Der Inhalt (im Überblick)

	Einführung	xxiii
1	Gute Software entwickeln: *Den Kunden zufrieden stellen*	1
2	Anforderungen sammeln: *Wissen, was der Kunde will*	29
3	Projektplanung: *Mit Planung zum Erfolg*	69
4	User-Stories und Tasks: *Jetzt aber an die Arbeit*	109
5	Gut genug-Design: *Fertig werden – mit gutem Design*	149
6	Versionskontrolle: *Defensives Programmieren*	177
6.5	Code erstellen: *Lasche A durch Schlitz B stecken …*	219
7	Tests und kontinuierliche Integration: *Damit nichts schiefgeht*	235
8	Testgetriebene Entwicklung: *Ihren Code verantwortlich machen*	275
9	Eine Iteration abschließen: *Alles fügt sich zusammen*	317
10	Die nächste Iteration: *Es ist nicht kaputt … reparieren Sie es trotzdem*	349
11	Bugs: *Bugs beheben wie ein Profi*	383
12	Das wahre Leben: *Ein Prozess fürs Leben*	417

Der Inhalt (jetzt ausführlich)

Einführung

Ihr entwicklungsfähiges Gehirn. Sie versuchen, etwas zu lernen, und Ihr Hirn tut sein Bestes, damit das Gelernte nicht hängen bleibt. Es denkt nämlich: »Wir sollten lieber ordentlich Platz für wichtigere Dinge lassen, z.B. für das Wissen, welche Tiere einem gefährlich werden könnten, oder dass es eine ganz schlechte Idee ist, nackt Snowboard zu fahren.« Tja, wie schaffen wir es nun, Ihr Gehirn davon zu überzeugen, dass Ihr Leben davon abhängt, gute Software entwickeln zu können?

Für wen ist dieses Buch?	xxiv
Wir wissen, was Sie gerade denken.	xxv
Metakognition: Nachdenken übers Denken	xxvii
Und das können SIE tun, um sich Ihr Gehirn untertan zu machen	xxix
Lies mich!	xxx
Die Gutachter	xxxii
Danksagungen	xxxiii

Gute Software entwickeln

1 Den Kunden zufrieden stellen

Wenn Ihr Kunde unzufrieden ist, sind *alle* unzufrieden!

Jedes gute Stück Software fängt damit an, dass ein Kunde eine tolle Idee hat. Ihr Job als Softwareentwickler ist es, **diese Idee real werden zulassen.** Aber aus einer vagen Idee funktionsfähigen Code zu machen – Code, **der den Kunden zufrieden stellt** – ist nicht ganz einfach. In diesem Kapitel werden Sie lernen, wie man verhindert, in der Softwareentwicklung Schiffbruch zu erleiden: indem Sie genau die Software abliefern, die **benötigt** wird, und zwar **pünktlich** und **im Rahmen des Budgets**. Jetzt schnappen Sie sich mal Ihren Laptop, und dann ziehen wir los, um gute Software auf den Weg zu bringen.

iel

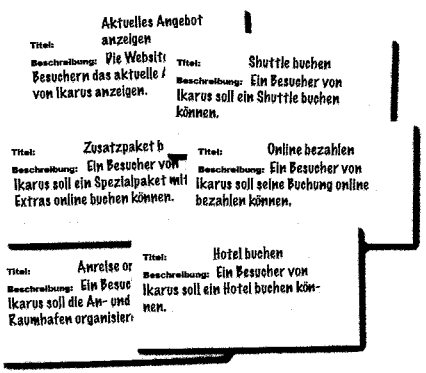
Toms Touren gehen online	2
Die meisten Projekte starten mit zwei wichtigen Fragen	3
Entwicklung nach dem »Big Bang«-Ansatz	4
Wir spulen vor: zwei Wochen später	5
Big Bang-Entwicklung endet meist mit einer üblen BAUCHLANDUNG	6
Gute Software entwickeln heißt …	9
Das Ziel mit ITERATIONEN erreichen	10
Jede Iteration ist ein Mini-Projekt	14
Jede Iteration entspricht QUALITÄTS-Software	14
Der Kunde WIRD etwas verändern	20
Es ist Ihre Aufgabe, Anpassungen vorzunehmen	20
Da gibt es aber ein paar MÄCHTIGE Probleme ...	20
Iteration berücksichtigt Veränderungen automatisch (irgendwie jedenfalls)	22
Ihre Software ist erst fertig, wenn sie VERÖFFENTLICHT ist	25
Werkzeuge für Ihre Softwareentwickler-Kiste	26

2

Anforderungen sammeln

Wissen, was der Kunde will

Man kann nicht alles haben, was man will … der Kunde allerdings schon! Gute Softwareentwicklung liefert, **was der Kunde haben will.** In diesem Kapitel geht es ausschließlich darum, **wie wir von unseren Kunden erfahren können,** welche **Anforderungen** sie an die Software stellen. Sie werden lernen, wie **User-Stories, Brainstorming** und das **Planungsspiel** Ihnen helfen, Ihrem Kunden in den Kopf zu sehen. Auf diese Weise können Sie sicherstellen, am Ende des Projekts das angefertigt zu haben, was der Kunde will … und nicht bloß eine armselige Imitation.



Ikarus Shuttle-Trips wird modernisiert 30

Sprechen Sie mit Ihrem Kunden, um an MEHR Informationen zu kommen 33

Blueskying – mit dem Kunden Wolken schieben 34

Manchmal endet Blueskying so … 36

Finden Sie heraus, wie sich die Leute KONKRET verhalten 37

Ihre Anforderungen müssen KUNDENzentriert sein 39

Kundenfeedback zum Entwickeln von Anforderungen nutzen 41

User-Stories definieren das WAS Ihres Projekts … Abschätzungen definieren das WANN 43

Bürogespräch 47

Spielen wir das Planungsspiel 48

Machen Sie Annahmen den Prozess 51

Für eine User-Story ist eine GROSSE Abschätzung eine SCHLECHTE Abschätzung 54

Das Ziel heißt Konvergenz 57

Der Iterationszyklus von der Anforderung bis zur Abschätzung 60

Jetzt endlich sind Sie so weit und können das gesamte Projekt abschätzen … 63

Projektplanung

3 Mit Planung zum Erfolg

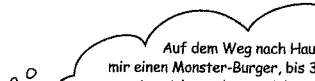
Jedes gute Stück Software beginnt mit einer soliden Planung.

In diesem Kapitel werden Sie lernen, wie Sie diese Planung erstellen. Sie werden lernen, mit dem Kunden zusammenzuarbeiten, um **Prioritäten für die Anforderungen festzulegen**. Sie werden **Iterationen definieren**, auf die Sie anschließend gemeinsam mit Ihrem Team hinarbeiten können. Und schließlich werden Sie einen **realisierbaren Projektplan** ausarbeiten, dem Sie sich mit Ihrem Team anvertrauen und **dessen Durchführung Sie kontrollieren können.** Wenn Sie sich das erarbeitet haben, werden Sie genau wissen, wie Sie von den Anforderungen zu Ihrer ersten lieferbaren Leistung kommen.

Kunden wollen ihre Software SOFORT!	70
Prioritäten festlegen – mit dem Kunden zusammen	73
Wir wissen, was in Meilenstein 1.0 hineinkommt (ja, vielleicht)	74
Wenn die Features nicht passen, überarbeiten Sie die Prioritäten	75
Manchmal bedeuten mehr Leute weniger Ertrag	77
Arbeiten Sie sich zu einem vernünftigen Meilenstein 1.0 vor	78
Bei Iterationen liegt die Würze in der Kürze	85
Vergleich von Planung und Realität	87
Velocity berücksichtigt Overhead in Ihren Abschätzungen	89
Programmierer rechnen in UTOPISCHEN Tagen …	90
Entwickler rechnen in REALEN Tagen …	91
Wann dauert Ihre Iteration zu lange?	92
Befassen Sie sich mit dem Durchsatz, BEVOR Sie Iterationen einteilen	93
Zeit für eine Beurteilung	97
Mit [redacted] Kunden umgehen	98
Die große Tafel da an Ihrer Wand	100
Wie Sie den Leuten in Ihrem Team das Leben vermiesen	103

ammierer Folgendes ...

irklichkeit allerdings das hier:



User-Stories und Tasks

4 Jetzt aber an die Arbeit

Es wird Zeit, sich an die Arbeit zu machen. Mit User-Stories erfassen Sie, was entwickelt werden soll, und jetzt wird es Zeit, sich an die Arbeit zu machen und **das alles aufzuteilen**, damit in diese User-Stories langsam Leben kommt. Im folgenden Kapitel werden Sie lernen, wie Sie Ihre **User-Stories in einzelne Tasks** (Aufgaben) **aufbrechen** und wie **Task-Abschätzungen** Ihnen helfen, Ihr Projekt von Anfang bis Ende zu überwachen. Sie werden lernen, wie Sie Ihre Projekttafel aktualisieren und Tasks von »In Arbeit« nach »Erledigt« verschieben, um am Ende **eine vollständig fertiggestellte User-Story** zu haben. Auf dem Weg dahin werden Sie erfahren, wie Sie mit der unvermeidlichen **unvorhergesehenen Arbeit** umgehen müssen, die Ihnen Ihr Kunde vorsetzen wird.

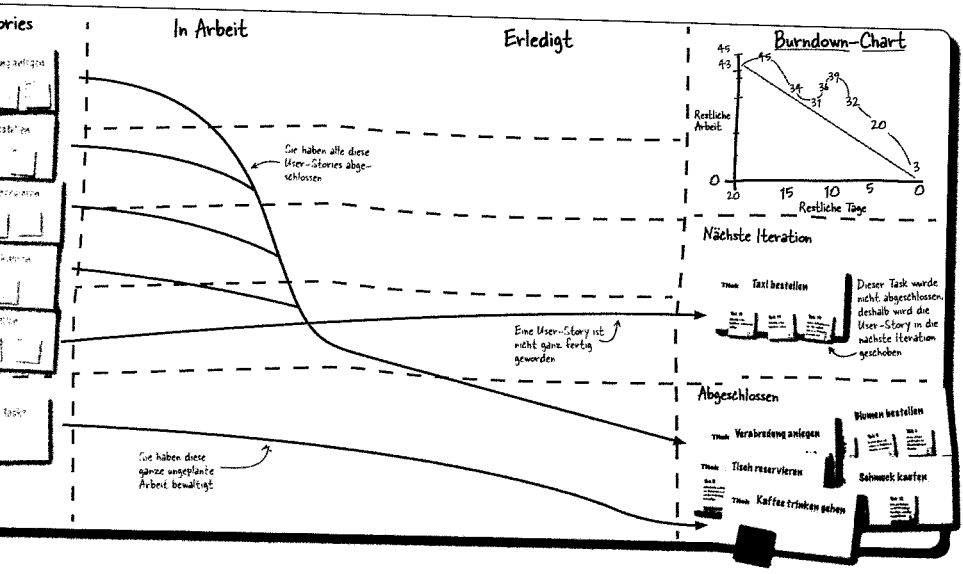
Dürfen wir vorstellen? – binHinUndWeg 110
Addieren sich Ihre Tasks zum richtigen Ergebnis? 113
Tragen Sie nur die Arbeit ab, die noch übrig ist 115
Hängen Sie Ihre Tasks an die Tafel 116
Fangen Sie an, an Ihren Tasks zu arbeiten 118
Ein Task ist erst dann IN ARBEIT, wenn er in Arbeit ist 119
Und wenn ich an zwei Sachen gleichzeitig arbeite? 120
Ihr erstes Stand-up-Meeting … 123
Task 1: die »Date«-Klasse anlegen 124
Stand-up-Meeting: Tag 5, also Ende von Woche 1 … 130
Stand-up-Meeting: Tag 2, Woche2 136
Wir unterbrechen dieses Kapitel … 140
Ungeplante Tasks müssen festgehalten werden 141
Unerwartete Tasks lassen Ihren Burndown-Verlauf ansteigen 143
Durchsatz ist hilfreich, aber … 144
Sie haben eine Menge zu tun … 146
… aber Sie wissen GENAU, wo Sie stehen 147
Durchsatz im Gespräch 148

Ihr erstes Stand-up-Meeting …

5 Fertig werden – mit gutem Design

Gutes Design hilft Ihnen dabei, liefern zu können. Im letzten Kapitel sah es reichlich düster aus. Schlechtes Design **hat allen das Leben schwer gemacht**, und zur Krönung des ganze Übels tauchte urplötzlich noch ein ungeplanter Task auf. In diesem Kapitel werden wir uns ansehen, wie Sie mit **Refactoring** Ihr Design so umgestalten, dass Sie und Ihr Team **produktiver arbeiten** können. Sie werden **Prinzipien für gutes Design** anwenden und sich gleichzeitig davor hüten, ein ***»perfektes Design«*** anzustreben. Abschließend werden Sie mit **ungeplanten Tasks** genau so umgehen wie mit der übrigen Arbeit in Ihrem Projekt, und dafür werden Sie die große Projekttafel nutzen, die Sie an Ihrer Wand hängen haben.

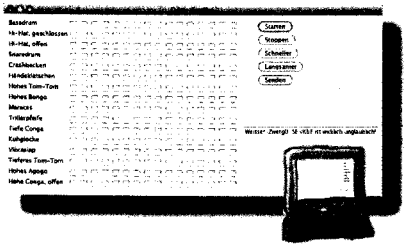
binHinUndWeg steckt in ernsten Schwierigkeiten … 150
Dieses Design verstößt gegen das Prinzip der einen Verantwortlichkeit 153
Mehrere Verantwortlichkeiten im Design aufspüren 156
Von mehreren Verantwortlichkeiten zu einer Verantwortlichkeit übergehen 159
Ihr Design sollte dem SRP folgen, aber ebenso dem DRY-Prinzip … 160
Das Post-Refactoring-Stand-up-Meeting … 164
Ungeplante Tasks sind trotz allem immer noch Tasks 166
Die Demo ist selbst Teil Ihres Tasks 167
Wenn alles erledigt ist, ist die Iteration beendet 170



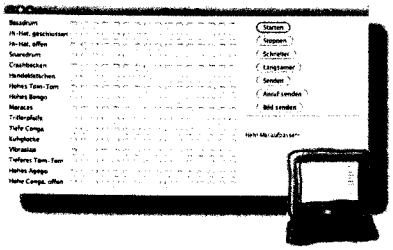
Versionskontrolle

6 Defensives Programmieren

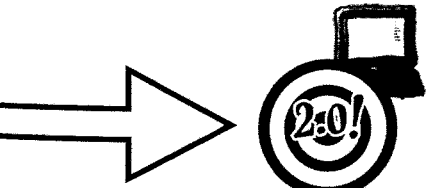
Sicherheit geht vor, wenn es darauf ankommt, wirklich gute Software zu schreiben! Gute Software zu schreiben ist nicht einfach … vor allem, wenn Sie sichergehen müssen, dass der Code **funktioniert und dass das auch so bleibt**. Ein einziger Tippfehler reicht aus, eine ungünstige Entscheidung eines Kollegen oder auch ein Festplatten-Crash, und Ihre gesamte Arbeit geht den Bach runter. Wenn Sie aber mit **Versionskontrolle** arbeiten, können Sie dafür sorgen, dass Ihr **Code immer sicher** in einem Repository **aufbewahrt** wird. Sie können **Fehler rückgängig machen**, und Sie können **Bugs beseitigen** – sowohl in neuen als auch in älteren Versionen Ihrer Software.



BeatBox Pro 1.0



BeatBox Pro 1.x



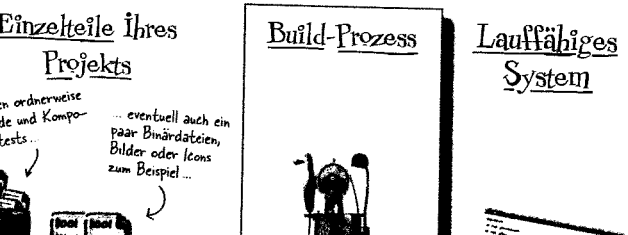
Sie haben einen neuen Auftrag - BeatBox Pro 178
Und jetzt die Arbeit am GUI … 182
Und ein schneller Test … 183
Führen Sie Ihrem Kunden die neue BeatBox vor 185
Kommen wir zum ThemaVERSIONSKONTROLLE 188
Zuerst legen Sie Ihr Projekt an … 190
… anschließend können Sie Code ein- und auschecken 191
Die meisten Werkzeuge zur Versionskontrolle versuchen, Probleme für Sie zu lösen 192
Der Server versucht, Ihre Änderungen ZUSAMMENZUFÜHREN 193
Wenn Ihre Software die Änderungen nicht zusammenführen kann, meldet sie einen Konflikt 194
Noch mehr Iterationen, noch mehr Stories … 198
Wir haben mehr als eine Version von unserer Software … 200
Gute Commit-Kommentare machen es Ihnen leichter, ältere Software zu finden 202
Jetzt können Sie Version 1.0 auschecken 203
(Notfall-)Stand-up-Meeting 204
Geben Sie Ihren Versionen ein Tag 205
Tags, Zweige und Trunks … au weia! 207
Version 1.0 korrigieren … diesmal aber wirklich! 208
Wir haben jetzt ZWEI Codestämme. 209
Wann man KEINEN Zweig anlegen sollte … 212
Zen oder die Kunst vom guten Verzweigen 212
Was Versionskontrolle macht … 214
Versionskontrolle kann nicht dafür sorgen, dass Ihr Code wirklich funktioniert … 215
Werkzeuge für Ihre Softwareentwickler-Kiste 216

Code erstellen

6½ Lasche A durch Schlitz B stecken ...

Es lohnt sich, sich an die Anweisungen zu halten ... vor allem wenn Sie sie selbst geschrieben haben. Versionskontrolle allein reicht nicht aus, um Ihren Code permanent abzusichern. Sie müssen sich Gedanken darüber machen, Ihren **Code zu kompilieren** und als vertriebsfertiges Softwarepaket aufzubereiten. Und vor allen Dingen: Welche Klasse soll die Hauptklasse sein? Wie soll sie ausgeführt werden? In diesem Kapitel lernen Sie, wie ein **Build-Tool** Ihnen ermöglicht, **Ihre eigenen Anweisungen** für den Umgang mit Ihrem Quellcode zu schreiben.

Entwickler sind keine Gedankenleser	220
Ihr Projekt in einem Schritt erstellen	221
Ant: Ein Build-Tool für Java-Projekte	222
Projekte, Properties, Targets, Tasks	223
Gute Build-Skripten ...	228
Gute Build-Skripten gehen über das Übliche HINAUS	230
Auch Ihr Build-Skript ist Code	232
Neuer Entwickler, Klappe, die zweite	233
Werkzeuge für Ihre Softwareentwickler-Kiste	234

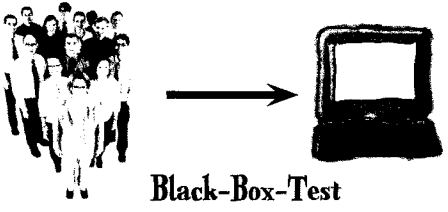


Tests und kontinuierliche Integration

7 Damit nichts schiefgeht

Auch die besten Entwickler können den Build beschädigen.

Jeder hat es schon einmal gemacht. Sie sind sich sicher: **Mein Code kompiliert**. Sie haben ihn immer wieder auf Ihrem System getestet und jetzt ins Repository eingepflegt. Aber auf dem Weg zwischen Ihrem System und der schwarzen Kiste, die man Server nennt, muss *jemand* Ihren Code geändert haben. Der arme Kerl, der den nächsten Check-out durchführt, wird einen harten Tag damit verbringen, das in Ordnung zu bringen, **was einmal funktionierender Code war**. In diesem Kapitel werden wir darüber reden, wie man ein **Sicherheitsnetz** aufbaut, um den Build funktionsfähig und Sie selbst **produktiv** zu halten.



Schiefgehen tut es IMMER	236
Ihr System können Sie auf dreierlei Weise betrachten	238
Black-Box-Tests konzentrieren sich auf EINGABE und AUSGABE	239
Grey-Box-Tests bringen Sie NÄHER an den Code	240
White-Box-Tests nutzen Insiderwissen	243
ALLES in einem Schritt testen	248
Tests mit einem Test-Framework automatisieren	250
Führen Sie Ihre Tests mit dem Framework aus	251
CI im Griff mit CruiseControl	254
Tests stellen sicher, dass alles funktioniert ... oder?	256
Den gesamten Code testen heißt, ALLE ZWEIGE zu testen	264
Mit einem Abdeckungsbericht prüfen, was abgedeckt wird	265
Eine gute Abdeckung zu erreichen ist nicht immer leicht ...	267
Was Versionskontrolle macht ...	270
Werkzeuge für Ihre Softwareentwickler-Kiste	274

Testgetriebene Entwicklung

8 Ihren Code verantwortlich machen

Manchmal geht es nur darum, Erwartungen zu erfüllen. Guter Code muss funktionieren, jeder weiß das. Aber woher wissen Sie, dass **Ihr Code funktioniert**? Selbst mit Unit-Tests gibt es in großen Codemengen immer noch Bereiche, die ungetestet bleiben. Aber was wäre, wenn Testen ein **grundlegender Teil der Softwareentwicklung wäre**? Was ist, wenn Sie ***alles*** mit Blick auf Tests machen würden? In diesem Kapitel werden wir auf dem aufbauen, was Sie über Versionskontrolle, CI und automatisiertes Testen wissen, und alles zu einer Umgebung zusammenbinden, in der Sie keine **Angst** vor dem **Reparieren von Bugs**, **dem Refactoring** und soger der **Neuimplementierung** von Teilen Ihres Systems haben müssen.

Testen Sie ERST, nicht zum Schluss	276
Wir werden also ERST testen ...	277
Willkommen bei der testgesteuerten Entwicklung	277
Ihr erster Test ...	278
... schlägt kläglich fehl	279
Machen Sie Ihre Tests GRÜN	280
Rot, grün, Refactoring ...	281
Bei TDD LENKEN Tests Ihre Implementierung	286
Einen Task abzuschließen bedeutet, dass Sie alle erforderlichen Tests haben und diese alle erfolgreich sind	288
Gehen Sie weiter, wenn Ihr Test erfolgreich ist!	289
Einfachheit heißt, Abhängigkeiten zu vermeiden	293
Schreiben Sie immer testbaren Code	294
Schauen Sie sich Ihren Entwurf an, wenn das Testen schwer wird	295
Das Strategy-Muster ermöglicht mehrere Implementierungen eines Interface	296
Halten Sie Ihren Testcode bei Ihren Tests	299
Testen führt zu besserem Code	300
Mehr Tests bedeuten immer viel mehr Code	302
Strategy-Muster, lockere Bindung, Mock-Objekte ...	303
Wir brauchen viele unterschiedliche, aber ähnliche Objekte	304
Können wir nicht Objekte generieren?	304
Ein Mock-Objekt vertritt echte Objekt	305
Mock-Objekte sind funktionierende Objektvertreter	306
Gute Software ist testbar	309
Es ist nicht immer leicht, grün zu sein ...	310
Ein Tag im Leben eines testgesteuerten Entwicklers	312
Werkzeuge für Ihre Softwareentwickler-Kiste	314

1 Rot: Ihr Test scheitert.

2 Grün: Ihr Test ist erfolgreich.

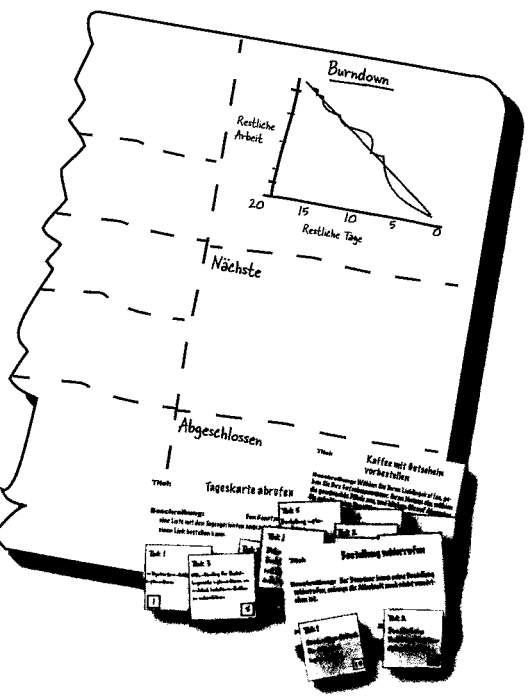
3 Refactoring: Alle Verdopplungen, Hässlichkeiten, und alten Code usw. aufräumen.

Eine Iteration abschließen

9 Alles fügt sich zusammen

Sie sind fast fertig! Das Team hat hart gearbeitet, und jetzt fügt sich alles zusammen. Ihre Aufgaben und User-Stories sind **vollständig**. Aber womit verbringen Sie jetzt am besten den Tag, den Sie noch übrig haben? Wo können Sie **Usability-Tests** einbauen? Können Sie eine weitere **Refactoring-** und **Umarbeitungsrunde** einbauen? Und sicher treiben sich da noch eine Menge **Fehler** herum ... wann werden die behoben? All das ist Teil des **Endes einer Iteration** ... machen wir uns also bereit, fertig zu werden.

Ihre Iteration ist fast abgeschlossen ...	318
... aber es gibt eine Menge, was Sie noch tun könnten	319
Systemtests sind NOTWENDIG ...	324
... aber WER führt sie durch?	325
Systemtests benötigen ein vollständiges System	326
Gute Systemtests erfordern ZWEI Iterationszyklen	327
Mehr Iterationen führen zu mehr Problemen	328
10 Kennzeichen effektiver Systemtests	333
Leben (und Sterben) eines Bugs	334
Sie haben also einen Bug gefunden ...	336
Anatomie eines Bug-Reports	337
Aber es gibt eine Menge Dinge, die Sie noch tun KÖNNTEN ...	338
Zeit für den Iterationsrückblick	342
Einige Fragen für den Iterationsrückblick	343
Eine ALLGEMEINE Prioritätsliste zur Erledigung ZUSÄTZLICHER Dinge ...	344
Werkzeuge für Ihre Softwareentwickler-Kiste	346



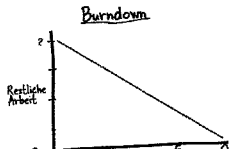
Die nächste Iteration

10 Es ist nicht kaputt ... reparieren Sie es trotzdem

Alles läuft gut? Warten Sie ab, das kann sich schnell ändern ...

Ihre Iteration ist wunderbar gelaufen, und Sie haben rechtzeitig funktionierende Software ausgeliefert. Zeit für die nächste Iteration? Kein Problem, stimmt's? Leider nicht ganz. In der Softwareentwicklung dreht sich alles um **Veränderung**, und der **Schritt zur nächsten Iteration** bildet darin keine Ausnahme. In diesem Kapitel werden Sie lernen, wie Sie sich auf die ***nächste*** Iteration vorbereiten. Sie müssen **Ihre Tafel neu aufbauen** und **Ihre Stories und Annahmen anpassen** – und das auf Basis dessen, was der Kunde ***JETZT*** will, nicht darauf, was er vor einem Monat wollte.

Was ist funktionierende Software?	350
Sie müssen die nächste Iteration planen	352
Der Durchsatz trägt dem WAHREN LEBEN Rechnung	359
Es geht IMMER noch um den Kunden	360
Auch die Software eines anderen ist BLOSS Software	362
Kundenzustimmung? Prüfen!	365
Ihren Code testen	370
Houston, wir haben ein ernsthaftes Problem ...	371
Vertrauen Sie KEINEM	373
Sie ohne Ihren Prozess	378
Sie mit Ihrem Prozess	379



11 Bugs beheben wie ein Profi

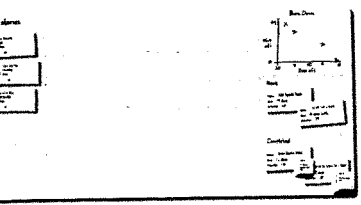
Ihr Code, Ihre Verantwortung ... Ihr Bug, Ihr Ruf! Wenn's hart wird, müssen Sie den Karren selbst **aus dem Dreck** ziehen. **Bugs**, egal ob sie in Ihrem Code stecken oder bloß in Code, den Ihre Software nutzt, gehören in der Softwareentwicklung zu den **Gegebenheiten des Lebens**. Und wie alles andere sollte sich auch das Verfahren, mit dem Sie Bugs behandeln, in Ihren restlichen Prozess einfügen. Sie müssen **Ihre Tafel vorbereiten**, **Ihren Kunden auf dem Laufenden halten**, **zuverlässig einschätzen**, wie viel Arbeit es ist, die Bugs zu beheben, und **Refactoring** sowie **Prefactoring** einsetzen, um Bugs zu reparieren und in der Zukunft zu vermeiden.

Neulich in Iteration 2	386
Zunächst müssen Sie mit dem Kunden reden	386
Priorität eins: Die Dinge erstellbar machen	392
Wir könnten den Code reparieren ...	394
... müssen aber die Funktionalität reparieren	395
Herausfinden, welche Funktionalität funktioniert	396
JETZT wissen Sie, was nicht funktioniert	399
Was würden Sie tun?	399
Spike-Test zum Abschätzen	400
Was sagen Ihnen die Ergebnisse der Spike-Tests?	402
Das Bauchgefühl Ihres Teams zählt	404
Geben Sie Ihrem Kunden die Bug-Fix-Abschätzung	406
Die Dinge sehen gut aus ...	410
... und Sie schließen die Iteration erfolgreich ab!	411
UND der Kunde ist zufrieden	412
Werkzeuge für Ihre Softwareentwickler-Kiste	414

Das wahre Leben

12 Ein Prozess fürs Leben

Sie haben eine Menge über die Entwicklung von Software gelernt. Aber bevor Sie jetzt anfangen, Burndown-Graphen in jedes Büro zu hängen, sollten Sie noch erfahren, wie man mit den Eigenarten des jeweiligen Projekts verfährt – in seinen eigenen Begriffen. Es gibt eine Menge **Ähnlichkeiten** und **Erfahrungswerte,** die Sie von einem Projekt ins nächste übernehmen sollten, aber überall gibt es auch **spezifische** Dinge, auf die Sie vorbereitet sein sollten. Jetzt ist es Zeit, einen Blick darauf zu werfen, was Sie an einem **bestimmten Projekt** gelernt haben und wohin Sie sich als Nächstes wenden solten, um noch **mehr zu lernen.**



Einen Softwareentwicklungsprozess fixieren 418

Ein guter Prozess liefert gute Software aus 419

Förmliche Kleidung erforderlich ... 424

Einige zusätzliche Ressourcen ... 426

Mehr Wissen = bessere Prozesse 427

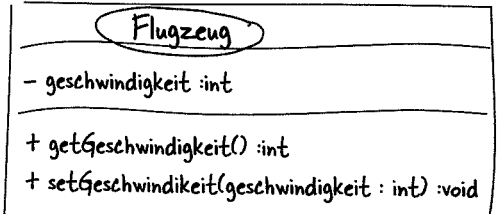
Werkzeuge für Ihre Softwareentwickler-Kiste 428

Was übrig bleibt

A Die Top Five der Themen (die wir nicht behandelt haben)

Schon mal das Gefühl gehabt, dass irgendetwas fehlt? Das kennen wir ... Gerade als Sie dachten, Sie wären fertig ... taucht was Neues auf. Wir konnten Sie einfach nicht entlassen, ohne noch ein paar Dinge zu erwähnen, die ins restliche Buch einfach nicht mehr gepasst haben. Zumindest nicht, wenn Sie dieses Buch immer noch mit sich herumtragen können möchten, ohne dazu einen Leiterwagen hinter sich herziehen zu müssen. Werfen Sie also einen Blick auf die Dinge, die Ihnen (immer noch) fehlen könnten.

1. UML-Klassendiagramme 434
2. Sequenzdiagramme 436
3. User-Stories und Anwendungsfälle 438
4. System-Tests vs. Unit-Tests 440
5. Refactoring 441



Techniken und Prinzipien

B Werkzeuge für den erfahrenen Softwareentwickler

Haben Sie sich auch schon gewünscht, es gäbe all diese tollen Werkzeuge und Techniken irgendwo an einem Platz versammelt? Nun, hier kommt eine Zusammenfassung aller Entwicklungstechniken und -prinzipien, die wir in diesem Buch behandelt haben. Schauen Sie sich alle noch einmal an und überprüfen Sie, ob Sie noch wissen, was sie im Einzelnen bedeuten. Vielleicht können Sie sich diese Seiten auch ausschneiden und damit Ihre Projekttafel garnieren, so dass sie jeder während Ihrer täglichen Stand-up-Meetings sehen kann.

Entwicklungstechniken 444

Entwicklungsprinzipen 446