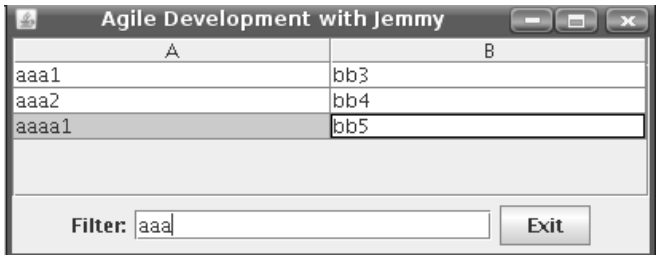


Abbildung 13-3: Filterung

Im Quellcode sieht das so aus: Wir betten unsere Tabelle in ein JFrame ein und führen der Einfachheit halber die Logik (Erstellung und Verwendung der Komponenten) direkt im Konstruktor durch. Neben dem üblichen Vorgehen, auf das wir nicht näher eingehen wollen, ist der Java 6-spezifische Teil von Interesse, in dem wir einen Tabellen-Sortierer erstellen und der Tabelle zuweisen:

```
sorter = new TableRowSorter<TableModel>(tm);
table.setRowSorter(sorter);
```

An das Eingabefeld `myFilter` vom Typ JTextField hängen wir ein `DocumentListener`, um über Eingaben und Änderungen in dem Feld (also auch Löschen von Zeichen) direkt informiert werden. Dieser anonyme `DocumentListener` implementiert dazu drei Methoden: `changedUpdate` (wird bei Änderungen benachrichtigt), `insertUpdate` (wird bei Einfügen eines Zeichens benachrichtigt) und `removeUpdate` (Callback zur Benachrichtigung bei Löschen eines Zeichens). In allen drei Fällen rufen wir die Methode `newFilter` auf, die dem von uns erstellen TableRowSorter `sorter` einen neuen Filter zuweist.

```
myFilter.getDocument().addDocumentListener(
        new DocumentListener() {
            public void changedUpdate(DocumentEvent e) {
                newFilter();
            }

            public void insertUpdate(DocumentEvent e) {
                newFilter();
            }

            public void removeUpdate(DocumentEvent e) {
                newFilter();
            }
        });
```

Dieser Filter schränkt eben die Auswahl der darzustellenden Zeilen ein und nimmt dazu den aktuellen Text des Eingabefelds.

```
/**
 * update the row filter regular expression according to text field
 */
protected final void newFilter() {
    try {
        sorter.setRowFilter(RowFilter.regexFilter(myFilter.getText(), 0));
    } catch (java.util.regex.PatternSyntaxException e) {
        /* */
    }
}
```

Unter Mithilfe von JUnit und Jemmy werden wir nun diese Basistabelle bzw. die Filterfunktionalität testen, indem wir die Tabelle automatisiert mit Daten füllen und die Filterung durchführen und das Ergebnis überprüfen. Dieser Test kann anschließend automatisch und isoliert immer wieder erneut durchgeführt werden. Wir sparen uns dabei das manuelle Starten der Anwendung, die mühsame Eingabe der Testdaten, das Durchführen der Filterung in verschiedenen Szenarien und die manuelle Überprüfung des Ergebnisses.

Wir erstellen dazu die Testklasse `UITest`. Sie nutzt neben Java 6 auch noch JUnit und Jemmy. Wir fügen die beiden *.jar*-Files zum Klassenpfad hinzu. Bei IDEA (aber auch bei anderen IDEs wie Eclipse) ist das ganz einfach: Ein Klick mit der rechten Maustaste auf das Projekt öffnet ein Kontextmenü. Hier müssen Sie den Eintrag MODULE SETTINGS wählen und dort die Bibliotheken dem Projekt hinzufügen.

Schauen wir uns nun die Testklasse an.

Die Testklasse

Die Testklasse `UITest`, siehe Beispiel 13-2 ist eine gewöhnliche JUnit 4-Testklasse, die verschiedene Klassenattribute hat, eine mit `@Before` initialisierende Methode, eine Testmethode, eine mit `@After` gekennzeichnete Nachbedingung und eine private Hilfsmethode.

Beispiel 13-2: UITest, die Testklasse

```
package com.huettermann.jemmy;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import org.junit.After;
import org.netbeans.jemmy.operators.JButtonOperator;
import org.netbeans.jemmy.operators.JFrameOperator;
import org.netbeans.jemmy.operators.JTableOperator;
import org.netbeans.jemmy.operators.JTextFieldOperator;

import javax.swing.*;
import javax.swing.event.DocumentListener;
import javax.swing.event.DocumentEvent;
import javax.swing.table.TableModel;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableRowSorter;

/**
 * @author Michael Huettermann
 */
public class UITest {
    private JFrameOperator frame;
    private TableRowSorter<TableModel> sorter;
    private JTextField filterText;

    @Before
    public void setUp() {
        //start application
        TableFilter.main(new String[]{});
        //fetch frame
        frame = new JFrameOperator(0);
    }
```

Beispiel 13-2: UITest, die Testklasse (Fortsetzung)

```
    @Test
    public void testFilter() {
        //fetch table
        JTableOperator mainTable = new JTableOperator(frame, 0);
        JTable table = (JTable) mainTable.getSource();

        //fill with test data
        TableModel model = createTableModel();
        mainTable.setModel(model);
        sorter = new TableRowSorter<TableModel>(model);
        table.setRowSorter(sorter);

        Assert.assertEquals(mainTable.getValueAt(0, 0), "Peter");
        Assert.assertEquals(mainTable.getValueAt(1, 0), "Frank");
        Assert.assertEquals(mainTable.getValueAt(2, 0), "Kurt");
        Assert.assertEquals(mainTable.getValueAt(3, 0), "Felix");
        Assert.assertEquals(mainTable.getValueAt(4, 0), "Marcelo");

        //change content
        mainTable.changeCellObject(2, 0, "Ferdinand");

        Assert.assertEquals(mainTable.getValueAt(2, 0), "Ferdinand");

        JTextFieldOperator inputField = new JTextFieldOperator(frame, 0);
        //new model means new filter stuff
        filterText = (JTextField) inputField.getSource();
        filterText.getDocument().addDocumentListener(
                new DocumentListener() {
                    public void changedUpdate(DocumentEvent e) {
                        sorter.setRowFilter(RowFilter.regexFilter(filterText.getText(), 0));
                    }

                    public void insertUpdate(DocumentEvent e) {
                        sorter.setRowFilter(RowFilter.regexFilter(filterText.getText(), 0));
                    }

                    public void removeUpdate(DocumentEvent e) {
                        sorter.setRowFilter(RowFilter.regexFilter(filterText.getText(), 0));
                    }
                });

        inputField.typeText("F");

        Assert.assertEquals(mainTable.getRowCount(), 3);
        Assert.assertTrue(((String) mainTable.getValueAt(0, 0)).startsWith("F"));
        Assert.assertTrue(((String) mainTable.getValueAt(1, 0)).startsWith("F"));
        Assert.assertTrue(((String) mainTable.getValueAt(2, 0)).startsWith("F"));
    }

    @After
    public void tearDown() throws Exception {
        //exit
```

Beispiel 13-2: UITest, die Testklasse (Fortsetzung)

```
        JButtonOperator cancelButton = new JButtonOperator(frame, "Exit");
        cancelButton.push();
    }

    /**
     * create table model with test data
     *
     * @return TableModel
     */
    private TableModel createTableModel() {
        //create a model consisting of 5 rows by 2 columns.
        DefaultTableModel model = new DefaultTableModel(5, 2);

        //assign headers
        String[] columnTitles = {
                "Name",
                "Age",
        };

        model.setColumnIdentifiers(columnTitles);

        //populate cells
        String[] names =
                {
                        "Peter",
                        "Frank",
                        "Kurt",
                        "Felix",
                        "Marcelo"
                };

        String[] ages =
                {
                        "31",
                        "28",
                        "12",
                        "66",
                        "71"
                };

        int rows = model.getRowCount();

        for (int i = 0; i < rows; i++) {
            model.setValueAt(names[i], i, 0);
            model.setValueAt(ages[i], i, 1);
        }

        return model;
    }
}
```

Wie ist unsere Testklasse im Detail aufgebaut? Bevor die Testmethode durchläuft, starten wir durch Aufruf der *main*-Methode die Klasse unter Test, die TableFilter-Klasse.

```
@Before
public void setUp() {
    TableFilter.main(new String[]{});
    frame = new JFrameOperator(0);
}
```

Wir holen uns anschließend mit Jemmy einen Handler auf das JFrame, die zugrundeliegende Containerinstanz. Der Handler `org.netbeans.jemmy.operators.JFrameOperator` gibt uns den ersten JFrame der zu testenden Applikation zurück, indem wir der Methode eine *0* übergeben (Java zählt in solchen Bereichen von 0 los). Wir wissen, dass wir nur einen JFrame haben, also der erste Frame unser gesuchtes Objekt ist (für weitere Details darüber, wie Jemmy die Instanzen findet, verweise ich an dieser Stelle auf die umfangreiche Jemmy-Dokumentation). Das brauchen wir nur einmal durchzuführen, es befindet sich daher als Vorbedingung außerhalb der Testmethode.

Nun kommen wir zur Testmethode. Unsere Ausgangstabelle ist eine leere Basistabelle. Entsprechend müssen wir dafür sorgen, dass die Tabelle mit Testdaten gefüllt wird. Das hat den Vorteil, dass die Testklasse unabhängig von Daten der Testanwendung ist. Wir erzeugen uns ganz ähnlich wie zuvor einen Handler auf die zu testende Tabelle. Dazu nutzen wir `JtableOperator` von Jemmy, der als Konstruktor unseren JFrame-`frame` und eine `0` aufnimmt. Das bedeutet, dass Jemmy in dem JFrame-`frame` die erste Tabelle sucht und eine Referenz darauf zurückliefert. Diese Referenz nennen wir `mainTable`; ihr mit `getSource` geliefertes Objekt casten wir auf JTable. So haben wir nun die Swing-Tabelle im Zugriff.

```
JTableOperator mainTable = new JTableOperator(frame, 0);
JTable table = (JTable) mainTable.getSource();
```

Danach füllen wir die Tabelle mit Testdaten. Diese Testdaten werden in Form eines `TableModel` in der Methode `createTableModel` erstellt und der Swing-Tabelle zugewiesen. Anschließend erstellen wir eine neue TableRowSorter-Instanz, die unser Tabellenmodell empfängt. Der zugrundeliegenden Swing-Tabelle weisen wir den RowSorter `sorter` zu (hier reicht es nicht, über den Jemmy-Handler zu gehen).

```
TableModel model = createTableModel();
mainTable.setModel(model);
sorter = new TableRowSorter<TableModel>(model);
table.setRowSorter(sorter);
```

Nun können wir testen, ob die anfänglichen Daten des Testmodells richtig in der Tabelle angelegt wurden. Dazu gehen wir auf die Tabelle `mainTable` (der Jemmy-Handler) und überprüfen jeweils die Zelleninhalte der ersten Spalte mit den Daten, die wir bei der Erstellung verwendet haben (zum Zweck der Anschaulichkeit wird hier auf Extrahierung der hart verdrahteten Werte verzichtet).

```
Assert.assertEquals(mainTable.getValueAt(0, 0), "Peter");
Assert.assertEquals(mainTable.getValueAt(1, 0), "Frank");
Assert.assertEquals(mainTable.getValueAt(2, 0), "Kurt");
Assert.assertEquals(mainTable.getValueAt(3, 0), "Felix");
Assert.assertEquals(mainTable.getValueAt(4, 0), "Marcelo");
```

Nun simulieren wir weitere Benutzerinteraktionen. Mithilfe der von Jemmy bereitgestellten Funktionalität `changeCellObject` simulieren wir die Modifikation des Inhalts der Zelle in der zweiten Zeile und ersten Spalte, und überprüfen direkt den Erfolg mit `Assert.assertEquals`. Wir ändern den Zellenwert auf `Ferdinand` und prüfen anschließend den Erfolg.

```
mainTable.changeCellObject(2, 0, "Ferdinand");
Assert.assertEquals(mainTable.getValueAt(2, 0), "Ferdinand");
```

Nun suchen wir das Eingabefeld. Es gibt nur eins, also ist es auch das erste des JFrame-`frame`. Hier müssen wir direkt auf die Swing-Komponente gehen und den `DocumentListener` mit dem Filter neu anhängen (der Jemmy-Handler reicht dafür nicht aus, wir müssen das zugrundeliegende JTextField ansprechen). Wir setzten den Filter so ähnlich, wie wir es in der ursprünglichen Klasse gemacht haben.

```
        JTextFieldOperator inputField = new JTextFieldOperator(frame, 0);
        //new model means new filter stuff
        filterText = (JTextField) inputField.getSource();
        filterText.getDocument().addDocumentListener(
                new DocumentListener() {
                    public void changedUpdate(DocumentEvent e) {
                        sorter.setRowFilter(RowFilter.regexFilter(filterText.getText(), 0
));
                    }

                    public void insertUpdate(DocumentEvent e) {
                        sorter.setRowFilter(RowFilter.regexFilter(filterText.getText(), 0
));
                    }

                    public void removeUpdate(DocumentEvent e) {
                        sorter.setRowFilter(RowFilter.regexFilter(filterText.getText(), 0
));
                    }
                });
```

In unserem Beispiel haben wir in der ersten Spalte Vornamen eingefügt. Zuerst begannen zwei mit dem Buchstaben `F`. Nachdem wir eine dritte Position entsprechend geändert haben, so dass sie auch mit `F` beginnt, haben wir nun in drei Zeilen drei mit F beginnende Vornamen. Wir simulieren eine Eingabe `F` in das Eingabefeld.

```
inputField.typeText("F");
```

Anschließend testen wir, ob der Filter tatsächlich ordnungsgemäß funktioniert hat. Die Tabelle muss nun drei Zeilen haben, wobei der Inhalt einer jeden ersten Spalte mit `F` beginnt.

```
Assert.assertEquals(mainTable.getRowCount(), 3);
Assert.assertTrue(((String) mainTable.getValueAt(0, 0)).startsWith("F"));
Assert.assertTrue(((String) mainTable.getValueAt(1, 0)).startsWith("F"));
Assert.assertTrue(((String) mainTable.getValueAt(2, 0)).startsWith("F"));
```

Nach dem Test wollen wir wieder in den Ausgangszustand zurück und die Applikation schließen. Das machen wir mit den Funktionen, die uns die Anwendung selbst zur Verfügung stellt.

```
@After
public void tearDown() throws Exception {
    //exit
    JButtonOperator cancelButton = new JButtonOperator(frame, "Exit");
    cancelButton.push();
}
```

Wir suchen uns dazu den JButton mit Hilfe des `JButtonOperator` von Jemmy. Ausgangspunkt ist hier wieder unser `frame`, nun aber suchen wir den Button über das Label `Exit`. Wir hätten auch hier wieder über die Position der Komponente zu den anderen Komponenten gehen können, was besonders einfach wäre, da wir nur einen Button nutzen.

Reihenfolgen von grafischen Komponenten auf dem UI sind genauso Änderungen unterworfen wie Änderungen bei den Labels. Ist Ihre Applikation lokalisiert bzw. internationalisiert? Falls Sie die Labels nicht hart verdrahtet in Ihren Klassen, sondern in *properties*-Dateien stehen haben, können Sie den Identifier, mit dem Sie auf diese Properties zugreifen, ebenfalls für Jemmy nutzen. Der eigentliche Text, der schließlich auf dem UI steht, ist dann nicht mehr wichtig, Sie nutzen nur noch einen technischen Schlüssel.

Der Testlauf

Rufen wir nun UITest über IDEA auf, so erhalten wir die in Abbildung 13-4 dargestellte Ausgabe. Der Test ist erfolgreich durchgelaufen.

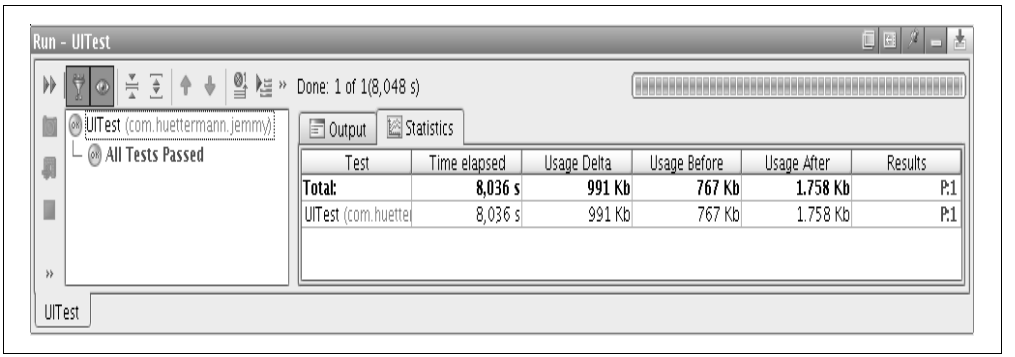


Abbildung 13-4: Der Testlauf in IDEA

Interessant ist noch ein Blick auf die Konsolenausgabe. Das Beispiel 13-3 enthält einen Ausschnitt der von Jemmy protokollierten Aktivitäten. Jemmy dokumentiert detailliert, wie es den Test begleitet. Fett unterlegt sind beispielhafte Passagen, in denen Jemmy das UI direkt beeinflusst.

Beispiel 13-3: Jemmy sucht und bedient Swing-Komponenten

```
...
Using org.netbeans.jemmy.drivers.DefaultDriverInstaller driver installer
Trace:
Start to wait frame "Any javax.swing.JFrame" opened
Trace:
Frame "Any javax.swing.JFrame" has been opened in 1 milliseconds
    com.huettermann.jemmy.TableFilter[frame0,0,0,408x161,layout=java.awt.BorderLayout,title=A
gile Development with Jemmy,resizable,normal,defaultCloseOperation=EXIT_ON_CLOSE,rootPane=jav
ax.swing.JRootPane[,4,22,400x135,layout=javax.swing.JRootPane$RootLayout,alignmentX=0.0,align
mentY=0.0,border=,flags=16777673,maximumSize=,minimumSize=,preferredSize=],rootPaneCheckingEn
abled=true]
Trace:
Start to wait action "Wait Any javax.swing.JTable loaded"
Trace:
"Wait Any javax.swing.JTable loaded" action has been produced in 2 milliseconds with result
...
Process finished with exit code 0
```

Zusammenfassung

Vermeiden Sie das explizite Testen von User Interfaces, wann immer es möglich ist. Erstellen Sie für Ihre Anwendung eine Architektur, in der das UI nur eine dünne Schicht über Ihrer eigentlichen Anwendung ist. Trennen Sie die Geschäftslogik vom UI. Berücksichtigen Sie ferner, dass auch Swing-Komponenten über ein Model verfügen. Sie können den Zustand einer Komponente (ist beispielsweise eine CheckBox markiert oder nicht?) am Model abfragen. Dazu müssen Sie nicht den Umweg über das UI machen. Die Geschäftslogik und das Model können Sie mit Unit-Tests versehen und umfassend von Benutzerinteraktionen unabhängig testen.

In den wenigen Szenarien, in denen Sie tatsächliche Benutzerinteraktionen auf Ihrer Swing-Anwendung simulieren möchten bzw. müssen, ist Jemmy eine ausgezeichnete Hilfe. Jemmy hilft Ihnen dabei, visuelle Komponenten zu lokalisieren und zu bedienen. Jemmy muss die Komponenten zunächst finden. Falls sich Ihre UI häufig ändert, Sie Ihre Komponenten auf dem UI umsortieren oder neue Komponenten regelmäßig ergänzen, kann das zu einem hohen Wartungsaufwand in Ihrer Jemmy-Testklasse führen.

Binden Sie Jemmy in Unit-Tests ein, um erwartete Sollwerte mit Istzuständen abzugleichen. Die zugrundeliegenden Unit-Tests können Sie entsprechend auch automatisiert ausführen.

KAPITEL 14

Webanwendungen testen mit Selenium

Java ist nicht nur für die Entwicklung von Client-Programmen sehr beliebt, sondern wird auch zur Programmierung von Webanwendungen viel genutzt. Webanwendungen lassen sich gewöhnlich von überall nutzen, wo ein Netzzugang und ein Browser zur Verfügung stehen. Der Browser stellt dabei vom Server gelieferte Informationen dar. Im Gegensatz zu Swing-Anwendungen interagiert in dieser traditionellen Client-Server-Umgebung der Browser bei jedem Benutzerverhalten mit dem Server. Das kann durch JavaScript auf dem Client umgangen werden.

Für das Testen solcher Anwendungen ergeben sich besondere Herausforderungen. In diesem Kapitel beschreiben wir die konzeptionellen Grundlagen von Java-Webanwendungen, und wie diese funktional getestet werden können. Dabei werden wir das Werkzeug Selenium kennenlernen und es agil auf zwei unterschiedliche Weisen nutzen.

Architektur von Webanwendungen

Technologisch betrachtet basieren Webanwendungen auf verschiedenen sich ergänzenden Konzepten.

Die Daten werden dem Benutzer in Form von *HTML* bereitgestellt. HTML ist eine Dokumentbeschreibungssprache, die den Inhalt einer Seite und ihren Aufbau definiert. Jede Seite verfügt dabei über eine *URL* (Uniform Resource Locator), mit deren Hilfe sie über einen Browser angesprochen werden kann. Der Browser ist eine Anwendung auf dem Arbeitsplatzrechner des Nutzers, mit deren Hilfe sich der Benutzer Webanwendungen anschauen kann. Gängige Browser sind *Internet Explorer*, *Firefox* und *Opera*.

HTTP ist ein Protokoll, mit dessen Hilfe der Browser den Webserver kontaktiert. Eine Anfrage an den Webserver hat eine Antwort zur Folge. Die Antwort ist mit einem HTTP-Statuscode verknüpft. Wird beispielsweise eine angeforderte Seite nicht gefunden oder ist der Webserver nicht verfügbar, wird ein HTTP-Fehlercode zum Browser übertragen. Die Anwendung selbst befindet sich auf dem Server. Dabei kommt ein Webserver zum Einsatz, eine zentrale Anwendung, auf der eine Webanwendung läuft. Beispiele dafür sind Apache Tomcat und Jetty.

Hyperlinks dienen dazu, verschiedene Seiten miteinander zu verlinken. Ein Element wird gewöhnlich mit einem Namen, einer ID oder einem `alt`-Attribut eindeutig gekennzeichnet. Im folgenden Beispiel werden exemplarisch drei typische HTML-Elemente vorgestellt: Ein Formular überträgt dabei Eingabedaten via HTTP-GET oder HTTP-POST zum Server. Mit dem GET-Protokoll werden die Formulardaten für jeden sichtbar an die URL gehängt. Übertragungen mit POST betten die Daten unsichtbar für den Benutzer der Webanwendung in den Body der HTTP-Nachricht ein.

```
<form action="/search" name=f>...</form>
<a name="Wiki id="WikiWiki">Ein link</a>
<img src="88x31.png" alt="Wiki" />
```

Ferner illustriert das Beispiel einen Anchor `<a>` sowie eine Grafik `<img>`. Sowohl Formular als auch Link und Bild werden gekennzeichnet: Das Formular hat einen Namen, der Link eine ID und das Bild einen sogenannten Alternativtext, der abgebildet wird, wenn es nicht dargestellt werden kann.

Bei der Entwicklung von Webanwendungen kommen weitere Standards zum Einsatz, die sich in *serverseitige* und *clientseitige* Technologien unterteilen lassen, siehe Abbildung 14-1. Serverseitige Technologien kommen ausschließlich auf dem zentralen Webserver zur Anwendung. So wird eine Java-Webanwendung beispielsweise mit *JSPs* (JavaServer Pages) und *Servlets* entwickelt. Diese serverseitigen Technologien sind dem Nutzer verborgen. JSPs und Servlets erzeugen HTML, das zum Browser geschickt wird. Frameworks wie JSF (JavaServer Faces) sind komfortabel zu nutzende Abstraktionen, die auf den angeführten Basistechnologien aufbauen.

Clientseitig kommen neben dem eigentlichen HTML weitere Standards zum Einsatz. Darunter fällt beispielsweise *CSS* (Cascading Style Sheets) zur optimalen Darstellung und Anordnung von Elementen und JavaScript.

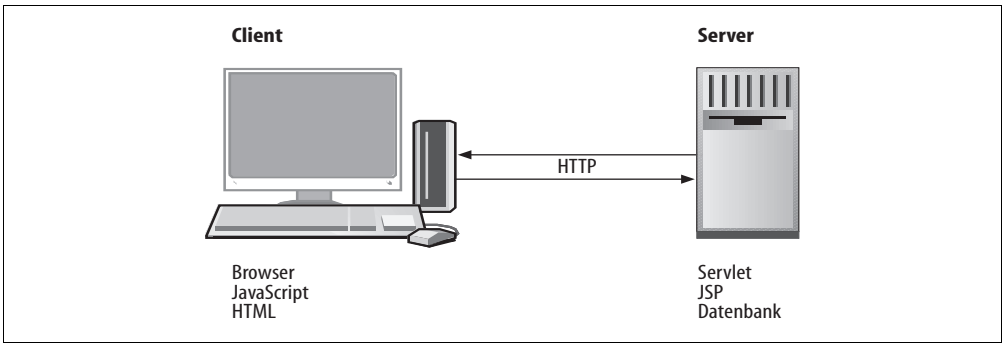


Abbildung 14-1: Architektur einer Webanwendung

JavaScript ist eine von Java gänzlich unabhängige Skriptsprache. JavaScript nutzt das hierarchische *DOM* (Document Object Model), mit dessen Hilfe alle Elemente auf einer Seite angesprochen und manipuliert werden können. Die Elemente liegen dabei in einer

Baumstruktur vor. XPath ist eine Abfragesprache, um Teile des DOM-Baums adressieren zu können. Hier sehen Sie ein Beispiel:

```
<dok>
    <kap title="Dokument">
        <pa>Ein Absatz</pa>
        <pa>Noch ein Absatz</pa>
        <pa>Und noch ein Absatz</pa>
    </kap>
    <kap title="Zweites Kapitel">
        <pa>Ein Absatz</pa>
    </kap>
</dok>
```

Folgende XPath-Ausdrücke liefern bei dem Beispiel die dargestellten Ergebnismengen:

```
/dok
```

selektiert das Wurzel-Element `dok`,

```
/dok/kap[1]
```

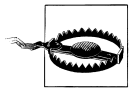
selektiert das erste `kap`-Element innerhalb des ersten `dok`-Elements,

```
//pa
```

selektiert alle `pa`-Elemente auf allen Ebenen und

```
//kap[@title="Dokument"]/pa
```

selektiert alle `pa`-Elemente direkt unterhalb aller `kap`-Elemente, die über ein Attribut `title` mit dem Wert `Dokument` verfügen.



XPath-Ausdrücke haben das Problem, dass sie recht fragil sein können. Wenn wir z.B. mit einen Ausdruck wie

```
//table[@class='defaultContent']//tr[10]/td[1]/input
```

auf ein Feld gehen, kann bereits eine neue Zelle den Such-String brechen.

JavaScript erlebte mit Ajax (Asynchronous JavaScript and XML) eine Renaissance: Ajax kombiniert bereits bestehende ältere Technologien (JavaScript und HTTP-Kommunikation) und entkoppelt die Kommunikation zwischen Client und Server (Asynchronität). Auf diese Weise hat nicht jede Aktivität im Browser, beispielsweise das Abschicken eines Formulars, eine Serveranfrage zur Folge. Das verhindert Wartezeiten. Die Anwendung im Browser lässt sich wie eine herkömmliche Anwendung bedienen (zum Beispiel eine Swing-Anwendung).

Selenium kann jede Form von Webanwendung testen, insbesondere auch Ajax-Anwendungen. Gerade bei der Unterstützung von Ajax-Anwendungen hat die Konkurrenz wie WebTest Nachholbedarf. WebTest baut auf HTTPUnit auf, einem Basis-Framework zum Testen von Webanwendungen, das HtmlUnit ähnelt.

Die Arbeitsweise von Selenium

Selenium ist ein von *ThoughtWorks* entwickeltes frei erhältliches Werkzeug zum Testen von Webanwendungen. Es entstand aus einem Testframework, das bei ThoughtWorks intern zum Einsatz kam. Die Tests laufen dabei direkt im Browser, simulieren also tatsächliche Nutzerinteraktionen. Selenium nutzt dabei JavaScript, um eine Testmaschine neben der eigentlichen Anwendung in den Browser einzuhängen, die die Interaktionen durchführt.

Selenium erlaubt das rasche Testen von Webanwendungen. Dabei lassen sich verschiedene allgemeine Testziele identifizieren:

1. *Smoke Test*: Schneller Durchlauf durch eine Anwendung durch Verfolgung von Links und rudimentären Eingaben. Es wird auf *kaputte Links* und Server-Errors getestet.
2. Testen des Verhaltens auf dem UI. Testskripten steuern die Anwendung so, wie der Nutzer es tun würde. An bestimmten Stellen werden umfangreiche Kontrollen eingebaut, z.B. ob auf einer Seite ein bestimmter Wert vorhanden ist (*Regressionstest*).
3. Rasche Identifikation von Browserinkompatibilitäten, indem die Tests in unterschiedlichen Browsern durchgeführt werden.

Mit Selenium-*Commands* können innerhalb eines Skripts Browseraktionen provoziert werden. Diese Befehle lassen sich in *Actions* und *Assertions* unterscheiden:

- Actions simulieren eine Interaktion innerhalb der Webanwendung, also etwa das Ausfüllen eines Formulars, das Drücken eines Buttons etc.
- Assertions überprüfen das zu erwartende Resultat eines Command. Hier kann beispielsweise überprüft werden, ob der Seiteninhalt korrekt ist.

Selenium nutzt gewöhnlich die Identifier `id` oder `name` aus dem HTML. Sie zu nutzen ist auch unabhängig von Selenium guter Stil. So gesehen kann die Nutzung eines Testwerkzeugs auch auf diesem Wege der Qualität der Anwendung und der Qualität des HTML dienen.

Selenium kommt in unterschiedlichen Ausprägungen daher. Wir möchten uns Selenium IDE und Selenium RC näher anschauen. Selenium IDE ist eine integrierte Entwicklungsumgebung für Selenium-Tests. Es wurde als Firefox-Erweiterung entwickelt und ermöglicht die Aufnahme und das Debuggen von Tests. Mit der IDE können Sie Testskripten aber nicht nur aufzeichnen, sondern auch editieren, erweitern oder komplett manuell schreiben. Selenium IDE verfügt über typische IDE-Funktionalitäten wie Autovervollständigen (die IDE kennt gültige Befehle und unterstützt bei der Eingabe) und speichert die Tests in einer Tabellennotation. Selenium RC (Remote Control) auf der anderen Seite erlaubt Ihnen, Web-UI-Tests in einer beliebigen Sprache zu schreiben. Sie sind nicht auf die Tabellennotation beschränkt. Selenium RC kontaktiert bei den Tests direkt den Server (durch Senden der GET- und POST-Kommandos von HTTP) und lässt sich in einen kontinuierlichen Integrationsprozess einbeziehen.

Selenium IDE

Selenium IDE ist eine integrierte Entwicklungsumgebung für Selenium-Tests. Sie ist als eine Firefox-Erweiterung implementiert. Mit ihrer Hilfe können Tests auf beliebigen Webanwendungen aufgenommen, editiert und kontrolliert werden. Die IDE zeichnet sich durch folgende Features aus:

- Record und Replay: Interaktionen können aufgenommen und wieder abgespielt werden.Autovervollständigen (Autocomplete) für alle Selenium-Kommandos: Die Kommandos werden von der IDE erkannt und der Anwender beim Verfassen unterstützt.
- Darstellung von Kontextmenüs im Browser für Einfügen von Assertions
- Schnelles Durchlaufen von Tests oder langsames, sukzessives Abspielen
- Debugging
- Speichern von Tests

Installation

Laden Sie sich von der Selenium IDE-Seite unter *http://www.openqa.org/selenium/* die Browsererweiterung für Firefox in der aktuellen Version herunter. Öffnen Sie die Datei mit der Endung *.xpi*, wird die Erweiterung automatisch eingerichtet und steht mit dem nächsten Neustart des Browsers zur Verfügung (alle Browserfenster müssen geschlossen werden!). Unter *EXTRAS* im Browsermenü können Sie Selenium IDE aufrufen.

Ein Szenario

Funktionalität und Arbeitsweise schauen wir uns nun an einem Beispiel an. Wir werden folgende Schritte durchlaufen:

1. auf *www.google.de* nach »Extreme Programming« suchen
2. überprüfen, ob auf der SERP (Search Engine Result Page, Suchergebnisseite) der Begriff »Extreme Programming« vorkommt
3. durch Klick auf einen entsprechenden Link auf der SERP auf die Wikipedia-Seite zum Thema gehen den dortigen Seitentitel mit einem erwarteten vergleichen
4. ein dort hinterlegtes Bild anklicken, was eine neue Seite öffnet
5. durch Klick auf den »Extreme Programming«-Artikellink wieder einen Schritt zurück auf den Artikel gelangen
6. dort überprüfen, ob ein Link einen bestimmten Text enthältauf einen externen Link klicken

Dieses Szenario enthält also die Tests, ob tatsächlich die Seiten erreichbar sind, ob sie in der Reihenfolge durch die Buttons oder Links gewechselt werden können und ob bestimmter Inhalt vorliegt. Es ist ein schöner Smoke Test, der nach Änderungen an den Seiten durchlaufen werden kann und feststellt, ob die bisherige Funktionalität weiterhin verfügbar ist.

Vorgehen

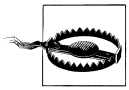
Wir öffnen Selenium IDE (siehe Abbildung 14-2) und geben die Basis-URL *http://www.google.de* manuell ein. Auf diese Weise öffnet Selenium bei jedem Abspielen zunächst diese Startseite, um dann die folgenden Schritte sukzessive von dort aus durchzuführen.

Aufnahme

Wir drücken den Aufnahmeknopf in der IDE oben rechts und gehen die einzelnen Schritte nun auf den Seiten manuell durch. Dort, wo wir eine Bedingung auf einer Seite überprüfen lassen wollen, gehen wir mit der Maus auf die jeweilige Komponente und betätigen die rechte Maustaste. Die aktuelle Komponente wird anhand der ID oder des Labels oder eines XPath-Ausdrucks identifiziert. Selenium bietet Ihnen dabei jeweils bestimmte Kommandos zur Auswahl an. Die Interaktionen werden mitgeschnitten und sind direkt in Selenium IDE sichtbar.

Keine Hektik bei der Aufnahme. Die Zeit spielt keine Rolle. Ein eingegebener Wert wird erst in die IDE übernommen, wenn er abschließend bestätigt wurde. Beispiel: Sie können ein Begriff innerhalb eines Formulars beliebig editieren, bis Sie ihn dann endgültig zum Server schicken. Das »clickAndWait« von Selenium sichert zu, dass der Browser auch beim erneuten Abspielen des Testskripts immer entsprechend auf die Antwort des Servers wartet.

Sind wir mit den Schritten durch, betätigen wir erneut den Aufnahmeknopf. Nun sehen wir alle Kommandos, die Selenium benötigt, um diese Schritte erneut auf der Zielanwendung abzuspielen.



Schließen Sie nicht leichtfertig den Browser. Selenium IDE ist eine Firefox Extension. Ist der Browser geschlossen, wird die IDE ebenfalls beendet. Sichern Sie das Testskript über FILE → SAVE TEST.

Abspielen

In der IDE sehen Sie im oberen Bereich unterhalb der Start-URL den Ausführungsmodus: Soll das Skript in normaler Geschwindigkeit ausgeführt werden oder in verfolgbaren Einzelschritten oder Schritt für Schritt?

Im mittleren Bereich sehen wir die von Selenium aufgenommenen Kommandos. Die erste Spalte der Tabelle enthält den Selenium-eigenen Namen (das Command), gefolgt von dem Ziel (also dem Element innerhalb der Seite) und einem Wert, der bei einem Vergleich herangezogen werden soll. Diese Kommandosequenz kann manuell verändert werden. Einzelne Kommandos werden durch *Entf* gelöscht, neue können durch die rechte Maustaste anderen vorgestellt werden. Dabei kann in den Feldern unterhalb von einer Eingabehilfe profitiert werden.

Im unteren Bereich existiert ein Logging-Bereich. Hier werden die beim Abspielen durchgeführten Aktionen protokolliert. Schlägt ein Test fehl, wird das rot vermerkt.

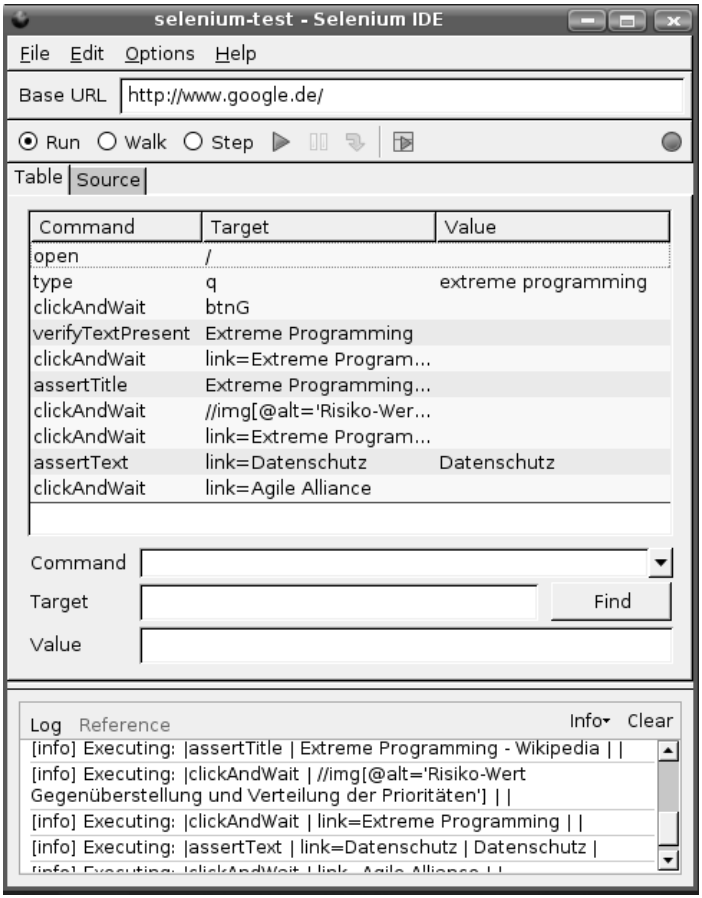


Abbildung 14-2: Selenium IDE

Das Skript wird in Tabellenform dargestellt. Das Tabellenformat ist nicht nur bei der Darstellung gewählt, die Testschritte werden auch in Tabellenform abgespeichert. Betätigen Sie dazu den Reiter *SOURCE*. Sie sehen dann die Selenium-eigene Form, siehe Beispiel 14-1.

Ein `clickAndWait` wartet nach einer Serveranfrage auf die Antwort. Wenn auf der Seite viel Logik implementiert ist und die Ausführung von JavaScript etwas Zeit braucht, reicht ein einfaches `click` nicht mehr aus. Hier müssen ggf. manuell Wartezeiten hineincodiert werden. Das kann auch bei Ajax-Applikationen notwendig sein, die auf asynchrone Serverkommunikation setzen. Schauen Sie sich in diesem Zusammenhang auch `waitForValue` an.

Beispiel 14-1: Der Test

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>New Test</title>
</head>
<body>
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">New Test</td></tr>
</thead><tbody>
<tr>
    <td>open</td>
    <td>/</td>
    <td></td>
</tr>
<tr>
    <td>type</td>
    <td>q</td>
    <td>extreme programming</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>btnG</td>
    <td></td>
</tr>
<tr>
    <td>verifyTextPresent</td>
    <td>Extreme Programming</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>link=Extreme Programming - Wikipedia</td>
    <td></td>
</tr>
<tr>
    <td>assertTitle</td>
    <td>Extreme Programming - Wikipedia</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>//img[@alt='Risiko-Wert Gegenüberstellung und Verteilung der Prioritäten']</td>
    <td></td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>link=Extreme Programming</td>
    <td></td>
</tr>
<tr>
    <td>assertText</td>
```

Beispiel 14-1: Der Test (Fortsetzung)

```
    <td>link=Datenschutz</td>
    <td>Datenschutz</td>
</tr>
<tr>
    <td>clickAndWait</td>
    <td>link=Agile Alliance</td>
    <td></td>
</tr>

</tbody></table>
</body>
</html>
```

Es gibt noch weitere Möglichkeiten, Selenium einzusetzen. So können Sie den Selenium TestRunner nutzen, der Ihnen in einem Browser eine komplette Kontrollzentrale bereitstellt.

Testskripte können mit Selenium IDE in unterschiedlichen Formaten gespeichert (bzw. in sie exportiert) werden, beispielsweise auch in eine Java-Form, die Sie mit Selenium RC weiternutzen können.

Beispiel 14-1 zeigt das in Tabellenform vorliegende Testskript. Dabei entspricht es dem Layout innerhalb der Selenium IDE: Befehle wie `clickAndWait` wechseln mit den UI-Komponenten, auf denen diese Kommandos ausgeführt werden. Tests wie `assertTitle` überprüfen, ob auf der aktuellen Webseite eine Bedingung erfüllt ist (also zum Beispiel der Titel der Anwendung einem erwarteten entspricht).

Selenium RC – Remote Control

Eine weitere Möglichkeit Selenium einzusetzen, ist über das Vorgehen über *Remote Control*. Selenium RC ermöglicht das Schreiben von automatischen UI-Tests in einer beliebigen Sprache unter Zuhilfenahme bereitgestellter Hilfsklassen (*Wrapper*) gegen eine beliebige Webanwendung mit Hilfe eines gängigen Browsers. Das erlaubt ein besonders agiles Vorgehen, da Sie sehr flexibel arbeiten können: Sie sind von keiner Sprache abhängig und können die Tests kontinuierlich und automatisiert laufen lassen.

Selenium RC kommt mit einem eigenen Server daher, der einen Browser starten, stoppen und beliebig steuern kann. Der Server kommuniziert dabei mit dem Browser via *XMLHttpRequest*, das auch in Ajax-Anwendungen häufig zum Einsatz kommt.

Der Selenium-Server ist ein HTTP-Proxy, der zwischen Browser und Webseite steht. Ein Proxy ist dabei ein Stellvertreter, der die Kommunikation regelt. Das ermöglicht das Ausführen von JavaScript auf beliebigen Webseiten.

Aufgrund von Sicherheitsvorkehrungen ist es generell nicht möglich, beliebigen JavaScript-Code auf fremden Seiten auszuführen. Selenium bietet durch den Proxy-Mechanismus eine Möglichkeit, beliebige Seiten zu testen und nicht nur eigene Seiten, die in einem lokalen Verzeichnis vorliegen.

Installation

Zum Betrieb von Selenium RC benötigen Sie Java 1.5.0 oder höher. Für eine Nutzung unter Linux installieren Sie Firefox und fügen Sie Firefox dem PATH hinzu. Selenium wird versuchen, die *firefox-bin* von Firefox direkt aufzurufen. Für weitere Details und Herrichtungen auf anderen Betriebssystemen oder mit anderen Browsern bemühen Sie bitte die eingehende Dokumentation unter *http://www.openqa.org/selenium-rc/*.

Der Selenium Server

Laden Sie sich den Server unterhalb des *selenium-rc*-Subprojekts von der Selenium-Seite herunter. Die *.zip*-Datei entpacken Sie an einen Ort Ihrer Wahl. Anschließend gehen Sie in das entpackte *server*-Verzeichnis und geben ein:

```
java -jar selenium-server.jar -multiWindow
```

Daraufhin ist der Server unter *localhost* und Port *4444* erreichbar. Den Port können Sie mit dem Startparameter `-port` selbst bestimmen. `MultiWindow` teilt den Browser in verschiedene Frames auf, was die Übersichtlichkeit fördert. Sie haben noch weitere Möglichkeiten, das Laufzeitverhalten zu konfigurieren. Der erfolgreiche Start des Servers wird auf der Konsole dokumentiert.

```
michael@BlueGene:~/selenium/selenium-remote-control-0.9.0/server$ java -
jar sele           nium-server.jar -multiWindow
application/xhtml+xml
03.12.2006 10:28:38 org.mortbay.http.HttpServer doStart
INFO: Version Jetty/0.9.0
03.12.2006 10:28:38 org.mortbay.util.Container start
INFO: Started HttpContext[/,/]
03.12.2006 10:28:38 org.mortbay.util.Container start
INFO: Started HttpContext[/selenium-server,/selenium-server]
03.12.2006 10:28:38 org.mortbay.util.Container start
INFO: Started HttpContext[/selenium-server/driver,/selenium-server/driver]
03.12.2006 10:28:38 org.mortbay.http.SocketListener start
INFO: Started SocketListener on 0.0.0.0:4444
03.12.2006 10:28:38 org.mortbay.util.Container start
INFO: Started org.mortbay.jetty.Server@a62fc3
```

JUnit-Testintegration

Wir möchten einen Selenium-Test nun komplett als JUnit-Test laufen lassen. Sie benötigen dazu JUnit und *selenium-java-client-driver.jar* aus dem *selenium/java*-Verzeichnis. Schauen wir uns zunächst einmal ein exemplarisches Testskript an. Beispiel 14-2 ist an das gleichnamige Beispiel aus der Selenium-Dokumentation angelehnt.

Der Test

Nachdem der Browser gestartet und die Seite *www.google.de* geöffnet ist, wird in das Eingabefeld namens `q` der String `Selenium RC` eingegeben, um anschließend den Button `BTNG` zu drücken. Es wird auf die Suchergebnisseite gewartet und überprüft, ob deren Titel dem gängigen Google-Muster entspricht (Such-String + »- Google-Suche«). Ist das der Fall, ist der JUnit-Test erfolgreich durchlaufen. Abschließend wird der Browser wieder geschlossen.

Beispiel 14-2: GoogleSearch, Selenium in JUnit eingebettet

```
import com.thoughtworks.selenium.DefaultSelenium;
import org.junit.Before;
import org.junit.*;
import static org.junit.Assert.*;

/**
 * Author: Michael Huettermann
 **/
public class GoogleSearch{
   private Selenium sel;
   private String firefox = "/usr/lib/firefox/firefox-bin";
   private String server = "http://www.google.de";
   private String search = "Selenium RC";

   @Before
   public void setUp(){
      sel=new DefaultSelenium("localhost",4444,"*firefox "+firefox,server);
      sel.start();
    }

   @Test
   public void testSearch(){
      sel.open(server+"/");
      sel.type("q", search);
       sel.click("btnG");
      sel.waitForPageToLoad("5000");
       assertEquals(search + " - Google-Suche", sel.getTitle());
   }

   @After
   public void tearDown(){
        sel.stop();
   }
}
```

Schauen wir uns jetzt die Testklasse nochmal genauer an.

```
       @Before
       public void setUp(){
          sel=new DefaultSelenium("localhost",4444,"*firefox
    "+firefox,server);
          sel.start();
        }
```

Bevor der Test durchgeführt werden soll, muss mit `DefaultSelenium` eine Arbeitsinstanz von Selenium (ein *Worker*) erzeugt werden. Dieser erwartet den Server, auf dem der Selenium-Server läuft, und den Port, unter dem er erreichbar ist. Anschließend folgt der String, der den Firefox steuert, und die Start-URL, die wir hier in Member-Variablen der Klasse halten.

```
@Test
public void testSearch(){
   sel.open(server+"/");
   sel.type("q", search);
    sel.click("btnG");
   sel.waitForPageToLoad("5000");
    assertEquals(search + " - Google-Suche", sel.getTitle());
}
```

Nun kommen wir zur Testmethode `testSearch`. Wir sehen hier die verschiedenen Kommandos, die wir der Instanz `sel` von Selenium übergeben. Zunächst öffnen wir unsere AUT (Application Under Test). Das ist *http://www.google.de*. Dann geben wir in das Feld, das über den Schlüssel `q` (der Elementname) identifiziert wird, den Suchbegriff `Selenium RC` ein. Das zugrundeliegende HTML der Webanwendung sieht so aus:

```
<input maxlength=2048 name=q size=55 title="Google-Suche" value="">
```

Danach klicken wir auf den Button mit dem Namen *btnG*.

```
<input name=btnG type=submit value="Google-Suche">
```

Nun warten wir, bis die Seite geladen ist. Hier kommen Time-outs ins Spiel (hier fünf Sekunden), da wir nicht unbedingt Stunden auf eine Seite warten wollen. Nun die JUnit-Abfrage: Wir überprüfen, ob der Titel der Suchergebnisseite dem erwarteten String (aus Konkatenation aus Suchbegriff, Slash und »Google-Suche« ist. Der Titel der HTML-Seite hat zur Ausführzeit die Form

```
<title>Selenium RC - Google-Suche</title><style>
```

so dass der Test gelingt.

Ausführen

Das Attribut `firefox` Ihrer Testklasse passen Sie bitte an Ihre ausführbare Firefox-Instanz auf Ihrem lokalen Rechner an. Nun kompilieren Sie die Datei und berücksichtigen dabei bitte, dass *selenium-java-client-driver.jar* und das *junit.jar* im Classpath sein müssen. Anschließend führen Sie die Klasse aus.

```
CP=selenium-java-client-driver.jar:junit-4.1.jar
javac -classpath $CP GoogleSearch.java
java -cp $CP:. org.junit.runner.JUnitCore GoogleSearch
```

Selenium öffnet nun eigenständig Firefox, führt die Schritte durch und schließt den Browser schließlich wieder. JUnit protokolliert wie gewohnt die Tests, die hier erfolgreich durchliefen.

Die Aufrufe des Servers auf der Webseite werden protokolliert. Beispiel 14-3 untermalt die Kommunikation von Selenium, insbesondere die per HTTP mit GET übertragenen Kommandos.

Beispiel 14-3: Das Selenium-Protokoll

```
Got result: OK on session 540574
GET: cmd=click&1=btnG&sessionId=540574
Browser 540574/:top sel_73563 posted OK
req: null:POST http://www.google.de/selenium-
server/core/driver/?&localFrameAddress=top&seleniumWindowName=&uniqueId=sel_73563&sessionId=5
40574&counterToMakeURsUniqueAndSoStopPageCachingInTheBrowser=1165138548855 HTTP/1.1
Host: www.google.de
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-
US; rv:1.8.0.3) Gecko/20060523 Ubuntu/dapper Firefox/1.5.0.3
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image
/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
keep-alive: 300
Proxy-Connection: keep-alive
Content-Length: 2
Content-Type: application/xml
Pragma: no-cache
Cache-Control: no-cache
Cookie: PREF=ID=3848924f684ac83d:TM=1165138548:LM=1165138548:S=oa54JoVkTGlGRe3W

Got result: OK on session 540574
GET: cmd=waitForPageToLoad&1=5000&sessionId=540574
Browser 540574/:top sel_73563 posted OK
req: null:POST http://www.google.de/selenium-
server/core/driver/?&localFrameAddress=top&seleniumWindowName=&uniqueId=sel_73563&sessionId=5
40574&counterToMakeURsUniqueAndSoStopPageCachingInTheBrowser=1165138549467 HTTP/1.1
Host: www.google.de
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-
US; rv:1.8.0.3) Gecko/20060523 Ubuntu/dapper Firefox/1.5.0.3
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image
/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
keep-alive: 300
Proxy-Connection: keep-alive
Content-Length: 2
Content-Type: application/xml
Pragma: no-cache
Cache-Control: no-cache
Cookie: PREF=ID=3848924f684ac83d:TM=1165138548:LM=1165138548:S=oa54JoVkTGlGRe3W
```

Beispiel 14-2 hat exemplarisch gezeigt, wie wir mithilfe von Selenium und JUnit Tests für Webanwendungen in Java schreiben können. Dabei können wir neben Selenium auf den

kompletten Sprachumfang von JUnit zurückgreifen. Die JUnit-Tests lassen sich wie gewohnt ausführen und in automatisierte Testreihen einbinden.

Kontinuierliche Integration mit Ant

Die Selenium-Skripten werden erst richtig wertvoll, wenn wir sie im Rahmen unserer Builds automatisch und in kurzen Abständen ausführen. Automatisierung kann viele Gesichter haben, wichtig ist nicht *wie*, sondern *dass* Sie automatisieren. Im Folgenden zeige ich Ihnen ein Beispiel, wie Sie Selenium und Ihr Ant-Skript integrieren können.

Das Ant-Skript

Die Erstellung des Ant-Skripts sollte aus den gewohnten Schritten bestehen, unter anderem dem Kompilieren der Testklassen. Neu im Selenium-Kontext ist, dass Sie separate Ant-Targets zum Starten und Stoppen des Selenium-Servers bereitstellen sollten und eine besondere Abfolge der Schritte berücksichtigen müssen (Starten des Selenium-Servers, Ausführen der Tests, Stoppen des Selenium-Servers). Die eigentlichen Tests befinden sich in JUnit-Klassen, so dass Sie diese Testklassen ausführen müssen. Ein exemplarisches Ant-Skript zeigt Beispiel 14-4.

Beispiel 14-4: Das Ant-Skript, das Selenium bedient

```
<project name="Selenium" basedir="." default="test">
<property name="selenium.home" value="/home/michael/selenium"/>

<target name="server.start" description="Start the selenium server">
   <java jar="${selenium.home}/selenium-rc/selenium-server.jar" fork="true" spawn="true"/>
</target>

<target name="ini">
   <delete dir="${selenium.home}/example/log"/>
   <mkdir dir="${selenium.home}/example/log"/>
</target>

<target name="test" depends="ini">
   <parallel>
     <antcall target="server.start"/>
     <sequential>
       <echo message="sleeping"/>
       <sleep seconds="3"/>
       <antcall target="junit"/>
       <antcall target="server.stop"/>
     </sequential>
   </parallel>
</target>

<target name="junit">
   <java classname="org.junit.runner.JUnitCore" output="${selenium.home}/example/log/output.
log" error="${selenium.home}/example/log/error.log">
```

Beispiel 14-4: Das Ant-Skript, das Selenium bedient (Fortsetzung)

```
      <arg value="GoogleSearch"/>
     <classpath>
        <pathelement location="selenium-java-client-driver.jar"/>
        <pathelement location="junit-4.1.jar"/>
        <pathelement path="/home/michael/selenium/example"/>
     </classpath>
  </java>
</target>

<target name="server.stop">
   <get taskname="selenium-shutdown" src="http://localhost:4444/selenium-
           server/driver/?cmd=shutDown" dest="${selenium.home}/example/log/selenium-
        server.txt" ignoreerrors="true" />
    <echo message="Stopping selenium-server" />
</target>
</project>
```

Kern der Verarbeitung ist das Standard-Target `test`. Das zuvor ausgeführte `ini` legt ein Log-Verzeichnis an, das Verarbeitungsprotokolle aufnehmen wird. Das *log*-Verzeichnis wird jedes Mal neu angelegt, das heißt, die Log-Dateien aus vergangenen Verarbeitungen gehen verloren. Das soll uns hier reichen. Denken Sie aber daran, Ihr Skript so zu erweitern, dass alle Protokolle erhalten bleiben (zum Beispiel durch Kopieren der älteren Dateien in ein Archiv).

Im Target *test* werden zwei Blöcke `parallel` und unabhängig voneinander ausgeführt, nämlich der `antcall` und der `sequential`-Block. Parallel bedeutet, separate Java-*Threads* kümmern sich um die Verarbeitung der einzelnen Blöcke innerhalb des Parallel-Tasks. Das ist notwendig, da wir hier bei jedem Durchlauf des Skripts den Selenium-Server starten möchten, er aber nach dem Start verharrt und auf Anfragen lauscht. Weitere Schritte (hier sind es die Schritte im `sequential`-Block) werden im zweiten Thread durchgeführt. Durch das Skript werden nur zwei und nicht noch mehr Threads erstellt, da das `sequential` alle enthaltenen Schritte zu einem atomaren Block zusammenfasst.

Den Server starten wir mit dem Ant-Target `server.start`. Hier wird das *.jar*-File mit dem Java-Task aufgerufen. Die Java-Anwendung läuft durch das `fork` in einer anderen VM als Ant. `spawn` erzeugt dabei einen Prozess, der die Laufzeit des Ant-Skripts überlebt.

```
<target name="server.start" description="Start the selenium server">
   <java jar="${selenium.home}/selenium-rc/selenium-
server.jar" fork="true"                    spawn="true"/>
</target>
```

In dem `sequential`-Block geben wir zunächst mit `echo` eine Kontrollinformation aus, um uns dann anschließend drei Sekunden schlafen zu legen. Das ist die Zeit, die wir einkalkulieren müssen, bis der Selenium-Server hochgefahren ist. Wir könnten das `seconds`-Attribut, also die Schlafzeit erhöhen oder verringern. Ein höherer Wert garantiert uns, dass der Server auch sicher hochgefahren ist. Da wir das ganze Skript automatisiert durchführen, könnten wir auch einen Wert von beispielsweise zehn Sekunden angeben. Ganz sicher können wir aber nicht sein, dass der Server wirklich schon gestartet ist, denn wir wissen

nicht, ob beim Start etwas schiefgegangen ist. Wir könnten dieses Warteintervall noch ausgeklügelter erstellen und beispielsweise einen Zustand des Servers abfragen. Uns reicht hier allerdings der pragmatische Ansatz. Bedenken Sie ferner, dass wir das Skript in recht kurzer Zeit regelmäßig laufen lassen. Wenn in *einem* von extrem vielen Läufen ein Fehler auftritt, wird er protokolliert, und beim nächsten Lauf ist das Problem nicht mehr vorhanden. Es ist außerdem jederzeit sichtbar, warum ein Lauf nicht funktioniert hat.

Nachdem der Thread wieder aufgewacht ist, geht es zum `junit`-Target. Hier führen wir die JUnit-Tests durch. Sind sie abgeschlossen, wird `server` wieder beendet. Auf diese Weise haben wir bei jedem Durchgang exakt dieselben Start- und Endbedingungen. Würden Sie nämlich davon ausgehen, dass der Server beim Lauf bereits gestartet ist und das als Vorbedingung ansehen, dann haben Sie eine weitere Fehlerquelle eröffnet. Ist der Server dann nämlich nicht gestartet, wird das Skript einen Fehler melden. Ferner bedeutet »alles automatisieren« eben, auch den Server-Start zu automatisieren. Da dafür diverse Parameter notwendig sind, ersparen Sie sich die Arbeit, das jedes Mal per Hand fehleranfällig selbst zu übernehmen. Sehen wir uns nun den Aufruf der JUnit-Testklasse genauer an.

```
<target name="junit">
   <java classname="org.junit.runner.JUnitCore" output="${selenium.home}/example/log/
output.log" error="${selenium.home}/example/log/error.log">
       <arg value="GoogleSearch"/>
      <classpath>
         <pathelement location="selenium-java-client-driver.jar"/>
         <pathelement location="junit-4.1.jar"/>
         <pathelement path="/home/michael/selenium/example"/>
      </classpath>
   </java>
</target>
```

Es existieren zahlreiche Möglichkeiten, das JUnit-Target für Ihre Bedürfnisse zu implementieren. Hier zeige ich Ihnen einen sehr flexiblen Weg: Nutzen Sie den `java`-Task von Ant, mit dessen Hilfe Sie beliebige ausführbare Java-Artefakte ausführen können (eben war es noch eine *.jar*-Datei).

Sie rufen die Klasse `org.junit.runner.JUnitCore` auf und übergeben die zu testende Klasse. Das Wissen darüber, wie Sie JUnit-Klassen nutzen und aufrufen, haben Sie bereits im JUnit-Kapitel erworben.

Ausgaben und Fehlermeldungen werden in das zu Beginn angelegte Verzeichnis *log* geschrieben. Dafür geben wir hier zwei *log*-Dateien an. Die Ausgabe sehen wir uns später noch genauer an. Vergessen Sie nicht, den Klassenpfad zu setzen. Wir zeigen hier einerseits auf die *.jar*-Artefakte (sie liegen in unserem Beispiel alle in einem Verzeichnis, wir geben aber nochmal jedes *.jar* an, damit nicht untergeht, welche *.jar*s Sie hier tatsächlich benötigen), andererseits auf einen ganzen Pfad (um die zu testende Klasse auffindbar zu machen). Und das war es auch schon.

Wir stoppen den Server mit dem Target `server.stop`:

```
<target name="server.stop">
   <get taskname="selenium-shutdown" src="http://localhost:4444/selenium-
            server/driver/?cmd=shutDown" dest="${selenium.home}/example/log/selenium-
        server.txt" ignoreerrors="true" />
    <echo message="Stopping selenium-server" />
</target>
```

Dazu nutzen wir den Ant-Task `get`, der eine mit `src` gekennzeichnete Seite anfordern kann. Hier machen wir uns zunutze, dass wir den Selenium-Server über die URL ansprechen können. Im Beispiel sehen wir das Kommando `shutDown`, das wir mit dem Attribut `cmd` anhängen. Mit `dest` wird ein Log-Verzeichnis angegeben, das uns weitere Informationen geben kann.

Sie können den Selenium-Server auch im interaktiven Modus starten. Dann haben Sie die Möglichkeit, auf der Konsole einzelne Kommandos an den Server zu schicken und auf diesem Wege einzelne Schritte auf einer Webanwendung manuell durchzugehen. Im interaktiven Modus kommen ebenfalls steuernde Serverkommandos wie `shutDown` zum Einsatz.

Der Aufruf

Wir rufen das Skript mit `ant` im Verzeichnis *${selenium.home}/example* auf.

```
michael@BlueGene:~/selenium/example$ ant
Buildfile: build.xml

ini:
   [delete] Deleting directory /home/michael/selenium/example/log
    [mkdir] Created dir: /home/michael/selenium/example/log

test:
     [echo] sleeping

server.start:

junit:

server.stop:
[selenium-shutdown] Getting: http://localhost:4444/selenium-server/driver/?cmd=shutDown
[selenium-shutdown] To: /home/michael/selenium/example/log/selenium-server.txt
     [echo] Stopping selenium-server

BUILD SUCCESSFUL
Total time: 11 seconds
michael@BlueGene:~/selenium/example$
```

Die Ausgabe auf der Konsole dokumentiert den Durchlauf der Schritte. Beim Durchlauf werden tatsächlich der Firefox gestartet und die einzelnen Testschritte auf der Zielanwendung durchgeführt.

Das Log

Die Datei *output.log* erhält die Ausgabe des Unit-Tests. Wir sehen, dass der Test erfolgreich verlaufen ist.

```
JUnit version 4.1
.
Time: 7,408

OK (1 test)
```

Die Datei *selenium-server.txt* enthält ein einfaches *OK*. Alles ist in Ordnung, der Server wurde ordnungsgemäß heruntergefahren.

Sie haben auch andere Möglichkeiten, die Selenium-Tests automatisiert laufen zu lassen. Es besteht beispielsweise die Möglichkeit, den Server zu starten und bereits zu diesem Zeitpunkt eine HTML-TestSuite zu übergeben und sie auswerten zu lassen. Auch das können Sie im Ant-Skript hinterlegen. Der Server wird dann nach Abarbeitung der HTML-Suite direkt heruntergefahren. Ich rate Ihnen allerdings dazu, diese Tests per Java und JUnit laufen zu lassen, nicht mit den HTML-Tabellen, die Sie über den Browser erstellt haben. Haben Sie ein Testskript aufgezeichnet, exportieren Sie das HTML-Protokoll über Selenium IDE nach Java.

Sie können den Aufruf des Ant-Skripts nun jederzeit anstoßen. Speichern Sie alle Ausführungsprotokolle und machen Sie sie für jedermann sichtbar. Eine Möglichkeit ist beispielsweise, dass Sie bei jedem Durchlauf ein neues eindeutiges Verzeichnis anlegen, das einen *Timestamp* im Namen hat. So wissen Sie, wann genau dieser Durchlauf stattgefunden hat. Dort legen Sie die Log-Dateien hinein bzw. kopieren Sie mittels Ant hin. Diese Verzeichnisse könnten außerdem im *htdocs*-Verzeichnis eines Webservers liegen bzw. dorthin mit Ant hochgeladen werden, damit Sie schnell per Browser einsehbar sind. Sie können Sie auch auf einer Projektseite einbinden. War der Test nicht erfolgreich, dann machen Sie visuell darauf aufmerksam.

Zusammenfassung

Selenium ist ein mächtiges Werkzeug zur Durchführung funktionaler Oberflächentests. Sie können Interaktionen auf Ihrer Webanwendung durchführen und diese protokollieren. Selenium IDE hilft Ihnen dabei, die einzelnen Schritte mit dem Browser Firefox auf Ihrer Anwendung durchzuführen und Testfälle zu erstellen. Die Schritte lassen sich in verschiedenen Formaten speichern, beispielsweise auch in einer von Selenium RC verarbeitbaren Java-Notation. Selenium RC bietet Ihnen die Möglichkeit, in Java geschriebene Tests zu erstellen und gegen eine beliebige Webanwendung laufen zu lassen. Diese Tests können als JUnit-Tests geschrieben werden und als solche mit Ant in einen kontinuierlichen Integrationsprozess eingebunden werden.

KAPITEL 15

Akzeptanztests mit Fit

Bisher haben wir uns neben den Komponententests die funktionalen Tests mit Selenium angeschaut. Dort konnten wir Benutzerinteraktionen auf einer Webanwendung aufzeichnen und manuell fortschreiben sowie Tests erstellen (einerseits in Selenium selbst, andererseits eingebunden in JUnit). Was macht den Unterschied zwischen einem funktionalen Test und einem *Akzeptanztest* aus? Ein Akzeptanztest treibt entsprechend dem Test-First-Ansatz die Entwicklung der fachlichen Klassen vor sich her. Es handelt sich dabei nicht um bloßes Testen von bereits existentem Code. Gegen was soll der Code eigentlich getestet werden? Gegen eine Anforderung? Wie kann die Anforderung, die beispielsweise als Word-Dokument vorliegt, agil (also schnell und wiederholbar) mit der Funktionalität des Codes abgeglichen werden?

Akzeptanztests

Akzeptanztests sind nicht ausschließlich funktionale Tests. Sie helfen,die Anforderungen der Kunden gleichzeitig zu verstehen (*Exploratory Testing*), zu erfassen und schriftlich festzuhalten. Dabei wird sichergestellt, dass die richtige Funktionalität entwickelt wird (»Build the right code«), im Gegensatz zum »Build the code right« bei Komponententests. Im Gegensatz zum traditionellen Vorgehen mit einem Lastenheft werden dabei ein Vorgehen und ein Medium zur Erfassung der Anforderungen gewählt, die in der Lage sind, den später vorliegenden Code automatisch zu testen. Die Anforderungen werden in der Sprache der Kunden bzw. am besten direkt von ihm selbst verfasst.

Fit ist ein Werkzeug, das die komfortable Erstellung und Ausführung von Akzeptanztests ermöglicht und die Brücke zwischen der fachlichen Domäne bzw. den Anforderungen einerseits und der tatsächlich geleisteten Funktionalität der Software andererseits schlägt. Durch eine für den Kunden verständliche Struktur und Schreibweise (HTML-Seiten) können die Fit-Tests Brücken schlagen, die Selenium und WebTest in dieser Form nicht schlagen können. Sowohl Selenium- als auch WebTest-Tests verfügen häufig über eine Nähe zur zugrundeliegenden Anwendung (Kopplung durch Aufzeichnen von Schritten auf einer bestehenden Anwendung, »Test Last«) und eine für den Nutzer schwieriger zu

verstehende Notation (zum Beispiel XML). Die »Nähe zur Anforderung« und die leichte Nutzung entsprechen der agilen Denkweise, den Kunden in den Entwicklungsprozess aktiv zu integrieren.

Der Übergang von funktionalen Tests zu Akzeptanztests ist fließend, und die Einordnung nicht nur von Fit, sondern auch von WebTest und Selenium als Werkzeuge zum Schreiben und Ausführen von Akzeptanztests möglich. Es ist auch mit WebTest und Selenium möglich, eine Anwendung »Test First«- konform und mit direkter Anlehnung an funktionale Kundenanforderungen zu entwickeln. Technisch versierte Kunden bevorzugen unter Umständen die XML-Notation von WebTest gegenüber der Tabellennotation von Fit und sind in der Lage, Akzeptanztests mit WebTest in Form von XML zu erfassen. Sie können mit allen drei Werkzeugen Akzeptanztests schreiben, wobei Fit als *das* Werkzeug für Akzeptanztests bekannt geworden ist.

Einführung in Fit

Fit ist die Abkürzung für *Framework for Integrated Test*. Es wurde von Ward Cunningham entwickelt und zeichnet sich dadurch aus, dass es Akzeptanztests in *Tabellenform* verwaltet. Diese Tabellen müssen nur wenigen Anforderungen genügen, um von Fit verarbeitet werden zu können.

Fit regt zu Kommunikation, Zusammenarbeit und gemeinschaftlichem Reflektieren über die Aufgabenstellung an. Es erleichtert außerdem, über die Domain zu sprechen: Fit hat den Vorteil, dass es auf der Sprache des Kunden aufsetzt. Es gibt keinen Gegensatz zwischen der Sprache der Kunden und der Sprache des Entwicklers, sondern eine gemeinsame fachliche Begriffswelt. So besteht keine Kluft zwischen Kunden und Entwicklern, und das Verständnis der zu entwickelnden Software bei den Entwicklern wird gefördert. Darüber hinaus lassen sich die Fit-Tabellen mit für den Kunden gewohnten Office-Anwendungen erstellen. Wie die Tabellen erstellt werden, ist nicht wichtig, ob nun mit einem HTML-Editor, Winword oder einem ähnlichen Werkzeug.

Dem Kunden bleibt die Realisierung verborgen. Fit ist bei der Umsetzung seiner Wünsche eine Brücke zwischen der fachlichen und der technischen Welt. Fit unterstützt die Sichtbarkeit des Projektfortschritts. Jederzeit kann genau und automatisch überprüft werden, welche Funktionalität die Anwendung anbietet.

Vorgehen

Der Kunde nimmt sich eine Story, die er spezifizieren will. Bei Scrum greift er sich dazu beispielsweise eine Story aus dem Backlog. Durch die gemeinschaftliche Erstellung der Fit-Dokumente durch Entwickler, Tester und Kunde entsteht eine *ausführbare Spezifikation*. Diese Spezifikation beschreibt die Funktionalität der Software. Die Beteiligten werden sich über die *Domain* (die Fachlichkeit) und das zu entwickelnde Feature klar. Sie sprechen alle dieselbe Sprache (analog zum *Domain-driven Design*).

Der Domain-Experte, der Kunde oder die Person, die sich mit der Fachlichkeit besonders gut auskennt, beschreibt ein neues Feature in wenigen Sätzen und führt einzelne Beispiele aus, wie das Feature arbeiten soll (*Specification Example*). Diese Spezifikationen detaillieren also die bestehenden User Stories und werden innerhalb der Entwicklungsiteration parallel zu und verzahnt mit den Programmierarbeiten der Entwickler geschrieben.

Zunächst werden ein paar einfache Fälle aufgeschrieben, die dann sukzessive durch kompliziertere Sonderfälle ergänzt werden. Das geschieht am besten an einem Whiteboard. Die Entwickler und Tester helfen dabei, die gewonnenen Daten in Tabellenform zu bringen. Tester regen weitere Fälle an und der Entwickler überwacht die Form der Tabelle und ob sie vom Aufbau her Fit-konform ist.

Das erstellte Excel- oder Word-Dokument wird als HTML abgespeichert. Nun können die Entwickler damit beginnen, die *Fixtures* zu schreiben. Fixtures sind dabei der Leim, der die Spezifikation an die eigentliche Applikation bindet. Fixtures sollen keine Geschäftslogik beinhalten, sondern lediglich die Funktionalität der Anwendung aufrufen.

Wenn die Fixtures fertig sind, kann Fit auf den Tabellen laufengelassen werden. Dabei wird eine neue HTML-Seite angelegt, die nicht nur die Ausgangsinformationen enthält. Sie wird darüber hinaus mit dem Ergebnis des Testlaufs angereichert. Zu Beginn sind die Features noch nicht implementiert, so dass Fit die Zellen rot färbt. Über die Tage kommen weitere Implementierungen hinzu, bis schließlich alle Tests grün hinterlegt werden.

Der Vorgang ist iterativ. Erinnern Sie sich, wie wichtig die Integration des Kunden in den Entwicklungsprozess ist? Stellen sich dem Entwickler bei der Implementierung Fragen, wird er zusammen mit dem Kunden diskutieren und ggf. die Tabelle modifizieren. Die Fit-Tabellen sind Teil des Build-Prozesses. Sie werden versioniert und bei jedem Build mit ausgeführt. Die Fit-Tabellen dokumentieren, was die Software augenblicklich leistet: Sie sind eine Dokumentation.

Die verschiedenen Fixtures

Fit unterstützt von Haus aus drei unterschiedliche Testszenarien. Neben der *ColumnFixture* gibt es dazu noch die *RowFixture* und die *ActionFixture*. Ich möchte in die drei kurz einführen, für Details aber auf die angeführte Literatur am Ende des Buchs verweisen.

ColumnFixture

Die ColumnFixture überprüft Eingabedaten gegen Ausgabedaten und ist prädestiniert für jede Art von Testen von Geschäftslogik.

Die Tabelle sieht so aus: Eine Spalte wird zu einem Eingabe- oder Ausgabefeld gebunden. Eine Zeile ist ein Testdatensatz. Die erste Zeile enthält die Parameter. Ausgabefelder besitzen ein `()`.

Die Implementierung geschieht so: ColumnFixtures werden von `fit.ColumnFixture` abgeleitet und um eigene Attribute und Methoden angereichert. Alle in der Tabelle definierten Felder müssen in der Fixture entsprechende Gegenstücke haben. Ausgabefelder besitzen in der Fixture eine gleichnamige öffentliche Methode.

RowFixture

Eine RowFixture untersucht Domain-Objekte. Sie dient der Überprüfung jeder Art von Ausgabelisten, sammelt Ergebnisse zusammen und prüft diese anschließend. Sie hat also nur Ausgabespalten, keine Eingabespalten. Die RowFixture dient der Überprüfung von Ergebnismengen.

Eine Zeile in der Tabelle entspricht einem Satz der Ergebnismenge. Die erste Zeile enthält die Datensatzattribute. Die Tabelle muss zumindest die geforderten Datensätze enthalten.

Die eigentliche Fixture leitet von `fit.RowFixture` ab und implementiert die Methoden `query` und `getTargetClass`. Die `query`-Methode gibt die zu untersuchende Ergebnismenge zurück. `getTargetClass` liefert den Typ der einzelnen Elemente der Ergebnisliste. Der Typ muss als öffentliches Feld vorliegen. Eine RowFixture ergibt meist nur im Zusammenspiel mit einer ColumnFixture oder ActionFixture Sinn.

ActionFixture

ActionFixtures testen Zustände durch Auslösen von Zustandsübergängen (`press`, `enter`) und Zustandsüberprüfung (`check`). Eine `TimedActionFixture` erweitert die ActionFixture um eine Zeitmessung der einzelnen Aktionen. Die Zeiten werden dabei mit in die Ergebnistabelle aufgenommen.

Die ActionFixture benötigt keinen Tabellenkopf. Jede Tabellenzeile entspricht einer Aktion und hat zwei oder drei Spalten. Die erste Spalte enthält immer einen der folgenden Befehle:

- `start`: Eine ActionFixture wird gestartet. Auf dieser Fixture werden die weiteren Kommandos ausgeführt.
- `enter`: Der Wert der dritten Spalte wird dem Identifier aus Spalte 2 zugewiesen.
- `press`: Ein Zustandsübergang durch einen virtuellen Button
- `check`: Überprüfung, ob der Identifier aus Spalte 2 dem Wert in Spalte 3 entspricht.

Wird die ActionFixture verwendet, muss lediglich die Klasse `Fixture` um Methoden erweitert werden, die die in der Testtabelle angelegten Aktionen repräsentieren. Die einzelnen Kommandos nutzen unterschiedliche Konventionen: Für das `enter`-Kommando muss eine entsprechend dem in der Tabelle angegeben Schlüssel benannte öffentliche Methode implementiert werden. Diese besitzt einen Parameter und den Rückgabewert `void`. Für das `check`-Kommando ist es andersherum: Dort ist kein Parameter notwendig, dafür aber ein Rückgabewert. Dem `press`-Befehl genügt eine entsprechend benannte Methode ohne Parameter und Rückgabewert.

Ein Rundgang: ColumnFixture

Wir möchten an dieser Stelle das Vorgehen anhand eines einfachen Beispiels illustrieren. Dazu gehen wir von einem Szenario aus, in dem wir eine Taschenrechner-Anwendung haben, die mathematische Bruchrechnung beherrscht und die wir um individuelle Regeln modifizieren.

Bevor wir mit dem Szenario beginnen können, benötigen wir eine Installation. Besorgen Sie sich von *http://fit.c2.com* die neueste Distribution von Fit. Was Sie dabei auf jeden Fall benötigen, ist *fit.jar*, um es bei all Ihren Arbeiten mit Fit in Ihren Classpath zu legen.

Nun erstellen wir eine HTML-Datei namens *division.html*. Diese sieht im Browser aus wie in Abbildung 15-1 dargestellt.

Ausgeklügelte Bruchrechnung

Unsere Implementierung eines Quotienten ergibt sich durch Division eines Zählers (numerator) durch einen Nenner (denominator).
Ist der Nenner 0, so ist der Quotient -1.
Ist der Zähler 12,201, so ist der Quotient immer 1.

Division		
numerator	denominator	quotient()
10	2	5
12.6	3	4.2
100	5	20
222	222	1
12.01	0	-1
12.201	1	1

Abbildung 15-1: Die Datei division.html im Browser

Sie können Ihre Tests auch im XML-Format abspeichern: Der Fit-Parser sucht die Elemente table, tr und td, die Sie natürlich auch innerhalb eines XML-Dokuments hinterlegen können.

Diese Seite beinhaltet neben erklärendem Text eine HTML-Tabelle, die einem bestimmten Aufbau gehorcht. Beachten Sie, welche Angaben für Fit in welchen Zeilen stehen müssen:

- Zeile 1: Name der Fixture
- Zeile 2: Fixture Mapping, Attribut Bindings
- Zeile 3-n: Akzeptanztests

Die erste Zeile hat eine Zelle und fixiert den Namen der Fixture (der Java-Klasse), um die wir uns ein wenig später kümmern. Wir schreiben im weiteren Verlauf der Tabelle über verschiedene Zeilen drei Spalten fort, wobei generell beliebig viele Spalten denkbar sind. Die zweite Zeile setzt dafür die Überschriften fest: mit *numerator* und *denominator* werden zwei Eingabewerte definiert, die Fit automatisch der Fixture übergeben wird. Die Fixture wird mit den jeweiligen Werten der folgenden Zeilen nacheinander ausgeführt

(beliebig viele Zeilen denkbar) und soll als Ergebnis jeweils die dritte Spalte zurückliefern. Das die dritte Spalte eine Ergebnisspalte ist, wird durch die Klammern () verdeutlicht.

Anschließend erstellen wir eine Java-Datei mit dem Namen *Divsion.java*.

```
import fit.ColumnFixture;

/**
 * @author Michael Huettermann
 */
public class Division extends ColumnFixture {
  public double numerator, denominator;
  public double quotient() {
      if(denominator!=0) return numerator/denominator;
      else return -1;
  }
}
```

Diese kompilieren wir nun (wir gehen hier davon aus, dass alle Dateien in einem Verzeichnis liegen):

```
javac -classpath fit.jar Division.java
```

Dann lassen wir Fit die Dateien, insbesondere die HTML-Datei, verarbeiten:

```
java -cp .:fit.jar fit.FileRunner division.html division-result.html
```

Als Ergebnis auf der Konsole erhalten wir

```
5 right, 1 wrong, 0 ignored, 0 exception
```

Fit hat nun eine neue von uns beim Aufruf spezifizierte Datei namens *division-result.html* angelegt, die uns das Ergebnis des Akzeptanztests darstellt. Das funktioniert so: Unsere HTML-Tabelle definiert einen Akzeptanztest. Diesem stellen wir ein wenig erläuternden Fließtext voran. Text, der vor- oder nachgestellt wird, dient der Dokumentation und ist hervorragend für eine ausführlichere Erläuterung geeignet.

```
Unsere Implementierung eines Quotienten ergibt sich durch Division eines Z&auml;hlers (nu
merator) durch einen Nenner (denominator).<br>
Ist der Nenner 0, so ist der Quotient -1.<br>
Ist der Z&auml;hler 12,201, so ist der Quotient immer 1.
```

Anschließend stellen wir die Tabelle auf. So sieht der Quelltext des HTML-Dokuments aus:

```
<html>
<head><title>Fit</title></head>
<body>
<h3>Ausgekl&uuml;gelte Bruchrechnung</h3>
Unsere Implementierung eines Quotienten ergibt sich durch Division eines Z&auml;hlers (nu
merator) durch einen Nenner (denominator).<br>
Ist der Nenner 0, so ist der Quotient -
1.<br>Ist der Z&auml;hler 12,201, so ist der Quotient immer 1.<br><br>
<table border=1>
<tr><td>Division</td></tr>
```

```
<tr><td>numerator</td><td>denominator</td><td>quotient()</td></tr>
<tr><td>10</td><td>2</td><td>5</td></tr>
<tr><td>12.6</td><td>3</td><td>4.2</td></tr>
<tr><td>100</td><td>5</td><td>20</td></tr>
<tr><td>222</td><td>222</td><td>1</td></tr>
<tr><td>12.01</td><td>0</td><td>-1</td></tr>
<tr><td>12.201</td><td>1</td><td>1</td></tr>
</table>
</body>
</html>
```

Der Inhalt der Tabelle wird von Fit ausgewertet. Der Tabelle haben wir einen HTML-Border zugeteilt, damit der Inhalt grafisch verständlicher dargestellt wird. Der Border ist lediglich kosmetischer Natur und hat keine Auswirkungen auf die Verarbeitung.

- Die erste Zeile der Tabelle beinhaltet den Namen der Java-Klasse, die wir angelegt haben. In dieser Klasse teilen wir Fit mit, wann ein Test fehlschlägt und wann nicht. Diese Feature-Klasse heißt bei uns Division:

  ```
  public class Division extends ColumnFixture {
  ```

Wir haben hier eine *ColumnFixture* erstellt, indem wir unsere Fixture davon ableiten. Fit beherrscht noch andere Verarbeitungsmechanismen (siehe Einschub). Anschließend haben wir in unserer Fixture zwei öffentliche Member definiert.

```
public double numerator, denominator;
```

Jeder Spaltenkopf in unserer Tabelle muss sich auf ein öffentliches Feld in der Fixture mappen lassen. Was nun überprüft werden soll, definieren wir in unserem HTML mit einem Begriff, der mit Klammern endet.

```
<tr><td>numerator</td><td>denominator</td><td>quotient()</td></tr>
```

Wir erkennen: `quotient()` ist die Methode unserer Fixture, die anhand der Attribute die Auswertungen vornimmt. Zu ihr gibt es eine gleichnamige Methode in unserer Java-Klasse. Der Einfachheit halber berechnet sie hier direkt einen Quotienten aus den beiden öffentlichen Attributen und liefert ihn zurück (in der Praxis würden Sie die Logik in der eigentlichen Anwendung haben und hier aufrufen). Wird durch `0` dividiert, wird bei uns kein Fehler zurückgeliefert, sondern eine `-1`. Die Methode beinhaltet also ein Regelwerk.

```
public double quotient() {
    if(denominator!=0) return numerator/denominator;
    else return -1;
}
```

Auch Exceptions können bewusst überprüft werden. Dazu muss in der Ergebnisspalte der Tabelle der Begriff *error* angegeben werden.

Die weiteren Zeilen der Tabelle beschreiben unterschiedliche Ausprägungen für *numerator* und *denominator* und spezifizieren das zu erwartende Ergebnis.

Das erstellte Ergebnis *division-result.html* wird generiert wie in Abbildung 15-2 zu sehen ist:

Ausgeklügelte Bruchrechnung

Unsere Implementierung eines Quotienten ergibt sich durch Division eines Zählers (numerator) durch einen Nenner (denominator).
Ist der Nenner 0, so ist der Quotient -1.
Ist der Zähler 12,201, so ist der Quotient immer 1.

Division		
numerator	denominator	quotient()
10	2	5
12.6	3	4.2
100	5	20
222	222	1
12.01	0	-1
12.201	1	1 *expected* 12.201 *actual*

Abbildung 15-2: Die Datei division-result.html

division-result.html entspricht dem Ursprungsdokument, ist aber zusätzlich angereichert mit den Testergebnissen. Konnte die Verarbeitung nicht erfolgreich durchgeführt werden, werden hier weiterführende Infos aufgenommen. Uns interessiert die letzte Spalte, die Ergebnisspalte. Fit stellt farblich dar, ob ein Sollwert einem Istwert entspricht oder nicht, die Zelle wird grün oder rot. Falls ein Wert nicht dem Sollwert entspricht, wird ferner ein Text hinzugefügt, der beide Werte aufführt. Analog zur Konsolenausgabe zu Beginn sehen wir es auch auch grafisch dargestellt: Fünf Tests sind erfolgreich verlaufen, einer ist fehlgeschlagen.

Warum ist ein Akzeptanztest fehlgeschlagen? Wir haben den Sonderfall bisher einfach noch nicht berücksichtigt. Wenn der Zähler *12201* ist (eine beliebige Regel, die wir uns für diesen Fall überlegt haben), soll der Quotient immer *1* sein. Wir haben also tatsächlich einen Test geschrieben, der noch nicht vorliegende Funktionalität beschreibt und testet.

Mit FitNesse liegt ein Fork von Fit vor, der die Tabellen in ein eigenes Wiki einbettet. »Fork« bedeutet, dass die Entwicklung von Fitnesse auf Fit aufsetzt, aber mittlerweile einen eigenen Weg geht. Tabellen müssen dabei im Wiki selbst editiert werden, lassen sich aber auch aus Excel importieren. Meine persönliche Meinung über Fitnesse ist, dass in Zeiten von Ajax-Anwendungen dieser Ansatz doch etwas altbacken wirkt. Kunden möchten Ihre Anforderungen lieber in gängigen Office-Produkten pflegen und nicht in einem Wiki.

Zusammenfassung

Fit-Dokumente sind keine Unit-Tests. Sie sind konkrete, beispielhafte Ausprägungen der Geschäftslogik. TDD mit Unit-Tests beschreibt die Testerzeugung für alle Artefakte, die potenziell Fehler produzieren können. Fit ist dagegen die Implementierung, also die inneren Details der Geschäftslogik egal – es schaut von einer höheren Warte auf die Funktionalität, und zwar aus einer fachlichen, nicht einer technischen Perspektive.

Schauen wir uns den Prozess aus diesem Blickwinkel an, wird ein Feature zunächst beschrieben, um es dann zu demonstrieren. »Demonstrieren« bedeutet hier, dass anhand einzelner Beispiele demonstriert wird, welches Verhalten das Feature haben soll. Anschließend wird das Verhalten implementiert. Der ganze Prozess ist iterativ und inkrementell. Während noch über andere Eigenschaften diskutiert wird, beginnt der Entwickler schon mit der Realisierung der ersten Eigenschaften. Bei dem Prozess können weitere Eigenschaften des Features zu Tage kommen.

Der Entwicklungsschritt an sich ist die Aktivität, in der der Entwickler mit Hilfe der Unit-Tests die Funktionalität des Codes technisch steuert. TDD und Fit sind also komplementär zu nutzen.

KAPITEL 16

Akzeptanztests für Webanwendungen mit WebTest

Wir haben uns Webanwendungen angesehen und wie wir sie mit Selenium funktional testen können. Mit Fit haben wir einen weiteren Ansatz kennengelernt, Anwendungen (Java-Applikationen und Webanwendungen) funktional zu testen und vom Kunden definierte Akzeptanztests gegen die Anwendung automatisiert auszuführen. Nun möchte wir mit WebTest ein Werkzeug vorstellen, das eine interessante Synergie aus den beiden Ansätzen von Selenium und Fit bietet. Es bietet wie Selenium die Möglichkeit, funktionale Regressionstests durchzuführen, und verknüpft dies mit der Eleganz, diese Tests *vor* der eigentlichen Entwicklung erstellen zu können.

Überblick

WebTest ist ein frei verfügbares Werkzeug, mit dessen Hilfe Sie funktionale Tests, aber auch Akzeptanztests für Ihre Webanwendungen durchführen können. Es entstand 2001 im Rahmen eines Bankprojekts und wird von Canoo gehostet. WebTest zeichnet sich durch seine einfache Nutzung und komfortable Integration in einen Test-First-Entwicklungsprozess aus.

WebTest baut auf Standards wie XML und Ant auf, so dass die Lernkurve relativ flach ist. WebTest-Skripten werden als Ant-Skripte ausgedrückt und können entweder über das Kommando `webtest` oder mit *ant* gestartet werden.

Die Tests können auch in Groovy formuliert werden (bitte schauen Sie dazu in die Dokumentation). Einzelne Ausführungsschritte, die sogennanten *Steps*, werden in prägnanter Form beschrieben. Beispiele dafür sind `<invoke../>` oder `<verifyTitle.../>`. WebTest stellt über 100 dieser Steps zur Verfügung, die von beliebigen Servern bereitgestellte HTML-Seiten testen können. Das Testwerkzeug ist allerdings nicht nur auf HTML beschränkt: Auch umfangreichere JavaScript-Seiten lassen sich steuern und testen. Darüber hinaus können auch XML-Dokumente, PDF-Dateien und sogar Excel-Dateien und E-Mails Tests unterzogen werden. Durch die Ausführung werden XML-Reporte erstellt, die das Testergebnis enthalten und die weiterverarbeitet und optisch dargestellt werden können.

Installation

Um mit WebTest arbeiten zu können, benötigen Sie Java 1.4 oder höher und Ant 1.6.5 oder höher (ist auch Bestandteil der WebTest-Distribution). Laden Sie sich von *http://webtest.canoo.com* einen aktuellen Build herunter und entpacken Sie ihn an einen Ort Ihrer Wahl. Wir nutzen im weiteren Verlauf WebTest in der Version 2.1. Sie benötigen die Artefakte *build.zip* und *doc.zip* (in beiden ist ein *doc*-Verzeichnis, nehmen Sie das aus *doc.zip*). Das *lib*-Verzeichnis enthält alle benötigten externen Bibliotheken, das *bin*-Verzeichnis die Skripten zur Steuerung von WebTest.

Stellen Sie sicher, dass das Skript *webtest.sh* ausführbar ist:

```
chmod a+x ./bin/webtest.sh
```

Anschließend fügen Sie das *bin*-Verzeichnis Ihrem Path hinzu

```
export PATH=${PATH}:<WEBTEST_HOME>/bin
```

oder direkt in Ihrem Profil.

Ein erstes Beispiel: Der WebTest-Selbsttest

Wir möchten die Funktionsweise anhand eines von WebTest mitgelieferten Beispiels veranschaulichen. Schauen wir in das */doc/sample*-Verzeichnis der Distribution, dann sehen wir ein WebTest-Skript namens *installTest.xml*. Das ist ein valides Testskript, ein Selbsttest. Es kann ausgeführt werden und überprüft, ob Ant in der notwendigen Version vorliegt und eine lokal vorliegende HTML-Seite mithilfe von WebTest überprüft werden kann. Diese Überprüfung beschränkt sich exemplarisch auf den Seitentitel, der einem vorgegebenen Wert entsprechen soll. Beispiel 16-1 zeigt das WebTest-Skript.

WebTest bietet auch die Möglichkeit, per Kommando ein exemplarisches WebTest-Testprojekt generieren zu lassen. Dazu geben Sie *webtest -f path\to\webtest\home\webtest.xml wt.createProject* ein. Sie werden nach einem Namen für das Testprojekt gefragt. Gehen Sie anschließend in ebendieses neu erstellte Testprojekt und geben *webtest* ein, um mitgenerierte Tests laufen zu lassen. Es wird ebenfalls ein Testreport erstellt.

Beispiel 16-1: Der Selbsttest: Ein einfaches WebTest-Skript

```
<project name="InstallationCheck" basedir="." default="all">

    <property name="webtest.home" value="${basedir}/../.."/>
    <import file="${webtest.home}/lib/taskdef.xml"/>

    <target name="all" depends="mayPrintANTError, checkWebTest"/>

    <target name="checkWebTest">
        <echo message="webtest.home is ${webtest.home}"/>
        <testSpec name="check calling and parsing a local file">
            <config
```

Beispiel 16-1: Der Selbsttest: Ein einfaches WebTest-Skript (Fortsetzung)

```
                host=""
                port="0"
                basepath="/"
                summary="false"
                saveresponse="false"
                haltonfailure="true"
                protocol="file"/>
            <steps>
                <invoke
                    description="get local file"
                    url="${basedir}/testfile.html"/>
                <verifyTitle
                    description="check the title is parsed correctly"
                    text="Test File Title"/>
            </steps>
        </testSpec>
    </target>

    <target name="checkANT">
        <available classname="org.apache.tools.ant.ProjectComponent"
            property="ant.version.ok"/>
    </target>

    <target name="mayPrintANTError" unless="ant.version.ok" depends="checkANT">
        <echo message="You have a non-compliant version of ANT"/>
        <echo message="Consider moving WEBTESTHOME/lib/ant.jar"/>
        <echo message="to ANT_HOME/lib."/>
    </target>

</project>
```

Zunächst müssen wir daran denken, *taskdef.xml* einzubinden. Dort werden die WebTest-Tasks definiert. Das Standardziel `all` ist abhängig von `mayPrintANTError` und `checkWebTest`. Diese beiden Targets werden also nacheinander ausgeführt. `MayPrintANTError` checkt in einer raffinierten Art und Weise und unter Zuhilfenahme des weiteren Targets `checkANT`, ob die Ant-Klasse `org.apache.tools.ant.ProjectComponent` auffindbar ist.

```
    <target name="checkANT">
        <available classname="org.apache.tools.ant.ProjectComponent"
            property="ant.version.ok"/>
    </target>
```

Ist das der Fall, wird das Property `ant.version.ok` gesetzt: Wir arbeiten auf Ant 1.6.5 oder höher! Ist `checkANT` erfolgreich verlaufen, ist der Test beendet, ansonsten gibt das Skript drei Meldungszeilen aus, dass etwas mit der Ant-Installation nicht stimmt.

Der zweite Selbsttest befasst sich mit WebTest selbst. Hier sehen wir auch schon einzelne WebTest-Kommandos in Aktion, die in eine Testspezifikation `testSpec` eingebettet und mit Konfigurationsinformationen (`config`) verknüpft werden.

Interessant soll für uns zunächst erstmal der Abschnitt `<steps>` sein. Hier wird eine Sequenz von Schritten ausgeführt. Der erste Step führt eine Anfrage auf einer spezifizierten URL aus. Das ist in diesem Fall eine lokal vorliegende HTML-Testdatei, die mit einem absoluten Pfad adressiert wird. Anschließend wird mit `<verifyTile>` der Titel der gerade aufgerufenen HTML-Seite untersucht und dem im Element spezifizierten Sollwert »Test File Title« gegenübergestellt. Die `description`-Attribute dienen lediglich der Dokumentation.

```
<steps>
    <invoke
        description="get local file"
        url="${basedir}/testfile.html"/>
    <verifyTitle
        description="check the title is parsed correctly"
        text="Test File Title"/>
</steps>
```

Nun rufen Sie

```
webtest.sh -buildfile installTest.xml
```

auf, und der Selbsttest läuft erfolgreich durch (siehe Abbildung 16-1). Das Target `checkWebTest` gibt via `echo` eine Meldung aus. Außerdem werden weitere Informationen ausgegeben, wie die für die Ausführung genutzte Java-Distribution.

Die Meldung »Trying to override old defintion of task sleep« resultiert daraus, dass WebTest in seinem `taskdef` einen `sleep`-Step definiert, unter dessen Namen auch Ant selbst ein Task nutzt. Schenken Sie dieser Meldung keine weitere Beachtung. In neueren Versionen ist dieses Problem behoben.

```
michael@BlueGene:~/webtest/doc/samples$ webtest.sh -buildfile installTest.xml
Will use /usr/local/java/jdk1.6.0/bin/java
Buildfile: installTest.xml
Trying to override old definition of task sleep

checkANT:

mayPrintANTError:

checkWebTest:
     [echo] webtest.home is /home/michael/webtest/doc/samples/../..

all:

BUILD SUCCESSFUL
Total time: 1 second
michael@BlueGene:~/webtest/doc/samples$
```

Abbildung 16-1: Der WebTest-Selbsttest

Ein weiteres Beispiel: Auf Googles Spuren

Schauen wir uns nun ein etwas umfangreicheres Beispiel an. Ausgangspunkt soll folgendes Szenario sein: Wir gehen auf *www.google.de* und geben als Suchbegriff »WebTest« ein. Die Suche führt uns zur Ergebnisseite, die den Titel »WebTest - Google-Suche« hat, also die Konkatenation des Suchbegriffs mit »- Google-Suche«. Das resultierende Testskript wird in Beispiel 16-2 dargestellt.

Beispiel 16-2: Eine Google-Suche

```
<?xml version="1.0"?>
<!DOCTYPE project SYSTEM "/home/michael/webtest/selftests/config/WebTest.dtd"[
  <!ENTITY config     SYSTEM "config.xml">
]>

<project name="SearchGoogle" basedir="." default="main">

  <property name="webtest.home" location="/home/michael/webtest" />
  <import file="${webtest.home}/lib/taskdef.xml"/>

  <target name="main">
    <webtest name="myTest">
      &config;
      <steps>
        <invoke
          description="start"
          url="/" />
        <verifyTitle
          description="title"
          text="Google" />
        <setInputField description="set search query"
          name="q"
          value="webTest" />
        <clickButton
          name="btnG"
          description="click submit button" />
        <verifyTitle description="verify title on search result page"
          text="WebTest - Google-Suche"  />
      </steps>
    </webtest>
  </target>
</project>
```

Vieles kennen wir schon von unserem ersten Test. Anders ist, dass wird das `testSpec`-Element vermissen und dafür ein `webtest`-Element vorfinden. Das `webtest`-Element hat das ältere `testSpec` ersetzt, das aber nach wie vor aus Kompatibilitätsgründen unterstützt wird.

Wir sehen im Dokumentenkopf eine `Doctype`-Defintion. Hier wird die zentrale WebTest-DTD hinterlegt. Diese liegt im Verzeichnis *selftests/config*. Eine Referenzierung der DTD teilt der von uns genutzten IDE mit, welche Syntax unsere Dokumente haben dürfen, und hilft somit bei der Editierung der Skripten.

Für WebTest gibt es mittlerweile auch einen als Firefox-Extension implementierten Recorder, der die Ausführung von Interaktionen auf einer Webanwendung aufzeichnen und als WebTest-Skript verfügbar machen kann. Diese Extension ist unter *http://webtestrecorder.canoo.com/http://webtestrecorder.canoo.com/* frei verfügbar. Da wir ein ähnliches Werkzeug bereits im Rahmen von Selenium kennengelernt haben, verzichten wir an dieser Stelle auf eine detaillierte Betrachtung.

Schauen wir in das `webtest`-Element unseres Standardziels:

```
<webtest name="myTest">
  &config;
  <steps>
                ....
  </steps>
</webtest>
```

Die Testsequenz beginnt mit `webtest` und hat den Namen `myTest`. Bevor die einzelnen Schritte auf der Webanwendung ausgeführt werden sollen, muss WebTest die zu kontaktierende Webanwendung mitgeteilt werden. Hier wird auf das im Kopf definierte XML-Entity `config` zurückgegriffen. Es muss in der Form

```
&config;
```

verankert werden, was XML-Standard und nicht typisch für WebTest ist. Durch die Referenzierung der DTD können wir die externe zentrale DTD um weitere eigene Elemente erweitern. Das ist hier das XML-Entity `config`, das auf das Dokument *config.xml* zeigt. Diese zentrale Definition ist wichtig, damit wir einzelne Snippets wiederverwenden können. Folgendes steht in *config.xml*:

```
<config
  host="www.google.de"
  port="80"
  protocol="http"
  basepath="/"/>
```

Hier haben wir also zentral die Startkonfiguration unserer Tests hinterlegt: Welche Seite unter welchem Port getestet werden soll, welches Protokoll dabei zum Einsatz kommt und wie der Basispfad auf dem Zielsystem lautet.

Schreiben wir viele verschiedene Testskripte, müssen wir immer das `config`-Element mit `host`, `port`, `protocol` und `basepath` angeben. Gewöhnlich sind diese aber immer gleich.

Anschließend kommen die Schritte

```
<invoke
    description="start"
    url="/" />
```

`invoke` akzeptiert auch komplette, absolute URLs. Um Webseiten anzusprechen, müssen Sie nicht zwingend den Weg über das `config`-Element gehen, um dort einen Host einzutragen.

Zunächst wird eine Anfrage auf die spezifizierte Startseite der Webanwendung durchgeführt. Das ist der Ausgangspunkt unseres Testszenarios. Wir überprüfen nun den HTML-Titel der Startseite, also ob dieser tatsächlich *Google* ist.

```
<verifyTitle
    description="title"
    text="Google" />
```

Danach geben wir in das Feld mit dem Namen *q* den Wert *webTest* ein.

```
<setInputField description="set search query"
    name="q"
    value="webTest" />
```

Das Formular schicken wir ab mit einem Druck auf den Such-Button. Auch hier dient das *description*-Attribut lediglich der Dokumentation.

```
<clickButton
    name="btnG"
    description="click submit button" />
```

Der Suchknopf heißt `btnG`. Das ist der Name des HTML-Buttons, der von WebTest gesucht und mit `clickButton` gedrückt wird.

```
<verifyTitle description="verify title on search result page"
    text="WebTest - Google-Suche"  />
```

Nun überprüfen wir, ob wir wirklich auf der neuen Seite, der Ergebnisseite gelandet sind. Diese sollte den Titel »WebTest - Google-Suche« haben. Führen wir nun unser WebTest-Skript *SearchGoogle.xml* mit *webtest.sh* und dem Startparameter *SearchGoogle.xml* aus, der den Namen des Skripts angibt:

```
michael@BlueGene:~/webtest/myexamples$ webtest.sh -buildfile SearchGoogle.xml
```

WebTest-Skripte sind Ant-Skripte. Sie können sie auch direkt über Ant aufrufen und müssen nicht zwingend über den Aufruf `webtest` gehen. Das ist insbesondere bei »Continuous Integration« von Interesse, da Sie hier bereits Ant-Skripte einsetzen bzw. das WebTest-Skript als Ant-Skript komfortabel einbinden können.

und wir erhalten als Antwort:

```
Will use /usr/local/java/jdk1.6.0/bin/java
Buildfile: SearchGoogle.xml
Trying to override old definition of task sleep

main:

BUILD FAILED
/home/michael/webtest/myexamples/SearchGoogle.xml:12: Canoo Webtest: R_1393.
Test failed.
Test step verifyTitle (/home/michael/webtest/myexamples/SearchGoogle.xml:28: )  named "ve
rify title on search result page" failed with message "Wrong document title found! Expect
ed "WebTest - Google-Suche" but got "webTest - Google-Suche""

Total time: 2 seconds
michael@BlueGene:~/webtest/myexamples$
```

Das sieht doch schon mal recht gut aus. Die Verarbeitung hat geklappt, allerdings ist ein Test fehlgeschlagen. Der Titel der Suchergebnisseite konnte nicht korrekt verglichen werden. Und das verwundert auch nicht. Haben Sie es gemerkt? Wir haben in das Suchfeld auf der Startseite mit `setInputField` »webTest« eingegeben, nicht »WebTest«. Die Tests sind alle schreibungsabhängig, es spielt also eine entscheidende Rolle, ob ein Buchstabe groß oder klein geschrieben wird. Ändern wir das

```
<setInputField description="set search query"
      name="q"
      value="WebTest" />
```

und führen den Test nochmal durch.

```
michael@BlueGene:~/webtest/myexamples$ webtest.sh -buildfile SearchGoogle.xml
Will use /usr/local/java/jdk1.6.0/bin/java
Buildfile: SearchGoogle.xml
Trying to override old definition of task sleep

main:

BUILD SUCCESSFUL
Total time: 3 seconds
michael@BlueGene:~/webtest/myexamples$
```

Und siehe da, er ist erfolgreich durchgelaufen. Wir haben sehr einfach einzelne Benutzerinteraktionen auf einer Webseite durchgeführt und bestimmte Ereignisse bzw. Eigenschaften überprüft. Und das, ohne einen Browser zu öffnen!

Google steht schon als Website im Internet bereit, so dass unsere Testschritte auf fertige Arbeit gestoßen sind. Aber bitte verinnerlichen Sie, dass Sie den Test unabhängig von der eigentlichen zu testenden Webanwendung schreiben können. Ist die Webanwendung oder sind einzelne Features noch nicht da, können die Tests trotzdem geschrieben werden. Mit den Tests anzufangen, schiebt die eigentliche Funktionalität vor uns her, wir schreiben Spezifikationen.

Wiederverwendung

Wir haben ganz beiläufig gesehen, wie wir mit WebTest-Tests einzelne Bausteine wiederverwenden können. Dazu gibt es zwei Mechanismen, die *Entities* und die *Imports*. Wollen wir innerhalb eines `<webtest>`-Elements einen Baustein wiederverwenden, müssen wir ein Entity nutzen. Oberhalb des `webtest`-Elements können wir auch ein `import` einsetzen. Im Beispiel wird *taskdef.xml* eingebunden, was unserem Dokument Zugriff auf zentral definierte Werte gestattet.

```
<import file="${webtest.home}/lib/taskdef.xml"/>
```

Anstelle von *taskdef.xml* kann auch *webtest.xml* eingebunden werden.

Testskripten können auch zu *Suites* aggregiert werden. Dazu nutzen wir einfach die Technik, die uns Ant von Haus aus anbietet:

```
<target name="testSuite">
   <ant antfile="UseCases/LoginAndStep.xml"/>
   <ant antfile="UseCases/UseCase2.xml"/>
</target>
```

Reporting

Nun haben wir bisher lediglich eine Kommandozeilenausgabe mit der Information erhalten, ob der Testlauf erfolgreich war oder nicht. Es wurden keine weiteren Reporting-Dateien erstellt. Das ist für einen integrierten Softwareentwicklungsprozess natürlich zu wenig. Was können wir hier machen? Wir fügen unserer zentralen *config.xml* zwei Attribute hinzu:

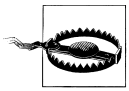
```
<config
  host="www.google.de"
  port="80"
  protocol="http"
  basepath="/"
  summary="true"
  saveresponse="true"
/>
```

Diese einfache Ergänzung des `summary`-Attributs und des `saveresponse`-Attributs an zentraler Stelle (wir haben ja die Testkonfiguration aus dem eigentlichen Testlauf extrahiert, wenn wir also mehrere Testskripten haben, ist diese Erweiterung tatsächlich nur einmal notwendig) führt dazu, dass der Testlauf innerhalb des gleichen Ordners ein Verzeichnis *webtest-results* anlegt. Dort befinden sich nun mehrere Dateien: zwei HTML-Dateien und *results.xml*. Führen Sie den Test erneut aus, werden Sie feststellen, dass WebTest den alten Report nicht löscht, sondern die neuen Daten anhängt. Für jeden weiteren Testlauf werden also zwei weitere HTML-Dateien angehängt und *results.xml* erweitert.

Die HTML-Dateien sind die Antworten des Servers auf unsere Anfragen, so wie sie auch im Browser dargestellt würden. Hier können wir also sehr genau verfolgen, was die Schritte unseres Testskripts für Server-Antworten produziert haben. In unserem Fall werden pro Testlauf zwei Antwortdateien erstellt, nach der Notation

```
response_XXX_invoke.html
response_XXX_clickButton.html
```

Der Name gibt bereits Aufschluss über den Kontext. *XXX* steht dabei für einen 13-stelligen numerischen Wert, der diese Response eindeutig kennzeichnet. Daneben gibt es *results.xml*. Das ist besonders interessant für uns. Sein Inhalt nach einem Testlauf ist in Beispiel 16-3 dargestellt. Diese Ergebnisdatei ist letztendlich für den Nutzer transparent, kann sie doch mit Stylesheets in grafische Repräsentationen umgewandelt werden. Dazu erfahren Sie später mehr. Wir möchten dennoch auf die Ergebnisdatei eingehen, um das Vorgehen von WebTest noch besser zu verstehen.



Mit neueren Versionen von WebTest kann die Ergebnisdatei anders aussehen. Sie ist nicht Bestandteil des öffentlichen API und dient hier lediglich dem Verständnis.

Beispiel 16-3: Die Ergebnisdatei results.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<summary Implementation-Title="Canoo Webtest" Implementation-Version="R_1393">
    <testresult endtime="Sat Dec 02 10:46:12 CET 2006"
        location="/home/michael/webtest/myexamples/SearchGoogle.xml:12: "
        starttime="Sat Dec 02 10:46:10 CET 2006" successful="yes" testspecname="myTest">
        <config>
            <parameter name="port" value="80"/>
            <parameter name="summary" value="yes"/>
            <parameter name="protocol" value="http"/>
            <parameter name="host" value="www.google.de"/>
            <parameter name="resultpath" value="/home/michael/webtest/myexamples/webtest-
results"/>
            <parameter name="saveprefix" value="response"/>
            <parameter name="basepath" value="/"/>
            <parameter name="resultfile" value="results.xml"/>
            <parameter name="haltonerror" value="yes"/>
            <parameter name="saveresponse" value="yes"/>
            <parameter name="haltonfailure" value="yes"/>
            <parameter name="timeout" value="300"/>
            <parameter name="showhtmlparseroutput" value="no"/>
        </config>
        <results>
            <step>
                <parameter name="description" value="start"/>
                <parameter name="method" value="GET"/>
                <parameter name="resultFilename" value="response_1165052771776_invoke.html"/>
                <parameter name="taskName" value="invoke"/>
                <parameter name="url" value="/"/>
                <result>
                    <completed duration="1003"/>
                </result>
            </step>
            <step>
                <parameter name="description" value="title"/>
                <parameter name="taskName" value="verifyTitle"/>
                <parameter name="text" value="Google"/>
                <result>
                    <completed duration="2"/>
                </result>
            </step>
            <step>
                <parameter name="description" value="set search query"/>
                <parameter name="name" value="q"/>
                <parameter name="taskName" value="setInputField"/>
                <parameter name="value" value="WebTest"/>
                <result>
                    <completed duration="2"/>
                </result>
```

Beispiel 16-3: Die Ergebnisdatei results.xml (Fortsetzung)

```
            </step>
            <step>
                <parameter name="description" value="click submit button"/>
                <parameter name="name" value="btnG"/>
                <parameter name="resultFilename" value="response_1165052772209_clickButton.
html"/>
                <parameter name="taskName" value="clickButton"/>
                <result>
                    <completed duration="422"/>
                </result>
            </step>
            <step>
                <parameter name="description" value="verify title on search result page"/>
                <parameter name="taskName" value="verifyTitle"/>
                <parameter name="text" value="WebTest - Google-Suche"/>
                <result>
                    <completed duration="0"/>
                </result>
            </step>
        </results>
    </testresult>
</summary>
```

Dieser Report enthält sehr viele wertvolle Informationen. Gehen wir der Reihe nach vor. Das Wurzelelement ist `summary`. Es enthält die Information darüber, wie die Testanwendung heißt und welche Version davon bei der Erstellung des Reports zum Einsatz kam.

```
<summary Implementation-Title="Canoo Webtest" Implementation-Version="R_1393">
```

Danach kommt ein `testresult`-Element. Es enthält den Start- und Endzeitpunkt des Tests, den Ort des Testskripts und den Namen des Tests, den wir im Testskript gewählt haben (hier `myTest`). Es gibt außerdem Aufschluss darüber, ob die Tests erfolgreich verlaufen sind. Dieses Element enthält alle weiteren Informationen zm Testlauf. Führen Sie den Test mehrfach aus, wird für jeden Testlauf ein `testresult`-Element in *results.xml* angehängt.

```
    <testresult endtime="Sat Dec 02 10:46:12 CET 2006"
        location="/home/michael/webtest/myexamples/SearchGoogle.xml:12: "
        starttime="Sat Dec 02 10:46:10 CET 2006" successful="yes" testspecname="myTest">
```

Anschließend folgen in einem `config`-Element diverse Konfigurationseinstellungen, mit denen der Testlauf durchgeführt wurde.

Danach werden alle Steps aufgezählt, die WebTest für uns durchgeführt hat und in einem `results`-Element einbettet. Diese Schritte befinden sich in einem `results`-Elementcontainer.

Generell sollten Sie Reports nur löschen, wenn sie, aufbauend auf einer einfach wiederherstellbaren Ausgangssituation, automatisch und schnell reproduzierbar sind. Bei vielen beteiligten technischen Komponenten, insbesondere bei Webanwendungen, kann das zeitintensiv und ineffizient sein. Sie sollten dann die Reports der Versionsverwaltung zuführen. Einmal in Subversion eingecheckt kann das Testergebnis jederzeit direkt angezeigt werden. Da heute aufgrund der erschwinglichen Kosten Speicherplatz keine kritische Größe mehr darstellt, sollten Sie im Zweifel eher zu viel als zu wenig ins zentrale Repository speichern. Begreifen Sie dann die Reports als weiteres zu versionierendes Artefakt neben den Testskripten.

Die Ausführungsinformationen der jeweiligen Steps werden als `parameter`-Elemente angeführt. Dazu gehört der `taskName`, also was WebTest in diesem Schritt genau macht. WebTest hat die mit HTTP per GET angeforderte HTML-Datei unter dem eindeutigen Namen abgespeichert.

Informationen zur Server-Antwort befinden sich im `result`-Element. Der Schritt dauert ca. eine Sekunde.

```
<step>
    <parameter name="description" value="start"/>
    <parameter name="method" value="GET"/>
    <parameter name="resultFilename" value="response_1165052771776_invoke.html"/>
    <parameter name="taskName" value="invoke"/>
    <parameter name="url" value="/"/>
    <result>
        <completed duration="1003"/>
    </result>
</step>
```

Danach hat WebTest mit `verifyTitle` den Test auf den Seitentitel durchgeführt. Auch hier sehen wir anhand der Beschreibung und von `taskName`, was gemacht wurde und was in diesem konkreten Fall der Sollwert des Tests war, nämlich der mit dem `text`-Parameter vorgegebene Google-Begriff.

```
<step>
    <parameter name="description" value="title"/>
    <parameter name="taskName" value="verifyTitle"/>
    <parameter name="text" value="Google"/>
    <result>
        <completed duration="2"/>
    </result>
</step>
```

Es folgt die Eingabe des Werts »WebTest« in das Suchfeld mit dem Namen `q`. Der WebTest-Step `setInputField` liegt zugrunde.

```
<step>
    <parameter name="description" value="set search query"/>
    <parameter name="name" value="q"/>
    <parameter name="taskName" value="setInputField"/>
    <parameter name="value" value="WebTest"/>
```

```
        <result>
            <completed duration="2"/>
        </result>
    </step>
```

Nach dem Step wird der Submit-Knopf namens *btnG* gedrückt. Die Server-Antwort wird in dem hier benannten Dokument im Verzeichnis gespeichert.

```
    <step>
        <parameter name="description" value="click submit button"/>
        <parameter name="name" value="btnG"/>
        <parameter name="resultFilename" value="response_1165052772209_clickButton.html"/>
        <parameter name="taskName" value="clickButton"/>
        <result>
            <completed duration="422"/>
        </result>
    </step>
```

Schließlich folgt die Überprüfung des Titels auf der Ergebnisseite:

```
    <step>
        <parameter name="description" value="verify title on search result page"/>
        <parameter name="taskName" value="verifyTitle"/>
        <parameter name="text" value="WebTest - Google-Suche"/>
        <result>
            <completed duration="0"/>
        </result>
    </step>
```

Wir hatten zu Beginn nicht so viel Glück mit unserem Testlauf. Der fehlgeschlagene Test hatte folgende Ausgabe:

```
    <step>
        <parameter name="description" value="verify title on search result page"/>
        <parameter name="taskName" value="verifyTitle"/>
        <parameter name="text" value="WebTest - Google-Suche"/>
        <result>
            <failed duration="0"/>
        </result>
    </step>
    <failure message="com.canoo.webtest.engine.StepFailedException: Wrong document title
found! Expected &quot;WebTest - Google-Suche&quot; but got &quot;webTest - Google-Suche
&quot;, Step: VerifyTitle at /home/michael/webtest/myexamples/SearchGoogle.xml:28:  with
(description=&quot;verify title on search result page&quot;, taskName=&quot;verifyTitle
&quot;, text=&quot;WebTest - Google-Suche&quot;)"/>
```

Wir sehen als Unterschied zum erfolgreichen Test, dass das `result`-Element diesmal nicht `completed` enthält, sondern `failed`. Darüber hinaus gibt es nun eine `failure`-Nachricht nach dem Schritt. Dieses XML-Dokument ist hilfreich, aber doch recht mühselig auszuwerten. Doch auch dazu gibt es eine Lösung.

Kontinuierliche Integration

Wir schauen uns nun ein Beispiel an, das einerseits neue Funktionen und Anwendungsmöglichkeiten von WebTest illustriert und andererseits einen Weg zeigt, wie wir die Testergebnisse zusätzlich noch visuell darstellen können. Unser Beispiel wird durch eine strukturierte Form aufgeräumter sein, weitere WebTest-Steps illustrieren und einen prototypischen Build-Prozess illustrieren.

In diesem Beispiel, das in Beispiel 16-4 dargestellt ist, führen wir erneut eine Google-Suche durch, um auf der Ergebnisseite mehrere Seiten vorzublättern, ehe wir dann zur Demonstration einen Testfehler provozieren. Das Skript werden wir mit Ant automatisiert laufen lassen. Testergebnisse werden wir in Form von HTML-Seiten darstellen lassen.

Das Testskript

Alle Dateien liegen der Einfachheit halber im selben Verzeichnis. Fangen wir mit dem Skript an, das die Testskripten enthält. Wir benennen es um nach *Testskripts.xml*, da es nun mehrere Testsequenzen beinhaltet.

Beispiel 16-4: Ein Testskript mit unterschiedlichen Testkommandos und Testwiederholungen sowie einem provozierten Testfehler

```
<?xml version="1.0"?>
<!DOCTYPE project SYSTEM "/home/michael/webtest/selftests/config/WebTest.dtd"[
  <!ENTITY config     SYSTEM "config.xml">
]>

<project name="SearchGoogle" basedir="." default="main">
  <property name="webtest.home" location="/home/michael/webtest" />
  <import file="${webtest.home}/lib/taskdef.xml"/>
  <target name="main">
    <webtest name="SearchGoogle">
      &config;
      <steps>
        <invoke
          description="start"
          url="/" />
        <verifyTitle
          description="title"
          text="Google" />
        <setInputField description="set search query"
          name="q"
          value="WebTest" />
        <clickButton
          name="btnG"
          description="click submit button" />
        <verifyTitle description="verify title on search result page"
          text="WebTest - Google-Suche"  />
      </steps>
    </webtest>
    <webtest name="SearchPictures">
```

Beispiel 16-4: Ein Testskript mit unterschiedlichen Testkommandos und Testwiederholungen sowie einem provozierten Testfehler (Fortsetzung)

```
      &config;
      <steps>
        <invoke
          description="start"
          url="/" />
        <verifyTitle
          description="title"
          text="Google" />
        <clickLink
          label="Bilder"
          description="click to pictures"/>
        <setInputField description="set search query"
          name="q"
          value="agile development" />
        <clickButton
          name="btnG"
          description="click submit button" />
        <verifyText description="verify if text exists"
          text="Ergebnisseite"  />
        <repeat count="3">
          <clickLink
             label="Vorwärts"
             description="forward"/>
        </repeat>
        <verifyText description="result count"
          text="55"/>
        <setInputField description="set field, provoke error"
          name="abc"
          value="break me"/>
        <verifyText description="search field"
          text="find me"/>
      </steps>
    </webtest>
  </target>
</project>
```

Direkt springt ins Auge, dass es mehrere `webtest`-Elemente gibt. Die erste Testsequenz namens `SearchGoogle` kennen wir bereits, die zweite mit dem Namen `SearchPictures` sucht den Begriff »agile development« in den Google-Bildern. Dafür wird zunächst auf den Bilder-Link der Google-Startseite geklickt, der Suchbegriff eingegeben und dann gesucht.

Ein `webtest`-Element sollte einen Akzeptanztest umfassen. Das Testskript in Beispiel 16-4 besitzt zwei `webtest`-Elemente, wobei das erste den Test des einfachen Vorbeispiels enthält. Es kann hilfreich sein, pro WebTest-Skript ein `webtest`-Element zu schreiben, um einen Test separat und isoliert testen zu können.

Nun überprüfen wir mit verifyText, ob der Begriff »Ergebnisseite« auf der von Google gelieferten Seite übertragen wird. Dies sollte der Fall sein.

```
<verifyText description="verify if text exists"
 text="Ergebnisseite"  />
```

Im Anschluss führen wir einen Schritt mehrfach durch. WebTest bietet nicht nur Kommandos und Tests, sondern steuernde Ausdrücke wie repeat. Alle im repeat-Element geschachtelten Elemente werden so oft durchgeführt, wie das Attribut count vorgibt.

```
<repeat count="3">
  <clickLink
     label="Vorwärts"
     description="forward"/>
</repeat>
```

Wir simulieren mit diesem Schritt, dass der Nutzer dreimal auf den Link »Vorwärts« geht, um sich weitere Ergebnisseiten anzeigen zu lassen. Wir überprüfen den Erfolg mit einem Test auf den Text »55«, da pro Ergebnisseite standardmäßig 18 Ergebnisse dargestellt werden.

```
<verifyText description="result count"
  text="55"/>
```

Nun möchten wir einen Fehler provozieren, um das Verhalten von WebTest zu illustrieren bzw. die Auswirkungen auf das im Anschluss vorgestellte Reporting zu verdeutlichen. Wir möchten in das Feld mit dem Namen abc einen Wert eingeben. Das Feld ist nicht da, so dass es hier einen Fehler geben wird. Der letzte WebTest-Step, die Überprüfung auf den Text »find me«, wird deswegen nicht mehr ausgeführt.

```
<setInputField description="set field, provoke error"
  name="abc"
  value="break me"/>
<verifyText description="search field"
  text="find me"/>
```

Die einbezogene *config.xml* belassen wir unverändert.

```
<config
  host="www.google.de"
  port="80"
  protocol="http"
  basepath="/"
  summary="true"
  saveresponse="true"
/>
```

Das Ant-Skript

Darüber hinaus fügen wir nun noch das Ant-Skript namens *build.xml* hinzu (siehe Beispiel 16-5). Das ist unser zentraler Einstieg für die Tests mit WebTest.

Beispiel 16-5: Das Ant-Skript zur Ausführung unseres Testskripts inklusive Reporting

```
<project name="WebTest" basedir="." default="main">
   <property name="webtest.home" value="/home/michael/webtest"/>
   <property name="webtest.config.resultpath" value="/home/michael/webtest/myexamples/webtest
-results"/>
   <property name="webtest.config.resultfile" value="results.xml"/>
   <property name="webtest.config.resultfile.html" value="${webtest.config.resultpath}/results.
html"/>

   <target name="main">
      <echo message="run WebTest"/>
      <exec dir="." executable="sh">
         <arg line="webtest.sh -buildfile TestScripts.xml"/>
      </exec>
      <formatResults/>
   </target>

   <macrodef name="formatResults" description="formats the xml result file using the xslt">
      <sequential>
         <tstamp>
            <format property="report.time" pattern="dd.MM.yyyy HH:mm"
             locale="de"/>
         </tstamp>
         <property name="resources.dir" value="${webtest.home}/resources"/>
         <xslt basedir="${webtest.config.resultpath}"
            destdir="${webtest.config.resultpath}"
            includes="${webtest.config.resultfile}"
            force="true" extension=".html"
            style="${resources.dir}/WebTestReport.xsl">
            <param name="reporttime" expression="${report.time}"/>
            <param name="title" expression="${ant.project.name}"/>
         </xslt>

       <copy todir="${webtest.config.resultpath}">
          <fileset dir="${resources.dir}">
             <include name="report.css"/>
             <include name="showHide.js"/>
               <include name="images/*.*"/>
               <include name="responseBr*.*"/>
            </fileset>
         </copy>

         <echo message="Webtest result overview available in ${webtest.config.resultfile.html
}"/>
      </sequential>
   </macrodef>
</project>
```

Was haben wir hier gemacht? Nachdem wir diverse wiederverwendbare zentrale Properties gesetzt haben, kommt auch schon direkt das standardmäßige Ansprungsziel *main*. Wir geben hier eine kleine Kontrollnachricht aus und führen mit dem *exec*-Task von Ant unser WebTest-Skript aus. Da wir hier auf Linux arbeiten und unsere auszuführende

Datei ein Shell-Skript ist, wählen wir als executable *sh*. Als Argumente übergeben wir den String, den wir sonst auf der Konsole eingeben würden. Das reicht, um alle Tests laufen zu lassen.

```
<target name="main">
   <echo message="run WebTest"/>
   <exec dir="." executable="sh">
      <arg line="webtest.sh -buildfile TestScripts.xml"/>
   </exec>
   <formatResults/>
</target>
```

Neben *webtest.sh* steht für Windows-Systeme eine *.bat*-Datei namens *webtest.bat* zur Verfügung. Um plattformübergreifend auf dieselbe Weise auf die Skripten zuzugreifen, sprechen Sie sie direkt als Ant-Skripten an und rufen Sie sie mit *ant* auf.

Doch es geht noch weiter: Anschließend führen wir das Ant-Makro namens `formatResults` aus. Der mit `macrodef` gekennzeichnete Block wird zur Laufzeit eingebunden.

Die im Verzeichnis *home* der WebTest-Distribution befindliche Datei *webtest.xml* enthält diverse Hilfskonstrukte, unter anderem auch das `formatResults`-Makro. Ein Import von *webtest.xml* mit `<import file="${webtest.home}/webtest.xml/>` in Ihrem Skript genügt, um das Makro direkt nutzen zu können.

Dieses Makro definiert einzelne Aktivitäten, die sequenziell durchgeführt werden: Hier wird zunächst ein aktueller Zeitstempel angelegt. Erinnern Sie sich: Für aussagekräftiges Tracing ist es sinnvoll, die Artefakte mit solchen Metadaten anzureichern. Der Zeitwert wird in *report.time* gesichert.

```
<tstamp>
   <format property="report.time" pattern="dd.MM.yyyy HH:mm"
      locale="de"/>
</tstamp>
```

Danach führen wir mit dem `xslt`-Task von Ant eine Stylesheet-Transformation durch. Als Eingabe wird der uns bekannte Testergebnisreport genommen und mit dem in der WebTest-Distribution mitgeliefertem Stylesheet *WebTestReport.xsl* von XML nach HTML transformiert. Als Parameter werden die Zeit und ein Titel übergeben. Diese Informationen werden in den Report eingebunden. Eine im Zielverzeichnis aus einer zuvor durchgeführten Transformation angelegte HTML-Datei wird durch die neue ersetzt.

```
<xslt basedir="${webtest.config.resultpath}"
   destdir="${webtest.config.resultpath}"
   includes="${webtest.config.resultfile}"
   force="true" extension=".html"
   style="${resources.dir}/WebTestReport.xsl">
   <param name="reporttime" expression="${report.time}"/>
   <param name="title" expression="${ant.project.name}"/>
</xslt>
```

Ist die HTML-Datei erstellt, werden alle zur Darstellung notwendigen Ressourcen in dasselbe Zielverzeichnis kopiert.

```
<copy todir="${webtest.config.resultpath}">
   <fileset dir="${resources.dir}">
      <include name="report.css"/>
      <include name="showHide.js"/>
     <include name="images/*.*"/>
     <include name="responseBr*.*"/>
  </fileset>
</copy>
```

Eine Nachricht, dass der Report verfügbar ist, schließt unseren Prozess ab.

Nun führen wir *build.xml* aus. Dazu geben wir `ant` ein und erhalten die in Beispiel 16-6 dargestellte Ausgabe.

Beispiel 16-6: Die Ausgaben von Ant

```
michael@BlueGene:~/webtest/myexamples$ ant
Buildfile: build.xml

main:
     [echo] run WebTest
     [exec] Will use /usr/local/java/jdk1.6.0/bin/java
     [exec] Buildfile: TestScripts.xml
     [exec] Trying to override old definition of task sleep

     [exec] main:

     [exec] BUILD FAILED
     [exec] /home/michael/webtest/myexamples/TestScripts.xml:32: Canoo Webtest: R_1393.
     [exec] Test failed.
     [exec] Test step setInputField (/home/michael/webtest/myexamples/TestScripts.xml:61: )
named "set field, provoke error" failed with message "No suitable form found having field
named "abc""

     [exec] Total time: 6 seconds
     [exec] Result: 1
     [xslt] Transforming into /home/michael/webtest/myexamples/webtest-results
     [xslt] Processing /home/michael/webtest/myexamples/webtest-
results/results.xml to /home/michael/webtest/myexamples/webtest-results/results.html
     [xslt] Loading stylesheet /home/michael/webtest/resources/WebTestReport.xsl
     [copy] Copying 14 files to /home/michael/webtest/myexamples/webtest-results
     [echo] Webtest result overview available in /home/michael/webtest/myexamples/webtest-
results/results.html

BUILD SUCCESSFUL
Total time: 8 seconds
```

Wir sehen, dass das Ant-Skript namens *Testskripts.xml* WebTest ausgeführt hat. Es wird dabei der von uns erwartete Fehler angezeigt das Feld »`abc`« sei nicht vorhanden. Das ist beispielsweise der Fall, wenn wir in einem TDD-Prozess dieses Feld noch nicht bereitge-

stellt haben. Der Test schlägt fehl. Wir könnten im weiteren Verlauf das Feld hinzufügen, so dass der Test erfolgreich durchläuft. Der Test treibt die Entwicklung der Anwendung vor sich her.

```
[exec] Test step setInputField (/home/michael/webtest/myexamples/TestScripts.xml:61: )
named "set field, provoke error" failed with message "No suitable form found having field
named "abc""
```

Dieser Schritt hat sechs Sekunden gedauert. Wir können weiterhin die XSLT-Transformation und das Kopieren der Dateien verfolgen. Das Ende des Durchlaufs wird signalisiert durch

```
/home/michael/webtest/myexamples/webtest-results/results.html
```

Nun schauen wir uns die Datei *results.html* an (siehe Abbildung 16-2). Es ist eine recht umfangreiche, gesprächige Datei, der wir alle spannenden Informationen entnehmen können. Die Startseite gibt einen umfassenden Überblick über unsere Tests bzw. die Schritte, die von WebTest durchgeführt wurden. Im oberen Bereich des Reports sehen wir auf der linken Seite die Anzahl der Steps auf der Zielanwendung bis auf ihr Aufrufen mit `invoke`. Wir haben 13 erfolgreiche Steps auf der Anwendung durchgeführt. Darüber hinaus ist eine fehlgeschlagen, und eine wurde deswegen nicht erreicht. Im rechten Bereich daneben haben wir die Schritte, die eine Server-Kommunikation zur Folge hatten, wie zum Beispiel das Drücken eines Buttons. Von allen acht Schritten haben sieben weniger als eine Sekunde benötigt. Die Ausführung des achten Steps hat zwischen einer und fünf Sekunden gedauert. Auch hier sind neben den reinen Zahlen auch Balkendiagramme zu sehen, die das Ergebnis grafisch aufbereiten.

Oben sehen Sie einen Überblick über unsere Tests und die Testszenarien

Im unteren Teil des Reports sehen wir unsere unterschiedlichen Testszenarien: *SearchGoogle* ist mit fünf erfolgreich durchgeführten Steps aufgelistet, *SearchPictures* mit insgesamt 13, von denen 11 erfolgreich durchgelaufen worden sind. Die fehlerhaften Steps werden rechts dokumentiert.

Klicken wir auf den Link zu *SearchGoogle*, können wir uns die Details zu dieser konkreten Sequenz anschauen (siehe Abbildung 16-3).

Das Gleiche gilt für *SearchPictures*. Die Oberfläche lässt es zu, geschachtelte Steps zu öffnen und zu schließen. Unsere Wiederholungssequenz kann somit aufgeklappt und komplett analysiert werden (siehe Abbildungen 16-4 und 16-5).

Zusammenfassung

Webanwendungen können wir funktional testen und mit Akzeptanztests versehen. Funktionale Tests, die Überprüfungen vornehmen, ob die Anwendung die zu erwartende Funktionalität besitzt, lassen sich mit Fit, Selenium und WebTest durchführen. WebTest baut auf Ant auf. Dadurch, dass die Skripten in XML geschrieben sind und eine WebTest-DTD vorliegt, können IDEs Sie beim Schreiben von WebTest-Skripten direkt unter-

canoo
WebTest Tests started at Sat Dec 02 13:02:56 CET 2006

Result Summary

Result	#	%	Graph
✓	13	87	
✗	1	7	
○	1	7	
Sum	**15**	**100**	

Server Roundtrip Timing Profile

Secs	#	%	Histogram
0 - 1	7	88	
1 - 5	1	13	
5 - 10	0	0	
10 - 30	0	0	
> 30	0	0	
Sum	**8**	**100**	
Avg		**1**	

Test Scenario Overview (4820)

No	Result	Name	# Steps	Timing profile			Failing step
				Duration	%	Graph	
1	✓	SearchGoogle	5 / 5	1468	30		
2	✗	SearchPictures	11 / 13	3352	70		**setInputField** set field, provoke error

Abbildung 16-2: Oben sehen Sie einen Überblick über unsere Tests und die Testszenarien

stützen (sofern sie eine XML-Unterstützung haben). Der WebTest-Recorder ermöglicht das Aufzeichnen von Oberflächeninteraktionen, um Interaktionen auf bestehenden Anwendungen (bzw. Teilen davon) aufzuzeichnen und als WebTest-Skript darzustellen.

Akzeptanztests sind funktionale Tests, legen aber zusätzlich noch den Schwerpunkt darauf, dass einerseits der Anwender (bzw. Kunde) selbst direkt bei der Spezifikation der Funktionalität beteiligt ist und das Medium, mit dem er diese Akzeptanztests erfasst, von einem Mechanismus automatisch auf die Anwendung angewendet wird, und dass andererseits diese Spezifikation *vor* dem Schreiben der Anwendung durchgeführt werden kann. Die Begriffe »funktionale Tests« und »Akzeptanztests« werden in der Praxis häufig synonym verwendet.

Die mithilfe von WebTest erstellten Akzeptanztests in Form von Ant-Skripten können manuell auf der Konsole und gleichzeitig mit jeder IDE angestoßen werden und gleichzeitig Bestandteil eines Continuous Integration-Prozesses sein. Die Information, ob ein Test erfolgreich durchlaufen worden ist, wird in Form von WebTest-Reportseiten gegeben.

✓ SearchGoogle

Test started at Sat Dec 02 13:02:56 CET 2006, lasting 1468 ms.
Source:
/home/michael/webtest/myexamples/TestScripts.xml:12:
Base URL (used by **invoke** steps with a relative URL):
http://www.google.de///

No	Result	Name	
1	✓	**invoke** start Resulting page	**method** GET **url** /
2	✓	**verifyTitle** title	**text** Google
3	✓	**setInputField** set search query	**name** q **value** WebTest
4	✓	**clickButton** click submit button Resulting page	**name** btnG
5	✓	**verifyTitle** verify title on search result page	**text** WebTest - Google-Suche

Abbildung 16-3: Details zu einer Sequenz

Um asynchronen Code zu testen, können die Asynchronität durch Mocks ausgeschaltet bzw. wohldefinierte isolierte Synchronisierungspunkte eingeführt werden (bei Unit-Tests) oder über die Oberfläche Interaktionen durchgeführt werden (funktionale Tests), und das Test-Framework (wie Selenium) behandelt die Asynchronität selbst.

Bei der Nutzung von Fit schreiben Kunden die Tests gewöhnlich mit Word. Die Selenium-Tests sind der ActionFixture aus Fit recht ähnlich. Selenium und Fit können integriert werden: Dazu werden Akzeptanztests mit Fit geschrieben, und die Fixtures steuern den Browser mit Selenium. Kann der Nutzer XML verstehen und schreiben (oder existiert ein Kunden-Proxy, der sich mit der Thematik befasst), ist das Vorgehen eleganter, als mit Fit Tabellen zu editieren (auch wenn das über eine Standardsoftware wie Word geschieht). Auf der anderen Seite geschieht die Rückkopplung bei Fit durch Anreicherung der Ausgangstabellen schneller und direkt.

Um JavaScript zu testen, kann *Rhino* eingesetzt werden, eine JavaScript-Engine, die ohne einen Browser auskommt. *JsTester* ermöglicht das JavaScript-Testen mit der Rhino-Engine ohne Browser und ist mit JUnit integrierbar. *Scriptaculous* ist ein Test-Framework, das die Tests im Browser ausführt. Dabei werden Unit-Tests als Java-Script entwickelt. *JsMock* ist ein Mock-Framework für JavaScript. JsMock ist EasyMock sehr ähnlich.

✕ SearchPictures

Test started at Sat Dec 02 13:02:57 CET 2006, lasting 3352 ms.
Source:
/home/michael/webtest/myexamples/TestScripts.xml:32:
Base URL (used by **invoke** steps with a relative URL):
http://www.google.de///

Expand/Collapse nested steps:

No	Result	Name	
1	✓	**invoke** start Resulting page	**method** GET **url** /
2	✓	**verifyTitle** title	**text** Google
3	✓	**clickLink** click to pictures Resulting page	**label** Bilder
4	✓	**setInputField** set search query	**name** q **value** agile development
5	✓	**clickButton** click submit button Resulting page	**name** btnG
6	✓	**verifyText** verify if text exists	**text** Ergebnisseite

Abbildung 16-4: Zweite Sequenz, Teil 1

WebTest wird nicht zum Einsatz kommen, wenn die Tests in JUnit Tests eingebunden werden sollen (dann Selenium), über dasselbe Test-Framework auch Java Swing-Anwendungen getestet werden sollen (dann Fit + Jemmy) oder wenn die zu testende Webanwendung sehr viel JavaScript enthält (dann Selenium). JavaScript in Webanwendungen kann hier der ausschlaggebende Punkt sein. JavaScript lässt sich zwar mittlerweile auch mit anderen Mechanismen testen, zum Beispiel mit dem GWT-Framework, das Java-Code nach JavaScript übersetzt (die Theorie dabei ist, dass dann nur noch der Java-Code getestet werden muss). In der Praxis kann aber nicht *jeder* Java-Code nach JavaScript übersetzt werden, und es ist fraglich, ob dieser Umweg in der Praxis praktikabel ist. Eine separate Betrachtung der JavaScript-Bestandteile einer Webanwendung scheint doch eher theoretischer Natur. Durch Nutzung eines Werkzeugs wie Selenium ist es dem Anwender egal, ob und wenn ja was für JavaScript in der Anwendung vorkommt. Hier sind funktionale Tests Komponententests gegenüber vorzuziehen bzw. die einzige Wahl.

7	✓	**repeat**	**count** 3

NoResult		Name
1	✓	**clickLink** **label** Vorwärts forward Resulting page
2	✓	**clickLink** **label** Vorwärts forward Resulting page
3	✓	**clickLink** **label** Vorwärts forward Resulting page

8	✓	**verifyText** result count	**text** 55
9	✗ Error	**setInputField** set field, provoke error	**name** abc **value** break me
10	▫	**verifyText** search field	**text** find me

Failure

Message

com.canoo.webtest.engine.StepFailedException: No suitable form found having field named "abc", Step: SetInputField at /home/michael/webtest/myexamples/TestScripts.xml:61: with (description="set field, provoke error", name="abc", taskName="setInputField", value="break me")

Abbildung 16-5: Zweite Sequenz, Teil 2

KAPITEL 17

Projektkommunikation mit Trac

Die Kommunikation innerhalb des Teams ist eine wesentliche Komponente agiler Projekte. Im Rahmen dieser Kommunikation werden die Beteiligten über den Projektinhalt und seinen Fortschritt stetig informiert. Außerdem sollen Schwierigkeiten bei der Realisierung der einzelnen Features genauso bekannt gemacht werden wie das Wissen des Teams selbst. Es muss eine Infrastruktur aufgesetzt werden, die sowohl Informationsaustausch als auch ständiges Lernen fördert. Auch die besten IT-Experten lernen jeden Tag etwas Neues. Und neue Erkenntnisse müssen so aufbereitet werden, dass die Person selbst jederzeit wieder darauf zugreifen kann. Aber auch die Kollegen sollen davon profitieren.

Als technische Lösung für diese Kommunikation möchte ich Ihnen Trac vorstellen. Es verbindet unterschiedliche Funktionen: Es ist einerseits ein Wiki, andererseits aber auch Ticketing-System und Projektmanagement-Werkzeug.

Wikis

Ein Wiki ist eine Seitensammlung, die von Benutzern sowohl gelesen als auch editiert werden kann. Ähnlich einem Content Management System (CMS) lässt sich auf diese Weise Inhalt beliebiger Art *kollaborativ* fortschreiben. Ein Wiki ist allerdings eine schlanke Variante des CMS, die sich durch viele interne und externe Querverweise auszeichnet. Konzeptionell kann dabei jeder Nutzer neue Seiten anlegen, ändern und auch löschen, wobei das durch ein Rechtesystem eingeschränkt werden kann. Ein CMS auf der anderen Seite unterliegt gewöhnlich umfassenden redaktionellen Prüfungen und einer detaillierten Berechtigungsstruktur.

Wikis können über das Internet verfügbar gemacht werden. Das beste Beispiel dafür ist die *Wikipedia*. Andererseits können Sie auch einer geschlossenen Nutzergruppe zur Verfügung gestellt werden, wie es in einem Intranet innerhalb einer Unternehmung der Fall ist.

Der Begriff Wiki stammt von dem hawaiischen Wort »wikiwiki«, das so viel wie *schnell* bedeutet. Das erste Wiki wurde 1995 von Ward Cunningham ins Leben gerufen. Die Grundidee eines Wiki ist nicht wirklich neu: Tim Berners-Lee, der Erfinder des World Wide Web, hat Anfang der 90er Jahre bei der Präsentation des *WWW* ein Konzept vorge-

stellt, bei dem Seiten öffentlich verfügbar und von jedermann editierbar sein sollten. Was heute als Wiki »verkauft« wird und durch den Hype *Web 2.0* neue Aufmerksamkeit erlangte, ist in Wirklichkeit, zumindest konzeptionell, ein alter Ansatz.

Eigenschaften

Wikis bieten eine ausgezeichnete Möglichkeit, Wissensmanagement aktiv zu praktizieren. Die sich durch den Einsatz ergebenden Vorteile (z.B. gegenüber klassischen Medien wie Word-Dokumenten) lassen sich wie folgt zusammenfassen:

- Die Seiten sind leicht editierbar.
- Es gibt eine Versionskontrolle (*wer* hat *was wann* geändert) durch Rückgriff auf ältere Versionen einer Seite.
- Die Seiten sind leicht zu verlinken und so zueinander in Beziehung zu setzen.
- Hierarchische Strukturen sind leicht zu erstellen. Durch eine geeignete Verlinkung können beispielsweise auf einer Einführungsseite zunächst die wesentlichen Informationen dargelegt werden. Unterseiten, die zu einzelnen Themen weiterführende Informationen liefern, werden einfach verlinkt.
- Das kollektive Projektwissen ist zentral verfügbar und kann gemeinsam weiterentwickelt werden (Texte, Informationen, strukturierte Daten).
- Die Bedienung ist einfach, auch für technisch nicht versierte Benutzergruppen.
- Eine Volltextsuche ist verfügbar.
- Der Zugang ist generell von überall aus möglich und geschieht einfach über einen Webbrowser.
- Eine Infrastruktur wird bereitgestellt, mit der das Projekt und die daran Beteiligten untereinander kommunizieren können.
- Eine E-Mail-Flut wird verhindert, da das Wiki das zentrale Medium ist, in dem Wissen hinterlegt wird.
- Es handelt sich um »spielerisches« Dokumentieren, was im Gegensatz zu schwergewichtigen Alternativen wie Word-Dokus sogar Spaß machen kann.
- Das Projektmanagement wird durch Wikis unterstützt.
- Wikis verfügen gewöhnlich über eine Preview-Funktion, durch die eine Änderung überprüft werden kann, bevor sie gespeichert wird.
- Wikis lassen die Grenze zwischen Besuchern und Autoren verschwimmen, indem beide Gruppen weitestgehend gleiche Rechte haben. Wikis verzichten auf eine feste Inhaltsstruktur und Metainformationen und emulieren diese durch Links. Wikis folgen ferner nicht dem Modell des Inhaltslebenszyklus (»Content Life Cycle«), sondern betrachten die Gesamtheit des Inhalts als evolutionäres Gebilde. Prozesse sollen durch Wikis nicht abgebildet, sondern von ihnen nur unterstützt werden. Mitarbeiter sollen motiviert und ermutigt werden, ihr Wissen einzubringen. Das gelingt auch dadurch, dass die Nutzungshürde nicht sehr hoch ist.

Trac

In der praktischen Arbeit in agilen Projekten hat sich Trac als sehr hilfreich erwiesen. Es zeichnet sich durch seine verhältnismäßig einfache Bedienung bei gleichzeitiger Mächtigkeit aus. Trac ist frei verfügbar.

Trac ist mehr als ein Wiki: Es enthält außerdem ein eigenes, leichtgewichtiges *Ticketing-System*, das sich gut konfigurieren lässt. Die Tickets werden dabei direkt in das Wiki eingebunden, das heißt sie können aus anderen Bereichen des Wikis referenziert werden: Wir werden z.B. über Änderungen an den Tickets über eine besondere Wiki-Seite aufmerksam gemacht o.Ä. Abhängig von Ihrer Projektsituation kann diese All-in-One-Lösung inklusive Ticketing-System perfekt geeignet sein.

Ticketing-System

Ein Ticketing-System ist eine Anwendung, über die Aktivitäten und Aufgaben (»Tickets«) erfasst und an Personen vergeben werden können. Auf diesem Weg ist transparent, welche Aufgaben derzeit vorliegen und wer sie bearbeitet. Ticketing-Systeme erlauben nicht nur das Anlegen von Tickets, sondern auch ihre Fortschreibung (Ergänzung neuer Erkenntnisse), Priorisierung und Statusabfragen.

Wenn Sie z.B. einen Kundenservice anrufen und ein Problem schildern, wird das gewöhnlich aufgenommen und mit Hilfe eines Ticketing-Systems verfolgt.

Trac verfügt über eine Integration von *Subversion*. Auf diese Weise können Sie aus dem Wiki auf Ihren Java-Quellcode zugreifen und ihn sich darstellen lassen. Subversion wird im Detail in Versionskontrolle mit Subversion diskutiert.

Agiles Arbeiten mit Trac

Trac unterstützt die agile Entwicklung

- als Wiki, um Ideen jederzeit aufzunehmen sowie Brainstorming und Wissensmanagement zu ermöglichen.
- als Wiki, um leichtgewichtig und dezentral *User Stories* fortzuschreiben und auszutauschen.
- als Ticketing-System, um sämtliche Arten von Defekten zu verfolgen, also Bugs, Tasks, Erweiterungen usw.
- als Browser für den Quellcode durch die Subversion-Unterstützung.
- um an zentraler Stelle die *Roadmap* zu veröffentlichen und mit den Einträgen des Ticketing-Systems zu synchronisieren.
- Als eine Zeitlinie, die jederzeit aktualisiert wird und über Änderungen informiert (chronologische Auflistung aller Änderungen auf einer expliziten Wiki-Seite).

- indem Sie die von Trac bei der Pflege von Tickets so genannten *Milestones* als Iterationen in Ihrem agilen Projekt begreifen
- indem Sie User Stories als Ticket vom Typ »Enhancement« fortschreiben
- indem Sie die im Ticketing-System eingestellten User Stories detailliert in technische Arbeitspakete zerschneiden und diese als *Tasks* fortschreiben.
- indem Sie sich die Trac-Funktionalität »Tickets per Milestone« zu Nutze machen und so verfolgen, welche Tasks und Stories in einer bestimmten Iteration bzw. einem Meilenstein abgearbeitet werden.

Ein zentrales Feature von Trac ist die *Roadmap*. Letztendlich ist es nur eine unabhängige Seite im Wiki, die aber Gold wert sein kann: Hier können Sie zentral Planungsdaten hinterlegen (Meilensteinname und Termin), die ggf. zusammen mit den Tickets Aufschluss über den Projektfortschritt geben. Daneben gibt es noch eine *Timeline*. Sie zeigt alle Ereignisse, also geänderte bzw. neue Inhalte im Wiki, Änderungen in Subversion und Änderungen im Ticketing-System in chronologischer Reihenfolge.

Technische Architektur

Trac ist in Python geschrieben und verfügt über einen eigenen Webserver namens `tracd`, kann aber auch mit CGI, FastCGI oder `mod_python` betrieben werden (falls Sie weitreichende Konfigurationen benötigen). Die Nutzung von Apache kann sinnvoll sein, wenn Ihnen das mit Trac mitgelieferte Berechtigungssystem nicht ausreicht oder Skalierung wichtig ist (weil Sie beispielsweise viele Nutzer haben). Gleiches gilt für die Datenbank: Die mitgelieferte transaktionale *SQLite*-Datenbank kann durch MySQL oder PostgreSQL ersetzt werden. Beides mag in besonderen Konstellationen sinnvoll sein, immer notwendig ist es sicher nicht. Die standardmäßige Konfiguration ist leichtgewichtig, so wie es ein Wiki sein sollte, kann aber gleichwohl mit Plugins angereichert werden, und bestimmte Funktionen können durch andere Lösungen substituiert werden.

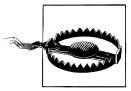
Installationshinweise

Im weiteren Verlauf gehen wir aber davon aus, dass wir unsere User Stories ausschließlich auf Story Cards schreiben, und werden die Ticketing-Funktion von Trac während seiner Installation deaktivieren. Auch die Subversion-Integration nutzen wir an dieser Stelle nicht, um uns auf das Wiki als solches zu konzentrieren. Wir schalten diese Funktionen ab, so dass Sie auf diesem Weg auch die umfassenden Konfigurationsmöglichkeiten von Trac kennenlernen.

Trac nutzt verschiedene Komponenten bzw. setzt sie voraus. Da es einige Abhängigkeiten gibt, empfiehlt sich ein Blick in die Installationsanweisungen auf der offiziellen Trac-Seite. Dort werden allgemeine Hinweise gegeben, ergänzt mit betriebssystemspezifischen Informationen und solchen zur Konfiguration. Auch die Mailing-Liste ist immer eine gute Anlaufstation. Dennoch kann die Installation eine Herausforderung sein.

Wir gehen im weiteren Verlauf von Kubuntu als Plattform aus. Trac kann dort auch über den Paketmanager eingerichtet werden.

Da einige Punkte zu beachten sind und die Versionen der Komponenten kompatibel zueinander sein müssen, gehe ich im Folgenden auf eine exemplarische Installation ein: Zunächst muss Python installiert sein. Ich nutze Ubuntu, kernel 2.6.15-23-386, mit Python 2.4.3. Zunächst besorgen wir uns *trac-0.10.tar.gz* von der Downloadseite auf *http://trac.edgewall.org/*.



Trac 0.10 ist die Folgeversion von Trac 0.9.6, also verwechseln Sie sie nicht mit Trac 0.1 oder Trac 0.1.0.

Dieses entpacken wir in ein Verzeichnis unserer Wahl.

```
python ./setup.py install --prefix=~/tools/trac
```

Anschließend installieren wir die von Trac verwendete Template-Engine *Clearsilver*, die für die eigentliche Darstellung der Inhalte zum Einsatz kommt. Dazu laden wir uns die Version *clearsilver-0.9.3.tar.gz* von *http://www.clearsilver.net/* herunter und entpacken sie. Mit neueren Versionen von Clearsilver ist es bis jetzt offensichtlich nicht möglich, eine lauffähige Installation vorzubereiten.

Clearsilver muss mit sogenannten Python-Bindings konfiguriert werden. Sie benötigen für die nächsten Schritte Root-Rechte auf Ihrem Rechner:

```
./configure --with-python=/usr/bin/python

make make
make
make install
```

Sie sollten sich auch *MediaWiki* und das kostenpflichtige *Confluence* anschauen. Wenn Trac nichts für Sie ist, wird Ihnen eines der beiden sicher helfen können.

Anschließend gehen Sie in das Verzeichnis *clearsilver-0.9.3/python/* und führen als Benutzer *root* folgenden Befehl aus:

```
python ./setup.py install
```

Nun editieren Sie Ihr *profile* und fügen ihm einen *PYTHONPATH* hinzu, beispielsweise so:

```
export PYTHONPATH=/home/myuser/tools/trac/lib/python2.4/site-packages
```

Wichtig dabei ist, dass dieser *PYTHONPATH* auf das Verzeichnis *site-packages* Ihrer Trac-Installation zeigt.

Anschließend legen wir ein neues Projekt für unseren Inhalt an. Dazu gehen wir in das Verzeichnis *~/tools/trac/bin* und führen Folgendes aus:

```
trac-admin ~/myproject initenv
```

Es gibt ein Web-Admin-Plugin für Trac, mit dessen Hilfe Sie Ihre Projekte über eine Weboberfläche konfigurieren können, so dass Sie nicht direkt das Kommandozeilenwerkzeug `trac-admin` nutzen müssen. Ab Trac 0.11 ist das Plugin Bestandteil der Standarddistribution.

Das legt das Trac-Projekt *myproject* im *home*-Verzeichnis an. Bevor das aber gelingt, werden von uns einige Antworten verlangt, die dem Werkzeug helfen, die Grundeinstellungen vorzunehmen. Wir können sie auch später noch nachreichen oder ändern.

Es beginnt mit dem Namen des Projekts. Hier könnten wir den vorgeschlagenen Wert, in eckigen Klammern vor der Eingabe, durch einfaches Betätigen der Eingabetaste übernehmen. Unser Projekt soll *Agile* heißen. Anschließend folgt die Eingabe der Datenbankverbindung. Hier geben wir nichts ein und übernehmen den Standard. Auch beim *Repository* übernehmen wir den Vorschlag. Den dann folgenden Pfad zum Repository übernehmen wir ebenfalls, wir werden in einem späteren Schritt die Repository-Integration deaktivieren. Auch das angebotene *template*-Directory übernehmen wir, um dann, nach diversen Informationsmeldungen, den Hinweis auf erfolgreiche Einrichtung zu bekommen: »Congratulations!«

Abhängig davon, wo Sie das Trac-Projekt eingerichtet haben und welche Rechte der technische User hat, unter dem das Wiki läuft, müssen Sie ggf. noch die Zugriffsrechte auf das *db*-Verzeichnis in Ihrem Projekt (hier *myproject*) erhöhen.

Um den integrierten Webserver starten zu können, müssen wir im *bin*-Verzeichnis unserer Trac-Installation Folgendes aufrufen:

```
./tracd -port 8000 /home/michael/myproject
```

Das startet den Server auf dem angegebenen Port und Projekt.

Nun kontrollieren wir mit einem Webbrowser, ob das Projekt zur Verfügung steht:

```
http://localhost:8000/myproject
```

Und siehe da: Da ist es (siehe Abbildung 17-1)!

Konfiguration

Die standardmäßige Konfiguration beinhaltet die volle Installation inklusive Subversion und Ticketing-System-Unterstützung, so dass wir nun schon fast fertig sind. Interessant wird es, wenn wir eine Funktionalität abschalten wollen, was in der Praxis häufig passieren wird. Wir machen das exemplarisch mit dem Ticketing-System, da wir unsere Aufgaben ausschließlich mit Story Cards verwalten wollen. Ich zeige Ihnen auch, wie man die Subversion-Unterstützung abstellt. Vielleicht möchten Sie Ihren Quellcode nicht offen im Intranet darstellen oder Sie nutzen ein anderes Versionierungssystem, z.B. CVS.

Abbildung 17-1: Die Trac-Startseite

Wir öffnen nun in unserem Projekt *myproject* im *conf*-Verzeichnis die Datei *trac.ini*. Dort fügen wir folgende Zeilen ein:

```
[components]
trac.ticket.* = disabled
```

Weiterhin können wir in *trac.ini* zahlreiche weitere Einstellungen vornehmen. Das reicht von den darzustellenden Logos über die sichtbaren Komponenten auf der Seite, Logging und E-Mail-Benachrichtigungen bis hin zu den Sucheinstellungen u.v.m.

Anschließend starten wir den Trac-Server neu (siehe Abbildung 17-2).

Markup

Trac hat ein eigenes Markup, eine eigene Metasprache, mit deren Hilfe Sie die Wiki-Seiten editieren können. Wir möchten uns auf die Kernfunktionalität des Wikis konzentrieren: Textinhalte. Doch wie können wir diesen Text eingeben bzw. formatieren?

Zeichenformate

Den Font des Texts können Sie durch Einbettung in bestimmte Zeichen beeinflussen. Im Folgenden sehen Sie ein paar praktische Beispiele:

```
'''bold'''
''italic''
'''''bold italic'''''
__unterstrichen__
~~durchgestrichen~~
^hochgestellt^
,,tiefgestellt,,
```

Abbildung 17-2: Die konfigurierte Trac-Seite

Lawinengefahr

Ich war einmal kurzzeitig an einem Projekt beteiligt, in dem es diverse interne Mailing-Listen gab. Diese Listen hatten ein so hohes Verkehrsaufkommen, dass schließlich niemand mehr die Mails las. Fand jemand etwas heraus oder wollte etwas kommunizieren, wurde diese Information über ebendiese Mailing-Liste verschickt. Auf der anderen Seite gab es keine koordinierte persönliche Kommunikation, keinen Erfahrungsaustausch, aber auch kein Wiki. Es musste dringendst ein Wiki her, um den Wissensaustausch anzukurbeln und die Mailing-Liste zu entschlacken.

Überschriften

Überschriften werden durch vor- und nachgestellte Gleichheitszeichen markiert. Ein Gleichheitszeichen kennzeichnet eine Überschrift auf höchster Ebene. Zwei bzw. drei Gleichheitszeichen stellen den eingeschlossenen Text als Überschrift der nächstkleineren Ebene dar.

```
= Überschrift Ebene 1 =
== Überschrift Ebene 2 ==
=== Überschrift Ebene 3 ===
```

Abschnitte

Abschnitte können auf verschiedene Arten dargestellt und geschachtelt werden. Sie können Sternchen, Zahlen und Buchstaben benutzen.

```
 * Text 1
   * Text 1.1
 * Text 2

 1. Text 1
   a. Text 1.a
   a. Text 1.b
      i. Text 1.b.i
      i. Text 1.b.ii
```

Tabellen

Ein wichtiges Medium sind Tabellen. Tabellen definieren Sie durch eine Konfiguration der Zellen, indem Sie die gewünschten Inhalte der Zellen jeweils durch zwei vertikale Linien voneinander trennen.

```
||Zelle 1||Zelle 2||Zelle 3||
||Zelle 4||Zelle 5||Zelle 6||
```

Links

Es existieren zwei Arten von Links: externe und interne. Externe Links sind Hyperlinks, die auf Seiten außerhalb des Wiki verweisen. Zur Definition schreiben sie in einfachen eckigen Klammern die URL gefolgt von einem darzustellenden Text auf. Nur der Text wird im Wiki sichtbar. Daneben gibt es interne Links, die auf Seiten innerhalb des Wiki verweisen. Hier reicht es »wiki:« dem Titel der entsprechenden Seite voranzustellen.

```
[http://www.edgewall.com/ Edgewall Software]
[wiki:ISO9000]
```

Zusammenfassung

Trac ist ein leistungsfähiges Wiki, das eine hervorragende Kommunikationsinfrastruktur für Ihr agiles Projekt bietet. Durch die Trac-eigenen Konzepte wie Roadmap und Milestone können Sie Ihr Projekt planen. Sie besitzen ein einheitliches Kommunikationsmedium. Das zentrale Wissen wird kontinuierlich in Trac fortgeschrieben. Zentrales Wissen sind beispielsweise wichtige Konfigurationsparameter für eine Anwendung, Kontaktpersonen oder eine Dokumentation. Handlungsrelevante Informationen müssen parallel zum Wiki aber auch immer persönlich mit dem Kommunikationspartner ausgetauscht werden, da Sie nicht völlig sicher sein können, dass die Daten im Wiki auch gelesen werden.

Durch die einfache Nutzung eines Wikis werden sich nach Einführung viele Personen aktiv beteiligen. Wie bei anderen Werkzeugen auch, werden Sie die größte Akzeptanz erzeugen, wenn Sie alle Nutzer aktiv an der Entscheidung für die Nutzung eines Wikis bzw. konkret Trac teilhaben lassen. Verankern Sie die Nutzung des Wikis als inhärenten Bestandteil Ihres Projekts und unterstützen Sie Personen, die bei der Nutzung Schwierigkeiten haben.

KAPITEL 18

Ant – Teil 2: Praktische Einsatzszenarien

Wir haben uns zu Beginn bereits rudimentär mit Ant befasst, um in den weiteren Kapiteln immer wieder bei bedarf darauf zurückzukommen und uns seiner zu bedienen. Wir haben unterschiedliche Werkzeuge mit Ant integriert.

Dieses Kapitel ist nun wieder Ant-zentriert und beschreibt, von Ant ausgehend, einzelne Anwendungsszenarien: Dabei werden wir teilweise vorherige Kapitel inhaltlich streifen, aber aus der Sicht des Prozesses und von Ant diskutieren. Das Kapitel illustriert, wie Sie mit Ant einen agilen Entwicklungsprozess konzeptionell unterlegen können und führt in die technische Umsetzung ein. Ein weiteres Szenario ist die Rolle von *Coding Guidelines*, und wie diese mit Ant überwacht werden können. Ich zeige Ihnen ferner eine alternative Lösung, wie man mit Ant Artefakte agil verwalten kann. Schließlich illustriere ich den *HotSwap*, den Austausch von Java-Artefakten in der Echtzeitumgebung zur Laufzeit mit Hilfe von Ant.

Mit *AntUnit* existiert ein Ansatz, der sich zum Ziel gesetzt hat, ein Unit-Testing-Framework für Ant-Skripten bereitzustellen.

Um die Artefakte zu verwalten, haben wir bisher ausschließlich Subversion genutzt. In der Praxis kommt auch häufig *CVS* zum Einsatz. Dieses Kapitel greift bewusst auf CVS anstelle von Subversion zurück (die Nutzung von Subversion kennen Sie bereits), um diesem Sachverhalt Rechnung zu tragen und Ihnen einen Eindruck davon zu vermitteln, wie die Nutzung von CVS aus Ant heraus aussieht.

Dieses Kapitel schließt die Diskussion um die Nutzung von Ant und die Nutzung und Integration der Werkzeuge ab.

Das fundamentale Ant-Gerüst für einen Entwicklungszyklus

Ant ist, wie gesagt, dafür geeignet, einen agilen Entwicklungsprozess zu unterstützen. Dabei werden insbesondere folgende agile Grundsätze erfüllt und Vorgehen unterstützt:

- Durch die vollautomatische Erstellung der Software ist es möglich, diesen Schritt sehr oft durchzuführen. Ein Build ist sehr schnell durchführbar (*10-Minute-Build*).
- Der Entwickler testet seine Änderungen mit Unit-Tests, die im Ant-Skript automatisch aufgerufen werden (*Code and Test*, *Test-first-Programming*).
- Integrationen mit Werkzeugen erlauben schnelle *Rückkopplungen*.
- Die Software wird mit Ant regelmäßig, mindestens einmal täglich, auf ein Integrationssystem gespielt, wobei auf dem Entwicklungsrechner alle Tests erfolgreich durchlaufen sein müssen, die komplette, endgültige Funktionalität aber nicht bereitstehen muss (*Continuous Integration*, *Incremental Design*, *Incremental Deployment*, *Daily Deployment*).
- Der Quellcode und die Build-Skripten werden im zentralen Repository gehalten. Es wird gemeinsam darauf gearbeitet, auch auf den Build-Skripten, den Unit-Tests etc. (*Single-Code Basis*, *Shared Code*). Das Ant-Skript kontaktiert das zentrale Repository direkt.
- Es existieren Tasks, mit deren Hilfe die für die agile Entwicklung wichtigen *Coding Standards* kontrolliert, *Deployment-Deskriptoren* generiert und Dokumentation mit *JavaDoc* und *Code-Metriken* erstellt werden können.

Sie können mit Ant während des Build-Prozesses den Quellcode Ihrer Anwendung mit zusätzlichen Informationen anreichern. Jede Form von Zusatzinformationen innerhalb der Sourcen ist sehr hilfreich. Ergänzen Sie den Build um Versionsnummern, ein Datum bzw. einen kompletten Timestamp, weiterführende Daten bzgl. des Versionierungsstatus (Labels) etc. Extrahieren Sie fremde Konfigurationseinstellungen wie einen Server immer aus dem eigentlichen Build-Skript und platzieren sie in ein Properties-File. Automatisieren Sie alles! Nehmen Sie nicht eine Aktivität aus dem Prozess heraus, die Sie dann manuell durchführen.

Ein Ant-Skript kann den vollen Software-Lebenszyklus unterstützen. Ein Build-Skript in Ihrem Projekt sollte die im Folgenden vorgestellten Bereiche abdecken. Diese Bereiche werden durch einzelne Ant-Basisschritte abgedeckt, die in separaten Ant-Targets implementiert werden können. Kombinieren Sie diese im weiteren Verlauf vorgestellten Basisschritte (siehe Abbildung 18-1) und passen Sie sie Ihren konkreten Anforderungen an. Aufbauend auf diesen Basisschritten schauen wir uns im Anschluss konkrete beispielhafte Nutzungsszenarien an.

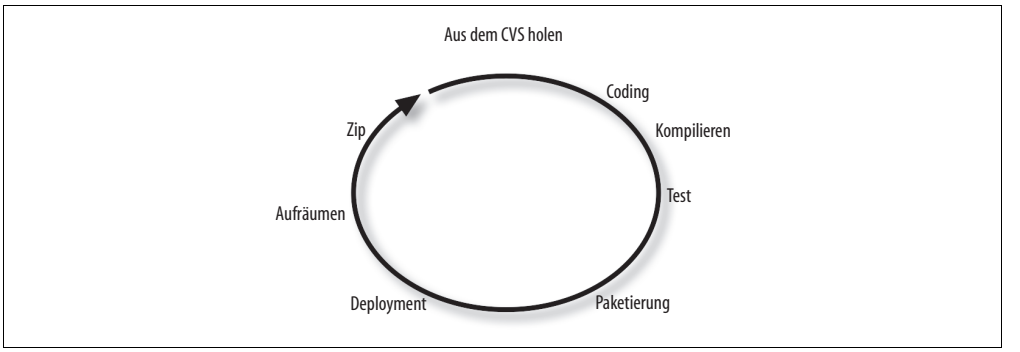


Abbildung 18-1: Die Ant-Basisschritte

Kompilieren

Sie sollten im Rahmen des Builds Ihre Sourcen kompilieren. Für das Kompilieren von Quellcode steht Ihnen der `javac`-Task zur Verfügung: Im Quellverzeichnis *srcdir* liegen die zu übersetzenden Klassen, die ins Zielverzeichnis *destdir* gelegt werden sollen. Sie können den Kompiliervorgang umfassend beeinflussen.

Geben Sie immer an, mit welchem Compiler Sie die Übersetzung durchführen wollen. Dazu steht das Attribut `executable` zur Verfügung. Es ist wichtig, diese Wahl treffen zu können, denn Compiler generieren teilweise unterschiedlichen Bytecode, was beispielsweise bei HotSwap zu Problemen führen kann. Weiterhin möchten Sie vielleicht nicht aus einer IDE wie Eclipse heraus den Eclipse-JDT Compiler nutzen, sondern zum Beispiel einen, von dem Sie sich leistungsstärkeren Bytecode versprechen (Jikes).

Die von Ihnen gewünschte *Source*- und *Target*-Kompatibilität geben Sie mit entsprechenden Attributen an. Source- und Target-Einstellungen müssen Sie beachten, falls auf Ihrem Zielsystem beispielsweise eine ältere Java-Version läuft. `fork` sollten Sie auf `yes` setzen, um die Kompilierung extern mit dem von Ihnen gewählten Compiler durchzuführen. Sie können ferner einen Klassenpfad angeben und Debugging-Informationen mit in die Klassen hineinkompilieren lassen.

```
<javac listfiles="true"
 srcdir="src"
 destdir="classes"
 compiler="javac1.5"
 source="1.5"
 target="1.5"
 fork="yes"
 debug="on"
 executable="/pfad/zu/java/jdk1.5.0_03/bin/javac">
    <classpath>
        <pathelement location="lib/lib1.jar"/>
        <pathelement location="lib/lib2.jar"/>
     <pathelement location="lib/lib3.jar"/>
    </classpath>
</javac>
```

Paketierung

Ein weiterer Basisschritt ist die Paketierung Ihrer Software. Dafür gibt es verschiedene Möglichkeiten. Artefakte können in Formate wie *.jar*, *.war*, *.ear*, *.zip* etc. gepackt werden. Möchten Sie beispielsweise Ihre Klassen zu einer *.jar*-Datei zusammenfassen, dann nutzen Sie den `jar`-Task. Die Nutzung des Tasks ist einfach: Die Dateien des *basedir* werden in die Zieldatei *jarfile* gepackt. Sie können mit *excludes* einzelne Dateien aus dem Schritt auslassen, wie hier die mit der Property `jarexclude` definierten Dateien.

```
<jar
  excludes="${jarexclude}"
  jarfile="file.jar"
  basedir="classes"
 />
```

Test

Das Testen Ihrer Software ist ebenfalls ein Grundschritt, den Sie unbedingt mit in Ihre Build-Datei aufnehmen müssen. Erwähnenswert ist hier vor allem der Task `junit`. Das angefügte Beispiel führt den Test `my.test.TestCase` aus. Es wird eine Warnung ausgegeben, wenn der Test fehlschlägt. Das Ergebnis des Tests wird in der Datei *result.xml* ausgegeben. Anschließend werden die Tests aus dem Verzeichnis *${src.tests}* ausgeführt. Die Testergebnisse werden in Dateien namens *TEST-name.txt* in dem Verzeichnis *${reports.tests}* ausgegeben.

Der Ant-Task `junit` bezieht sich auf JUnit 3.x. JUnit in der Version 4 wird zur Zeit noch nicht durchgehend unterstützt. Sie können JUnit 4-Testklassen per Wrapper für ältere APIs nutzbar machen.

```
<junit printsummary="yes" haltonfailure="no">
  <formatter type="plain"/>
  <test name="my.test.TestCase" haltonfailure="no" outfile="result">
    <formatter type="xml"/>
  </test>
  <batchtest fork="yes" todir="${reports.tests}">
    <fileset dir="${src.tests}">
      <include name="**/*Test*.java"/>
      <exclude name="**/AllTests.java"/>
    </fileset>
  </batchtest>
</junit>
```

Aufräumen

Ein immer wiederkehrender Schritt im Build-Prozess ist das Aufräumen von temporär angelegten Dateien bzw. Verzeichnissen. Sie können sie mit Hilfe des `delete`-Tasks löschen.

```
<delete includeEmptyDirs="true">
  <fileset dir="build"/>
</delete>
```

Deployment

Sie möchten die kompilierten, getesteten und gepackten Dateien sicherlich auch auf einem entfernten System verteilen. Möchten Sie Quellen und Dokumente zu einer zentralen Stelle, zum Beispiel einer Projektseite, transferieren? Übertragungen werden häufig mit dem FTP-Protokoll durchgeführt, für dessen Behandlung der `ftp`-Task konzipiert ist. Durch Angabe eines Server-Namens und der Credentials werden Ihre im Task spezifizierten Verzeichnisse übertragen.

```
<ftp server="ftp.apache.org"
     userid="anonymous"
     password="me@myorg.com">
  <fileset dir="htdocs/manual"/>
</ftp>
```

Rausholen

Wesentlicher Bestandteil eines jeden Build-Skripts ist es, Sourcen aus dem Versionierungssystem zu holen. Sie können beispielsweise den `cvs`-Task nutzen, um Artefakte aus dem Repository auszuchecken. Durch Angabe des CVSRoot und der auszucheckenden Packages legt Ant die Artefakte auf das mit *dest* spezifizierte Zielverzeichnis auf den lokalen Rechner.

```
<cvs cvsRoot="/home/cvspublic"
     package="ant"
     dest="${ws.dir}"
 />
```

JavaDocs

Mithilfe des `javadoc`-Tasks können Sie JavaDocs erstellen, die Dokumentation des öffentlichen API einer Anwendung. Pakete, die Sie verarbeitet haben möchten, können Sie genauso angeben wie Pakete, die von der Verarbeitung ausgeschlossen werden sollen. Angegeben werden ferner die Verzeichnisse (Quellcode- und Zielverzeichnis) und weitere informative Daten, die auf der JavaDoc-Seite dargestellt werden sollen. Die Seite (bzw. die Klassen) wird in zwei Gruppen unterteilt. Ant erfährt über den Klassenpfad, wo es weitere Bibliotheken finden kann.

```
<javadoc
   packagenames="com.huettermann.*"
   excludepackagenames="com.huettermann.private.*
   sourcepath="src"
   destdir="api"
   author="true"
   version="true"
```

```
    use="true"
    windowtitle="Agile Entwicklung"
    doctitle="Agile Entwicklung">
         <group title="Group A" packages="com.huettermann.xp*"/>
         <group title="Group B" packages="com.huettermann.scrum*"/>
         <classpath>
            <pathelement location="lib/comp1.jar"/>
            <pathelement location="lib/comp2.jar"/>
            <pathelement location="lib/comp3.jar"/>
            <pathelement location="lib/comp4.jar"/>
            <pathelement location="lib/comp5.jar"/>
         </classpath>
</javadoc>
```

Diese Minimalgerüste verknüpfen wir nun zu im Entwicklungsalltag typischen Szenarien.

Erstes Szenario: »Continuous Integration« auf dem Integrationsrechner

Sie stoßen automatisch mit `cron` ein Ant-Skript an. Dieses holt sich den Code (und andere Artefakte) aus dem zentralen Repository (`fetch`). Ihr Code wird dann komplett neu mit dem gewünschten Compiler kompiliert (`compile`). Anschließend werden Tests (wie Unit-Tests, Code Coverage, Coding Guidelines etc.) angestoßen, die ein Output generieren (`test`). Aufgrund der vielen Tests besteht die Software aus »self-testing code«. Die Klassen packen Sie anschließend und platzieren sie auf einem anderen Rechner, wo sie dann von einer Laufzeitumgebung entpackt und als Anwendung bereitgestellt werden (*Deployment*). Aus den Java-Sourcen erstellen Sie JavaDoc-Dokumentation. Die Sourcen, die Testergebnisse und Dokumentation oder auch das ganze Release verteilen Sie auf andere Rechner. Dabei publizieren Sie die Dokumente zu Informationszwecken auf einem Produktportal. Abschließend räumen Sie den Arbeitsbereich, also das Verzeichnissystem, in dem Ant die Schritte durchgeführt hat, auf.

Die Ausführung von mehreren Targets erreichen Sie, indem Sie mehrere kommaseparierte Targets innerhalb des `depends`-Attributs angeben. Sie können auch mit den Tasks `ant` und `antcall` arbeiten.

Zweites Szenario: »Continuous Integration« auf einem Arbeitsplatzrechner

Ein weiteres häufiges Einsatzszenario ist die Nutzung von Ant aus der IDE des Entwicklers heraus. Dabei holen Sie sich den Code (fachliche Klassen, Testklassen und andere Artefakte) mithilfe einer Entwicklungsumgebung aus dem zentralen Repository und schreiben ihn lokal fort. Über die Ant-Integration Ihrer IDE stoßen Sie zu bestimmten Zeiten einen Prozess an, der mit dem »Continuous Integration auf dem Integrationsrechner«-Szenario vergleichbar ist, aber nicht alle Schritte umfasst. Dabei ist es üblich, die

Unit-Tests auf den lokalen Sourcen laufen zu lassen und ein Release zu erstellen. Wenn die Unit-Tests erfolgreich durchlaufen und der Code kompiliert sind, können Sie darüber nachdenken, Ihre Artefakte in das zentrale Repository einzuchecken. Natürlich machen das auch andere, so dass Sie sich zunächst den neuesten Stand der Software holen müssen. Sie machen ein Update auf Ihrer Arbeitskopie. Falls das Update keine Konflikte provoziert, lassen Sie Ihre Tests noch einmal laufen. Falls es Konflikte gibt, lösen Sie sie zunächst auf. Ist der Stand Ihrer Software im Workspace mit dem im Repository abgeglichen und die Tests laufen durch, können Sie Ihre Änderungen einchecken.

Häufiges Committen und Continuous Integration können und sollen Kommunikation nicht ersetzen. Reden Sie mit Ihren Kollegen darüber, was Sie wo gemacht haben. Laden Sie einen Kollegen zu einem Kaffee ein und sprechen Sie über diese Dinge insbesondere, wenn ein Update Konflikte produziert hat.

Drittes Szenario: »Single Code Basis«-Redundanzen eliminieren unter CVS

Ein sehr häufig zu beobachtendes Antimuster in Projekten ist die redundante, unkoordinierte Verwaltung von Softwaremodulen, sowohl von eigenen als auch von denen anderer (»3rd Party Libraries«). Java-Projekte ab einer signifikanten Größe besitzen eine große Anzahl von Komponenten, die zu allem Übel auch noch in verschiedenen, untereinander kompatiblen oder nicht kompatiblen Versionen vorliegen. Sehr ungünstig ist es, wenn all diese Artefakte in ihrer aktuellen Version zusammen mit dem eigentlichen, fachlichen Quellcode an derselben Stelle im zentralen Repository gehalten werden. Etwas besser, aber immer noch suboptimal, ist es, ein eigenes Projekt im Repository anzulegen, das die aktuell genutzten Artefakte enthält.

Was machen Sie, wenn Sie mehrere Projekte (oder Softwareprodukte) haben und diese unterschiedliche Versionen der bereitgestellten »3rd Party Libraries« nutzen? Aufgrund der ausbaufähigen Struktur wird sich jedes Projekt die notwendigen Bibliotheken in ein eigenes Verzeichnis legen. Wiederverwendbarkeit sieht anders aus. Haben Sie noch den Überblick, welche Komponente in welcher Version genutzt wird? Was machen Sie, wenn es mit einer Komponente Probleme gibt und sie nicht schnell herausfinden können, welche Version davon Sie einsetzen? Um ein neues Projekt aufzusetzen, müssen alle Komponenten aus den unterschiedlichen Stellen mit der IDE in den lokalen Arbeitsbereich geholt werden. Dies ist sehr fehleranfällig und würde von einem Entwickler euphemistisch eher als Fleißarbeit beschrieben.

Die diskutierte Form des Konfigurationsmanagements ist technologieunabhängig. Wir gehen in diesem Beispiel allerdings davon aus, dass Sie CVS und Ant im Einsatz haben. Wir werden später sehen, dass das, was wir hier manuell durchführen, von Maven 2 weitestgehend »from scratch« unterstützt wird.

Die von Ihnen entwickelte Anwendung benötigt drei Bibliotheken aus dem Apache Jakarta-Projekt. Sie möchten von standardisierten, gut unterstützten, durchgetesteten Lösungen für übliche Herausforderungen profitieren. Sie nutzen

- *commons-lang* in der Version 2.2 und daraus insbesondere *commons-lang-2.2.jar*
- *commons-collections* in der Version 3.2 und daraus insbesondere *commons-collections-3.2.jar*
- *commons-io* in der Version 1.2 und daraus insbesondere *commons-io-1.2.jar*

Sie sind zwar an allen Bestandteilen der von Ihnen heruntergeladenen Komponenten interessiert, wie beispielsweise auch der Dokumentation, die drei *.jar*s benötigen Sie aber mindestens, damit Ihre Anwendung übersetzbar ist. Diese *.jar*s sollen im Verzeichnis *lib* auf der Root-Ebene Ihres Projekts liegen.

Schritt 1: Zentrale Bereitstellung

Erstellen Sie sich in Ihrem zentralen Repository ein Modul, das diese wiederverwendbaren Komponenten aufnehmen soll.

In dem Versionsverzeichnis (wie *v3.2*, *v1.2* usw.) befinden sich alle Dateien und Unterverzeichnisse der Komponente, also die zur Komponente mitgelieferten *.jar*s, Dokumentation, Quellcode etc. Wichtig dabei ist, dass Sie für jede Version, die Sie von der jeweiligen Komponente haben, ein neues dediziertes Verzeichnis anlegen. Sie übernehmen also bewusst selbst die Versionierung und überlassen sie nicht dem CVS. Für alle Versionen gibt es im HEAD des CVS ein eigenes Verzeichnis. Sie erkennen also sofort, welche Komponenten in welcher Version vorliegen (siehe Abbildung 18-2).

Abbildung 18-2: Das Versionsverzeichnis

Schritt 2: Alias anlegen

Für genau die Artefakte, die Sie für Ihre Anwendung benötigen (häufig ist es nur ein einzelnes *.jar*, wie im vorliegenden Fall), legen sie im CVS-Repository ein »Alias-Modul« an. Ein Alias-Modul ist ein Modul, unter dessen Namen die referenzierten Artefakte nun ebenfalls ansprechbar sind. Sie fügen der *modules*-Datei im *CVSROOT*-Verzeichnis Ihres CVS-Repositories für jede Komponente einen entsprechenden Eintrag hinzu. Als Namen des Alias-Moduls nehmen Sie den Namen der *.jar*-Datei, die über eine Versionsnummer verfügt und mit *-dist* (für »Distribution«) abschließt. Die drei neuen Zeilen in der *modules*-Datei sehen in diesem Fall so aus:

```
commons-collections-3.2-dist -a dev/3rdparty/commons-collections/v3.2/commons-
collections-3.2.jar
commons-io-1.2-dist -a dev/3rdparty/commons-io/v1.2/commons-io-1.2.jar
commons-lang-2.2-dist -a dev/3rdparty/commons-lang/v2.2/commons-lang-2.2.jar
```

Wenn wir ein ganzes Verzeichnis *v3.2* auschecken, werden alle Dateien in Ihren Arbeitsbereich übertragen. Diese Übertragung kann lange dauern. Da wir nur an einzelnen Dateien interessiert sind, können wir die Übertragungsdauer und das Dateiaufkommen in unserem lokalen Arbeitsbereich deutlich reduzieren. Bei unserem Vorgehen können wir die *.jar*-Files direkt unter dem Alias-Namen ansprechen (auf dem HEAD direkt erreichen) und auschecken, ohne uns über die anderen Bestandteile der Verzeichnisse Gedanken machen zu müssen. *commons-collections3.2-dist* ist nun beispielsweise direkt ansprechbar und verweist auf die Datei *dev/3rdparty/commons-collections/v3.2/commons-collections-3.2.jar*.

Schritt 3: component.properties

Legen Sie innerhalb Ihrer Anwendung, die die Abhängigkeiten zu den Jakarta-Bibliotheken hat, ein Verzeichnis *install* an. Dort erstellen Sie die Property-Datei namens *components.properties*. Für jede Ihrer drei Jakarta-Komponenten fügen Sie zwei Zeilen zu diesen *components.properties* hinzu. Property-Dateien bestehen aus Schlüssel/Wert-Paaren. Der Schlüssel folgt dem Muster `Komponentenname.component` gefolgt von `repository` respektive `version`. Als Wert geben Sie entsprechend das Repository bzw. die Version an, die Sie benötigen. Das ist eine hinreichende Definition Ihrer Projektabhängigkeiten, also welche Komponenten Sie in welcher Version zur Übersetzung Ihrer Anwendung benötigen.

```
commons-lang.component.repository=/usr/local/cvsrepository
commons-lang.component.version=2.2

commons-collections.component.repository=/usr/local/cvsrepository
commons-collections.component.version=3.2

commons-io.component.repository=/usr/local/cvsrepository
commons-io.component.version=1.2
```

Schritt 4: build.xml

Innerhalb des *install*-Verzeichnisses erstellen Sie nun ein Ant-Skript *build.xml* (siehe Beispiel 18-1).

Beispiel 18-1: Die Datei build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Author: Michael Huettermann -->
<project name="agiledevelopment" default="getComponents" basedir=".">

    <property file="components.properties"/>

    <macrodef name="getComponent">
        <attribute name="component.name"/>
        <attribute name="target"/>
        <sequential>
            <ant antfile="getComponentFromCVS.xml">
                <property name="component.repository" value="${@{component.name}.component.
repository}"/>
                <property name="component.name" value="@{component.name}"/>
                <property name="component.version" value="${@{component.name}.component.
version}"/>
                <property name="component.target" value="@{target}"/>
            </ant>
        </sequential>
    </macrodef>

    <target name="getComponents">
        <mkdir dir="temp"/>
        <delete dir="../lib"/>
        <mkdir dir="../lib"/>

        <getComponent component.name="commons-lang" target="../lib"/>
        <getComponent component.name="commons-collections" target="../lib"/>
        <getComponent component.name="commons-io" target="../lib"/>

        <delete dir="temp"/>
    </target>

</project>
```

Das Skript besteht aus einem standardmäßig aufgerufenen *getComponents*-Target. Zunächst wird ein temporäres Arbeitsverzeichnis namens *temp* erstellt und das Zielverzeichnis, das die *.jar*s aufnehmen soll, gelöscht und wieder angelegt. So haben wir bei jedem Skriptdurchlauf die gleichen Startbedingungen. Nun wird für jede Library das Ant-Makro `getComponent` aufgerufen. Diese mit Ant 1.5 eingeführten Makros (Templates) erlauben uns, ähnliche Tasks mit abweichenden Parametern beliebig oft aufzurufen, und sind für die fortgeschrittene Ant-Entwicklung bestens geeignet. Wir übergeben dem Template den jeweiligen Komponentennamen und das Zielverzeichnis, in das die *.jar*s gelegt werden sollen.

Innerhalb des Templates werden nun mit dem `ant`-Task eine weitere Ant-Datei (bzw. genauer: das Default-Target der Ant-Datei) aufgerufen und `property`-Werte übergeben. Die Werte kommen uns bekannt vor: Neben dem Zielverzeichnis übergeben wir bei jedem der drei Aufrufe den jeweiligen Komponentennamen und die Komponentenversion. Die Werte werden mehrfach expandiert: Zunächst wird der Parameter an das Template übertragen (mit `@` gekennzeichnet), dann werden die konkreten Werte aus *components.properties* übernommen. Das geschieht mit Hilfe der »`${...}`«-Syntax und setzt voraus, dass zu Beginn des Skripts die Schlüssel/Wert-Paare unserer *component.properties* über den `property`-Task verfügbar gemacht werden. Dieser Task stellt alle Schlüssel/Wert-Paare der referenzierten Datei zur Verfügung. Aber was passiert nun in dem Skript *getComponentFromCVS.xml*?

Schritt 5: getComponentFromCVS.xml

Als dritte Datei legen Sie nun *getComponentFromCVS.xml* im *install*-Verzeichnis an. Sie sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Author: Michael Huettermann -->
<project name="getComponentFromCVS" default="getComponent" basedir=".">

    <target name="getComponent" >
        <cvs cvsRoot="${component.repository}"
             package="${component.name}-${component.version}-dist"
             dest="temp"
             failonerror="false"/>

        <move todir="${component.target}" flatten="yes">
            <fileset dir="temp">
                <include name="**/*.jar"/>
            </fileset>
        </move>
    </target>

</project>
```

Bei jedem Aufruf des `getComponent`-Targets aus *build.xml* heraus wird nun die *.jar*-Datei aus dem CVS ausgecheckt. Es wird das auf `-dist` endende Alias-Modul in der notwendigen Version in das *temp*-Verzeichnis ausgecheckt. Da das ausgecheckte Modul noch Verzeichnisinformationen enthält (die Verzeichnisstruktur zum *.jar* geht nicht verloren), kopieren wir das *.jar* mit dem `move`-Task in das endgültige Zielverzeichnis. Das geschieht generisch: In dem Verzeichnis und all seinen Unterverzeichnissen werden *.jar*s gesucht und kopiert. Im aufrufenden Skript haben wir bereits gesehen, dass nach erfolgter Verarbeitung das Verzeichnis *temp* gelöscht wird.

Rufen Sie nun Ant über die IDE oder mit `ant` im *install*-Verzeichnis auf (siehe Abbildung 18-3), werden die drei Komponenten wie gewünscht in das lokale *lib*-Verzeichnis gelegt (siehe Abbildung 18-4).

```
michael@BlueGene:~/Agile/install$ ls
build.xml  components.properties  CVS  getComponentFromCVS.xml
michael@BlueGene:~/Agile/install$ ant
Buildfile: build.xml

getComponents:
    [mkdir] Created dir: /home/michael/Agile/install/temp
   [delete] Deleting directory /home/michael/Agile/lib
    [mkdir] Created dir: /home/michael/Agile/lib

getComponent:
      [cvs] U dev/3rdparty/commons-lang/v2.2/commons-lang-2.2.jar
     [move] Moving 1 file to /home/michael/Agile/lib

getComponent:
      [cvs] U dev/3rdparty/commons-collections/v3.2/commons-collections-3.2.jar
     [move] Moving 1 file to /home/michael/Agile/lib

getComponent:
      [cvs] U dev/3rdparty/commons-io/v1.2/commons-io-1.2.jar
     [move] Moving 1 file to /home/michael/Agile/lib
   [delete] Deleting directory /home/michael/Agile/install/temp

BUILD SUCCESSFUL
Total time: 3 seconds
michael@BlueGene:~/Agile/install$
```

Abbildung 18-3: Aufruf von Ant

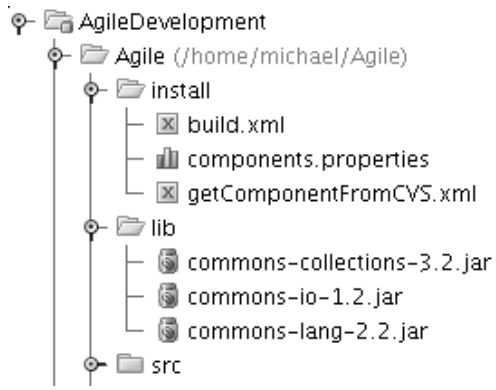


Abbildung 18-4: Bereitstellung der Komponenten

Berücksichtigen Sie bitte, dass es hier um das Konzept geht, wie sich Redundanzen bei Softwarekomponenten verhindern, Versionen verwalten und Module wiederverwenden lassen. Dieser Abschnitt richtete also den Fokus auf ein Konfigurationsmanagement mithilfe von Ant. In diesem Zusammenhang würde eine komplette Darstellung sämtlicher Möglichkeiten des hier zugrundeliegenden CVS den Rahmen sprengen. Wenn Sie das Konzept in der Praxis umsetzen möchten, sollten Sie Folgendes berücksichtigen: Sie sollten eine weitere Property-Datei anlegen, die über den CVS-Benutzernamen und den Verweis auf die

CVS-Passwortdatei verfügt. Da das *install*-Verzeichnis und die hier vorgestellten Skripten Bestandteil der zu entwickelnden Anwendung sind und mit in die Versionsverwaltung der Anwendung übernommen werden, möchten Sie diese entwicklerspezifischen Login-Informationen nicht mit in die *component.properties* aufnehmen. *component.properties* muss für alle Entwickler gleich sein. Erstellen Sie also eine eigene Property-Datei mit den notwendigen Daten, zum Beispiel mit dem Namen *user.properties*. Sie könnte so aussehen:

```
cvs.user=agile
cvs.passfile=/home/user/.cvspass
```

cvsignore

In der Datei, die die individuellen CVS-Login-Daten enthält, sollten Sie ein `.cvsignore` aufnehmen. So verhindern Sie, dass diese Datei mit in das Repository eingecheckt wird. Dort hat sie nichts zu suchen. Das Gleiche gilt für das *lib*-Verzeichnis, in dem die Komponenten in Ihrem lokalen Arbeitsbereich liegen. Es sollte auch nicht mit eingecheckt und versioniert werden. Denken Sie bitte daran, dass Ihre Anwendung im CVS nicht die Komponenten beinhaltet, sondern nur Verweise auf die Abhängigkeiten und die *install*-Skripten. Die Komponenten liegen ausschließlich an zentraler Stelle, unabhängig von Ihren Anwendungen. Die einzelnen *.cvsignore*-Dateien müssen eingecheckt werden. Sie sehen bei allen Entwicklern gleich aus.

Wie Sie Benutzer für CVS anlegen und eine Passwortdatei hinterlegen und für andere Personen unzugänglich machen, die Kommunikation mit dem Repository über ein Netzwerk sicher gestalten und mit den CVS-Nutzerinformationen in dem CVS-Ant-Task das Repository kontaktieren, lesen Sie bitte in den entsprechenden Büchern nach (siehe Hinweise am Ende des Buchs).

Viertes Szenario: Coding Guidelines überprüfen

Bei agilen Prozessen wird großer Wert auf gemeinschaftliche Verantwortlichkeit gelegt. Jeder darf (und soll sogar!) die unterschiedlichen Artefakte bearbeiten dürfen. Das ist aber nur dann richtig produktiv, wenn der Quellcode des ganzen Teams ähnlich aufgebaut ist. Gibt es große Unterschiede, benötigt der einzelne Entwickler eine gewisse Zeit, um sich mit den von anderen Entwicklern erstellen Quellen vertraut zu machen. Hier schaffen Coding Guidelines Abhilfe: Das sind klare Regeln dafür, wie der Quellcode auszusehen hat. Jeder Entwickler hält sich daran. Auf diese Weise kann der Quellcode zwar inhaltlich für einen Entwickler relativ neu sein, die Struktur erkennt er aber wieder, so dass er sich schneller orientieren und zurechtfinden kann.

Coding Guidelines haben daneben noch eine weitere Dimension: Wird der Quellcode ohne Befolgung von Standards geschrieben, liegt die Qualität der Quellen völlig in der Verantwortung des einzelnen Entwicklers. Er muss selbst an alles denken. Das birgt die Gefahr, dass die Qualität der Quellen nicht optimal ist. Sie muss jedoch vom ersten Tag,

vom Start des Projekts an, zu 100 % perfekt sein. Qualität bauen wir in das Produkt mit ein. Wir dürfen nicht den Fehler machen, ein Produkt zu programmieren und erst im Anschluss zu versuchen, eine gewissen Qualität nachzureichen. Refactorings wird es immer geben. Qualität aber muss von Beginn an angestrebt werden. Das ist übrigens bei weiteren Komponenten wie nicht funktionalen Anforderungen ähnlich gelagert. Schauen wir uns die Performance von Anwendungen an: Viele Projekte haben Performance-Schwierigkeiten. Diese fallen allerdings erst auf, wenn das Kind schon in den Brunnen gefallen ist. Auch hier gilt: Die Performance muss von der ersten Stunde an mit in das Produkt eingebaut werden. Versucht man sich später mit der Materie zu befassen, ist die Umsetzung entweder nicht mehr möglich oder viel kostspieliger als eine begleitende Aktivität. Also integrieren Sie in Ihre kleinen Entwicklungsschleifen – die unter anderem aus Testen, Entwicklung und dem Einholen von Feedback bestehen – immer auch die Qualitätsstandards und die Performanceziele der Anwendung.Auch wenn Sie mit vielen Pair Programming-Sessions schon etwas für die Qualität tun, können Sie doch zusätzlich von Werkzeugen Gebrauch machen, um die Codequalität zu validieren.

Checkstyle ist ein mächtiges Werkzeug, das automatische Audits und Metriken unterstützt. Neben Checkstyle existieren aber noch andere Werkzeuge. Werfen Sie einen Blick auf FindBugs, PMD und JDepend.

Hier kommt Checkstyle ins Spiel, das über *http://checkstyle.sourceforge.net* bezogen werden kann. Checkstyle unterstützt den Entwickler dabei, Code zu schreiben, der vorgefertigten Standards entspricht, indem es ihn auf diese Standards hin überprüft. Diese Automatisierung ist weniger fehleranfällig als eine manuelle Prüfung. Sie nimmt dem Entwickler eine lästige Fleißarbeit ab. Wie wir später noch sehen werden, gibt es so viele Ausprägungen von Standards, dass ihre manuelle Überprüfung kaum vertretbar ist. Wir können bei Checkstyle individuell einstellen, welche Standards überprüft werden sollen. Ein mitgelieferter Standard sind beispielsweise die Sun Code Conventions. Sie sind ein guter Ansatz, um von ausgeklügelten Best Practices zu profitieren.

Sicher können Sie auch selbst Standards aufstellen. Das habe ich in der Vergangenheit auch schon häufiger erlebt. Ich sehe es aber als Antimuster an: Nehmen Sie stattdessen wohldurchdachte, in der Praxis bewährte Standards, die Sie situationsabhängig gern anpassen können.

Checkstyle hatte ursprünglich die Funktion, das Layout des Codes zu überprüfen. Sie wurde mit der Zeit immer mehr erweitert. Nun können das Design der Klasse überprüft, doppelter Code identifiziert und auch bekannte »Bug Patterns« gefunden werden.

Wann überprüfen Sie die Guidelines?

Checkstyle existiert als Plugin für viele Entwicklungsumgebungen. Sie können die Konfiguration also zentral bereitstellen und jedem Entwickler das Plugin zur Integration in seine jeweilige IDE geben. Schauen Sie für Details bitte in die Dokumentation Ihrer Lieb-

lings-IDE bzw. in die Dokumentation von Checkstyle. Ein anderer Ansatz ist, die Überprüfung mit Checkstyle zum Bestandteil des Build-Prozesses zu machen, indem Sie den einzelnen Personen das Plugin zur Hand geben. Doch es ist nur menschlich, dass auch mal vergessen wird, während der Entwicklung das Plugin anzuwerfen. Die zentrale kontrollierbare Stelle, die die Tests permanent automatisch durchführt, ist also eine gute Lösung.

Einigen Sie sich im Projekt auf eine Guideline, erstellen Sie dazu eine Checkstyle-Policy und checken Sie sie ins Repository ein. Jeder Entwickler nutzt diese Policy und darf keinen Code einchecken, der Warnungen enthält. Machen Sie davon keine Ausnahmen. Sie können ggf. darüber nachdenken, einzelnen Klassen eine aufgeweichte Policy zuzuweisen, beispielsweise Testklassen.

Coding Standards und Legacy-Code

Wenn Sie viel Legacy-Code haben, also Code von eher schlechten Qualität, dann ist es ein sportliches Vorhaben, diesen Code komplett umzustellen, so dass er dem neuen Standard genügt. Sie können Ihren Build-Prozess aber so konfigurieren, dass der Legacy-Code nicht ganz so hart getestet wird wie der Code, den Sie neu entwickeln, und versuchen, den alten Code sukzessive zu verbessern.

Um die Struktur des bestehenden Codes zu verbessern, benötigen Sie auf jeden Fall Tests. Diese Tests müssen vor dem Refactoring erfolgreich durchlaufen und auch nach den Änderungen noch wie erwartet laufen. Haben Sie keine Tests? Es ist fragwürdig, für Legacy-Code Unit-Tests nachzuschieben, vor allem testen Sie damit nur den Code, wie er gerade vorliegt. Eine Option wäre es, auf einer höheren Ebene neue Tests durchzuführen. Führen Sie also keine Unit-Tests durch, sondern funktionale Tests, mit denen Sie die Verarbeitung überprüfen.

Diese zentrale Stelle ist innerhalb unseres Build-Prozesses. Also müssen wir das in unser Ant-Skript aufnehmen. Hier können wir regelmäßig überprüfen, wie es mit der Qualität des Quellcodes bestellt ist. Das kann so weit gehen, dass ein identifizierter Defekt während des Builds den ganzen Build fehlschlagen lässt. Sun beschreibt Auswirkungen und Bedeutung von Code-Konventionen folgendermaßen:

1. 80% der Kosten eines Produkts (verteilt auf seine Lebenszeit) fallen für die Wartung an.
2. Software wird über ihre Lebenskurve selten von ein und demselben Entwickler betreut.
3. Code-Konventionen überprüfen die Lesbarkeit von Software und erlauben es Entwicklern, neuen Code schneller und besser zu verstehen

Checkstyle bringt von Haus aus zahlreiche Codeprüfungen mit, von denen Sie im Folgenden eine Auswahl sehen:

JavaDoc-Kommentare
: Überprüft, ob die JavaDocs vorhanden und vollständig sind.

Namenskonventionen
: Überprüft, ob Klassen, Methoden und Variablen einen Namen haben, der *einem* Muster folgt.

Header der Klassen
: Überprüft, ob die Klassen mit einem bestimmten Header beginnen (z. B. Lizenzbestimmungen).

Importe
: Überprüft auf *-Importe, doppelte Importe, nicht genutzte Importe usw.

Größe
: Checkt die Länge von Statements, Methoden, Zeilen, Klassen etc. Zu große Klassen weisen beispielsweise auf ein suboptimales Design hin.

Whitespaces
: Überprüft die falsche Nutzung von Leerzeichen bzw. stellt zu viele fest.

Modifier
: Überprüft die empfohlene Reihenfolge der Modifier, angefangen mit `public`, und mögliche Redundanzen (beispielsweise ein `public` für Methoden von Interfaces).

Block Checks
: Checkt den Aufbau von Codeblöcken und findet leere Blöcke, überprüft, ob die geschweifte Klammer in einer neuen Zeile steht u.Ä.

Coding
: Checkt auf Coding-spezifische Defekte. Beispiele dafür sind überschriebene *equals*, wenn es *hashCode* nicht ist, und leere Statements, also alleinstehende Semikola, was bei Schleifen und Alternationen fatale Folgen haben kann.

Class Design
: Checkt, ob die Klasse für eine Erweiterung entworfen ist oder nicht, kontrolliert, dass nur »`static final`« Members »`public`« sind und andere nicht u.Ä.

Duplicate Code
: Hilft bei der Identifikation von doppeltem Code, der ja häufig von der Unsitte des Copy-and-Paste herrührt.

Metriken
: Überprüft den Code anhand von Metriken. Interessant ist dabei zum Beispiel die »Fan-out complexity«, also die Anzahl der Klassen, von der eine Klasse abhängig ist, und die »cycomatic complexity«, die Durchlaufpfade durch ein Programm.

Eine Geschichte über Coding-Standards

Es war einmal ein Projekt, in dem über Coding-Standards nachgedacht wurde. Das entwickelte Produkt war nun schon einige Jahre alt und historisch gewachsen. Mein Vorschlag war, für neuen Code hohe Qualitätsansprüche vorzulegen und alten Code zwar auch auf Metriken zu testen, diese sollten aber nicht so anspruchsvoll sein. Jemand schlug vor, dabei die zahlreichen Warnungen des alten Codes zu zählen und sich zu merken. Würde dann an dem Legacy-Code eine Änderung oder Erweiterung durchgeführt, würde man erneut die Defekte zählen und der ersten Zählung gegenüberstellen. Meine Reaktion war: »Oh mein Gott! Die Quantität von Qualität zählen? Was ist, wenn wir im Vorbeigehen einen alten Defekt eliminieren, dafür aber einen neuen einbauen? Die Zahl der Defekte bleibt gleich! Was ist, wenn ein neuer Defekt zehnmal so schlimme Konsequenzen hat wie der alte, eliminierte?« Die Antwort darauf war: »Dann müssen wir die gewichten und eine relative Quantität ermitteln«. Merken Sie sich: Lassen Sie sich auf so etwas nicht ein. Führen Sie eine Ursachenanalyse durch. Zählen Sie um Gottes Willen keine Defekte – eliminieren Sie sie!

Daneben existieren noch andere, wie speziell für J2EE-Bereiche entwickelte. Auch eigene Tests können komfortabel geschrieben werden.

Wenn Sie alten Code übernehmen müssen, den Sie nicht kennen, dann führen Sie mit CheckStyle Tests durch, wie die Codestruktur aussieht, wo Redundanzen liegen usw.

Ein konkretes Beispiel für eine Checkstyle-Prüfung

Sehen wir uns an, wie es konkret aussieht, wenn wir Klassen mit Checkstyle untersuchen möchten. Wir laden uns zunächst Checkstyle von *checkstyle.sourceforge.net* herunter und entpacken die Artefakte in das Verzeichnis *checkstyle-dist*. Auf derselben Ebene haben wir einen *src*-Ordner. Dort liegt eine beispielhafte Java-Datei, die wir checken möchten. Wir illustrieren das weitere Vorgehen an der Klasse *Agile.java*. Selbstverständlich werden Sie im produktiven Einsatz mehrere Klassen bzw. Pakete testen wollen bzw. das ganze Projekt. Im vorliegenden Beispiel haben wir im Verzeichnis *src* das Paket *com/huettermann* mit der Klasse *Agile.java* (siehe Beispiel 18-2).

Beispiel 18-2: Die Basisklasse

```
package com.huettermann;

/**
 * @author Michael Huettermann
 **/
public class Agile {
   private final String name;
```

Beispiel 18-2: Die Basisklasse (Fortsetzung)

```
   /**
    * the name
    * @param name myName
    **/
   public Agile(String name) {
      this.name = name;
   }

   /**
    * get the name
    * @return name the name
    **/
   public String getName() {
      return name;
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      Agile agile = (Agile) o;
      if (!name.equals(agile.name)) return false;
      return true;
   }

   @Override
   public int hashCode() {
      return name.hashCode();
   }
}
```

Es ist die Klasse, die bereits zum Einsatz kam, ein Value Object mit überschriebenen `equals`- und `hashCode`-Methoden. Nun wollen wir Checkstyle in den Build-Prozess einbinden. Ein exemplarisches *build.xml* zeigt Beispiel 18-3. Denken Sie daran, dass wir hier nur den Checkstyle-spezifischen Teil illustrieren. In einem produktiv eingesetzten Ant-Skript werden Sie sicherlich je nach Anforderung verschiedene der hier vorgestellten Ant-Bausteine einbinden wollen.

Beispiel 18-3: Die Einbindung von Checkstyle in den Build-Prozess

```
<project name="Conventions" default="check">
   <property name="check.home" value="/home/michael/checkstyle"/>

   <taskdef resource="checkstyletask.properties"
         classpath="${check.home}/checkstyle-dist/checkstyle-all-4.2.jar"/>

   <target name="check" depends="clear,checkstyle"
        description="generate report of coding violations."/>
   <target name="clear">
      <delete dir="${check.home}/report"/>
      <mkdir dir="${check.home}/report"/>
```

Beispiel 18-3: Die Einbindung von Checkstyle in den Build-Prozess (Fortsetzung)

```
   </target>
   <target name="checkstyle">
       <checkstyle config="${check.home}/checkstyle-dist/sun_checks.xml"
              failOnViolation="false">
           <formatter type="xml" tofile="${check.home}/report/checkstyle_report.xml"/>
           <fileset dir="${check.home}/src" includes="**/*.java"/>
       </checkstyle>

       <xslt in="${check.home}/report/checkstyle_report.xml" out="${check.home}/report/check
style_report.html" style="${check.home}/checkstyle-dist/contrib/checkstyle-simple.xsl"/>

   </target>
</project>
```

Zunächst ist wichtig, dass wir den Checkstyle-Task definieren und dabei *checkstyle-all-4.2.jar* einbinden (bzw. nach Wunsch eine andere Version). Wir haben auch in diesem Build-Skript wieder ein anfängliches `clear`-Target, das das *report*-Verzeichnis bei jedem Durchlauf initialisiert. Kern der Verarbeitung ist das `checkstyle`-Target. Hier definieren wir über das `config`-Attribut, welche Checks wir beim Test zu Hilfe nehmen. Innerhalb der Standarddistribution sind die Coding Conventions von Sun im Dokument *sun_checks.xml* in Checkstyle-Notation inkludiert. Der nächste Punkt von Interesse ist der Formatierer. Wir lassen einen Report in XML-Syntax innerhalb unseres *report*-Verzeichnisses erstellen. Mit einem geschachtelten `fileset`-Element geben wir die Menge der zu untersuchenden Klassen an, die sich hier der Übersichtlichkeit halber auf eine Datei beschränkt.

```
<checkstyle config="${check.home}/checkstyle-dist/sun_checks.xml"
  failOnViolation="false">
   <formatter type="xml" tofile="${check.home}/report/checkstyle_report.xml"/>
   <fileset dir="${check.home}/src" includes="**/*.java"/>
     </checkstyle>
```

Das generierte XML-Dokument enthält alle gewünschten Informationen, ist aber doch für den Menschen nicht wirklich gut lesbar. Aus diesem Grund nutzen wir ein der Checkstyle-Distribution beigefügtes Stylesheet, um aus dem XML-Dokument schön in einem Browser darstellbares HTML zu machen.

```
<xslt in="${check.home}/report/checkstyle_report.xml" out="${check.home}/report/checkstyl
e_report.html"
style="${check.home}/checkstyle-dist/contrib/checkstyle-simple.xsl"/>
```

Wir nutzen dazu den `xslt`-Task, der als Eingabedatei unser *.xml*-Dokument im *report*-Verzeichnis enthält und in dasselbe Verzeichnis als Ausgabe *checkstyle_report.html* schreibt. Checkstyle bietet unterschiedliche Stylesheets an. Wir nehmen hier das im *contrib*-Verzeichnis vorhandene *checkstyle-simple*-Stylesheet.

Nach Ausführung des Ant-Skripts mit

```
ant
```

erscheint folgende Ausgabe:

```
michael@BlueGene:~/checkstyle$ ant
Buildfile: build.xml

clear:
   [delete] Deleting directory /home/michael/checkstyle/report
    [mkdir] Created dir: /home/michael/checkstyle/report

checkstyle:
[checkstyle] Running Checkstyle 4.2 on 1 files
     [xslt] Processing /home/michael/checkstyle/report/checkstyle_report.xml to /home/mi
chael/checkstyle/report/checkstyle_report.html
     [xslt] Loading stylesheet /home/michael/checkstyle/checkstyle-
dist/contrib/checkstyle-simple.xsl

check:

BUILD SUCCESSFUL
Total time: 4 seconds
michael@BlueGene:~/checkstyle$
```

und es wird tatsächlich ein *report*-Verzeichnis angelegt, in dem sich die zwei Dateien wiederfinden:

```
checkstyle_report.xml
checkstyle_report.html
```

Öffnen wir nun das generierte HTML-Dokument, ergibt sich das in Abbildung 18-5 dargestellte Bild. Zunächst bekommen wir eine Übersicht über den Erfolg der Untersuchung. In unserem Fall wird zum einen die Datei *package.html* auf ihre Existenz hin geprüft. Sie ist integraler Bestandteil der JavaDoc-Dokumentation. Sie ist nicht vorhanden und entsprechend wird ein Fehler angezeigt. Dann folgt die Untersuchung von *Agile.java*. Für die Bedeutung der jeweiligen Fehler konsultieren Sie bitte die Dokumentation.

Fünftes Szenario: HotSwap

Sie können mit Ant auch HotSwap ermöglichen. HotSwap bedeutet, dass Sie Java-Klassen in Echtzeit austauschen können, ohne dass Sie Ihren Server bzw. die Java Virtual Machine neu starten müssen. Dies ist ein *inkrementelles Deployment* und steigert die Produktivität ungemein. Sie sparen ausgesprochen viel Zeit, und das nicht nur einmal, sondern dauerhaft.

Eine gute Automatisierung fällt desto mehr ins Gewicht, je häufiger die automatisierten Schritte durchlaufen werden. Aber Achtung: Es ist genauso sinnvoll, Schritte zu optimieren oder für eine ganz bestimmte Aufgabenstellung eine effiziente Lösung zu finden. Das geht häufig unter.

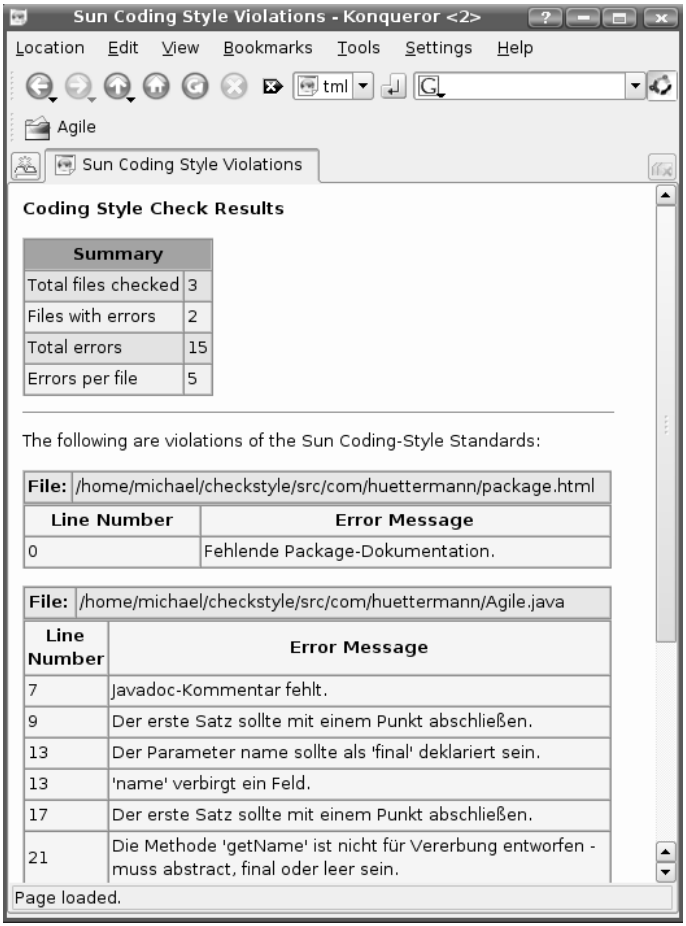


Abbildung 18-5: Der Checkstyle-Report

Das hier vorgestellte HotSwap-Skript kann unter vielen Bedingungen hilfreich sein, insbesondere wenn Sie mit einer IDE arbeiten, die Ihre Dateien mit einem Compiler kompiliert, der nicht dem produktiv eingesetzten entspricht. Arbeiten Sie zum Beispiel mit der Eclipse-IDE und kompilieren damit Ihre Sourcen, werden die Sourcen mit dem Eclipse-eigenen Java Compiler JDT kompiliert. Das kann beim HotSwap zu Problemen führen, wenn im Rahmen Ihres Build-Skripts die Sourcen mit dem javac-Kompiler von Sun Microsystems kompiliert wurden. Verschiedene Compiler erzeugen aus demselben Quellcode teilweise unterschiedlichen Bytecode. Sind die Signaturen der Methoden verschieden (ist also das »Schema« unterschiedlich), misslingt der HotSwap.

Laden Sie sich *hotswap.jar* von *https://hotswap.dev.java.net/* herunter und legen Sie es in das Verzeichnis *ant/lib*. Durch den Aufruf des Targets `hotswap` werden Ihre Klassen mit dem im Ant-Skript eingestellten Compiler neu kompiliert (siehe Beispiel 18-4). In unse-

rem Fall geschieht die Kompilierung mit dem javac-Compiler. Anschließend werden die Klassen Ihres laufenden Systems durch aktuelle ersetzt, falls es welche gibt. Das `taskdef`-Element zu Beginn zeigt, wie Ant erweitert werden kann: In der referenzierten Java-Klasse wird der Task `hotswap` implementiert.

Beispiel 18-4: Ein Ant-Ausschnitt für HotSwap

```
<taskdef name="hotswap" classname="dak.ant.taskdefs.Hotswap"/>
<target name="hotswap">
  <tstamp>
    <format property="class.tstamp" pattern="MM/dd/yyyy kk:mm:ss" />
  </tstamp>

  <javac destdir="${build.classes.dir}>
    <fileset dir="${dev.src.dir}" includes="**/*.java"/>
  </javac>

  <hotswap verbose="true" port="9000">
    <fileset dir="${build.classes.dir}" includes="**/*.class">
      <date datetime="${class.tstamp}" pattern="MM/dd/yyyy kk:mm:ss" when="after" granularity
="0"/>
    </fileset>
  </hotswap>
</target>
```

Application Server – Start und Stop

Sie können auch einen ganzen Application Server skriptgesteuert starten und stoppen. Dazu nutzen Sie den Ant-Task `exec` und rufen das Shell-Kommando auf, das Sie auch direkt auf der Konsole eingeben würden. Daneben existieren Projekte, die diesen Weg abstrahieren und ein Framework dafür anbieten. Dieses Framework ist einfach zu nutzen und steuert dann selbst den Application Server. *Cargo* ist ein solches Framework.

Best Practices und Ant-Tipps

Im Folgenden fasse ich wesentliche Punkte bei der Nutzung von Ant zusammen.

Abhängigkeiten verwalten
: Nutzen Sie die Möglichkeit, Abhängigkeiten zu definieren. Kompilieren Sie also beispielsweise eine Komponente, die selbst Input für die Kompilierung einer anderen Komponente ist. Versichern Sie sich, dass die Abhängigkeiten Sinn machen und keine Abschnitte unnötig doppelt ausgeführt werden.

Pfade geschickt einsetzen
: Nutzen Sie absolute Pfade. Definieren Sie dabei ein Wurzelverzeichnis absolut mithilfe einer Ant-Property. Ausgehend davon können Sie weitere Pfade relativ ansprechen.

Konfigurieren über Properties
: Lagern Sie Konfigurationsparameter aus, und zwar in Properties innerhalb oder außerhalb des Ant-Skripts.

Zentrales Skript
: Platzieren Sie *build.xml* in das Wurzelverzeichnis des Projekts.

Nutzen Sie SCM
: Das Build-Skript ist Bestandteil des Projekts. Es muss genauso der Softwareversionierung zugeführt werden wie der eigentliche Quellcode. Stellen Sie die mit Ant erstellten Artefakte an zentraler Stelle bereit, beispielsweise durch Kopieren auf eine Projektseite. Checken Sie die Artefakte aber nicht in das Versionierungssystem ein. Das Ant-Skript selbst kann jederzeit die Artefakte neu generieren.

Ant als gemeinsame Basis
: Unabhängig davon, welche Werkzeuge bei Ihnen eingesetzt werden, ist Ant der kleinste gemeinsame Nenner. Ant-Skripten können von allen IDEs genutzt und aufgerufen werden.

Namen
: Es ist zwar nicht zwingend notwendig, aber benennen Sie alle Ant-Skripten einheitlich *build.xml*.

Nutzen Sie »zipfileset«
: »zipfileset« hilft Ihnen dabei, einen Stapel Dateien in ein Archiv zu packen, wobei Sie Quell- und Zielpfad übergeben können. So ersparen Sie sich das Kopieren in temporäre Arbeitsverzeichnisse.

Unterschiedliche Umgebungen, unterschiedliche Skripten
: Für unterschiedliche Umgebungen wird es sich häufig anbieten, physisch getrennte *build.xml* vorliegen zu haben.

Projektstruktur
: Nutzen Sie die gleiche Projektstruktur für alle Projekte. Auf diese Weise werden die Ant-Skripte schnell auffindbar sein, da Sie die Pfade wiedererkennen.

Agile Entwicklung
: Erstellen Sie auch das Ant-Skript agil. Implementieren Sie beispielsweise nur die Targets, die Sie gerade benötigen, und refaktorisieren Sie, wenn es nötig ist. Das ist der Kern des Build-Prozesses.

Modulare Build
: Es kann sinnvoll sein, einzelne Targets wiederholt auszuführen. Beispielsweise möchten Sie ggf. Teil-Builds erstellen. Verwenden Sie die vorgestellte Modularisierung.

Benennung
: Folgen Sie für alle Targets innerhalb Ihrer *build.xml*-Skripten einem einheitlichen Vorgehen.

Modularisierung und Wiederverwendung
: Nutzen Sie alle Möglichkeiten der Modularisierung und Strukturierung, die Ihnen XML bzw. Ant bietet. Dazu gehören Aufrufe von `ant` und `antcall` sowie der Import von wiederverwendbaren Teilen. Einige Targets werden projektunabhängig und somit wiederverwendbar sein.

API
: Fügen Sie Ihren Targets gute Beschreibungen hinzu. Diese können, mit einem Hinweis auf das Default-Target, ohne Blick ins Build-File mit `ant -projecthelp` gelistet werden. Lassen Sie bei internen Targets (also Targets, die separat nicht nutzbar sein sollen und nicht Bestandteil der offenen Schnittstelle sind) das `description`-Attribut des Targets weg. Auf diese Weise werden diese Targets nicht mehr gelistet.

Gleiche Vorbedingungen
: Ein Build sollte immer bei null anfangen. Fügen Sie also beispielsweise ein `clean`-Target ein, das bei jedem Durchlauf die Startbedingungen exakt gleich gestaltet.

Überraschungen
: Wählen Sie das Default-Target des Build-Skripts vernünftig, so dass Sie keine Überraschungen erleben, wenn jemand das Ant-Skript ausführt.

Zusammenfassung

Ant/XML ist keine Programmiersprache. Mit XML per Hand zu programmieren ist fehleranfällig und kann sehr nervenaufreibend sein, insbesondere wenn keine IDE-Unterstützung vorliegt, die z.B. über öffnende und schließende Tags wacht. Der Entwickler von Ant, James Duncan Davidson, sagt im Rahmen des Buchs »Pragmatic Project Automation« von Mike Clark, dass XML für Einsatzgebiete wie Ant nicht geeignet sei und er im Nachhinein betrachtet eine Skriptsprache wie JavaScript dafür hätte nehmen sollen.

Fangen Sie mit Ant an, wenn Sie erste Erfahrungen mit Build-Prozessen sammeln wollen, Ihr Produkt oder der Build-Prozess nicht umfangreich ist oder für Sie eine umfangreichere Lösung wie Maven 2 oder CruiseControl zunächst nicht in Frage kommt, da Sie beispielsweise erstmal für Ihren eigenen Arbeitsplatz einen agilen Prozess aufbauen möchten (siehe Kapitel 6, *Agile Prozesse einführen*). Ant ist die Minimallösung, die sie nutzen müssen. Sagt Ihnen jemand, Sie bräuchten kein Ant, die »paar Schritte« mache doch jeder manuell, dann hat er nicht verstanden, wie sich Software effizient entwickeln lässt. Kaufen Sie ihm dieses Buch!

Ant erlaubt zwar, deklarative Tasks zu definieren, es existiert allerdings kein komfortabler Mechanismus, um Fehler zu verarbeiten (obwohl es Listener-Klassen gibt). Ferner sind keine komplexen konditionalen Statements möglich. Ant-Skripten liegen zwar in einem für Menschen lesbaren XML-Format vor, aber die Tasks werden schnell unübersichtlich, wenn die Aufgaben zunehmen. Bei großen Projekten ist das Build-Skript sehr groß und nur noch schwierig zu überblicken.

Ant ist ein De-facto-Standard. Es ist mächtig, plattformunabhängig und erweiterbar. Für viele Werkzeuge existieren Ant-Anbindungen. Aus diesen Gründen und wegen der enormen Verbreitung und der weit verbreiteten Hemmung gerade großer Unternehmen, ein lange genutztes Werkzeug zu ersetzen, ist davon auszugehen, dass Ant auch weiterhin nichts von seiner immensen Bedeutung verlieren wird.

Wir haben uns in diesem Kapitel auf Ant und auf für Ant typische Fragestellungen konzentriert. In verschiedenen Szenarien habe ich illustriert, wie Sie mit Ant Ihren Entwicklungsprozess agil gestalten können. Einzelne Punkte werden wir uns später nochmal aus einem anderen Blickwinkel anschauen, detaillierter und unter Betrachtung anderer, organischerer Werkzeuge, die »out of the Box« besser unterstützen, insbesondere Maven 2, das den Anspruch erhebt, der Nachfolger von Ant zu sein.

KAPITEL 19

Kontinuierliche Integration mit CruiseControl

Wir haben uns in den letzten Kapiteln verschiedenste Dimensionen angeschaut, wie man mit Artefakten agil umgeht und sie erstellt, überprüft und verteilt. Ant war dabei stets Bindeglied, unser Schweizer Taschenmesser, das die unterschiedlichen Werkzeuge integrierte. Es half uns dabei, einen Prozess abzubilden. Was fehlt jetzt noch zu der Infrastruktur, die »Continuous Integration« umfassend abbildet und ermöglicht?

Die Erstellung von Software ist nicht nur ein Ingenieursprozess, eine Kunst, es ist auch der kontinuierliche Umgang mit Menschen und einer Menge Abhängigkeiten und Artefakte. Entwickler checken permanent etwas in das Repository ein, und wir haben unsere Ant-Skripten, die eine Integration durchführen. Doch wie und wann starten wie diese Skripten?

Die Ant-Skripten müssen häufig laufen. Es scheint also fehleranfällig und für den einzelnen eher lästig, wenn eine Person den Zeitpunkt jedes Mal selbst bestimmt und das Skript manuell anstößt, so wie wir es in den letzten Kapiteln gemacht haben. Also warum sollen wir nicht auch das automatisieren?

Der pragmatische Ansatz: cron

Der erste naheliegende Weg ist, einen Dienst des zugrundeliegenden Betriebssystems zu nutzen. Ist der Integrationsrechner ein Linux-System, können wir beispielsweise ganz pragmatisch mit `cron` einen kontinuierlichen Job aufsetzen. `cron` ist ein im Hintergrund laufendes Programm, um zu spezifizierten Zeiten angegebene Aktivitäten durchzuführen. Gewöhnlich ist das bei einfachen Skripten praktisch, kann aber auch für eine Continuous Integration eine pragmatische Strategie sein.

Zunächst legen wir ein Skript namens *cronscript.sh* in unserem Home-Verzeichnis */home/myuser/cron* an:

```
#!/bin/bash
date
exit 0
```

Anschließend machen wir diese neue Datei ausführbar

```
chmod 755 cronscript.sh
```

und editieren die *crontab*

```
crontab -e
5 11 31 * * /home/myuser/cron/cronscript.sh >> /home/myuser/cron/cronprotocol.txt
```

Jede Zeile der `cron`-Tabelle ist eine ausführbare Aktivität. Innerhalb der Zeile gibt die Position des Werts Aufschluss über die Funktion: An erster Stelle wird die Minute angegeben, es folgen Stunde, Tag, Monat und Wochentag. Zu diesem Zeitpunkt soll der Job ausgeführt werden, also das Skript, das anschließend angegeben wird. Mehrere Zeitpunkte können durch Kommata getrennt und mehrere Intervalle mit Bindestrichen getrennt angegeben werden, oder auch Joker (Sternchen). Den generellen Aufbau illustriert Tabelle 19-1.

Tabelle 19-1: Die Bestandteile einer crontab-Zeile

Platzhalter	Werte
Minute	0-59
Stunde	0-23
Tag	1-31
Monate	1-12
Wochentag	0-6
Befehl	Der auszuführende Befehl

Der im Beispiel definierte Eintrag stößt also unser *cronscript.sh* an jedem 31. um 11:05 an. Monate und Wochentage werden nicht weiter eingeschränkt. Die Ausgabe des Skripts fließt in eine Log-Datei *cronprotocol.txt*.

Nachdem das Skript gelaufen ist, finden wir die neu angelegte Datei im Verzeichnis wieder. Da wir in dem ausgeführten Skript das Datum ausgeben, sieht *cronprotocol.txt* so aus:

```
So 31 Dez 11:05:01 CET 2006
```

Wir können auf diese Weise eine Automatisierung implementieren und Skripten automatisch anstoßen. Leider hat das auch Nachteile: Die Flexibilität ist doch recht eingeschränkt. So können wir beispielsweise keine Bedingungen hinterlegen, unter welchen Umständen das Skript ausgeführt wird oder nicht.

Der umfassende Ansatz: CruiseControl

CruiseControl ist ein frei erhältliches Werkzeug, um einen kontinuierlichen Build-Prozess zu steuern. So bietet es Integrationen zu E-Mail, Ant und Versionskontrollsystemen. Es bietet eine Webschnittstelle an, um Informationen über den aktuellen und zurückliegende Build-Prozesse zu erhalten.

CruiseControl definiert einen Build-Zyklus. Dieser besteht im Wesentlichen aus folgenden Schritten:

1. Bestimmen, ob ein Build notwendig ist
2. Build durchführen
3. Log-File erstellen
4. Informationen bereitstellen

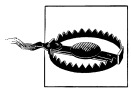
Installation

Laden Sie sich CruiseControl von *http://cruisecontrol.sourceforge.net/* herunter und entpacken es. Rufen Sie dann die im Wurzelverzeichnis Ihrer CruiseControl-Installation liegende Datei auf: *./cruisecontrol.sh*. Das startet den Integrationsserver. Der CruiseControl-Dämon-Prozess wacht zur vom Nutzer bestimmten Zeit auf und möchte einen Build-Zyklus anstoßen. Die CruiseControl-Konfiguration wird bei jedem Durchlauf gelesen.

Konfiguration

Sie müssen nun diverse Einstellungen vornehmen. Obwohl es auch andere Möglichkeiten gibt, zeige ich Ihnen hier das in der Praxis etablierte Vorgehen.

Sie gehen in das Root-Verzeichnis Ihrer CruiseControl-Installation. Auf diese Weise sind Ihre Daten sauber von der Installation getrennt, wodurch Sie später CruiseControl besser updaten können.



Es wäre noch besser, die einzelnen Verzeichnisse in einem separaten Arbeitsverzeichnis anzulegen. Auf diese Weise wären Ihre Daten noch sauberer von der Installation getrennt, wodurch Sie CruiseControl *noch* besser updaten könnten. Allerdings ist CruiseControl dafür nicht sauber konzipiert. *cruisecontrol.sh* geht von relativen Pfaden aus, die abhängig vom Ort des Aufrufs sind, und lädt die Bibliotheken und startet die Webanwendung unter Jetty. Wundern Sie sich nicht, wenn das nicht rund läuft. Es ist einfacher, den hier vorgestellten Weg zu gehen.

```
michael@BlueGene:~/cc/cruisecontrol-bin-2.5$ ls -la
insgesamt 216
drwxr-xr-x 10 michael michael   4096 2006-12-31 18:02 .
drwxr-xr-x  6 michael michael   4096 2006-12-31 16:58 ..
drwxr-xr-x  6 michael michael   4096 2006-04-27 22:39 apache-ant-1.6.5
drwxr-xr-x  3 michael michael   4096 2006-12-31 17:26 artifacts
-rw-r--r--  1 michael michael    478 2006-12-31 17:25 build-myproject.xml
drwxr-xr-x  3 michael michael   4096 2006-12-31 18:01 checkout
-rw-r--r--  1 michael michael    903 2006-12-31 17:19 config.xml
-rw-r--r--  1 michael michael   4922 2006-04-24 23:13 cruisecontrol.bat
-rw-r--r--  1 michael michael 145449 2006-12-31 18:11 cruisecontrol.log
-rwxr-xr-x  1 michael michael   3365 2006-04-24 23:18 cruisecontrol.sh
drwxr-xr-x  4 michael michael   4096 2006-04-27 22:39 docs
drwxr-xr-x  3 michael michael   4096 2006-04-23 20:48 lib
```

```
drwxr-xr-x  3 michael michael   4096 2006-12-31 17:25 logs
-rw-r--r--  1 michael michael    349 2006-12-31 18:01 myproject.ser
drwxr-xr-x  3 michael michael   4096 2006-04-27 22:39 projects
-rw-r--r--  1 michael michael    283 2006-04-27 22:39 README.txt
drwxr-xr-x  3 michael michael   4096 2006-04-27 22:39 webapps
```

Zunächst legen Sie die Verzeichnisse *artifacts* und *checkout* manuell an. Das müssen Sie nur ein einziges Mal tun. Dann erstellen Sie auf dieser Ebene ein Ant-Skript, das Sie nach dem Muster *build-* + Name Ihres Projekts benennen. Es ist ein Wrapper zu dem Ant-Skript, das Ihre Anwendung, also in diesem Fall *myproject*, besitzt. So können Sie vorbereitende Maßnahmen durchführen, siehe Beispiel 19-1. In dieser Wrapper-Build-Datei löschen wir das *checkout*-Verzeichnis, in das wir bei jedem Durchlauf mit CruiseControl die Sourcen frisch aus dem Subversion-Repository auschecken. Abschließend wird der Verarbeitungsprozess an das eigentliche Ant-Skript des Projekts, nämlich *build.xml* im Trunk, delegiert.

Beispiel 19-1: Ein praktischer Wrapper für unsere eigentliche Projekt-Build-Datei

```
<project name="build-myproject" default="build" basedir="checkout">
   <target name="build">
      <delete dir="myproject"/>
      <exec executable="svn" failonerror="true">
         <arg value="co"/>
         <arg value="svn://localhost/agile/myproject"/>
         <arg value="--username"/>
         <arg value="michael"/>
         <arg value="--password"/>
         <arg value="michael"/>
      </exec>
      <ant antfile="myproject/trunk/build.xml"/>
   </target>
</project>
```

Die zentrale Konfiguration für CruiseControl befindet sich in *config.xml*. Hier haben Sie zahlreiche Möglichkeiten, die Verarbeitung zu steuern. Für ein Projekt haben sie jeweils ein `project`-Element. Es können also auch mehrere Projekte parallel behandelt werden. In unserem Beispiel (siehe Beispiel 19-2) definieren wir zunächst einen `currentbuildstatuslistener`, der in eine Datei den aktuellen Build-Status schreibt. Anschließend definieren wir mit einem `modificationset`, dass CruiseControl den Build nur durchführen soll, wenn sich im Subversion-Repository etwas geändert hat (davon abzuweichen kann sinnvoll sein, wenn Sie zu festen Zeitpunkten einen Build anstoßen wollen). Damit CruiseControl das feststellen kann, geben eine bereits ausgecheckte Arbeitsversion an bzw. den Ort, wo unser Wrapper die Quellen auschecken wird. Anhand der dortigen Log-History überprüft CruiseControl, ob Änderungen vorliegen. Auf diese Weise ersparen Sie sich auch, die Verbindung zum Repository hart in *config.xml* coden zu müssen.

Mit den `bootstrapper`-Elementen können Sie Aktivitäten anstoßen, bevor der eigentliche Build startet. `svnbootstrapper` beispielsweise erlaubt es, einzelne lokale Dateien (wie *build.xml* oder *build.properties*) mit den Versionen im Repository abzugleichen, ehe der Build startet.

`quietperiod` gibt an, dass seit der letzten Änderung im Repository 60 Sekunden vergangen sein müssen, ehe tatsächlich ein Build angestoßen wird. Werden gerade viele verschiedene Sachen eingecheckt, so wird der Build-Prozess um die angegebene Zeit verzögert. Das war bei Nutzung von CVS noch hilfreicher als bei Subversion, da CVS die Commits auf das Repository nicht atomar durchgeführt hat. Startete der Build mitten in einem umfangreichen Commit, führte das unweigerlich zu einem kaputten Build.

Beispiel 19-2: Die Datei config.xml für unseren CruiseControl-Ablauf

```
<cruisecontrol>
   <project name="myproject" buildafterfailed="false">
      <listeners>
         <currentbuildstatuslistener file="logs/myproject/status.txt"/>
      </listeners>
      <modificationset quietperiod="60">
         <svn localworkingcopy="checkout/myproject"/>
      </modificationset>
      <schedule interval="14400" >
         <ant anthome="/home/myuser/ant/apache-ant-1.7.0"
              buildfile="build-myproject.xml"
              uselogger="true"
              usedebug="false" />
      </schedule>
      <log dir="logs/myproject">
        <merge dir="checkout/myproject/trunk/junit-report"/>
      </log>
      <publishers>
         <artifactspublisher dir="checkout/myproject"
             dest="artifacts/myproject/"/>
      </publishers>
   </project>
</cruisecontrol>
```

`schedule` definiert – man vermutet es fast schon – das Intervall, in dem jeweils ein neuer Build angestoßen werden soll. Das geschieht hier alle 14400 Sekunden (also alle vier Stunden) und wird tatsächlich nur durchgeführt, wenn sich im Repository etwas geändert hat. Wie groß Sie dieses Intervall nun in Ihrem Projekt wählen, kann schwerlich verallgemeinert gesagt werden. Bei manchen Projekten möchte man bei jeder Änderung im zentralen Repository einen neuen Build erstellen, bei anderen reicht einmal nachts aus. Berücksichtigen Sie dabei, wie lange der Build dauert. Dauert der komplette Build eine Stunde, macht es keinen Sinn, alle zehn Minuten einen neuen erstellen zu wollen. Je mehr Builds Sie auf der anderen Seite haben, und je schneller Sie über kaputte Builds informiert werden, desto besser.

Nachdem wir angegeben haben, wann CruiseControl bauen soll, können wir nun beschreiben, *wie* CruiseControl es machen soll. Wir hinterlegen also das Build-Werkzeug. Hier nutzen wir Ant. Wollen wir nicht das mit CruiseControl mitgelieferte Ant nutzen (es könnte uns zu alt sein), können wir eine eigene Distribution angeben. Als Build-Datei wählen wir unseren Wrapper.

Wir führen in unserem Projekt-Build Unit-Tests durch. Diese möchten wir fortschreiben. Wir mischen unseren in XML vorliegenden JUnit-Report in die Log-Datei von CruiseControl. So können wir über die Weboberfläche auf die Testergebnisse zugreifen.

Schließlich geben wir noch einen Publisher an, also eine Direktive, die bestimmt, wie wir die Build-Ergebnisse kommunizieren möchten: Hier definieren wir ein Verzeichnis, in das die Arbeitsgrundlage, die ausgecheckten Dateien geschoben werden sollen. Auf diese Weise können wir durch diese Artefakte über die Web-Oberfläche von CruiseControl browsen.

Achten Sie darauf, wie Sie die einzelnen Einträge nennen. Die Datei *status.txt* muss zum Beispiel tatsächlich genau so heißen. Wenn Sie sie anders nennen, wird sie nicht gefunden. Der Name muss korrelieren mit `context-param` in *web.xml* innerhalb von *webapps/cruisecontrol*.

Sie können CruiseControl in verschiedensten Konfigurationen nutzen. Dazu gehört die Einbindung über das Build-Werkzeug Maven statt Ant, oder eine Nutzung anderer Versionskontrollsysteme. Ant ist sicher eine gängige Wahl. Ich zeige Ihnen hier das generelle Vorgehen. Selbstverständlich können Sie mit CruiseControl und Einbindung von Ant alle Features nutzen, die Sie auch separat mit Ant nutzen. Exemplarisch führen wir hier eine Verarbeitung mit Unit-Tests durch.

Bei jedem Durchlauf wird das Projekt, also auch dessen *build.xml*, ausgecheckt. Diese Build-Datei, siehe Beispiel 19-3, entspricht im Wesentlichen dem Build-Skript aus dem Kapitel über TestNG. Wir mussten hier aber die Pfade anpassen.

Beispiel 19-3: Die zentrale Build-Datei unseres Projekts

```
<project default="reports" basedir=".">
    <property name="root.dir" value="/home/michael/cc/cruisecontrol-bin-2.5/checkout/
myproject/trunk"/>
    <path id="cp">
        <pathelement location="${root.dir}/lib/testng-5.3.1-jdk15.jar"/>
        <pathelement location="${root.dir}/classes"/>
        <pathelement location="${root.dir}/test"/>
    </path>

    <taskdef name="testng" classpathref="cp" classname="org.testng.TestNGAntTask"/>

    <target name="ini">
        <delete dir="${root.dir}/classes"/>
```

Beispiel 19-3: Die zentrale Build-Datei unseres Projekts (Fortsetzung)

```
        <delete dir="${root.dir}/junit-report"/>
        <delete dir="${root.dir}/testng-report"/>
        <delete>
            <fileset dir="${root.dir}/test" includes="**/*.class"/>
        </delete>
        <mkdir dir="${root.dir}/classes"/>
        <mkdir dir="${root.dir}/junit-report"/>
        <mkdir dir="${root.dir}/testng-report"/>
    </target>

    <target name="create">
       <testng classpathref="cp" groups="create,update" haltOnFailure="true" verbose="2" suite
name="Agile Development" testname="Some agile tests" outputdir="${root.dir}/testng-report">
          <classfileset dir="${root.dir}/test" includes="**/*.class"/>
       </testng>
    </target>

    <target name="compile">
        <javac srcdir="${root.dir}/src" destdir="${root.dir}/classes" classpathref="cp">
           <include name="**/*.java"/>
        </javac>
        <javac srcdir="${root.dir}/test" destdir="${root.dir}/test" classpathref="cp">
           <include name="**/*.java"/>
        </javac>
    </target>

    <target name="reports" depends="ini,compile,create">
      <junitreport todir="${root.dir}/junit-report">
        <fileset dir="${root.dir}/testng-report/Agile Development">
          <include name="*.xml"/>
        </fileset>
        <report format="noframes" todir="${root.dir}/junit-report"/>
      </junitreport>
    </target>

</project>
```

Wir starten nun CruiseControl aus dem Installationsverzeichnis mit

```
./cruisecontrol.sh
```

und erhalten folgende Ausgabe

```
...
Project myproject:  waiting for next time to build
2006-12-31 17:05:02,044 [Project myproject thread] INFO  ProjectController -
 myproject Controller: build progress event: waiting for next time to build
```

Abhängig davon, ob Sie bereits eine lokale Arbeitsversion vorliegen haben oder sie – wie in unserem Fall – beim ersten Lauf von CruiseControl zunächst erstellt werden muss, bekommen Sie eine Meldung, dass CruiseControl nicht anhand der in *config.xml* definierten Strategie auf eine Änderung im Repository schließen kann. Nichtsdestotrotz läuft CruiseControl. Das beunruhigt uns nicht, beim nächsten Build wird es ohne Probleme

durchlaufen. Beispiel 19-4 zeigt uns, dass CruiseControl nach Änderungen im Repository gesucht hat und fündig geworden ist. Nach der Ruhepause wird ohne weitere Änderungen ein Build gestartet.

Beispiel 19-4: CruiseControl angestoßen

```
2006-12-31 18:00:37,566 [Thread-48] INFO  ProjectController -
 myproject Controller: forcing build
2006-12-31 18:00:37,567 [Project myproject thread] INFO  Project          -
 Project myproject:  in build queue
2006-12-31 18:00:37,567 [Project myproject thread] INFO  ProjectController -
 myproject Controller: build progress event: in build queue
2006-12-31 18:00:37,571 [BuildQueueThread] INFO  BuildQueue       -
 now adding to the thread queue: myproject
2006-12-31 18:00:37,626 [Thread-49] INFO  Project          -
 Project myproject:  bootstrapping
2006-12-31 18:00:37,627 [Thread-49] INFO  ProjectController -
 myproject Controller: build progress event: bootstrapping
2006-12-31 18:00:37,627 [Thread-49] INFO  Project          -
 Project myproject:  checking for modifications
2006-12-31 18:00:37,628 [Thread-49] INFO  ProjectController -
 myproject Controller: build progress event: checking for modifications
2006-12-31 18:00:37,775 [Thread-49] INFO  ModificationSet  -
 1 modification has been detected.
2006-12-31 18:00:37,776 [Thread-49] INFO  ModificationSet  -
 A modification has been detected in the quiet period.
2006-12-31 18:00:37,777 [Thread-49] INFO  ModificationSet  -
 Sleeping for 52 seconds before retrying.
...
2006-12-31 18:01:35,181 [Thread-49] INFO  ProjectController -
 myproject Controller: build progress event: merging accumulated log files
2006-12-31 18:01:35,184 [Thread-49] INFO  Project          -
 Project myproject:  build successful
2006-12-31 18:01:35,185 [Thread-49] INFO  Project          -
 Project myproject:  publishing build results
2006-12-31 18:01:35,185 [Thread-49] INFO  ProjectController -
 myproject Controller: build progress event: publishing build results
2006-12-31 18:01:35,297 [Thread-49] INFO  Project          - Project myproject:  idle
2006-12-31 18:01:35,337 [Thread-49] INFO  ProjectController -
 myproject Controller: build progress event: idle
```

Die Weboberfläche

CruiseControl enthält eine eigene Weboberfläche. Auf dem eigenen Rechner erreichen Sie sie unter *http://localhost:8080* (siehe Abbildung 19-1). Die Startseite gibt Ihnen direkt Aufschluss darüber, welche Projekte Sie mit CruiseControl überwachen. Außerdem wissen Sie direkt, ob der letzte Build erfolgreich war und welchem CruiseControl-Label er zugewiesen wurde. Das ist kein Label oder Tag im Repository, sondern ein Kennzeichen, das nur von CruiseControl genutzt wird.

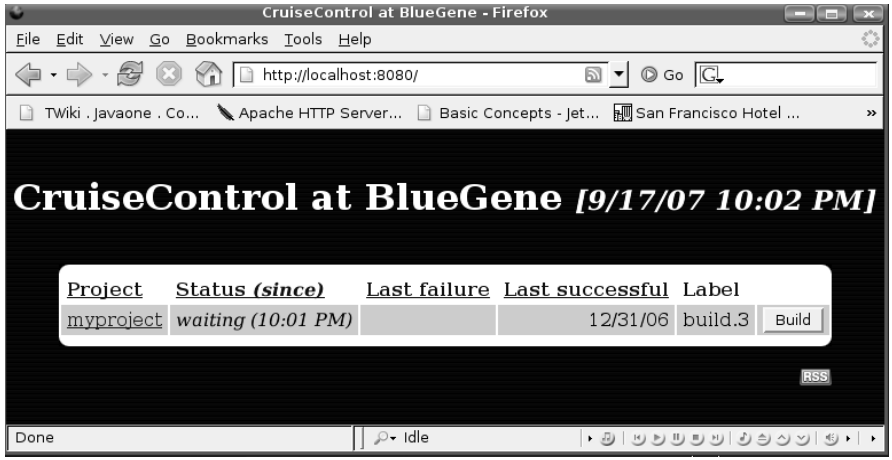


Abbildung 19-1: CruiseControl-Startseite

Sie sollten nach einem erfolgreichen Build den Stand taggen, um – unabhängig von CruiseControl – jederzeit Versionen Ihrer Software in Ihrem Versionskontrollsystem identifizieren und nutzen zu können. Nehmen Sie die notwendige Logik dazu in das Wrapper-Skript von Ant auf. Das CruiseControl-Label können Sie selbst bestimmen, wenn Sie es möchten.

Sie können über die Startseite einen neuen Build anstoßen. Klicken Sie auf das Projekt *myproject*, erhalten Sie weitere Informationen zu diesem Projekt (siehe Abbildung 19-2).

Auf der Projekt-Detailseite sehen Sie in der linken Leiste die History der Builds und rechts die Details zu einem konkreten Build. Sie können über die Übersicht ältere Builds anspringen und entsprechend die Details dazu jederzeit abfragen. Im vorliegenden Fall ist gerade ein erfolgreicher Build durchgelaufen. Die Artefakte des Builds können Sie sich über BUILD ARTIFACTS anschauen. Sie sehen das Datum des Builds und wie lange er gedauert hat, und die letzte Änderung im Subversion-Repository und wann sie gemacht wurde. Im mittleren Bereich der rechten Seite sehen Sie, ob die Unit-Tests aus *build.xml* erfolgreich gelaufen sind. Alle 13 Tests sind erfolgreich durchgelaufen. Im unteren Bereich schließlich sehen Sie die Änderungen im Repository seit dem letzten erfolgreichen Build.

Gehen Sie von dem Reiter BUILD RESULTS auf TEST RESULTS, werden Details zu den Unit-Tests geliefert (siehe Abbildung 19-3). Spannend ist auch METRICS, wo Sie die History sehen können, nämlich wie viele Builds erfolgreich und nicht erfolgreich durchlaufen sind.

Sie können auch die Weboberfläche von CruiseControl Ihren eigenen Projektanforderungen anpassen.

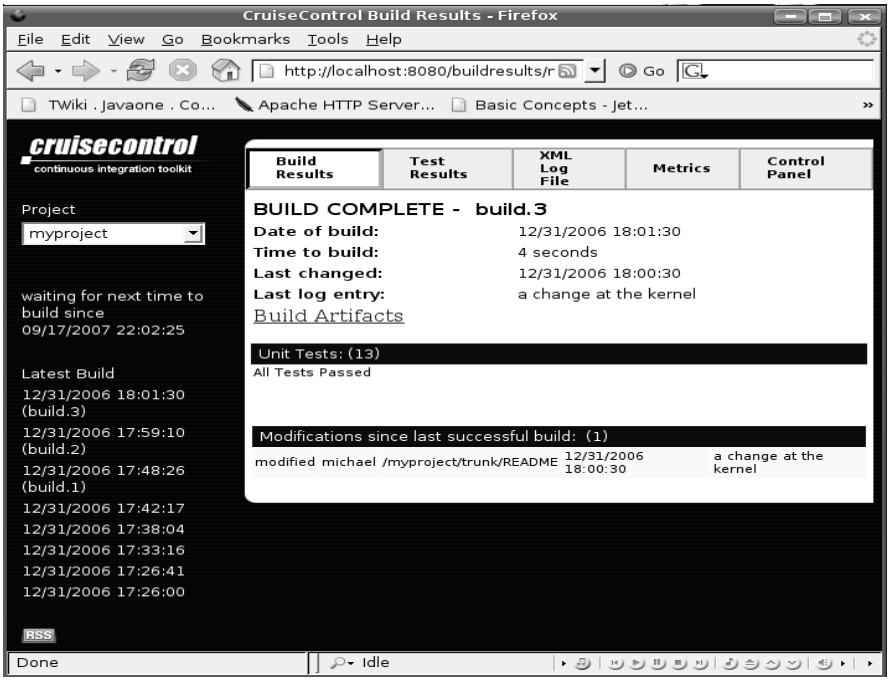


Abbildung 19-2: Die CruiseControl-Detailansicht

Sie können abschließend noch die Entwickler vom Ausgang des Builds in Kenntnis setzen. Hier gibt es verschiedene Möglichkeiten, wie das Senden einer Information an einen Jabber-Client, eine Mail oder eine Info an eine x10-fähige Einrichtung wie eine Lava-Lampe.

Zusammenfassung

CruiseControl unterstützt kontinuierliche Integrationen. Zu von Ihnen bestimmten Zeitpunkten werden neue Builds Ihrer Software automatisch angestoßen. Dabei greift CruiseControl auf Ihre bestehenden Build-Skripten zu. Ob Sie WebTest, JUnit oder andere Werkzeuge in Ihrem Build-Skript einsetzen, spielt für den Einsatz über CruiseControl keine Rolle. CruiseControl startet den Build unabhängig davon, wie er konkret aussieht. CruiseControl setzt eine Ebene oberhalb der Build-Skripten an.

Die Ergebnisse dieser Builds werden über eine zentrale Weboberfläche zugänglich gemacht. Testergebnisse sind direkt sichtbar. Ferner haben Sie die Möglichkeit, auf ältere Builds und auf die während des Builds erstellten Artefakte zuzugreifen.

Die zeitnahe Kommunikation des Stands der Software steht im Mittelpunkt: Werden beispielsweise nachts umfassende Integrationstests durchgeführt, ist das Team am nächsten Morgen direkt über ihren Erfolg oder Misserfolg informiert. Ist der Build fehlgeschlagen,

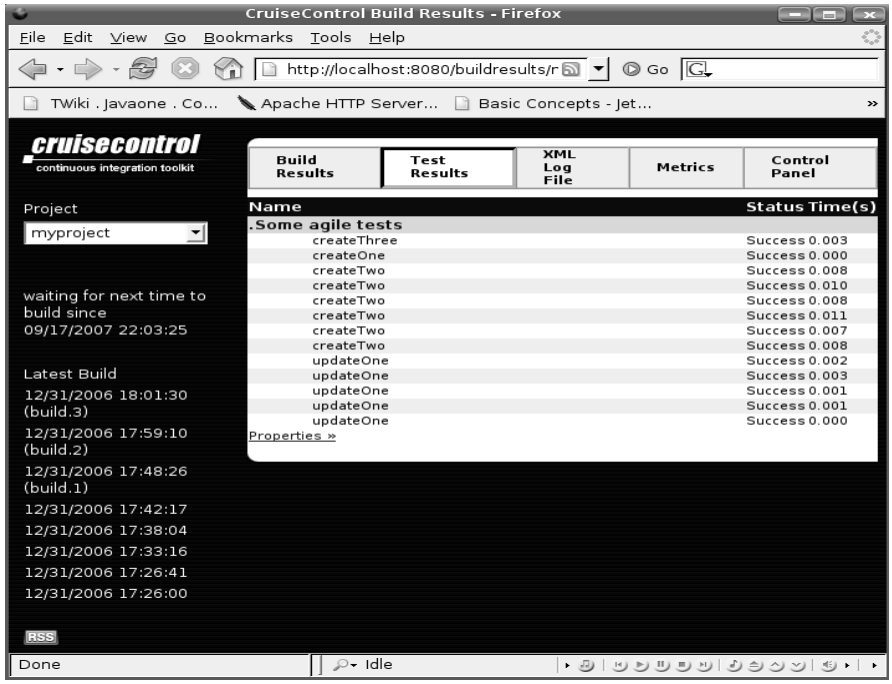


Abbildung 19-3: Übersicht Unit Tests

können am nächsten Tag die Entwickler direkt als Erstes die Probleme beseitigen. Lassen Sie CruiseControl in kurzen Intervallen neue Builds anstoßen, wenn der Build-Lauf nicht zu lange dauert. Informationen über die Builds, insbesondere auch die Benachrichtigung über fehlgeschlagene Builds, werden von CruiseControl direkt über verschiedene von Ihnen konfigurierte Kommunikationsstränge publiziert.

CruiseControl ist der De-facto-Standard unter den Build-Servern. Es wird komplett über XML konfiguriert (daneben gibt es noch eine JMX-Konsole, die über die CruiseControl-Weboberfläche nutzbar ist). Optik und Inhalt der Weboberfläche lassen sich durch Anpassung von XSL-Stylesheets ändern. Andere Build-Ergebnisse wie Code-Metriken lassen sich so in die Weboberfläche direkt einbinden. Durch seine mächtigen Konfigurationsmöglichkeiten ist CruiseControl flexibel einsetzbar, durch die jahrelange Nutzung in vielen Projekten erprobt und ausgereift. Falls Ihnen CruiseControl nicht zusagt, werfen Sie einen Blick auf andere Build-Server wie TeamCity aus dem Hause JetBrains. Dort können Sie alle Einstellungen über eine Weboberfläche durchführen. Das geht allerdings zu Lasten der Flexibilität.

KAPITEL 20

Der Einsatz von Maven für Build-Prozesse

Wir haben in den letzten Kapiteln ausführlich Ant behandelt, es mit anderen Werkzeugen integriert und auf diese Weise Build-Prozesse abgebildet. Diese Prozesse beinhalteten im Wesentlichen das Holen von Java-Artefakten aus dem Repository, die Kompilierung der fachlichen Klassen und Testklassen und die Ausführung von Tests. Der beschriebene Ansatz wird für die Mehrzahl von Projekten ausreichend und sinnvoll sein, da er eine hohe Flexibilität bietet. Doch gibt es Szenarien, wo dieses Vorgehen an seine Grenzen stößt. Haben Sie sehr große Ant-Skripten, wird es schwierig, den eigentlichen Prozess daraus abzulesen und auf einen Blick zu überschauen, was die Skripten genau tun. Das fällt umso mehr ins Gewicht, wenn es versäumt wird, die Skripten zu modularisieren, Teile wiederverwendbar zu machen und eine einheitliche Implementierung zu verfolgen.

Hier kommt Maven ins Spiel. Maven ist wie Ant ein Werkzeug für die Durchführung von Build-Prozessen und die Integration mit Werkzeugen und Funktionen. Maven vereinheitlicht Build-Prozesse und Implementierungsaufwand, hilft also auch dabei, die Kosten zu minimieren. Anders als Ant setzt Maven auf einen *deklarativen* Ansatz, bei dem der Nutzer definiert, *was* er machen möchte. Ohne weiteres Zutun deckt Maven bereits einen einfachen Standard-Build-Prozess ab, der deklarativ erweitert oder umgeschrieben werden kann. Bei Ant dagegen muss der Nutzer Skripten schreiben, *wie* er etwas erreichen will (er geht dort also *imperativ* vor). Er muss den kompletten Build-Prozess implementieren und kann nicht auf einen vorgefertigten mitgelieferten Build-Prozess aufsetzen.

Ein weiterer großer Unterschied ist, dass Maven von Haus aus eine Verwaltung von Abhängigkeiten zwischen im Softwareprojekt genutzten Artefakten ermöglicht.

Maven ist genau wie Ant ein Top-Level-Projekt von Apache. Es entstand 2004 im Rahmen des Apache-Projekts Jakarta-Alexandria, eines mittlerweile eingestellten Projekts, das sich zum Ziel gesetzt hatte, ein einheitliches Quellcode- und Dokumentenmanagement-System auf Basis von Java und XML zu etablieren.

Obwohl Maven noch recht jung ist, blickt das Projekt auf eine abwechslungsreiche Geschichte zurück. Böse Zungen behaupten, Maven 1.0 sei ein schwacher, architektonisch indiskutabler Schnellschuss gewesen. Darum steht Maven seit Oktober 2005 komplett überarbeitet in der Version 2.0 bereit. Das hat zur Folge, dass auf Maven 1 auf-

bauende Projekte mit teils hohen Migrationsaufwänden zu kämpfen haben. Andererseits ist Maven 2 nun so ausgereift, dass per se nichts mehr gegen seinen Einsatz spricht und es zukünftig keine großen Kompatibilitätsprobleme bei Upgrades geben wird: Es ist zukunftssicher geworden.

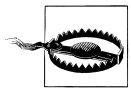
Funktionsumfang

Wir haben Maven bisher als Werkzeug zur automatischen Unterstützung und Ausführung des Build-Prozesses betrachtet. Das ist zwar richtig und tatsächlich der Haupteinsatzbereich, aber nur ein Drittel der Wahrheit. Neben der Durchführung des Build-Prozesses übernimmt Maven noch die Verwaltung der Abhängigkeiten zwischen genutzten Bibliotheken und bietet einen impliziten Vorgang zur Erstellung von Projektdokumentation.

Build-Prozess

Der mitgelieferte Build-Prozess besteht aus standardisierten Einzelphasen, z.B. dem Holen von Artefakten aus dem Repository, ihrem Kompilieren und dem Ausführen von Tests. In den mitgelieferten Standardprozess sind viele in der Praxis bewährte Ansätze eingeflossen, die direkt unterstützt werden. Während bei der Nutzung von Ant für diesen standardisierten Prozess innerhalb eines Projekts erneut Skripten geschrieben werden müssen, kann der Default-Prozess mit Maven out-of-the-Box angewendet werden. Der Default-Prozess wird für viele Projekte direkt in der mitgelieferten Form anwendbar sein.

Starten Sie ein Projekt und haben noch keinen etablierten Build-Prozess, können Sie den Standardprozess von Maven nutzen. Es ist flexibel konfigurierbar. Entspricht der standardmäßig abgedeckte Build-Prozess nicht dem eigenen, in dem Projekt tatsächlich angewendeten, kann Maven entsprechend angepasst werden.



Ich habe auch schon Projekte erlebt, bei denen die eigenen Prozesse dem Werkzeug angepasst wurden. Das sollten Sie aber vermeiden. Häufig ist das auch gar keine Option, da gerade in kommerziellen Projekten und Unternehmen wohldurchdachte Prozessstandards zum Tragen kommen. Passen Sie das Werkzeug Ihrem Build-Prozess an!

Der mitgelieferte Build-Prozess (die einheitliche Projektstruktur) vereinfacht das Vorgehen für die meisten Fälle sehr.

Abhängigkeiten

Maven ist darüber hinaus auch eine Plattform, mit deren Hilfe Sie die Abhängigkeiten Ihrer Software dokumentieren und sichtbar machen können. Obwohl jede Software seit Urzeiten aus unterschiedlichen voneinander abhängigen Komponenten und Bibliotheken besteht, hat es doch bis Maven gebraucht, bis eine akzeptable, standardisierte Lösung für die Verwaltung dieser Abhängigkeiten vorlag.

Mit *Ivy* steht ein Apache-Projekt zur Verfügung, das sich ebenfalls die Verwaltung von Abhängigkeiten zum Ziel gesetzt hat (siehe *http://incubator.apache.org/ivy/)*. Es lässt sich mit Ant integrieren.

Häufig ist es so, dass in Projekten unklar ist, welche Bibliotheken mit welchen Versionen genutzt werden. Mit Maven können Sie die eingesetzten Bibliotheken (wie beispielsweise JUnit) und ihre Versionen (wie JUnit 4.1) und deren Abhängigkeiten im Build-Prozess dokumentieren, und zwar nicht nur für das System, sondern auch für Sie selbst.

In Maven werden die benötigten externen Bibliotheken im Projektmodell (POM) festgelegt. Daneben werden die externen Bibliotheken im eigenen Maven-Repository zentral hinterlegt und koordiniert. Das Maven-Repository ist eine Verzeichnisstruktur mit weiteren Informationen für die Identifikation der Komponenten in den jeweiligen Versionen in XML. Die einzelnen Versionen der abhängigen Komponenten werden durch ihren Dateinamen unterschieden. Wird der Build nun ausgeführt, lädt Maven die notwendigen Bibliotheken in den geforderten Versionen aus seinem Repository.

Dokumentation und Kommunikation

Wie Sie wissen, ist in agilen Projekten die Kommunikation sehr wichtig. Das betrifft hier die deklarative Kommunikation darüber, wie der Build-Prozess aussieht, so dass jeder einzelne Entwickler durch einen Blick auf das Projektmodell den Vorgang direkt versteht. Eine weitere agile Eigenschaft von Maven ist die Fähigkeit, eine Projektdokumentation zu generieren. Diese Dokumentation wird in Form einer Projektseite erstellt. Also bietet auch Maven hier einen standardisierten modellbasierten Ansatz, eine angenehme Alternative zu den manuellen Schritten, die bei Ant notwendig wären.

Architektur

Maven ist eine Java-Anwendung und besteht aus einem kompakten Kern und einer beliebigen Anzahl von Plugins. Installieren Sie Maven, wird ausschließlich der Kern eingerichtet. Dieser kann zwar ein Projektmodell interpretieren, besitzt aber noch keine Funktionalität, um den Build-Prozess tatsächlich auszuführen. Notwendige Build-Schritte sind bei Maven die Build-*Phasen*. Diese Phasen zusammengenommen ergeben den Build-Prozess. Die Phasen werden von einzelnen Build-Zielen ausgeführt, den *Goals*, für die es jeweils Plugins gibt.

Da diese Plugins nicht gleich am Anfang auf Ihrem Rechner abgelegt werden, müssen die eingesetzten Plugins vor dem ersten Build automatisch aus dem Internet heruntergeladen werden. Der Maven-Kern sucht dazu zunächst die notwendigen Plugins im lokalen Repository, das unter Linux unter *.m2\repository* im Home-Verzeichnis angelegt wird. Ist das Plugin dort nicht verfügbar, zum Beispiel bei der ersten Ausführung, greift Maven auf das zentrale Plugin-Repository zu, das sich unter *http://repo1.maven.org/maven2* befindet. Nach dem Herunterladen steht das Plugin für jede weitere Nutzung direkt lokal zur Verfügung.

Die Verwaltung über Repositories betrifft aber nicht nur die Plugins. Auch die externen Bibliotheken werden durch dasselbe Vorgehen eingebunden und zunächst im Remote Repository gesucht. Die URL ist standardmäßig dieselbe wie bei den Plugins, kann aber auch konfiguriert werden.

Am Ende des Build-Prozesses entsteht eine Auslieferungsdatei, die Maven im lokalen oder im Remote Repository bereitstellen kann. Hier wird direkt das Potenzial klar: Arbeiten Sie mit vielen Projekten, erstellen Sie pro Projekt ein Projektmodell. Die Auslieferungsdateien stellen Sie lokal zur Verfügung. Die Auslieferungsdatei kann dann für andere (interne) Projekte eine externe Bibliothek darstellen, die Maven automatisch einbindet. Auch eine Hierarchisierung ist möglich: Ein Projektmodell greift auf andere Projektmodelle zu und aggregiert dessen Ergebnisse. So kann ein ganzes Produkt erstellt werden.

Prinzipien

Maven zeichnet sich durch vier Prinzipien aus.

Konvention ist wichtiger als Konfiguration

Konventionen über Konfigurationen zu stellen ist ein weiter Trend, der auch in die Java-Entwicklung Einzug gehalten hat. Üppige XML-Konfigurationen werden bei leichtgewichtigen Lösungen wie EJB ab Version 3 oder Spring durch mehr Konventionen ersetzt. Auch Ruby bzw. Ruby On Rails macht sich diesem zunutze.

Bei Maven gibt es folgende Konventionen:

1. Eine Standard-Verzeichnisstruktur: Sie macht einen schnellen Einstieg möglich. Jeder in einem agilen Team findet sich direkt zurecht. Man weicht nur vom Standard ab, wenn es die vorliegenden Projektprozesse notwendig machen. Die Bestandteile aller Projekt-Sourcen wie Quelltext, Konfigurationsdateien, generierter Output und Dokumentation liegen standardmäßig an einem festen, bekannten Ort.
2. Konzept eines zentralen Maven-Projekts und eines einzigen daraus erstellten Artefakts: Wiederverwendung durch zentrale Bereitstellung von kohärenten Bausteinen. Ein umfangreiches Projekt wird in einzelne Komponenten zerlegt. Diese Modularisierung macht die Abhängigkeiten transparenter und fördert das Aufteilen des Gesamtsystems in logische Schichten (Separation of Concerns).
3. Standardisierte Namenskonventionen: Allein beim Anblick der Artefakte wissen wir bereits, um was es sich handelt. Als Beispiel kann das Artefakt *commons-logging-1.2.jar* dienen: Hier ist direkt ersichtlich, dass es sich um eine *.jar*-Datei in der Version 1.2 von *commons-logging* handelt.

Deklarative Ausführung

Bei Maven wird nicht wie bei Ant implementiert, was das Werkzeug tun soll (*imperativ*), es wird *deklarativ* beschrieben, wie der Build-Prozess aussieht. Dazu werden in einem Maven-Projektmodell die Metadaten des Projekts zentral dokumentiert. Das Projektmodell wird als *POM* (Project Object Model) bezeichnet (siehe unten) und ist eine XML-Repräsentation eines Maven-Projekts, das in einer Datei namens *pom.xml* gehalten wird.

Aufbauend auf den dort vorgenommenen Einstellungen bzgl. Projektstruktur und notwendigen Bibliotheken übernimmt Maven dann den Build-Prozess. Das hat einerseits den Charme, dass der Build-Prozess nur noch an einer zentralen Stelle deklariert wird. Dieser modellbasierte Ansatz hat andererseits den Nachteil, dass es, wenn das Projekt in der Realität von den im Modell definierten Annahmen abweicht, zu Schwierigkeiten kommt. Von Haus aus bietet Maven einen standardmäßigen Build-Prozess an. Ist der eigene Build-Prozess mit ihm kongruent, ist alles in Ordnung. Je mehr er allerdings abweicht, desto mehr Schwierigkeiten hat man. Wo Ant flexibel ist und recht gesprächige Skripten erfordert, ist Maven ein verschwiegener, starrer Gehilfe. Aber keine Angst: Wir können Maven und Ant auch kombinieren bzw. den Projektablauf konfigurieren.

Die deklarative Art von Maven lässt sich als modellbasierter Ansatz bezeichnen. Sie bauen sich ein Modell Ihres Build-Prozesses mit von Maven bereitgestellten XML-Konstruktionen.

Wiederverwendung der Build-Logik

Der mitgelieferte Build-Prozess ist für eine Vielzahl von Projekten ausreichend. Wo er nicht ausreicht oder vom konkreten Prozess abweicht, kann er angepasst werden. Das geschieht an zentraler Stelle im POM und kann auch für andere Projekte genutzt werden. POMs lassen sich ebenfalls schachteln. Ein Build-Prozess kann abhängig von einem anderen Build-Prozess sein oder diesen enthalten.

Kohärente Organisation der Abhängigkeiten

Die in den Build-Prozess eingebundenen Artefakte liegen in ihren unterschiedlichen Versionen an zentraler Stelle im Maven-Repository. Das im Projekt vorliegende *lokale Repository* bezieht seine Artefakte (wie JUnit) aus einem entfernten zentralen Maven-Repository im Internet, dem *Remote Repository*. In das POM eingebundene Artefakte werden im entfernten Repository gesucht und ins lokale Repository heruntergeladen, um sie dann im weiteren Verlauf stetig von dort zu beziehen.

Unterschiedliche Projekte können auf dieses Repository zugreifen und diese Bibliotheken gemeinschaftlich in den jeweiligen Build-Prozess einbinden. Die individuellen Build-Prozesse halten also nicht separate Versionen der Artefakte, sondern referenzieren jeweils die zentral vorliegenden.

Unser Beispiel enthält eine Projektabhängigkeit zu JUnit. Damit Maven das JUnit-Artefakt finden kann, muss eine Referenz im POM angelegt werden. Dazu gehört die Angabe, in welchem *Repository* das Artefakt zu finden ist, sowie die eindeutige Artefaktkennzeichnung. Diese eindeutige Kennzeichnung besteht aus einer Kombination aus *groupId*, *artifactId* und *version*. Wir definieren also nicht: »Wir brauchen *Junit-3.8.1.jar*«, sondern definieren: »Unser Projekt hat eine Abhängigkeit vom Artefakt »junit« (artifactId) vom Anbieter »junit« (groupId) in der Version 3.8.1 (version).«

Die ersten Schritte

Wir möchten nun Maven 2 installieren und ein neues Projekt anlegen.

Laden Sie sich die aktuelle Version von Maven 2 von *http://maven.apache.org/* herunter und entpacken Sie die Distribution auf Ihren Rechner an einen Ort Ihrer Wahl. Fügen Sie Ihrem Pfad in Ihrem Profil das Home-Verzeichnis und das dort befindliche *bin*-Verzeichnis hinzu.

Nun geben Sie an dem Ort, wo sie das Maven Projekt anlegen wollen, ein:

```
michael@BlueGene:~/maven$ mvn archetype:create -DgroupId=com.huettermann.agile -
DartifactId=agile
```

Damit wird nicht nur das Projekt *agile* angelegt, sondern auch die notwendigen Artefakte in das lokale Repository unter *~/.M2* gelegt. Außerdem legt dieser Befehl das Verzeichnis *agile* an, das das Projektmodell als *pom.xml* und ein Source-Verzeichnis mit einer kleinen Testklasse inklusive eines korrespondierenden Unit-Tests enthält. Dazu haben wir die Funktionalität des im Maven-Plugin *archetype* hinterlegten Aktionsblocks `create` genutzt. Ein Plugin ist eine Ansammlung von Aktionsblöcken, die in einem fachlichen Zusammenhang separate Aufgaben erfüllen und sich mit den Ant-Tasks vergleichen lassen. Die Aktionsblöcke, denen zur eindeutigen Identifizierung der Name des Plugins vorangestellt wird, heißen *Goals*.

Mit anderen Worten: Der von uns genutzte Befehl hat das Maven-Goal `archetype:create` ausgeführt und diverse Parameter übergeben. Durch die Erstellung des Projekts wird ein POM angelegt.

Das POM

Das angelegte POM ist das Projektmodell, das die Metadaten des Projektes enthält. Aus diesen Informationen erstellt Maven dann den Build-Prozess. Unser im vorigen Schritt bei der Erstellung des Projekts angelegtes POM sieht so aus:

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XML
Schema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.huettermann.agile</groupId>
```

```
  <artifactId>agile</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>agile</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Neben dem zentralen *pom.xml* unterstützt Maven eine benutzerspezifische Konfigurationsdatei namens *settings.xml*. Sie liegt im jeweiligen Benutzerverzeichnis unter *.m2/settings.xml*. Während in *pom.xml* übergreifende projektweite Einstellungen vorgenommen werden, enthalten die jeweiligen *settings.xml*-Dateien der einzelnen Nutzer benutzerspezifische Einstellungen.

pom.xml beschreibt das Projekt *agile* bzw. das unter diesem Schlüssel bekannte Artefakt, das eigenständig ausgeliefert werden kann. Die konzeptionelle Trennung von Artefakt und Projekt wird dann interessant, wenn Sie in einem Projekt mehrere Artefakte unabhängig voneinander erstellen wollen. Alle Artefakte haben ein eigenes *pom.xml*, wobei dann ein separates *pom.xml* die einzelnen Artefakte zusammenfasst und alle Artefakte als ein Produkt bereitstellt. Dieses Vorgehen ist gerade im Bereich JEE interessant.

pom.xml enthält folgende Informationen:

modelVersion
: Die benutzte Version.

groupId
: Die eindeutige Kennzeichnung des Anbieters dieses Artefakts. Wir haben diesen Schlüssel bei der Erstellung angegeben.

artifactId
: Die eindeutige Kennzeichnung dieses mit *pom.xml* beschriebenen Artefakts. Dieser Schlüssel wird bei der Benennung des Artefakts genutzt.

packaging
: Die Form, wie das Artefakt gepackt wird. Hier sind viele verschiedene Typen denkbar, die den Build-Prozess direkt beeinflussen.

version
: Die Version des Artefakts. Sie wird standardmäßig automatisch in den generierten Artefaktnamen aufgenommen. Snapshot ist dabei ein Zwischenstand, der während der täglichen Weiterentwicklung entsteht.

name
Der Name des Projekts.

url
Eine URL des Projekts.

description
Eine Beschreibung des Projekts, erscheint beispielsweise auf der Projektseite.

Anhand der Koordinaten finden Sie auch die Plugins. Wenn Sie zu einem Werkzeug ein Plugin suchen, gehen Sie auf *http://repo1.maven.org/maven2/*. Das ist eine Anlaufstelle, die zentral und für Maven aufbereitet und verarbeitbar die gängigen Bibliotheken anbietet. Suchen Sie dort den Ordner, der das Werkzeug enthält. Innerhalb des Ordners finden Sie dann die verschiedenen Versionen, und innerhalb einer Version eine *pom.xml*:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.1</version>
  <url>http://junit.org</url>
  <name>JUnit</name>
  <description>
    JUnit is a regression testing framework written by Erich Gamma and Kent Beck. It is
used by the developer who implements unit tests in Java.
  </description>

  <licenses>
    <license>
      <name>Common Public License Version 1.0</name>
      <url>http://www.opensource.org/licenses/cpl1.0.txt</url>
    </license>
  </licenses>
  <dependencies>
  </dependencies>
</project>
```

Wenn ein Plugin oder eine Bibliothek lokal nicht verfügbar ist, werden sie in das lokale Repository heruntergeladen. Im weiteren Verlauf greifen dann alle auf das lokale Repository zu, so dass es schnell zum Nadelöhr werden kann. Maven bietet hier Abhilfe durch das Konzept der Spiegel-Repositories. Sie können diese Mirrors einrichten, um die Last zu verteilen. Details entnehmen Sie bitte der Dokumentation.

Die Verarbeitung von Maven, also das Herunterladen der Artefakte in das lokale Repository und die weitere Verarbeitung, geschieht automatisch. Sie müssen also nicht mehr zu den Webseiten der einzelnen Anbieter gehen und dort lange nach dem Downloadbereich und der richtigen Version suchen, diese herunterladen und an einen von Ihnen zu bestimmenden Ort platzieren. Abbildung 20-1 illustriert das für JUnit.

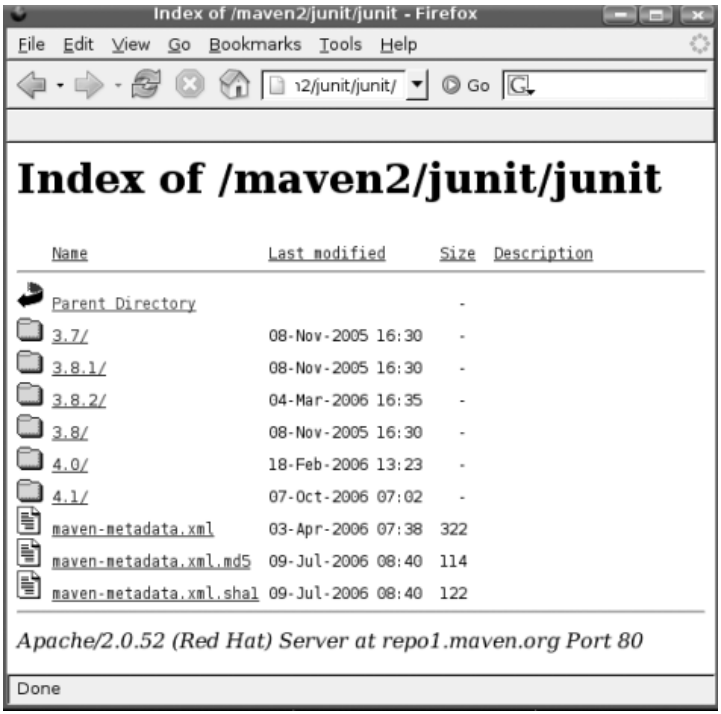


Abbildung 20-1: Die JUnit-Artefakte in unterschiedlichen Versionen im entfernten Repository

Phase und Goal

Nachdem wir *pom.xml* der JUnit 4.1-Version vorliegen haben, nutzen wir Maven, um aus den hinterlegten Klassen eine *.jar*-Datei zu erstellen. Dazu nutzen wir den Befehl

```
mvn package
```

der nicht über einen Plugin-Teil verfügt. Mussten wir beim Einrichten des Projekts dem Goal (`create`) noch einen Plugin-Namen (`archetype`) vorwegstellen, genügt an dieser Stelle ein Wort. Hier beinhaltet der Maven-Befehl kein Goal, sondern eine *Phase*. Eine Phase ist ein Schritt im Build-Prozess. Intern werden diese Phasen zu Goals gemappt. Daraus ergeben sich verschiedene Implikationen. Beispielswiese ruft die Phase *package* das Goal *jar:jar* auf, aber nur, da wir als Typ des Artefakts *.jar* angegeben haben. Würden wir das Webarchiv *war* erstellen wollen, wäre das aufgerufene Goal *war:war*.

Es werden erneut notwendige Plugins ins lokale Repository heruntergeladen und die *.jar*-Datei angelegt. Die Konsolenausgabe endet mit

```
[INFO] [jar:jar]
[INFO] Building jar: /home/michael/maven/agile/target/agile-1.0-SNAPSHOT.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESSFUL
```

```
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 39 seconds
[INFO] Finished at: Tue Jan 02 13:22:42 CET 2007
[INFO] Final Memory: 4M/12M
[INFO] ------------------------------------------------------------------------
michael@BlueGene:~/maven/agile$
```

und resultiert in folgender Verzeichnisstruktur:

```
michael@BlueGene:~/maven/agile/target$ ls -la
insgesamt 24
drwxr-xr-x 5 michael michael 4096 2007-01-02 13:22 .
drwxr-xr-x 4 michael michael 4096 2007-01-02 13:23 ..
-rw-r--r-- 1 michael michael 2341 2007-01-02 13:22 agile-1.0-SNAPSHOT.jar
drwxr-xr-x 3 michael michael 4096 2007-01-02 13:22 classes
drwxr-xr-x 2 michael michael 4096 2007-01-02 13:24 surefire-reports
drwxr-xr-x 3 michael michael 4096 2007-01-02 13:22 test-classes
```

Wenn Sie die Konsolenausgabe aktiv verfolgt haben und sich darüber hinaus das erstellte Verzeichnis genau anschauen, fällt Ihnen sicherlich direkt auf, dass auch die JUnit-Tests ausgeführt wurden.Den Grund dafür sehen wir uns nun an.

Lifecyles

Der Build-Prozess wird von Maven als *Build*-Lebenszyklus (*Lifecycle*) bezeichnet. Unter dem Begriff »Lifecycle« fasst Maven verschiedene Phasen in einer bestimmten Reihenfolge zusammen. Springt man eine Maven-Phase innerhalb des Build-Lifecycle an, werden bis zur übergebenen alle Phasen ausgeführt, die im Build-Prozess vor dieser angesiedelt sind. Der Lifecycle ist in Tabelle 20-1 abgebildet:

Tabelle 20-1: Der Build-Lifecycle in Maven

Phase	Bedeutung
`validate`	Prüft auf Vorbedingungen
`compile`	Kompiliert den Quelltext
`test`	Führt die Unit-Tests aus
`package`	Generiert die Auslieferungsdatei (das Artefakt)
`integration-test`	Installiert die Auslieferungsdatei in der Testumgebung und führt Integrationstests aus
`verify`	Prüft das Artefakt auf Vollständigkeit und Korrektheit
`install`	Überträgt das Artefakt in das lokale Maven-Repository
`deploy`	Überträgt das Artefakt in ein Remote-Repository.

Der Build ergibt sich also aus der Aneinanderreihung der einzelnen Phasen von oben nach unten.

Neben dem eben vorgestellten standardmäßigen Build-Lifecycle kennt Maven noch zwei weitere: den `clean`-Lifecycle und den `site`-Lifecycle:

clean
: Löscht in einem früheren Build erstellte Artefakte.

site
: Generiert ausgehend vom POM eine Seitendokumentation.

Wir haben mit der Paketierung also auch das Kompilieren und Testen der Anwendung angestoßen. Der Quellcode der Anwendungen liegt in *src/main/java*, die Klassen und das *.jar*-File in *target/classes*, und die Testklassen in *src/test/java*.

Führen wir

```
mvn clean
```

aus, um den Ausgangszustand zu erhalten, und dann

```
mvn install
```

(springen also die Phase `install` des Build-Lifecycle an), werden alle Schritte im Build-Lifecycle bis inklusive `install` ausgeführt. Wenn wir das nicht das erste Mal machen, werden auch keine neuen Plugins heruntergeladen.

Agile Entwicklung mit Maven

Maven selbst unterstützt durch seine Architektur vorbildlich agile Entwicklung. Die bereits angeführten Eigenschaften bzw. Vorteile wie hohe Wiederverwendbarkeit und der modulare Aufbau sowie die Pflege der Abhängigkeiten reduzieren dauerhaft die täglich verschwendete Zeit. Das Tool, ob Ant oder Maven, ist unabhängig vom Build-Prozess. Sie können mit beiden Werkzeugen Ihren individuellen Prozess aufsetzen und genau die Werkzeuge einbinden, die ich vorgestellt habe, und sie so kombinieren, wie es für Sie den größten Nutzen bringt. Auch hier sollte man nicht dogmatisch sein.

Die Beschreibung von Maven in diesem Kapitel wird nun abgeschlossen mit einem Beispiel, in dem wir Klassen kompilieren, Tests laufen lassen und Code mit Checkstyle überprüfen. Um Tests mit JUnit 4.1 durchführen (hier möchten wir Testklassen beliebig benennen dürfen) und Checkstyle nutzen zu können, müssen Einstellungen am POM durchgeführt werden. Wir weichen somit von der Basiskonfiguration, vom Standard-Build-Prozess ab. Darüber hinaus wird eine Projektseite erstellt, auf der wir die Test- und Checkstyle-Ergebnisse sehen können. Sie gewinnen einen Überblick, wie Sie das POM konfigurieren und externe Plugins einbinden und ihr Verhalten steuern können.

Die Klassen

Wie schauen uns die Klassen an. Zunächst haben wir unter

```
maven/agile/src/main/java/com/huettermann/agile
```

die Datei

```
Agile.java
```

hinterlegt. Sie sieht folgendermaßen aus:

```
package com.huettermann.agile;

/**
 * @author Michael Huettermann
 **/
public class Agile {
   private final String name;

   /**
    * the name
    * @param name myName
    **/
   public Agile(String name) {
      this.name = name;
   }

   /**
    * get the name
    * @return name the name
    **/
   public String getName() {
      return name;
   }

   @Override
   public boolean equals(Object o) {
      if (this == o) return true;
      if (o == null || getClass() != o.getClass()) return false;
      Agile agile = (Agile) o;
      if (!name.equals(agile.name)) return false;
      return true;
   }

   @Override
   public int hashCode() {
      return name.hashCode();
   }
}
```

Sie ist eine einfache exemplarische Klasse mit einem Konstruktor, einem Getter und einer `equals`- und einer `hashCode`-Methode. Diese Klasse ist uns nicht ganz unbekannt, haben wir sie doch schon in Komponententests mit JUnit genutzt. Nun haben wir noch zwei Testklassen unter

```
maven/test/java/com/huettermann/agile
```

hinterlegt. `AgileTest` ist eine JUnit 4.x-konforme Testklasse mit zwei Testmethoden (siehe Beispiel 20-1), `AgileTestLegacy` eine JUnit 3.x-konforme Testklasse mit ebenfalls zwei Testmethoden (siehe Beispiel 20-2).

Beispiel 20-1: Eine JUnit 4-konforme Testklasse

```
package com.huettermann.agile;

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

/**
 * @author Michael Huettermann
 **/
public class AgileTest {

    @Test
    public void agileName() {
        Agile agile = new Agile("XP");
        assertEquals("XP", agile.getName());
    }

    @Test
    public void agileEquals() {
        Agile agile1 = new Agile("XP");
        Agile agile2 = new Agile("XP");
        assertTrue(agile1.equals(agile2));
    }
}
```

Die Beispielklassen sind bewusst einfach gehalten. Der Fokus liegt hier auf Maven, nicht auf den Testklassen. Wie komplex die Klassen sind, ist für die Funktionsweise von Maven unerheblich.

Beispiel 20-2: Eine JUnit 3-konforme Testklasse

```
package com.huettermann.agile;

import junit.framework.TestCase;

/**
 * @author Michael Huettermann
 **/
public class AgileTestLegacy extends TestCase {

    public void testAgileName() {
        Agile agile = new Agile("XP");
        assertEquals("XP", agile.getName());
    }

    public void testAgileEquals() {
        Agile agile1 = new Agile("XP");
        Agile agile2 = new Agile("XP");
        assertTrue(agile1.equals(agile2));
    }
}
```

Das POM

Das in Beispiel 20-3 abgebildete Projektmodell definiert die für die Verarbeitung notwendigen Schritte.

Beispiel 20-3: Das dem Beispiel zugrundeliegende POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.huettermann.agile</groupId>
  <artifactId>agile</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>agile</name>
  <url>http://huettermann.net</url>
  <organization>
    <name>Michael Huettermann</name>
    <url>http://huettermann.net</url>
  </organization>
  <description>Agile Development with Java</description>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <includes>
            <include>**/*.class</include>
          </includes>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>surefire-report-maven-plugin</artifactId>
        <configuration>
          <includes>**/*</includes>
          <reportsDirectory>target/surefire-reports</reportsDirectory>
          <testFailureIgnore>true</testFailureIgnore>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>net.unto.maven.plugins</groupId>
        <artifactId>maven-junit4-plugin</artifactId>
```

Beispiel 20-3: Das dem Beispiel zugrundeliegende POM (Fortsetzung)

```
        <version>1.0-SNAPSHOT</version>
        <executions>
          <execution>
            <phase>test</phase>
            <goals>
              <goal>test</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
  <reporting>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-checkstyle-plugin</artifactId>
        <configuration>
          <configLocation>config/sun_checks.xml</configLocation>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>surefire-report-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </reporting>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <pluginRepositories>
    <pluginRepository>
      <id>unto.net</id>
      <url>http://repository.unto.net/maven/</url>
      <releases>
        <updatePolicy>daily</updatePolicy>
        <checksumPolicy>warn</checksumPolicy>
      </releases>
    </pluginRepository>
  </pluginRepositories>
</project>
```

Zunächst fällt auf, dass wir den Build-Prozess ändern und erweitern. Dazu haben wir ein *build*-Element hinzugefügt, dessen *plugin*-Elemente jeweils einen Eingriff in den Build-Prozess vornehmen.

Die Plugins

Die Plugin-Einträge enthalten eine `groupId` und eine `artifactId`, um das Maven-Plugin zu bestimmen, das wir konfigurieren wollen. Hier wird `maven-surefire-plugin` adressiert. Es ist für die JUnit-Tests zuständig.

Derzeit ist der Standard, dass die Unit-Testklassen mit »`Test`« enden müssen, um als Testklasse identifiziert und verarbeitet zu werden. Um diese Restriktion zu beheben, ändern wir die Standardkonfiguration des Plugins und fügen ein `configuration`-Element hinzu. Hier möchten wir die Klassen bestimmen, die einem JUnit-Test unterzogen werden sollen. In objektorientierter Sprache gesprochen: Wir überschreiben Funktionen des Plugins. Wir führen ein `include`-Element ein, das sämtliche Klassen (im Testverzeichnis) als Testklassen ansieht.

```
<build>
   ...
     <plugin>
       <groupId>org.apache.maven.plugins</groupId>
       <artifactId>maven-surefire-plugin</artifactId>
       <configuration>
         <includes>
           <include>**/*.class</include>
         </includes>
       </configuration>
     </plugin>
   ...
</build>
```

Wir möchten definieren, wo Maven die Testergebnisse platziert. Dazu nutzen wir das `reportsDirectory`-Element. Wir überschreiben außerdem die Einstellung `testFailureIgnore`. Als Standard bricht der ganze Vorgang bei einem fehlerhaften Test ab, was durch diese Einstellung verhindert wird.

```
<build>
   ...
     <plugin>
       <groupId>org.codehaus.mojo</groupId>
       <artifactId>surefire-report-maven-plugin</artifactId>
       <configuration>
         <includes>**/*</includes>
         <reportsDirectory>target/surefire-reports</reportsDirectory>
         <testFailureIgnore>true</testFailureIgnore>
       </configuration>
     </plugin>
   ...
</build>
```

Maven lädt automatisch alle benötigten Plugins und referenzierten Bibliotheken herunter. Darüber hinaus lädt es auch automatisch diejenigen Bibliotheken in das lokale Repository, die von diesen referenzierten Bibliotheken benötigt werden. Das sind die transitiven Abhängigkeiten. Voraussetzung ist, dass diese Bibliotheken mit der Projektbeschreibung im

entfernten Repository zu finden sind. Diese automatische Arbeit, die Maven für uns übernimmt, ist ausgesprochen hilfreich, da die manuelle Suche und Beschaffung der – gerade bei größeren Projekten – unzähligen Komponenten doch sehr mühselig und fehleranfällig ist. Haben Sie die transitiven Abhängigkeiten im Hinterkopf. Treffen Sie auf einen Fehler bei der Erzeugung Ihres Produkts, kann es der Fall sein, dass unterschiedliche Versionen von fremden Bibliotheken gleichzeitig genutzt werden. Sie können in solchen Fällen manuell eingreifen.

Nun haben wir aber JUnit 4-Testklassen, die wir von Maven ausführen lassen möchten. Wir nutzen auch in den fachlichen Klassen Java 5-Sprachmerkmale. Standardmäßig kompiliert Maven die Klassen (fachliche Klassen und Testklassen) Java 1.3-konform. Wir sprechen den Java-Compiler von Maven direkt an, um im Plugin `maven-compiler-plugin` die Kompilierung auf 1.5 zu ändern. Wir stellen `source`- und `target`-Level auf 1.5. Der POM-Eintrag dazu sieht so aus:

```
<build>
  ...
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0</version>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
  ...
</build>
```

Zum aktuellen Zeitpunkt ist das `surefire`-Plugin von JUnit immer noch im Migrationsstadium und unterstützt keine JUnit 4-Testklassen. Wir müssen also Maven mitteilen, dass wir JUnit 4-Testklassen ausführen wollen. Dazu nutzen wir das `maven-junit4-plugin`, das die Testphase modifiziert. Es ist ein Plugin, das es ermöglicht, JUnit 4-Testklassen zu verarbeiten. Dieses Plugin, das nun auch in der Testphase durch Aufruf des `test`-Goals ausgeführt wird (wir haben die Verarbeitung in die Testphase eingehängt und dem Goal `test` zugeordnet), unterstützt eine einfache Verarbeitung der Testklassen. Das Plugin hat allerdings die Einschränkung, dass es die standardmäßige Verzeichnisstruktur, also den Ort der Quellen und Testklassen, voraussetzt und keine Reporting-Funktionalität anbietet. Es gibt die Testergebnisse auf der Konsole aus. Der relevante Abschnitt im POM sieht folgendermaßen aus:

```
<build>
  ...
    <plugin>
      <groupId>net.unto.maven.plugins</groupId>
      <artifactId>maven-junit4-plugin</artifactId>
      <version>1.0-SNAPSHOT</version>
      <executions>
        <execution>
          <phase>test</phase>
          <goals>
```

```
            <goal>test</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  ...
</build>
```

Wir möchten nun Checkstyle und sein Reporting konfigurieren. Wir möchten definieren, dass wir unsere fachliche Klasse mit Checkstyle einer Qualitätskontrolle unterziehen wollen. Innerhalb der Konfiguration geben wir die Richtlinie an, mit deren Hilfe Checkstyle prüfen soll. Wir definieren hier die Sun Coding Conventions, die mit dem durchgeführten Eintrag genutzt werden.

```
<reporting>
  ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <configuration>
        <configLocation>config/sun_checks.xml</configLocation>
      </configuration>
    </plugin>
  ...
</reporting>
```

Neben den Sun Coding Conventions stehen noch andere vorbereitete Konventionen zur Verfügung, die auch durch eigene ergänzt werden können und an dieser Stelle referenziert werden.

Nun möchten wir noch einstellen, dass die JUnit-Testergebnisse mit in das Reporting einbezogen werden sollen. Dazu wird unter `reporting` das JUnit-Plugin referenziert.

```
<reporting>
  ...
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>surefire-report-maven-plugin</artifactId>
    </plugin>
  ...
</reporting>
```

Hier wird JUnit (also das `surefire`-Plugin) angewiesen, die Testergebnisse auf unserer Projektseite zur Verfügung zu stellen.

Nach den Reporting-Einstellungen folgen die Abhängigkeiten. Wir möchten JUnit in der Version 4.1 nutzen. Diesen Abschnitt kennen wir zwar schon, hier haben wir allerdings eine neue Einstellung vorgenommen, so dass wir nun JUnit in der Version 4.1 nutzen.

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.1</version>
```

```
    <scope>test</scope>
  </dependency>
</dependencies>
```

Kommen wir nun noch zu einer weiteren praktischen Eigenschaft von Maven: Jeder kann aufbauend auf der offenen Maven-Architektur selbst Plugins schreiben und bereitstellen. Das haben wir mit dem JUnit-Workaround gemacht. Nun müssen wir Maven aber noch mitteilen, wo das entfernte Repository ist, von dem das Plugin bezogen werden kann. Das geschieht im Rahmen des `url`-Elements unter dem neuen `pluginRepository`-Eintrag.

```
<pluginRepositories>
  <pluginRepository>
    <id>unto.net</id>
    <url>http://repository.unto.net/maven/</url>
    <releases>
      <updatePolicy>daily</updatePolicy>
      <checksumPolicy>warn</checksumPolicy>
    </releases>
  </pluginRepository>
</pluginRepositories>
```

Neben den Koordinaten des Plugins teilen wir Maven noch mit, wie häufig es nach neuen Versionen schauen soll und wie es mit Prüfsummen umgehen soll. Es wird täglich nach neuen Versionen zu dem Plugin geschaut.

Die Ausführung

Nun können wir in dem Verzeichnis, in dem *pom.xml* liegt, den Befehl

```
mvn clean site
```

eingeben, um auf einer grünen Wiese damit anfangen zu können, die Projektseite zu erstellen. Wir können also auch zwei Phasen hintereinander in einer Zeile aufrufen. Sie werden sequenziell abgearbeitet. Das gilt übrigens auch für Goals, die sie aneinanderreihen können.

Es liegen folgende Verzeichnisse vor: *classes* mit den Klassen, *surefire-reports* mit den Reports und *test-classes* mit den Testklassen. Darüber hinaus gibt es nun die Datei *agile-1.0-SNAPSHOT.jar*.

```
michael@BlueGene:~/maven/agile/target$ ls -la
insgesamt 24
drwxr-xr-x 5 michael michael 4096 2007-01-02 13:22 .
drwxr-xr-x 4 michael michael 4096 2007-01-02 13:23 ..
-rw-r--r-- 1 michael michael 2341 2007-01-02 13:22 agile-1.0-SNAPSHOT.jar
drwxr-xr-x 3 michael michael 4096 2007-01-02 13:22 classes
drwxr-xr-x 2 michael michael 4096 2007-01-02 13:24 surefire-reports
drwxr-xr-x 3 michael michael 4096 2007-01-02 13:22 test-classes
```

Die Verarbeitung protokolliert Maven auf der Konsole. Von besonderem Interesse ist folgender Abschnitt:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running com.huettermann.agile.AgileTestLegacy
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.026 sec
Running com.huettermann.agile.AgileTest
Tests run: 0, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.008 sec

Results :
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0

[INFO] [junit4:test {execution: default}]
[INFO]
-------------------------------------------------------
  T E S T S
-------------------------------------------------------

Results:
[junit] Tests run: 4, Failures: 0
```

Sie sehen zunächst das JUnit-Plugin `surefire`, wie es die Testklassen verarbeitet. Leider erkennt es nur die Legacy-Klasse. Anschließend wird die JUnit-Individuallösung aufgerufen. Sie erkennt beide Klassen und durchläuft alle vier Testmethoden erfolgreich. Der Legacy-Test wurde also doppelt ausgeführt. Das ist keinesfalls eine Best Practice, sondern eine Konsequenz aus dem Verhalten von Maven und dem eingesetzten JUnit 4-Plugin. Beide Testläufe (mit zwei bzw. vier gelaufenen Tests) sind unabhängig voneinander zu sehen.

Weiterhin interessant ist

```
[INFO] Generate "Checkstyle" report.
[INFO] There are 15 checkstyle errors.
```

Wir können verfolgen, wie der Checkstyle-Report erstellt wird und dass er 15 Errors enthält. Aber wir haben `site` nicht aufgerufen, um allein von der Konsole abhängig zu sein. Im *target*-Verzeichnis befindet sich nun auch ein *site*-Ordner, dessen *index.html* die Projektseite startet. Wir sehen das standardmäßige Layout mit den Standardmenüpunkten und ein wenig Text, den wir in *pom.xml* deklariert haben (siehe Abbildung 20-2).

Im unteren Bereich der Menüleiste erscheint ein Menüpunkt PROJECT REPORTS. Wenn wir den aufklappen, sehen wir die einzelnen Reports, die wir in *pom.xml* angefordert haben: JUnit und Checkstyle (siehe Abbildung 20-3). Die Abbildung 20-4 stellt die Checkstyle-Ergebnisse, die Abbildung 20-5 die JUnit-Testergebnisse dar.

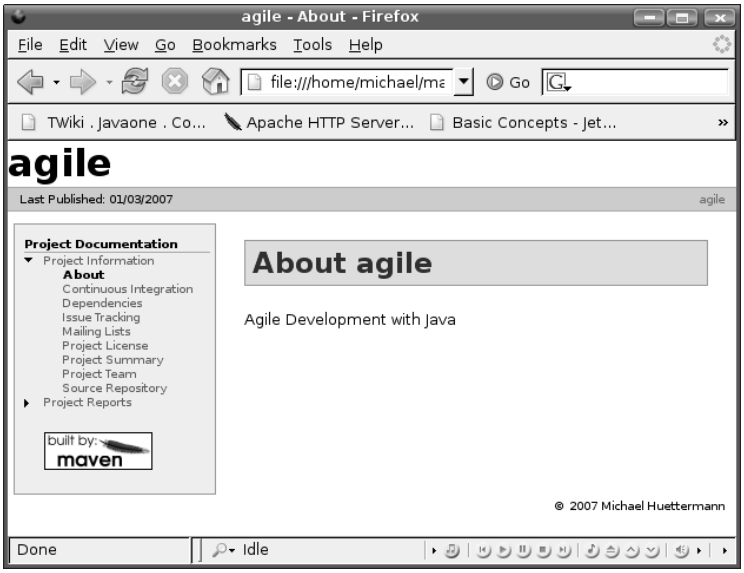


Abbildung 20-2: Die Standard-Startseite

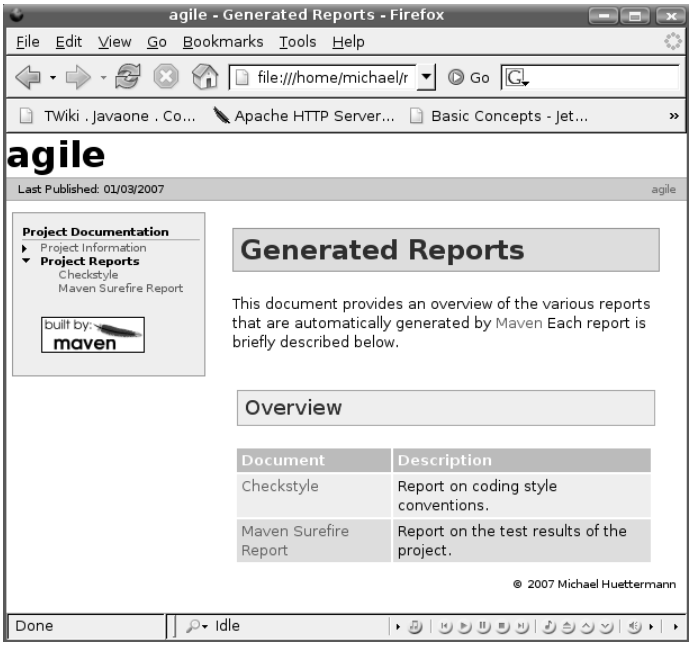


Abbildung 20-3: Projekt-Reports

IDE-Integration

Maven unterstützt die Entkopplung von Artefakten im Subversion-Repository und im lokalen Arbeitsbereich. Es ist unschön, Dateien mit einzuchecken, die abhängig von einer IDE sind. Die IDE muss von der Software unabhängig bleiben. So existieren bei der Entwicklung mit IntelliJ-IDEA beispielsweise die Projektdatei **.ipr*, die Workspace-Dateien **.iws* und Modul-Dateien **.iml*. Diese mit ins Repository einzuchecken ist unsauber, da sie mit dem Projekt nichts zu tun haben.

Um diese »Verunreinigung« zu verhindern, checken Sie die Artefakte in das Subversion-Repository ein bzw. aus ihm aus und nutzen das dedizierte Maven-Plugin, um aus *pom.xml* die entsprechenden IDEA- Dateien zu generieren. Auf diese Weise können Sie das erstellte Projekt direkt mit IDEA öffnen und mit der Arbeit beginnen. Die Basisbefehle sind in Tabelle 20-2 dargestellt.

Diese Unterstützung gibt es auch für Eclipse. Auch dort ist sie stabil und komfortabel.

Tabelle 20-2: Maven-Plugin für IDEA: Goals

Goal	Funktion
`Idea:idea`	Kann genutzt werden, um die drei anderen Goals aufzurufen (`project`, `module`, `workspace`).
`Idea:project`	Generiert die Projekt-Dateien (**.ipr*).
`Idea:module`	Generiert die Module-Dateien (**.iml*).
`Idea:workspace`	Generiert die Workspace-Dateien (**.iws*).
`Idea:clean`	Löscht die zuvor generierten IDEA-Dateien.

Zusammenfassung

Dieses Kapitel war eine Einführung in Maven unter dem Gesichtspunkt der agilen Entwicklung. Wie bei den anderen Kapiteln überlasse ich es anderen Autoren, sich dem Werkzeug in einem einzelnen Buch ausführlichst in allen Facetten zu widmen. Was wir hier gar nicht angesprochen haben, sind die Möglichkeiten, mit Maven Projekte zu schachteln und dabei *pom.xml*-Einträge zu vererben. Auch können wir die Verzeichnisstruktur ändern und die Projektseite komplett selbst entwerfen. Daneben können Einstellungen in Form von Properties ausgelagert oder eine individuelle Datei *settings.xml* genutzt werden.

Anhand dieser Beispiele haben Sie hoffentlich ein Gefühl dafür bekommen, wie Sie mit Maven ein Projekt organisieren, Tests durchführen und Informationen über das Projekt und die Testergebnisse auf einer Projektseite veröffentlichen können. Ich habe gezeigt, wie Sie den mit Maven ausgelieferten Standard nutzen, aber auch davon abweichen können. Beispiele dafür waren, den abgebildeten Build-Prozess zu modifizieren, den Compiler zu ändern und weitere Plugins einzubinden und zu konfigurieren.

agile - Checkstyle Results - Firefox

File Edit View Go Bookmarks Tools Help

file:///home/michae Go

TWiki . Javaone . Co... Apache HTTP Server... Basic Concepts - Jet...

agile

Last Published: 01/03/2007 agile

Project Documentation
- Project Information
- Project Reports
  - **Checkstyle**
  - Maven Surefire Report

built by: maven

Checkstyle Results

The following document contains the results of Checkstyle. XML

Summary

Files	Infos	Warnings	Errors
2	0	0	15

Rules

Rules	Violations	Severity

Done Idle

Abbildung 20-4: Die Ergebnisse von Checkstyle

agile - - Firefox

File Edit View Go Bookmarks Tools Help

file:///home/michael/maven/agile Go

TWiki . Javaone . Co... Apache HTTP Server... Basic Concepts - Jet...

agile

Last Published: 01/03/2007 agile

Project Documentation
- Project Information
- Project Reports
  - Checkstyle
  - **Maven Surefire Report**

built by: maven

Summary

[Summary][Package List][Test Cases]

Tests	Errors	Failures	Success Rate	Time
2	0	0	100.00%	0.02

Note: failures are anticipated and checked for with assertions while errors are unanticipated.

Package List

[Summary][Package List][Test Cases]

Package	Tests	Errors	Failures	Success Rate	Time
com.huettermann.agile	2	0	0	100.00%	0.02

Done Idle

Abbildung 20-5: Die Ergebnisse von JUnit

Maven ist eine sehr gute Alternative zu Ant, kann aber auch mit Ant zusammen eingesetzt werden. Ant-Skripten können verhältnismäßig einfach in *pom.xml* eingebunden werden. Sie können natürlich auch an Maven vorbei mithilfe von Ant Automatisierungen durchführen.

Maven kann genauso wie Ant über Continuous Integration-Server wie CruiseControl oder TeamCity automatisiert bedient werden.

Hier zeigt sich aber schnell der Nachteil von Maven: Je mehr Sie vom Standard abweichen, desto mehr Arbeit haben Sie damit, die gewünschten Schritte zu konfigurieren. Dabei stoßen Sie regelmäßig auf das Problem der Dokumentation: Obwohl Maven selbst noch recht gut dokumentiert ist (es existieren zwar kaum Bücher, dafür aber eine sehr gute Onlinedokumentation unter *http://www.mergere.com*), kann es ein Abenteuer werden, bestimmte Plugins einzubinden und wie gewünscht zu konfigurieren. Hier fehlt häufig eine vernünftige Dokumentation, und die, die es gibt, sind häufig veraltet.

Maven ist zwar nach dem Quantensprung zu Maven 2 recht stabil in der Entwicklung geworden, aber das eine oder andere verändert sich noch. Und wenn sich Maven bzw. das Plugin nicht verändert, ändert sich die einzubindende Applikation selbst. Um das zu illustrieren, habe ich exemplarisch JUnit eingebunden. JUnit wurde als Version 4.x bereitgestellt, doch mit Maven können Sie diese JUnit 4.x-Skripten noch nicht nutzen. Stattdessen müssen Sie auf ältere Versionen wie JUnit 3.x oder Workarounds zurückgreifen, wie das in Einzelinitiative entwickelte JUnit 4-Plugin, das wiederum keine Reporting-Funktionalität bietet.

Ein weiterer Punkt, den Sie berücksichtigen sollten, ist, dass Sie nicht nur von den Plugins abhängig sind, Sie müssen auch hoffen, dass diese zeitnah und lauffähig in dem öffentlichen Repository bereitgestellt werden.

Ein weiterer Punkt ist *pom.xml*. Wie wir am Beispiel gesehen haben, kann es schnell sehr umfangreich werden. Dieses Skript zu warten und zu erweitern, ist eine Aufgabe für einen Mitarbeiter, der sich genau darauf spezialisiert. Das allerdings ist nicht mit dem »Collective Code Ownership« in Einklang zu bringen. Deswegen gilt auch hier, genau wie bei den Ant-Skripten: Wechseln Sie nach wenigen Monaten die Aufgabe des Build-Managers innerhalb des Teams.

Viele werden Ant bewusst Maven vorziehen, da Ant besser zu verstehen ist. Mit Ant ist jederzeit klar, was man gerade tut. Maven ist eine Abstraktionsebene höher, man muss die Konventionen und Interna genau kennen. Aber dann kann sich der Einsatz von Maven durchaus lohnen. Eine geschickte Nutzung von Ant erlaubt zwar auch Wiederverwendbarkeit, doch wird mit Ant häufig das Rad neu erfunden. Vor allem im Bereich der Verwaltung der Abhängigkeiten glänzt Maven und ist Ant deutlich überlegen.

ANHANG

Weiterführende Literatur

Zur Methodik

Andrew Stellmann, Jennifer Greene. *Applied Software Project Management.* O'Reilly, 2005.

Henning Wolf et al. *Extreme Programming.* dpunkt.verlag, 2005.

Mary Poppendieck, Tom Poppendieck. *Lean Software Development.* Addison-Wesley, 2003.

Mary and Tom Poppendieck. *Implementing Lean Software Development.* Addison-Wesley, 2007.

Venkat Subramaniam, Andy Hunt. *Practices of an Agile Developer.* O'Reilly, 2006.

Esther Derby, Diana Larsen. *Agile Retrospectives.* O'Reilly, 2006.

Mike Cohn. *User Stories Applied.* Addison-Wesley, 2004.

Mike Cohn. *Agile Estimating and Planning.* Prentice Hall, 2006.

Eric Burke, Brian Coyner. *Java Extreme Programming Cookbook.* O'Reilly, 2003.

Matthew Doar. *Practical Development Environments.* O'Reilly, 2005.

Kent Beck. *Extreme Programming Explained.* Addison-Wesley, 2005.

Ron Jeffries et al. *Extreme Programming Installed.* Addison-Wesley, 2001.

Kent Beck, Martin Fowler. *Planning Extreme Programming.* Addison-Wesley, 2001.

Scott Berkun. *The Art of Project Management.* O'Reilly, 2005.

Ken Schwaber. *Agile Project Management with Scrum.* Microsoft Press, 2004.

Peopleware – der Faktor Mensch

Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley, 1995.

Tom DeMarco, Timothy Lister. *Bärentango*. Hanser, 2003.

Tom DeMarco. *Spielräume*. Hanser, 2001.

Tom DeMarco. *Warum ist Software so teuer?* Hanser, 1997.

Gerald M. Weinberg. *Die Psychologie des Programmierers*. Mitp, 1998.

Abraham H. Maslow. *Motivation und Persönlichkeit*, Rowohlt Tb., 2002

Tom DeMarco. *Der Termin*. Hanser, 1998.

Tom DeMarco, Timothy Lister. *Wien wartet auf Dich*. Hanser, 1999.

Testen

Johannes Link et al. *Softwaretests mit JUnit*. dpunkt.verlag, 2005.

Thomas Hammell. *Test-Driven Development*. Apress, 2005.

Andreas Spillner, Tilo Linz. *Basiswissen Softwaretest*. dpunkt.verlag, 2006.

Frank Westphal. *Testgetriebene Entwicklung mit JUnit & Fit*. dpunkt.verlag, 2006.

Andrew Hunt, David Thomas. *Pragmatic Unit Testing*. O'Reilly, 2003.

Kent Beck. *Test-Driven Development*. Addison-Wesley, 2003.

Fit

Frank Westphal. *Testgetriebene Entwicklung mit JUnit & Fit*. dpunkt.verlag, 2006.

Rick Mugridge, Ward Cunningham. *Fit for Developing Software*. Prentice Hall, 2005.

Refactoring

Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, 2005.

Martin Fowler. *Refactoring*. Addison-Wesley, 1999

TestNG

Stefan Edlich, Mark Rambow. *Next Generation Testing mit TestNG & Go*. Entwickler. press, 2007.

Ant, Maven

Mike Clark. *Pragmatic Project Automation.* O'Reilly, 2004.

Gunther Popp. *Konfigurationsmanagement mit Subversion, Ant und Maven.* dpunkt.verlag, 2006.

Subversion

Ben Collins-Sussman et al. *Subversion*. O'Reilly, 2005.

Mike Mason. *Pragmatic Version Control using Subversion.* O'Reilly, 2006.

Gunther Popp. *Konfigurationsmanagement mit Subversion, Ant und Maven.* dpunkt.verlag, 2006.

Index

A

Abschätzung 86
Abwechslung 56
Advocatus Diaboli 116
Agile Alliance 4
Agile Entwicklung 3
  Ausprägungen 26
  Geschichte der 4
  und Nutzen 13
  und was sie nicht ist 26
Agiles Manifest 3
  und Prinzipien 6
  und Wertesystem 4
Ajax 123, 267, 271
Aktive Pause 67
Akzeptanztests 283, 292, 312
Akzeptanztests/Story-Tests 97
Anforderungen 69
Angst 48
Ant 100, 124–125, 326, 351, 362
  Abhängigkeiten verwalten 332
  antcall 278
  Aufräumen 329
  batchtest 180
  build.number 152
  build.properties 152
  Checkstyle 342
  classpath 180
  Coding Guidelines 338
  cron 331
  cvsignore 338
  Default-Target 127
  Deployment 330
  echo 278
  Entwicklungszyklus 327
  exec 151
  fileset 180
  get 279
  HotSwap 345
  Installation von 126
  jar 126
  java 278
  javac 180, 357
  JavadDocs 330
  junit 180
  junitreport 180, 357
  jvmarg 199
  Kompilieren 328
  Kontinuierliche Integration auf einem Arbeitsplatzrechner 331
  Kontinuierliche Integration auf Integrationsrechner 331
  Legacy-Code 340
  macrodef 308
  mkdir 126
  Paketierung 329
  parallel 278
  Parameterübergabe an Skript 152
  path 200, 356
  property 179, 343
  Rausholen 330
  report 180, 357
  sequential 278
  sleep 278
  Targets 127
  taskdef 198
  Tasks 127
  Test 329
  Tipps 347
  xslt 344
ant-contrib 127
AntUnit 326
Applaus 110
Architekt 82
Asynchroner Code 313
Aufbauorganisation 81

Aufwandsschätzungen 21
Authentification 133
Authentifizierung 132
Authentisierung 132
AWT 248

B

Balanced Scorecard 70
Bedürfnisebene 57–58
Beziehungsebene 57, 58
Branch 129, 145
Branching 96
Browser 265
Bugzilla 109
Build Manager 82
build.xml 125
Build-Datei 127
Building Block 123
Build-Prozess 124–125
Burndown-Chart 73, 105

C

Canoo 292
Capture & Replay 97, 251
Change Log 129
Changeset 129
Checkstyle 103, 124, 342
Chrysler Comprehensive Compensation System-Projekt 28
ClearCase 129
CMS 316
Coach 81, 115
Code Repository 95
Code Smells 98
Coding-Standards 124, 326
CollabNet 130
Collective-Code Ownership 129
Continuous Integration 93, 99, 126
cron 331, 351
CruiseControl 100, 124, 349–352
361
bootstrapper 355
currentbuildstatuslistener 354
Installation von 353
JMX -Konsole 361
Konfiguration 353
modificationset 354
quietperiod 355
taskdef 356
Weboberfläche 358
Wrapper-Skript 354
XSL-Stylesheets 361
CVS 129

D

Daily Scrums 27
Date Driven Commitment 87
Death March 54
Death Sprint 54
deklarativ 362
Design 16
Dienst nach Vorschrift 46
Diff 129
Dokumentation 73
Domain Model 35
Domain-driven Design (DDD) 35–36, 284
Doomsayer 116
Dummy 207

E

EasyMock 99, 124, 209
andReturn 221
createMock 221
createStrictMock 213
expectLastCall 214
Installation von 210
und JUnit 213
replay 213
times 214
verify 213
Eigenverantwortlichkeit 17
Einführungsstrategien 114
E-Mail 60
EMMA 124, 195
und Ant 198
atomare Statements 206
Blockabdeckung 203
Delegates 206
Fremdkomponenten 206
getter 206
Installation von 195
Negativtests 205
Randwertebereichtests 205
setter 206
Entscheidungen 78
Entscheidungen verschieben 39

Entwickler 82
Entwicklung 93
Entwicklungszweige 145
Erich Gamma 158
Event Dispatch Thread 249
Exploratory Testing 283
Extreme Programming 3, 28
  und Folgepraktiken 33
  und Primärpraktiken 31
  und Prinzipien 30

F

Fallstricke 117
FDD 42
Feature 42
Feature-driven Development 42
Featuritis 15
FindBugs 339
Finger Pointing 95
Fit 98, 123, 283, 292, 313
  ActionFixture 286
  ColumnFixture 285–286
  FitNesse 290
  RowFixture 286
  Spezifikationen schreiben 284
  und TDD 291
Fixpreisprojekte 121
Fixtures 285
Flexibilität 77
Freiräume 56
funktionale Tests 312

G

Gesamtproduktivität 22
Groovy 292

H

Head 129
HotSwap 326
HtmlUnit 267
HTTPUnit 123
Humble Dialog Box 250

I

Idealzeit 92
imperativ 362
Industrial XP 35
Informationen 51
Informationsmedien 22
Infrastruktur 123
inkrementelles Deployment 345
Instrumentalisierung von Bytecode 195
Iterationen 29, 84
Iterationsplanung 91
Ivy 364

J

Java Annotations 157
Java Client Anwendungen 247
Java Swing 123
JavaScript 266, 313
JDepend 339
Jeff DeLuca 42
Jeff Sutherland 26
Jemmy 99, 124, 247
  Delegates 252
  Installation von 252
  JButtonOperator 260
  JFrameOperator 258
  JTableOperator 259
  JTextFieldOperator 259
  mit Fit 253
  mit JUnit 253
  mit TestNG 253
  und JUnit, TestNG 253
Jira 109
JMock 210
JSPs 266
JUnit 98, 124, 157
  @After 160
  @AfterClass 160
  @Before 160
  @BeforeClass 160
  @RunWith 167
  @Test 159
  ActiveTestSuite 173
  und Annotations 163
  und die zu testenden Artefakte 176
  assertEquals 160
  und Assertions 162
  assertTrue 159
  Assumptions 158
  Delegates 176
  Erweiterungen 178
  Fallstricke beim Mischen der Versionen 165

Fixture 160
getter 176
HTTPUnit 178
und IDEs 163
Installation von 158
Interne Funktionalität 176
JUnit4TestAdapter 165
JUnitPerf 178
MasterSuite 175
und eine Methode parametrisieren 169
mit Ant 178
Nebenläufigkeit 171
RepeatedTest 174
Report 184
setter 176
Suites 165, 167
Synchronisationspunkte 177
Testaggregation 167, 171
TestRunner 159, 161
Theories 158
Überprüfung von Exceptions 163
und verschiedene Instanzen 170
Versionen 3.x vs. 4.x 164
XMLUnit 178
und eine zentrale Fixture 167
Just-in-Time Flow 37

K

Kano-Modell 71
Ken Schwaber 26
Kent Beck 4, 28, 158
Kiichiro Toyoda 37
Kohäsion, hohe 186
Kollaboration 316
Kommunikation 45, 57
Komponententests 124, 158, 207, 223
Konfigurationsmanagement 93, 95
Kontinuierliche Integration 178, 351
Kopplung, niedrige 186
Kundensicht 13

L

Label 129
Lean Development 3, 37
Lean Production 37
Legacy-Systeme 122

M

Magisches Viereck 52
make 125
Management nach Zahlen 70
Manuelle Tests 99
Maven 100, 124, 349, 362
Abhängigkeiten 363, 366
artifactId 367
Build 371
Build-Prozess 363
clean 372
Dokumentation 364
Goals 364, 370
groupId 367
und IDEs 383
Kommunikation 364
Konventionen 365
Lifecyles 371
Maven 367
maven-compiler-plugin 378
maven-junit4-plugin 378
Phasen 364, 370
Plugin-Repository 364, 380
POM 366–367, 375
Repository 367
settings.xml 368
site 372
surefire-report-maven-plugin 377, 379
Version 367
MBO 41
Merge 96, 129
Metamodell 3
Methodik 24
Mitarbeiterzufriedenheit 21
Mock-Objekte 99, 207
Model-Delegate 249
Modultest 158
Motivation 45
Multi-Threading 249
Murphy's Law 101
MVC 249

N

Nearshoring 121
Nutzen agiler Entwicklung
Kundensicht 13
Programmierersicht 16
Projektsicht 19
Soziale und betriebswirtschaftliche Sicht 21

O

Offshoring 121
Optimierungen 42
OSGi 224

P

Pair Programming 93
Physische Grundbedürfnisse 47
Pilotprojekt 114
Planspiele 87, 117
PMD 339
Priorisierung 71
Product Owner 27, 81
Product-Backlog 27
Projektkommunikation 316
Projektmanager 82
Prozessmodell 23

Q

Qualität 38, 52, 77

R

Rahmenwerk 24
Rational Unified Process (RUP) 23
Refactoring 33, 93, 97, 102, 185, 190
Regressionstest 268, 292
Release 28, 83
Release-Planung 83
Repository 95, 128–129, 351, 355, 362
Respekt 40
Retrospektive 28, 110
Return On Investment 15
Reviews 104
Risikomanagement 16–20
Rollentrennung 17
Ron Jeffries 28

S

Sachebene 57–58
Sandbox 129
Schweizer Taschenmesser 125
SCM 95, 128
Scrum 3, 26
Scrum-Master 28
Selbstverwirklichung 55
Selenium 98, 123, 265, 283, 292, 313
  und Ant 278
  clickAndWait 271
  IDE 269
  Interaktionen abspielen 270
  Interaktionen aufnehmen 270
  und JUnit 274
  RC 273
  Tabellenrepräsentation 273
  waitForValue 271
  XMLHttpRequest 273
Selenium RC 273
  Implementierung von Tests 276
  Installation von 274
Servlets 266
Smoke Test 100, 268
Software Configuration Management 128
Soziale Anerkennung 50
Specification-By-Example 97, 285
Spezifikation, ausführbare 284
Spiralmodell 24
Sprint Planning Meeting 27
Sprint-Backlog 27
Staging 100
Standards 102
Stand-up-Meetings 62, 93–94
Stop-the-Line 37
Story Cards 14, 92
Story Driven Commitment 87
Story Points 29, 83, 105
Stub 207
Subversion 123, 128–129
  Änderungen einchecken 138
  Änderungen untersuchen 137
  anlegen 145
  und Ant 150
  Artefakte auschecken 136
  Berkeley DB 131
  Branch 129, 131, 145
  Change Log 129
  Changeset 129
  Client 131, 155
  Copy-Edit-Merge 148
  Dependency-Package 131
  Diff 129
  Fast Track 156
  FSFS 131
  Head 129

IDE 154
Installation 131
KDiff3 155
Konflikte auflösen 139
Label 129
Locking 148
Lokalisierung 135
Mergen 129
Neon 131
Optimistisches Sperren 148
Pessimistisches Sperren 149
Property Sets 149
Revisionen 142
Sandbox 129
Subversion Pro 156
Subversive 156
SVK 156
svn 135
SVN Importer 156
svn_add_ 137
svn_co 136
svn_commit 139
svn_copy 144
svn_diff 137, 141
svn_import 135
svn_info 139
svn_list 136, 144
svn_log 142
svn_resolved 141
svn_revert 138
svn_status 138
svn_update 139
svnadmin 136
svnserve 134
svnserve.conf 133
Tag 129, 131, 143
Tortoise 131
TortoiseSVN 155
Trunk 129
Verbindungsmöglichkeiten 134
Versionierung 143
WebClient for SVN 156
WebDAV- 131
und Werkzeuge 154
Workspace 129
Zlib 131
Zugriffsrechte 132
zusammenführen 147
Swing 247
  Testen von 250
Synchronisation 100

T

Tag 129, 143
Task Cards 92
Tasks 27, 91
TDD *siehe* Test-driven Development 185
Team 45, 64, 82, 122
TeamCity 361
Teamgröße 68
Teamkiller 67
Telefon 61
Tempo 76
Testabdeckung 195, 202
Testaggregation 171
Test-driven Development 93, 97, 185
  und Refactoring 190
Test-driven Development (TDD) 124
Testen 15, 123
  Akzeptanztests 123
  funktionale Tests 123
  Komponententests 123
Test-First 97
TestNG 98, 124, 157, 165, 170, 223
  @BeforeSuite 233
  Abhängigkeiten 236
  und Annotations 224
  und Ant 236
  groups 232
  und Gruppierungen 225
  und IDEs 246
  und Nebenläufigkeit 225
  Reporting 239
  testng.xml-Datei 242
  Thread_Pool 233
Ticketing-System 318
Time-Boxing 29
Trac 109, 124, 316
  Installation von 319
  Konfiguration 321
  Markup 322
  Milestones 319
  Roadmap 319
  Subversion 318
  trac.ini 322

tracd 319
User Stories 318
Tracking 105
Tracking-Datenbank 109
Trunk 129

U

Überflüssiges 38
Under Cover 113
Usability 99
User-Stories 14, 29, 83

V

VCS 128
Velocity 29, 85, 105
Verantwortung 55
Verschwender 79
Versionierung 128
Versionskontrolle 128
Verteilte Teams 121
V-Modell 24

W

Ward Cunningham 28, 316
Wasserfallmodell 24
Web 2.0 317
Webanwendungen 123, 265, 283, 292
testen 267
WebTest 98, 123, 283, 292
clickButton 296
config.xml 297
Generierung eines Projektes 293
und IDEs 311
Installation 293
invoke 294
Kontinuierliche Integration 305
Recorder 297
Reporting 300
Selbsttest 293
setInputField 296
Steps 292
Testergebnisse 311
verifyTitle 294
webtest.xml 309
Wiederverwendung 299
XML Entities 297
Werkzeuge 123
Wertesystem 3
Wertschöpfungsprozess 22
Whiteboard 105
Wiki 61, 124, 316
Wikipedia 316
Wissen 39, 45, 69
Wissenszuwachs 21
Workspace 95, 129
WWW 316

X

XP 28
XP@Scrum 36
XPath 267

Z

Zeit 45, 76
Zertifizierungen 121

Über den Autor

Michael Hüttermann ist Sun Java Champion, Freelancer, Autor, Trainer und Berater für Java/JEE und agile Softwareentwicklung. Er verfügt über mehr als zehn Jahren Projekterfahrung und teilt sein Wissen in Fachartikeln, Büchern, in der Wikipedia und auf Konferenzen. Für die Java User Group Köln organisiert er regelmäßig Vorträge mit hochkarätigen, international bekannten Referenten. Michael ist einer von drei »java.net JUGs Community Leaders«. Er ist ferner aktiv in der Agile Alliance und dem Java Community Project und besitzt mit dem SCJA, SCJP, SCJD und SCWCD mehrere Java-Zertifizierungen. Weil er sich jahrelang nachhaltig für die Weiterentwicklung und Verbreitung der Java Plattform engagiert hat, wurde er im Dezember 2006 mit dem Titel »Java Champion« geehrt.

Wenn er nicht gerade Tag und Nacht vor dem Rechner sitzt, tobt Michael mit seiner Frau und seinen Söhnen herum oder frönt dem Fußball. Im Ruhrgebiet geboren, studierte er Wirtschaftsinformatik in Paderborn, ehe es ihn vor Jahren nach Köln verschlug. Seine Webseite ist *http://huettermann.net*.

Kolophon

Auf dem Cover von *Agile Java-Entwicklung in der Praxis* ist ein Mondspinner abgebildet (Actias luna). Der Mondspinner ist eine der weltweit größten und auffälligsten Mottenarten und lebt in Mittelamerika und im östlichen Nordamerika. Der zu den Pfauenspinnern (Saturniidae) gehörende Nachtfalter hat hellgrüne Flügel mit markanten tief violetten Augenflecken. Die Hinterflügel sind schwanzartig verlängert. Beide Geschlechter können bis zu dreizehn Zentimeter groß werden und unterscheiden sich nur durch die langen fiedrigen Fühler der Männchen, ihren Geruchsorganen. Mit ihnen können sie die von den Weibchen ausgestoßenen Pheromone über große Distanzen orten. Diese Art der Kontaktaufnahme ist auch notwendig, da das Leben der ausgewachsenen Falter nur kurz ist und sich voll und ganz auf die Fortpflanzung konzentriert: Ihre Mund- und Verdauungswerkzeuge sind zurückgebildet, so dass sie kein Futter zu sich nehmen können und nach kurzer Zeit sterben.

Die Weibchen legen ca. 200 Eier auf den Futterpflanzen der Raupen ab, z.B. auf Bäumen wie Kastanien, Walnuss, Birke und Eiche, von deren Blättern sich die Raupen ernähren. Diese schlüpfen nach ca. zehn Tagen und verbringen fortan ihre gesamte Zeit mit der Nahrungsaufnahme. In diesem Stadium wachsen die Insekten in kürzester Zeit von drei Millimetern auf sechs bis sieben Zentimeter heran und müssen sich dabei viermal häuten. Nach ca. fünf Wochen verpuppen sie sich in einem spindelförmigen Kokon. Aus dem Seidengespinst schlüpft im Frühjahr dann der junge Schmetterling. Auch wenn der Mondspinner, wie sein Verwandter, der Echte Seidenspinner (Bombyx mori), Kokonfäden produziert, hat man die Versuche, daraus Seide herzustellen, bald wieder aufgegeben, da die Fäden von schlechterer Qualität sind.

Der Umschlagsentwurf dieses Buches basiert auf dem Reihenlayout von Edie Freedman und stammt von Michael Oreal, der hierfür einen Stich des *Dover Pictorial Archive* aus dem 19. Jahrhundert verwendet hat. Als Textschrift verwenden wir die Linotype Birka, die Überschriftenschrift ist die Adobe Myriad Condensed und die Nichtproportionalschrift für Codes ist LucasFont's TheSans Mono Condensed. Die in diesem Buch enthaltenen Abbildungen stammen von Michael Oreal. Geesche Kieckbusch hat das Kolophon geschrieben.