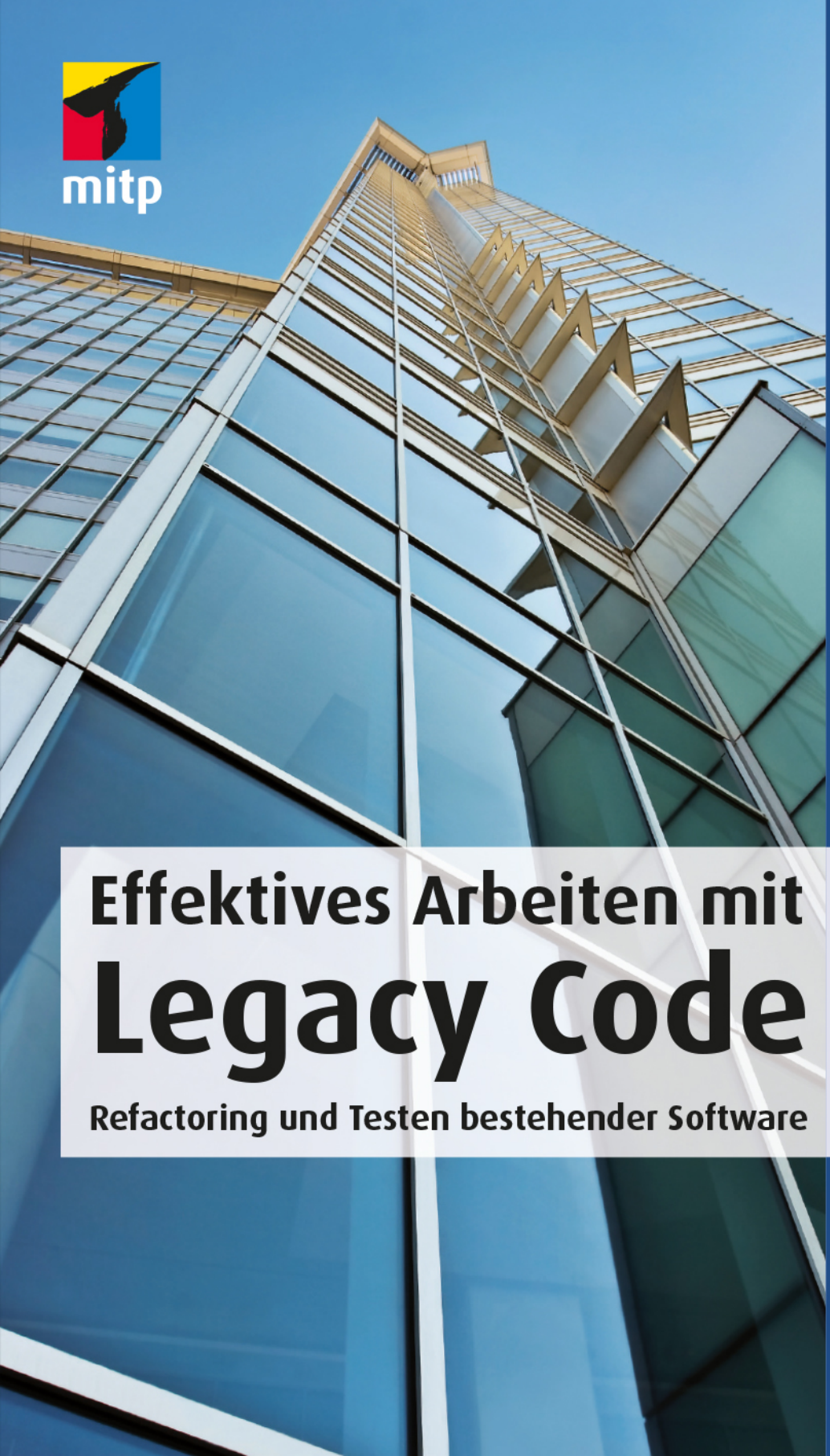

mitp

Michael C. Feathers

# Effektives Arbeiten mit Legacy Code

## Refactoring und Testen bestehender Software

## Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Michael Feathers

# Effektives Arbeiten mit Legacy Code

## Refactoring und Testen bestehender Software

Übersetzung aus dem Amerikanischen
von Reinhard Engel

mitp

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

ISBN 978-3-95845-902-1
1. Auflage 2011

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

© 2011 mitp Verlags GmbH & Co. KG, Frechen

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

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

Authorized translation from the English language edition, entitled *Working Effectively with Legacy Code* by Feathers, Michael; published by Pearson Education, Inc., publishing as Prentice Hall PTR, Copyright © 2005. Pearson Education, Inc. Tenth Printing, May 2009

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. GERMAN language edition published by mitp, eine Marke der Verlagsgruppe Hüthig Jehle Rehm GmbH, Copyright © 2011.

Lektorat: Sabine Schulz
Sprachkorrektorat: Petra Heubach-Erdmann
Satz: III-satz, Husby, www.drei-satz.de
Cover: © Lux2008 – fotolia.de

*Für Ann, Deborah und Ryan,*
*die strahlenden Zentren meines Lebens.*
– Michael

# Inhaltsverzeichnis


**Vorwort** . . . . . 13

**Geleitwort** . . . . . 15

**Danksagungen** . . . . . 21

**Einführung – Wie man dieses Buch lesen sollte** . . . . . 23

**Teil I Wie Wandel funktioniert** . . . . . 25

1 **Software ändern** . . . . . 27
1.1 Vier Gründe, Software zu ändern . . . . . 27
1.2 Riskante Änderungen . . . . . 31

2 **Mit Feedback arbeiten** . . . . . 33
2.1 Was sind Unit-Tests? . . . . . 36
2.2 Higher-Level-Tests . . . . . 39
2.3 Testabdeckung . . . . . 39
2.4 Der Algorithmus zur Änderung von Legacy Code . . . . . 42

3 **Überwachung und Trennung** . . . . . 45
3.1 Kollaborateure simulieren . . . . . 47

4 **Das Seam-Modell** . . . . . 53
4.1 Ein riesiges Blatt mit Text . . . . . 53
4.2 Seams . . . . . 54
4.3 Seam-Arten . . . . . 57

5 **Tools** . . . . . 69
5.1 Automatisierte Refactoring-Tools . . . . . 69
5.2 Mock-Objekte . . . . . 71
5.3 Unit-Test-Harnische . . . . . 72
5.4 Allgemeine Test-Harnische . . . . . 77

**Teil II Software ändern** ... 79

**6 Ich habe nicht viel Zeit und ich muss den Code ändern** ... 81
6.1 Sprout Method ... 83
6.2 Sprout Class ... 87
6.3 Wrap Method ... 91
6.4 Wrap Class ... 95
6.5 Zusammenfassung ... 100

**7 Änderungen brauchen eine Ewigkeit** ... 101
7.1 Verständlichkeit ... 101
7.2 Verzögerungszeit ... 102
7.3 Dependencies aufheben ... 103
7.4 Zusammenfassung ... 108

**8 Wie füge ich eine Funktion hinzu?** ... 109
8.1 Test-Driven Development (TDD) ... 110
8.2 Programming by Difference ... 116
8.3 Zusammenfassung ... 125

**9 Ich kann diese Klasse nicht in einen Test-Harnisch einfügen** ... 127
9.1 Der Fall des irritierenden Parameters ... 127
9.2 Der Fall der verborgenen Dependency ... 134
9.3 Der Fall der verketteten Konstruktionen ... 138
9.4 Der Fall der irritierenden globalen Dependency ... 140
9.5 Der Fall der schrecklichen Include-Dependencies ... 148
9.6 Der Fall der Zwiebel-Parameter ... 152
9.7 Der Fall des Alias-Parameters ... 154

**10 Ich kann diese Methode nicht in einem Test-Harnisch ausführen** ... 159
10.1 Der Fall der verborgenen Methode ... 159
10.2 Der Fall der »hilfreichen« Sprachfunktion ... 163
10.3 Der Fall des nicht erkennbaren Nebeneffekts ... 166

**11 Ich muss eine Änderung vornehmen. Welche Methoden sollte ich testen?** ... 173
11.1 Effekte analysieren ... 173
11.2 Vorwärtsanalyse (Reasoning Forward) ... 179
11.3 Effektfortpflanzung (Effect Propagation) ... 184
11.4 Tools für Effektanalysen ... 186
11.5 Von der Effektanalyse lernen ... 188
11.6 Effektskizzen vereinfachen ... 189

**12 Ich muss in einem Bereich vieles ändern. Muss ich die Dependencies für alle beteiligten Klassen aufheben?** ..... 193
12.1 Abfangpunkte ..... 194
12.2 Ein Design mit Einschnürpunkten beurteilen ..... 201
12.3 Fallen bei Einschnürpunkten ..... 203

**13 Ich muss etwas ändern, weiß aber nicht, welche Tests ich schreiben soll** ..... 205
13.1 Charakterisierungs-Tests ..... 206
13.2 Klassen charakterisieren ..... 209
13.3 Gezielt testen ..... 210
13.4 Eine Heuristik für das Schreiben von Charakterisierungs-Tests ..... 215

**14 Dependencies von Bibliotheken bringen mich um** ..... 217

**15 Meine Anwendung besteht nur aus API-Aufrufen** ..... 219

**16 Ich verstehe den Code nicht gut genug, um ihn zu ändern** ..... 227
16.1 Notizen/Skizzen ..... 228
16.2 Listing Markup ..... 229
16.3 Scratch Refactoring ..... 230
16.4 Ungenutzten Code löschen ..... 231

**17 Meine Anwendung hat keine Struktur** ..... 233
17.1 Die Geschichte des Systems erzählen ..... 234
17.2 Naked CRC ..... 238
17.3 Conversation Scrutiny ..... 241

**18 Der Test-Code ist im Weg** ..... 243
18.1 Konventionen für Klassennamen ..... 243
18.2 Der Speicherort für Tests ..... 244

**19 Mein Projekt ist nicht objektorientiert. Wie kann ich es sicher ändern?** ..... 247
19.1 Ein einfacher Fall ..... 248
19.2 Ein schwieriger Fall ..... 248
19.3 Neues Verhalten hinzufügen ..... 252
19.4 Die Objektorientierung nutzen ..... 255
19.5 Es ist alles objektorientiert ..... 258

**20 Diese Klasse ist zu groß und soll nicht noch größer werden** ..... 261
20.1 Aufgaben erkennen ..... 264
20.2 Andere Techniken ..... 278

20.3 Die nächsten Schritte ... 278
20.4 Nach dem Extrahieren von Klassen ... 281

**21 Ich ändere im ganzen System denselben Code** ... 283
21.1 Erste Schritte ... 286

**22 Ich muss eine Monster-Methode ändern und kann keine Tests dafür schreiben** ... 301
22.1 Spielarten von Monstern ... 301
22.2 Monster mit automatischer Refactoring-Unterstützung zähmen ... 306
22.3 Die Herausforderung des manuellen Refactorings ... 308
22.4 Strategie ... 316

**23 Wie erkenne ich, dass ich nichts kaputtmache?** ... 319
23.1 Hyperaware Editing ... 319
23.2 Single-Goal Editing ... 321
23.3 Preserve Signatures ... 322
23.4 Lean on the Compiler ... 325

**24 Wir fühlen uns überwältigt. Es wird nicht besser.** ... 329

**Teil III Techniken zur Aufhebung von Dependencies** ... 333

**25 Techniken zur Aufhebung von Dependencies** ... 335
25.1 Adapt Parameter ... 335
25.2 Break Out Method Object ... 339
25.3 Definition Completion ... 345
25.4 Encapsulate Global References ... 347
25.5 Expose Static Method ... 353
25.6 Extract and Override Call ... 356
25.7 Extract and Override Factory Method ... 358
25.8 Extract and Override Getter ... 360
25.9 Extract Implementer ... 363
25.10 Extract Interface ... 368
25.11 Introduce Instance Delegator ... 374
25.12 Introduce Static Setter ... 376
25.13 Link Substitution ... 382
25.14 Parameterize Constructor ... 383
25.15 Parameterize Method ... 386

25.16 Primitivize Parameter . . . . . 388
25.17 Pull Up Feature . . . . . 390
25.18 Push Down Dependency . . . . . 394
25.19 Replace Function with Function Pointer . . . . . 397
25.20 Replace Global Reference with Getter . . . . . 400
25.21 Subclass and Override Method . . . . . 402
25.22 Supersede Instance Variable . . . . . 405
25.23 Template Redefinition . . . . . 409
25.24 Text Redefinition . . . . . 412

**A Refactoring** . . . . . 415
A.1 Extract Method . . . . . 415

**B Glossar** . . . . . 421

**Stichwortverzeichnis** . . . . . 423

# Vorwort

*»... damit fing es an ...«*

In der Einführung zu diesem Buch verwendet Michael Feathers diesen Ausdruck, um den Beginn seiner Leidenschaft für Software zu beschreiben.

*»... damit fing es an ...«*

Kennen Sie dieses Gefühl? Erinnern Sie sich an den einen Moment in Ihrem Leben, über den Sie sagen könnten: »... damit fing es an ...«? Gab es einen einzigen Moment, der den Lauf Ihres Lebens änderte und schließlich dazu führte, dass Sie zu diesem Buch gegriffen haben und begannen, dieses Vorwort zu lesen?

Ich war in der sechsten Klasse, als ich einen solchen Moment erlebte. Ich interessierte mich für Naturwissenschaften, den Weltraum und alles Technische. Meine Mutter hatte in einem Katalog einen Plastikcomputer entdeckt und für mich bestellt. Er hieß *Digi-Comp I*. Vierzig Jahre später hat dieser kleine Plastikcomputer auf meinem Bücherregal einen Ehrenplatz. Er war der Katalysator, an dem sich meine lebenslange Leidenschaft für Software entzündete. Er vermittelte mir eine erste Ahnung davon, welche Freude das Schreiben von Programmen machen kann, die für Menschen Probleme lösen. Er bestand nur aus drei S-R-Flip-Flops und sechs Und-Gates aus Plastik, aber das reichte aus – er erfüllte seinen Zweck. Damit begann es ... – für mich.

Aber meine Freude wurde bald getrübt, als ich erkannte, dass Softwaresysteme fast immer in einem Chaos enden. Was als kristallklares Design im Geist der Programmierer entstanden war, verrottete im Laufe der Zeit wie ein Stück verdorbenes Fleisch. Das hübsche kleine System, das wir im letzten Jahr erstellt haben, entwickelte sich im nächsten Jahr in einen schrecklichen Morast aus verschlungenen Funktionen und Variablen.

Warum passiert das? Warum verrotten Systeme? Warum können sie nicht sauber bleiben? Manchmal schieben wir die Schuld auf unsere Kunden. Manchmal beschuldigen wir sie, die Anforderungen zu ändern. Wir trösten uns mit dem Glauben, das Design wäre schon in Ordnung gewesen, wären die Kunden nur mit dem zufrieden gewesen, was sie ihrer Aussage nach brauchten. Der Kunde ist selber schuld, wenn er seine Anforderungen an uns ändert.

Na ja, falls Sie es noch nicht wissen: *Anforderungen ändern sich*. Designs, die nicht flexibel auf Änderungen der Anforderungen reagieren können, sind per se schlechte Designs. Kompetente Software-Entwickler wollen Designs erstellen, die Änderungen tolerieren.

Dieses Problem scheint unlösbar schwierig zu sein. Tatsächlich ist es so schwierig, dass fast jedes jemals produzierte System langsam und kräftezehrend verrottet. Diese Verrottung ist so weit verbreitet, dass wir für verrottete Programme einen besonderen Begriff geprägt haben: *Legacy Code*.

*Legacy Code* – ein Begriff, der Programmierer abstößt. Er ruft Bilder von einem unergründlichen Sumpf mit verschlungenem Wurzelwerk, Blutegeln in trüben Gewässern und sirrenden Stechmücken hervor. Es riecht nach Verfall, Moder, Schleim und Verwesung. Auch wenn unsere erste Freude am Programmieren überschäumend gewesen sein mag, reicht das Elend beim Umgang mit Legacy Code oft aus, um diese Begeisterung zu ersticken.

Viele haben versucht, Methoden zu entwickeln, um zu *verhindern*, dass aus Code überhaupt Legacy Code werden kann. Wir haben Bücher über Prinzipien, Patterns und Verfahren geschrieben, die Programmierern helfen können, ihre Systeme sauber zu halten. Aber Michael Feathers hat etwas erkannt, das vielen anderen entgangen ist. Vorbeugung ist nicht perfekt. Selbst das disziplinierteste Entwicklungsteam, das die besten Prinzipien und Patterns beherrscht und die besten Verfahren anwendet, produziert immer wieder einmal chaotische Systeme. Der Morast wächst immer noch. Es reicht nicht aus, Verrottung zu verhindern – Sie müssen diesen Prozess *umkehren* können.

Darum geht es in diesem Buch: die Umkehrung der Verrottung. Wie kann man aus einem verschlungenen, undurchschaubaren, verworrenen System langsam und allmählich Schritt für Schritt ein einfaches, sauber strukturiertes System mit einem makellosen Design machen? Wie kann man die Entropie umkehren?

Bevor Sie sich von Ihrer Begeisterung fortreißen lassen, möchte ich Sie warnen: Verrottung umzukehren, ist nicht leicht und braucht Zeit. Die Techniken, Patterns und Tools, die Feathers in diesem Buch präsentiert, sind wirksam, aber sie erfordern Arbeit, Zeit, Ausdauer und *Sorgfalt*. Dieses Buch ist keine Silberkugel. Es sagt Ihnen nicht, wie Sie den ganzen Mist, der sich in Ihren Systemen angesammelt hat, über Nacht beseitigen können, sondern beschreibt einen Satz von Disziplinen, Konzepten und Einstellungen, den Sie für den Rest Ihrer Karriere mit sich tragen werden und der *Ihnen helfen wird, aus Systemen, die sich im Laufe der Zeit verschlechtern, Systeme zu machen, die sich im Laufe der Zeit verbessern*.

Robert C. Martin
29. Juni 2004

# Geleitwort

Erinnern Sie sich noch an das erste Programm, das Sie geschrieben haben? Ich erinnere mich an meins. Es war ein kleines Grafikprogramm, das ich auf einem frühen PC geschrieben habe. Ich begann später als die meisten meiner Freunde mit dem Programmieren. Sicher, Computer waren mir von klein auf bekannt; und ich erinnere mich daran, wie nachhaltig mich ein Minicomputer beeindruckt hat, den ich in einem Büro sah. Aber jahrelang hatte ich keine Gelegenheit, mich an einen Computer zu setzen. Später in meiner Teenagerzeit kaufte einer meiner Freunde einige der ersten TRS-80s. Ich war interessiert, aber auch ein wenig besorgt. Ich wusste, dass ich dem Computer verfallen würde, sollte ich anfangen, mit ihm zu spielen. Er sah einfach zu verlockend aus. Ich weiß nicht, woher ich mich so gut kannte, aber ich hielt mich zurück. Später im College hatte ein Zimmergenosse einen Computer; und ich kaufte mir einen C-Compiler, um mir das Programmieren beizubringen. Damit fing es an. Ich blieb jede Nacht auf, um dieses und jenes auszuprobieren und den Quellcode des Emacs-Editors zu studieren, der dem Compiler beilag. Es machte süchtig, es war anspruchsvoll, und ich liebte es.

Ich hoffe, Sie haben ähnliche Erfahrungen gemacht und haben die reine Freude erlebt, Dinge auf einem Computer zum Laufen zu bringen. Fast jeder Programmierer, den ich frage, kennt dieses Gefühl. Diese Freude gehört zu den Motiven, die uns diese Arbeit haben wählen lassen; aber wohin ist sie im Alltag verschwunden?

Vor einigen Jahren rief ich abends nach erledigter Arbeit meinen Freund Erik Meade an. Ich wusste, dass Erik gerade einen Beratungsauftrag mit einem neuen Team angenommen hatte, und fragte ihn deshalb: »Wie läuft's?« Er sagte: »Nicht zu glauben, die schreiben Legacy Code.« Dies war einer der wenigen Male in meinem Leben, bei denen mich eine Äußerung eines Kollegen wie ein unerwarteter Schlag erwischte. Ich fühlte ihn direkt in der Magengrube. Erik hatte das Gefühl punktgenau ausgedrückt, das mich oft beschleicht, wenn ich zum ersten Mal mit einem fremden Team zu tun habe. Seine Mitglieder bemühen sich redlich, aber letztlich schreiben viele Menschen einfach nur Legacy Code. Die Gründe? Vielleicht ist der Termindruck zu stark. Vielleicht wiegt die Last des überkommenden Codes zu schwer. Vielleicht ist einfach nur kein besserer Code vorhanden, mit dem sie ihre Anstrengungen vergleichen könnten.

Was ist Legacy Code? Ich habe diesen Terminus bis jetzt undefiniert verwendet. Betrachten wir die strenge Definition: *Legacy Code ist Code, den wir von jemand anderem übernommen haben*. Vielleicht hat unser Unternehmen Code von einem anderen Unternehmen übernommen; vielleicht sind die Mitarbeiter des ursprünglichen Teams zu anderen Projekten abgewandert. Legacy Code ist Code eines anderen. Aber im Programmiererjargon bedeutet der Terminus viel mehr als das. Der Terminus *Legacy Code* hat im Laufe der Zeit zusätzliche Bedeutungen und mehr Gewicht angenommen.

Was denken Sie, wenn Sie den Terminus *Legacy Code* hören? Denken Sie wie ich an eine verworrene, unverständliche Struktur, an Code, den Sie ändern müssen, den Sie aber nicht wirklich verstehen? Denken Sie an schlaflose Nächte, in denen Sie versuchen, Funktionen hinzuzufügen, die leicht hinzuzufügen sein sollten? Fühlen Sie sich entmutigt? Haben Sie den Eindruck, der Code sei den Mitgliedern Ihres Teams so über, dass ihnen alles egal ist und sie den Code am liebsten sterben sähen. Ein Teil von Ihnen fühlt sich sogar schlecht bei dem Gedanken, den Code zu verbessern. Er scheint Ihre Anstrengungen nicht zu verdienen. Diese Definition von Legacy Code hat nichts damit zu tun, wer ihn geschrieben hat. Die Qualität von Code kann durch viele Faktoren verschlechtert werden; und viele haben nichts damit zu tun, ob der Code von einem anderen Team geschrieben wurde.

In der Branche wird *Legacy Code* oft salopp zur Bezeichnung von Code verwendet, den man nicht versteht und der schwer zu ändern ist. Aber im Laufe der Jahre, in denen ich verschiedenen Teams geholfen habe, ernste Code-Probleme zu beseitigen, habe ich eine andere Definition entwickelt.

Für mich ist *Legacy Code* ganz einfach Code ohne Tests. Mit dieser Definition habe ich mir einigen Kummer eingehandelt. Was haben Tests damit zu tun, ob Code schlecht ist? Darauf hab ich eine unkomplizierte Antwort, die ich in diesem Buch immer wieder aus verschiedenen Blickwinkeln darstelle:

> Code ohne Tests ist schlechter Code. Es spielt keine Rolle, wie gut er geschrieben ist; es spielt keine Rolle, wie schön oder objektorientiert oder gut eingekapselt er ist. Mit Tests können wir das Verhalten unseres Codes schnell und verifizierbar ändern. Ohne Tests wissen wir nicht wirklich, ob unser Code besser oder schlechter wird.

Vielleicht halten Sie das für streng. Was ist mit sauberem Code? Reicht es nicht aus, wenn eine Code-Basis sehr sauber und gut strukturiert ist? Bitte verstehen Sie mich nicht falsch. Ich liebe sauberen Code. Ich liebe ihn mehr als die meisten Menschen, die ich kenne; doch sauberer Code ist zwar gut, aber allein nicht gut genug. Teams gehen erhebliche Risiken ein, wenn sie große Änderungen ohne Tests durchführen wollen. Es ähnelt der Hochseilartistik ohne Sicherheitsnetz. Es erfordert unglaubliches Können und ein klares Verständnis, was bei jedem Schritt

passieren wird. Die Auswirkung der Änderung einiger Variablen genau zu überschauen, ist oft mit der Gewissheit vergleichbar, dass Sie nach einem Salto von einem anderen Artisten an den Armen aufgefangen werden. Wenn Sie in einem Team an Code arbeiten, der derartig übersichtlich ist, sind Sie in einer besseren Position als die meisten Programmierer. Bei meiner Arbeit sind mir Teams mit einem solchen Code selten begegnet. Sie scheinen eine statistische Anomalie zu sein. Und wissen Sie was? Wenn sie nicht mit Testunterstützung arbeiten, brauchen sie für Code-Änderungen immer noch länger als Teams, die systematisch testen.

Es stimmt: Teams werden besser und schreiben von Anfang an klareren Code; aber es dauert sehr lange, bis älterer Code klarer wird. In vielen Fällen wird dieses Ziel nie ganz erreicht. Deshalb habe ich kein Problem damit, Legacy Code als Code ohne Tests zu definieren. Es ist eine brauchbare Arbeitsdefinition, die auf eine Lösung verweist.

Obwohl ich bis jetzt ausführlich auf Tests eingegangen bin, handelt dieses Buch nicht vom Testen. In diesem Buch geht es darum, eine beliebige Code-Basis erfolgssicher zu ändern. In den folgenden Kapiteln beschreibe ich Techniken, mit denen Sie Code verstehen, in eine Testumgebung integrieren, refaktorisieren und funktional erweitern können.

Sie werden beim Lesen dieses Buches bemerken, dass es hier nicht um »schönen« Code geht. Die Beispiele in diesem Buch sind konstruiert, weil ich mit Kunden unter einer Geheimhaltungsvereinbarung arbeite. Aber in vielen Beispielen bemühe ich mich, den »Geist« des Codes wiederzugeben, der mir im Feld begegnet ist. Ich will nicht behaupten, alle Beispiele wären repräsentativ. Sicher gibt es im Feld Oasen mit großartigem Code, aber, ganz ehrlich, es gibt dort auch Code-Basen, die viel schlechter als alles sind, was ich in diesem Buch als Beispiel verwenden kann. Abgesehen von der Vertraulichkeit konnte ich einfach keinen derartigen Code in dieses Buch einfügen, ohne Sie zu Tode zu langweilen und wichtige Punkte in einem Morast von Details zu versenken. Folglich sind viele Beispiele relativ kurz. Wenn Sie ein solches Beispiel sehen und denken: »Der hat ja keine Ahnung – meine Methoden sind viel länger und viel schlechter«, nehmen Sie bitte meinen zugehörigen Ratschlag für bare Münze und prüfen Sie seine Anwendbarkeit, auch wenn das Beispiel einfacher zu sein scheint.

Die hier vorgestellten Techniken sind mit erheblich umfangreicherem Code getestet worden. Die Beispiele sind nur wegen des Buchformats kürzer. Insbesondere können Sie Ellipsen (`...`) in einem Code-Fragment wie folgt interpretieren: »Fügen Sie hier 500 Zeilen mit hässlichem Code ein.« Ein Beispiel:

```
m_pDispatcher->register(listener);
...
m_nMargins++;
```

In diesem Buch geht es nicht nur nicht um »schönen« Code, sondern noch weniger um »schönes« Design. Gutes Design sollte zu den Zielen jedes Programmierers gehören; aber bei *Legacy Code* nähern wir uns diesem Ziel schrittweise. In einigen Kapiteln beschreibe ich Methoden, wie man eine vorhandene Code-Basis mit neuem Code erweitern und dabei gute Designprinzipien berücksichtigen kann. Sie können in eine Legacy-Code-Basis Bereiche mit qualitativ hochwertigem Code einführen, sollten aber nicht überrascht sein, wenn bei einigen Änderungen andere Teile des Codes etwas hässlicher werden. Diese Arbeit gleicht einem chirurgischen Eingriff. Wir müssen Einschnitte vornehmen, und wir müssen durch die Eingeweide gehen und gewisse ästhetische Überlegungen beiseitelassen. Könnten die Hauptorgane und Eingeweide dieses Patienten in besserer Verfassung sein? Ja. Vergessen wir deshalb das anstehende Problem, nähen ihn wieder zu und raten ihm zu einer besseren Ernährung und regelmäßigem Training? Dies könnten wir tun; doch hier und jetzt müssen wir den Patienten nehmen, wie er ist, die Mängel beseitigen und ihn gesünder machen. Vielleicht wird er nie um olympische Medaillen kämpfen, aber wir dürfen das »Beste« nicht zum Feind des »Besseren« machen. Die Code-Basis kann gesünder und leichter handhabbar werden. Wenn sich ein Patient ein wenig besser fühlt, ist oft der geeignete Zeitpunkt, ihn zu einem gesünderen Lebensstil zu führen. Genau dies möchten wir mit *Legacy Code* erreichen. Wir versuchen, die dringenden Probleme zu beheben und dann den Code schrittweise zu verbessern, indem wir Änderungen erleichtern. Wenn es uns gelingt, diese Vorgehensweise fest in einem Team zu etablieren, wird auch das Design besser.

Die hier beschriebenen Techniken habe ich im Laufe der Jahre entdeckt oder von Kollegen gelernt, als ich bei meiner Arbeit mit Kunden versuchte, die Kontrolle über widerspenstige Code-Basen zu gewinnen. Dieser Legacy-Code-Schwerpunkt bildete sich zufällig heraus. Meine Arbeit bei Object Mentor bestand anfangs hauptsächlich darin, Teams mit ernsten Problemen bei der Entwicklung ihrer Fähigkeiten und der Verbesserung ihrer Interaktionen so weit zu unterstützen, dass sie regelmäßig qualitativ hochwertigen Code abliefern konnten. Wir verwendeten oft Extreme-Programming-Verfahren, um den Teams zu helfen, ihre Arbeit zu kontrollieren, intensiv zusammenzuarbeiten und Ergebnisse zu liefern. Oft glaube ich, Extreme Programming (XP) ist weniger eine Methode der Software-Entwicklung, sondern eher eine Methode zur Bildung funktionierender Teams, die nebenbei auch noch im Abstand von zwei Wochen großartige Software abliefern.

Doch von Anfang an gab es ein Problem. Viele der ersten XP-Projekte waren »Greenfield«-Projekte. Meine Kunden verfügten über umfangreiche Code-Basen, und sie hatten Probleme. Sie brauchten eine Methode, um ihre Arbeit in den Griff zu bekommen und termingerecht abzuliefern. Im Laufe der Zeit stellte ich fest, dass ich mit meinen Kunden immer wieder dieselben Probleme behandelte. Dieser Eindruck verdichtete sich bei einer Arbeit mit einem Team der Finanzbranche.

Bevor ich dazukam, hatte man erkannt, dass Unit-Testing eine beeindruckende Sache war, aber die Tests, die man ausführte, testeten das komplette Szenarium, griffen wiederholt auf eine Datenbank zu und führten umfangreiche Code-Fragmente aus. Die Tests waren schwer zu schreiben, und das Team führte sie nicht oft aus, weil sie so lange liefen. Als ich mich mit dem Team zusammensetzte, um die Dependencies (Abhängigkeiten) aufzulösen und den Code in kleineren Einheiten zu testen, hatte ich ein schreckliches Déjà-vu-Gefühl. Es schien, dass ich diese Art von Arbeit mit jedem Team, das ich traf, erneut leisten musste, und es war eine Art von Arbeit, über die niemand wirklich gerne nachdenkt. Es handelte sich um eine Drecksarbeit, die man erledigt, wenn man die Kontrolle über seinen Code gewinnen will und weiß, was man tun muss. Damals beschloss ich, dass es sich wirklich lohnen würde, über die Methoden zur Lösung dieser Probleme nachzudenken und sie aufzuschreiben, um Teams bei der Verbesserung ihrer Code-Basis zu helfen.

Eine Anmerkung zu den Beispielen: Ich habe Beispiele in mehreren verschiedenen Programmiersprachen verwendet. Die meisten Beispiele sind in Java, C++ und C geschrieben. Ich habe Java ausgewählt, weil diese Sprache weit verbreitet ist, und ich habe C++ eingeschlossen, weil diese Sprache in einer Legacy-Umgebung einige besondere Herausforderungen präsentiert. Ich habe C ausgewählt, weil es viele Probleme in prozeduralem Legacy Code hervorhebt. Zusammen decken diese Sprachen einen großen Teil des Spektrums der Legacy-Code-Probleme ab. Doch auch wenn Sie mit anderen Sprachen arbeiten, sollten Sie sich die Beispiele anschauen. Viele der behandelten Techniken können auch in anderen Sprachen, wie etwa Delphi, Visual Basic, COBOL oder FORTRAN, verwendet werden.

Ich hoffe, dass Ihnen die Techniken in diesem Buch bei Ihrer Arbeit helfen und dazu beitragen, die Freude am Programmieren wiederzufinden. Programmieren kann eine sehr lohnenswerte und erfreuliche Arbeit sein. Wenn Sie dieses Gefühl bei Ihrer Alltagsarbeit nicht haben, hoffe ich, dass Ihnen die Techniken in diesem Buch helfen werden, dieses Gefühl zu entdecken und in Ihrem Team zu kultivieren.

# Danksagungen

Vor allem schulde ich meiner Frau, Ann, und meinen Kindern, Deborah und Ryan, einen tief empfundenen Dank. Ihre Liebe und ihre Unterstützung machten dieses Buch und die vorhergehende Zeit des Lernens möglich. Außerdem möchte ich »Uncle Bob« Martin, dem Chef und Gründer von Object Mentor, danken. Sein strenger pragmatischer Ansatz zu Entwicklung und Design und seine Trennung des Kritischen vom Belanglosen gaben mir vor etwa einem Jahrzehnt Halt, als ich in einer Woge unrealistischer Ratschläge zu ertrinken schien. Und danke, Bob, dass du mir die Gelegenheit verschafft hast, in den vergangenen fünf Jahren mehr Code zu sehen und mit mehr Menschen zu arbeiten, als ich jemals für möglich gehalten hätte.

Ich muss auch Kent Beck, Martin Fowler, Ron Jeffries und Ward Cunningham für ihre gelegentlichen Ratschläge und ihre Lehren über Teamarbeit, Design und Programmieren danken. Mein besonderer Dank richtet sich an alle Menschen, die die Entwürfe lasen. Die offiziellen Gutachter waren Sven Gorts, Robert C. Martin, Erik Meade und Bill Wake; die inoffiziellen Gutachter waren Dr. Robert Koss, James Grenning, Lowell Lindstrom, Micah Martin, Russ Rufer und die Silicon Valley Patterns Group sowie James Newkirk.

Dank auch an die Gutachter der allerersten Entwürfe, die ich ins Internet stellte. Ihr Feedback hat die Richtung dieses Buches erheblich beeinflusst, nachdem ich sein Format umstrukturiert hatte. Ich entschuldige mich im Voraus bei allen, die ich vielleicht ausgelassen haben. Die ersten Gutachter waren: Darren Hobbs, Martin Lippert, Keith Nicholas, Phlip Plumlee, C. Keith Ray, Robert Blum, Bill Burris, William Caputo, Brian Marick, Steve Freeman, David Putman, Emily Bache, Dave Astels, Russel Hill, Christian Sepulveda und Brian Christopher Robinson.

Dank auch an Joshua Kerievsky, der wesentliche Anmerkungen zu einem der ersten Entwürfe beitrug, und Jeff Langr, der meine ganzen Schreibprozesse mit seinen Ratschlägen und zeitnahen Kritiken begleitete.

Die Gutachter halfen mir, meinen Entwurf erheblich zu glätten; doch sollte das Buch noch Fehler enthalten, bin ich dafür verantwortlich.

Dank an Martin Fowler, Ralph Johnson, Bill Opdyke, Don Roberts und John Brant für ihre Arbeit über das Refactoring. Sie war mir eine Inspiration.

Besonderen Dank schulde ich auch Jay Packlick, Jacques Morel und Kelly Mower von Sabre Holdings und Graham Wright von Workshare Technology für Unterstützung und Feedback.

Besonderen Dank schulde ich auch Paul Petralia, Michelle Vincenti, Lori Lyons, Krista Hansing und dem Rest des Teams bei Prentice-Hall. Danke, Paul, für die Hilfe und Ermutigung, die dieser Erstautor brauchte.

Mein besonderer Dank gilt auch Gary und Joan Feathers, April Roberts, Dr. Raimund Ege, David Lopez de Quintana, Carlos Perez, Carlos M. Rodriguez und dem verstorbenen Dr. John C. Comfort für ihre Hilfe und Ermutigung im Laufe der vergangenen Jahre. Ich muss auch Brian Button für das Beispiel in Kapitel 21, *Ich ändere im ganzen System denselben Code*, danken. Er schrieb diesen Code in etwa einer Stunde, als wir zusammen einen Refactoring-Kursus entwickelten. Dieser Code ist heute eines meiner Lieblingsbeispiele in meinen Programmierkursen.

Besondere danke ich auch Jannick Top, dessen Instrumentalstück *De Futura* mich als Soundtrack während meiner letzten Wochen bei der Arbeit an diesem Buch begleitete.

Schließlich möchte ich allen danken, mit denen ich im Laufe der letzten Jahre zusammengearbeitet habe und deren Einsichten und Herausforderungen das Material in diesem Buch verbessert haben.

Michael Feathers

mfeathers@objectmentor.com
www.objectmentor.com
www.michaelfeathers.com

# Einführung – Wie man dieses Buch lesen sollte

Ich habe verschiedene Formate ausprobiert, bevor ich mich für das gegenwärtige Format dieses Buches entschied. Viele der verschiedenen Techniken und Verfahren, die beim Arbeiten mit Legacy Code nützlich sind, lassen sich isoliert nur schwer erklären. Die einfachsten Änderungen sind oft einfacher, wenn Sie Andockpunkte finden, Objekte simulieren und Dependencies mit einschlägigen Techniken aufheben. Schließlich kam ich zu dem Schluss, der Hauptinhalt des Buches (*Teil II, Software ändern*) ließe sich am besten durch die FAQ-Methode im FAQ-Format (FAQ = Frequently Asked Questions; häufig gestellte Fragen) erschließen. Weil besondere Techniken oft den Einsatz anderer Techniken erfordern, sind die FAQ-Kapitel stark verknüpft. Fast jedes Kapitel referenziert andere Kapitel und Abschnitte, in denen besondere Techniken und Refactorings beschrieben werden. Ich möchte mich entschuldigen, wenn Sie deshalb wild in diesem Buch hin und her blättern müssen, um Antworten auf Ihre Fragen zu finden; aber ich bin davon ausgegangen, dass Sie lieber blättern als das Buch von Deckel zu Deckel durchlesen würden, um die Arbeitsweise aller Techniken zu verstehen.

In *Software ändern* habe ich versucht, häufig gestellte Fragen zu beantworten, die bei der Legacy-Code-Arbeit auftauchen. Jedes Kapitel ist nach einem besonderen Problem benannt. Dadurch werden die Kapitelüberschriften ziemlich lang; aber hoffentlich können Sie so schnell einen Abschnitt finden, der Ihnen hilft, Ihr besonderes Problem zu lösen.

*Software ändern* wird von einem Satz einführender Kapitel (*Teil I, Wie Wandel funktioniert*) und einem Katalog von Refactorings eingerahmt, die bei Legacy-Code-Arbeit sehr nützlich sind (*Teil III, Techniken zur Aufhebung von Dependencies*). Bitte lesen Sie das einführende Kapitel, insbesondere Kapitel 4, *Das Seam-Model*. Diese Kapitel liefern den Kontext und die Nomenklatur für alle folgenden Techniken. Zusätzlich sollten Sie Termini, die nicht im Kontext beschrieben werden, im Glossar nachschlagen.

Die Refactorings in *Techniken zur Aufhebung von Dependencies* sind etwas Besonderes, da sie ohne Tests angewendet werden sollen. Sie dienen der Einrichtung von Tests. Ich rate Ihnen, jede einzelne Technik durchzulesen, damit Sie mehr Möglichkeiten kennen lernen, um Ihren Legacy Code zu zähmen.

# Teil I

## Wie Wandel funktioniert

**In diesem Teil:**


- **Kapitel 1**
  Software ändern . . . . . 27
- **Kapitel 2**
  Mit Feedback arbeiten . . . . . 33
- **Kapitel 3**
  Überwachung und Trennung . . . . . 45
- **Kapitel 4**
  Das Seam-Modell . . . . . 53
- **Kapitel 5**
  Tools . . . . . 69

Kapitel 1

# Software ändern

Code zu ändern, ist etwas Großartiges. Wir verdienen damit unseren Lebensunterhalt. Aber es gibt Methoden, Code zu ändern, die das Leben erschweren, und es gibt Methoden, die es erheblich erleichtern. In der Branche wurde nicht viel darüber geredet. Der Sache am nächsten kommt noch die Literatur über Refactoring. Ich glaube, wir sollten die Diskussion etwas breiter anlegen und überlegen, wie wir in den schlimmsten Situationen mit Code umgehen sollten. Zu diesem Zweck müssen wir uns zunächst näher mit der Mechanik von Änderungen befassen.

## 1.1 Vier Gründe, Software zu ändern

Der Einfachheit halber möchte ich vier Hauptgründe unterscheiden, Software zu ändern:

1. Eine Funktion hinzufügen
2. Einen Fehler beseitigen
3. Das Design verbessern
4. Die Nutzung von Ressourcen optimieren

### 1.1.1 Funktionen hinzufügen und Fehler beseitigen

Eine Funktion hinzuzufügen, scheint mir die unkomplizierteste Art von Änderungen zu sein. Die Software zeigt ein Verhalten, und der Anwender erwartet von dem System auch noch ein anderes Verhalten.

Angenommen, wir arbeiteten an einer Webanwendung und ein Manager teilte uns mit, das Unternehmenslogo solle nicht auf der linken, sondern auf der rechten Seite stehen. Wir sprechen mit ihm darüber und stellen fest, dass dies nicht ganz so einfach ist. Er möchte das Logo verschieben, aber erwartet auch andere Änderungen. Es soll beim nächsten Release animiert werden. Gehört dies in die Kategorie »Fehler beseitigen« oder »Neue Funktion hinzufügen«? Das hängt von Ihrem Standpunkt ab. Aus der Sicht des Kunden handelt es sich definitiv um die Beseitigung eines Problems. Vielleicht hat er die Webseite gesehen und ein Meeting mit Mitarbeitern seiner Abteilung veranstaltet; und sie haben beschlossen, das Logo an eine andere Stelle zu setzen und ein wenig mehr Funktionalität zu fordern. Auf der Sicht eines Entwicklers kann die Änderung als vollkommen neue

Funktion eingestuft werden. »Wenn die Abteilung einfach aufhören würde, ständig ihre Meinung zu ändern, wären wir jetzt fertig.« Aber in einigen Unternehmen wird eine Verschiebung eines Logos einfach als Beseitigung eines Fehlers gesehen, ungeachtet der Tatsache, dass das Team dafür umfangreiche neue Arbeit leisten muss.

Man könnte dies leicht als subjektive Einschätzung abtun. Für Sie ist dies ein zu beseitigender Fehler, für mich eine neue Funktion, also was? Leider müssen in vielen Unternehmen Korrekturen von Fehlern und Erweiterungen um neue Funktionen unterschiedlich überwacht und abgerechnet werden, weil es auch noch Verträge, Garantien oder Qualitätsinitiativen gibt. Zwar können wir endlos darüber diskutieren, ob wir Funktionen hinzufügen oder Fehler beheben, aber letztlich müssen Code und andere Artefakte geändert werden. Dieser Streit auf semantischer Ebene, was die Tätigkeit denn nun letztlich sei, maskiert etwas für uns technisch viel Wichtigeres: die Verhaltensänderung eines Systems. Und dabei bedeutet es einen großen Unterschied, ob neues Verhalten hinzugefügt oder altes geändert wird.

> Verhalten ist der wichtigste Aspekt von Software. Es ist der Grund, warum Anwender Software verwenden. Anwender lieben es, wenn wir Verhalten hinzufügen (vorausgesetzt, es leistet, was sie wirklich wollten), aber wenn wir Verhalten ändern oder entfernen, das sie benötigen (Fehler einführen), verlieren wir ihr Vertrauen.

Fügen wir in unserem Unternehmenslogo-Beispiel Verhalten hinzu? Ja; denn nach der Änderung wird das System ein Logo auf der rechten Seite anzeigen. Beseitigen wir Verhalten? Ja; denn es gibt kein Logo mehr auf der linken Seite.

Betrachten wir einen schwierigeren Fall. Angenommen, ein Kunde wolle ein Logo rechts auf einer Webseite anzeigen, aber es gäbe kein Logo auf der linken Seite, mit dem wir anfangen könnten. Ja; wir fügen Verhalten hinzu; aber entfernen wir auch Verhalten? Wurde an der Stelle, an der das Logo erscheinen soll, irgendetwas anderes angezeigt?

Ändern wir Verhalten, fügen wir Verhalten hinzu, oder beides?

Wir können eine Unterscheidung treffen, die für uns als Programmierer nützlicher ist. Wenn wir Code modifizieren müssen (und HTML zählt in diesem Fall als Code), könnten wir Verhalten ändern. Wenn wir nur Code hinzufügen und ihn aufrufen, fügen wir oft Verhalten hinzu. Betrachten wir ein anderes Beispiel, eine Methode einer Java-Klasse:

```
public class CDPlayer
{
   public void addTrackListing(Track track) {
```

```
        ...
    }
    ...
}
```

Die Klasse enthält eine Methode, mit der wir Track-Listings (etwa mit den Songs einer CD) hinzufügen können. Fügen wir eine Methode hinzu, mit der wir Track-Listings ersetzen können:

```
public class CDPlayer
{
    public void addTrackListing(Track track) {
        ...
    }

    public void replaceTrackListing(String name, Track track) {
        ...
    }
    ...
}
```

Haben wir mit dieser Methode neues Verhalten zu unserer Anwendung hinzugefügt oder haben wir Verhalten geändert? Weder noch. Eine Methode hinzuzufügen, ändert Verhalten erst, wenn die Methode irgendwie aufgerufen wird.

Ändern wir den Code erneut. Wir wollen einen neuen Button in die Benutzerschnittstelle des CD-Players einfügen und ihn mit der `replaceTrackListing`-Methode verknüpfen. Damit fügen wir das Verhalten hinzu, das wir in der `replaceTrackListing`-Methode spezifiziert haben; aber wir ändern auch Verhalten auf subtile Weise; denn die Benutzerschnittstelle wird mit diesem neuen Button etwas anders dargestellt. Möglicherweise dauert es eine Mikrosekunde länger, bis es komplett angezeigt wird. Es scheint fast unmöglich zu sein, Verhalten hinzuzufügen, ohne zugleich vorhandenes Verhalten bis zu einem gewissen Grad zu ändern.

### 1.1.2 Das Design verbessern

Das Design zu verbessern, ist eine weitere Art von Software-Änderung. Wir wollen die Software umstrukturieren, etwa damit sie wartungsfreundlicher wird, wobei ihr Verhalten im Allgemeinen bewahrt werden soll. Wird dabei Verhalten, vielleicht aus Versehen, entfernt, bezeichnen wir dies oft als Bug (Fehler). Einer der Hauptgründe, warum viele Programmierer nicht versuchen, das Design zu verbessern, liegt darin, dass dabei leicht Verhalten verloren gehen, beschädigt oder unerwünscht verändert werden kann.

Der Prozess, Design zu verbessern, ohne Verhalten zu ändern, wird als *Refactoring* bezeichnet. Es basiert auf der Idee, dass wir Software wartungsfreundlicher machen können, ohne ihr Verhalten zu ändern, wenn wir Tests schreiben, mit

denen wir kontrollieren, dass das vorhandene Verhalten nicht geändert wird, und in kleinen Schritten vorgehen, um dies nach jedem Schritt zu verifizieren. Entwickler säubern schon seit Jahren den Code vorhandener Systeme; aber erst in den letzten Jahren hat sich das Refactoring verbreitet. Es unterscheidet sich von allgemeinen Säuberungen darin, dass wir nicht einfach risikoarme Änderungen wie etwa eine Umformatierung von Quellcode oder invasive und riskante Dinge wie etwa das Umschreiben ganzer Code-Fragmente vornehmen, sondern dass wir eine Reihe kleiner struktureller Änderungen vornehmen und dabei von Tests unterstützt werden, die das Ändern des Codes erleichtern. Die Essenz des Refactorings besteht darin, Verhalten zu bewahren, während funktionale Änderungen Verhalten modifizieren.

### 1.1.3 Optimierung

Optimierung ähnelt dem Refactoring, verfolgt aber ein anderes Ziel. Bei beiden wird die Funktionalität nicht geändert, aber beim Refactoring wird die Programm-Struktur geändert, während bei der Optimierung die Nutzung von Ressourcen (Zeit, Speicherplatz usw.) verbessert wird.

### 1.1.4 Alles im Überblick

Doch ähnelt das Refactoring der Optimierung tatsächlich viel stärker als dem Hinzufügen von Funktionen oder der Beseitigung von Fehlern? Refactoring und Optimierung haben gemeinsam, dass die Funktionalität invariant bleibt, während etwas anderes geändert wird.

Im Allgemeinen können wir bei der Arbeit an einem System drei verschiedene Aspekte ändern: Struktur, Funktionalität und Ressourcenverbrauch.

Was ändert sich normalerweise und was bleibt im Wesentlichen konstant, wenn wir vier unserer verschiedenen Arten von Änderungen vornehmen (ja, oft ändern sich alle drei Aspekte, aber wir wollen das Typische betrachten):

| | Funktion hinzufügen | Fehler beseitigen | Refactoring | Optimierung |
|---|---|---|---|---|
| **Struktur** | Ändert sich | Ändert sich | Ändert sich | - |
| **Funktionalität** | Ändert sich | Ändert sich | - | - |
| **Ressourcenverbrauch** | - | - | - | Ändert sich |

Oberflächlich sehen sich Refactoring und Optimierung sehr ähnlich. Sie halten die Funktionalität invariant. Aber was passiert, wenn wir neue Funktionalität separat betrachten? Wenn wir eine Funktion hinzufügen, führen wir eine neue Funktionalität ein, aber ohne vorhandene Funktionalität zu ändern.