Vorwort zur ersten Auflage

»Refactoring« wurde ursprünglich in Smalltalk-Kreisen konzipiert. Es dauerte allerdings nicht lange, bis es auch den Weg in die Lager anderer Programmiersprachen fand. Da Refactoring ein integraler Bestandteil bei der Entwicklung von Frameworks ist, kommt dieser Ausdruck häufig zur Sprache, wenn sich »Frameworker« über ihr Handwerk unterhalten. Er wird verwendet, wenn sie ihre Klassenhierarchien überarbeiten und davon schwärmen, wie viele Codezeilen sie löschen konnten. Frameworker wissen, dass ein Framework nicht gleich beim ersten Versuch perfekt ist – vielmehr muss es sich aus ihrer stetig anwachsenden Erfahrung entwickeln. Sie wissen auch, dass Code häufiger gelesen und modifiziert als geschrieben wird. Beim Refactoring handelt es sich um den Schlüssel zu lesbarem und modifizierbarem Code. Das gilt insbesondere für Frameworks, trifft aber auch auf Software im Allgemeinen zu.

Wo also liegt das Problem? Ganz einfach: Refactoring ist riskant. Es erfordert Änderungen an funktionsfähigem Code, die zu schwer aufspürbaren Bugs führen können. Wird ein Refactoring nicht sorgfältig durchgeführt, kann es Sie um Tage oder sogar Wochen zurückwerfen. Und Refactoring wird noch riskanter, wenn es außerhalb eines formellen Rahmens oder gar spontan durchgeführt wird. Zunächst beginnt man an der Oberfläche des Codes zu schürfen. Dabei findet man schnell Verbesserungsmöglichkeiten und beginnt, tiefer zu graben. Je tiefer man gräbt, desto mehr Verbesserungsmöglichkeiten werden aufgedeckt ... und umso mehr Änderungen nimmt man vor. Unter Umständen gräbt man sich dabei ein sehr tiefes Loch, aus dem man nicht wieder herauskommt. Um zu verhindern, dass man sich sein eigenes Grab schaufelt, muss ein Refactoring systematisch durchgeführt werden. Als meine Co-Autoren und ich das Buch *Design Patterns* verfassten, wiesen wir darauf hin, dass Design Patterns gut geeignete Ziele für Refactorings darstellen. Allerdings ist das Identifizieren des Ziels nur ein Teil des Problems – den Code dementsprechend umzustrukturieren, stellt eine weitere Herausforderung dar.

Martin Fowler und die weiteren mitwirkenden Autoren leisten zur Entwicklung objektorientierter Software einen Beitrag von unschätzbarem Wert, indem sie den Prozess des Refactorings ausführlich darstellen. Dieses Buch erklärt Prinzipien und bewährte Praktiken des Refactorings und zeigt auf, wann und wo Sie beginnen sollten, Ihren Code auf Verbesserungsmöglichkeiten hin zu untersuchen. In der Mitte des Buchs ist eine umfangreiche Liste an Refactorings enthalten. Jedes dieser Refactorings beschreibt die Motivation sowie die jeweilige Vorgehensweise einer bewährten Transformation des Codes. Einige der Refactorings, wie beispielsweise *Funktion extrahieren* (Extract Function, Abschnitt 6.1) oder *Feld verschieben* (Move Field, Abschnitt 8.2), scheinen auf der Hand zu liegen.

Aber lassen Sie sich nicht täuschen. Die Vorgehensweisen solcher Refactorings zu verstehen, ist von entscheidender Bedeutung, um ein Refactoring auf disziplinierte Weise auszuführen. Die Refactorings in diesem Buch werden Ihnen dabei helfen, Ihren Code Schritt für Schritt zu modifizieren und auf diese Weise das Risiko bei der Weiterentwicklung Ihres

Designs zu verringern. Als Entwickler werden Sie schon bald die Refactorings und ihre Bezeichnungen in Ihren Wortschatz aufnehmen.

Meine ersten Erfahrungen mit diszipliniertem »Schritt-für-Schritt«-Refactoring machte ich während des Pair Programmings mit Kent Beck 30.000 Fuß über dem Erdboden. Er stellte sicher, dass wir die Refactorings aus diesem Buch schrittweise anwendeten. Ich war erstaunt, wie gut dieses Verfahren funktionierte. Ich hatte nicht nur größeres Vertrauen in den geschriebenen Code, ich fühlte mich auch weniger gestresst. Ich empfehle Ihnen nachdrücklich, diese Refactorings auszuprobieren: Ihnen und Ihrem Code wird es damit viel besser ergehen.

–Erich Gamma, Object Technology International, Inc.

Januar 1999

Über den Fachkorrektor der deutschen Ausgabe

Dr. Jan Linxweiler ist Geschäftsführer des *Center for Mechanics, Uncertainty and Simulation in Engineering* an der TU Braunschweig. Er verfügt über mehr als 15 Jahre Erfahrung in Forschung und Lehre im Bereich der Softwareentwicklung und war als Softwarearchitekt im Automotive-Sektor tätig. Darüber hinaus entwickelt er erfolgreich Apps für iOS und macOS. Seine Leidenschaft für die Entwicklung hochqualitativer Software vermittelt Linxweiler auch den Studierenden in seinen Vorlesungen.

Einleitung

Es war einmal ein Berater, der sich zu einem Softwareprojekt aufmachte, um sich den Code anzusehen, der im Laufe des Projekts entstanden war. Als er sich die Klassenhierarchie im Kern des Systems ansah, stellte er fest, dass es ein ziemliches Durcheinander war. Klassen auf höherem Abstraktionsniveau gingen von bestimmten Annahmen bezüglich der Funktionsweise anderer Klassen aus – Annahmen, die sie in Form von Code an ihre Unterklassen vererbten. Der Code war allerdings nicht für alle Unterklassen geeignet, daher wurde er ziemlich häufig überschrieben. Schon geringfügige Änderungen an der Basisklasse hätten die Notwendigkeit, den Code zu überschreiben, weitestgehend beseitig. An anderen Stellen hatte man die Intentionen der Basisklasse offenbar nicht richtig verstanden, denn Teile des in der Basisklasse bereits vorhandenen Verhaltens wurden in Unterklassen erneut implementiert. An wieder anderen Stellen erledigten mehrere Unterklassen die gleichen Aufgaben durch Code, der in der Klassenhierarchie eindeutig nach oben verschoben werden konnte.

Der Berater empfahl dem Projektmanagement, den Code zu untersuchen und zu bereinigen – was allerdings nicht gerade für Begeisterung sorgte. Der Code funktionierte offenbar, und es gab erheblichen Termindruck. Das Management meinte, man werde sich später darum kümmern.

Den Programmierern, die an der Klassenhierarchie arbeiteten, hatte der Berater ebenfalls gezeigt, was da tatsächlich vor sich ging. Sie waren eifrig und erkannten das Problem. Ihnen war klar, dass es nicht wirklich ihr Fehler war. Manchmal benötigt es eben ein weiteres Paar an Augen, um ein Problem zu erkennen. Die Programmierer befassten sich also ein paar Tage lang damit, die Hierarchie aufzuräumen. Als sie fertig waren, hatten sie die Hälfte des Codes aus der Hierarchie entfernt, ohne dadurch die Funktionalität einzuschränken. Sie waren mit dem Ergebnis sehr zufrieden und stellten fest, dass sowohl das Hinzufügen neuer Klassen als auch die Verwendung der Klassen des übrigen Systems schneller und einfacher von der Hand gingen.

Das Projektmanagement war nicht erfreut. Der Terminplan war eng, und es gab eine Menge Arbeit zu erledigen. Die beiden Programmierer hatten zwei Tage mit Arbeiten verbracht, die nichts zu den vielen Features beitrugen, die das System in wenigen Monaten besitzen musste. Der alte Code hatte doch tadellos funktioniert. Ja, zugegeben, das Design war jetzt etwas »reiner« und etwas »sauberer«, aber das Projekt musste Code liefern, der funktioniert, keinen Code, der Akademikern gefällt. Der Berater hingegen schlug vor, weitere zentrale Teile des Systems auf ähnliche Weise aufzuräumen, wodurch das Projekt womöglich ein bis zwei Wochen zum Stillstand käme. Und das Ganze, um den Code schöner zu machen, nicht, damit er etwas kann, was er nicht jetzt schon könnte.

Was halten Sie von dieser Geschichte? Glauben Sie, dass der Berater zu Recht vorschlug, weiter aufzuräumen? Oder neigen Sie eher zur alten Ingenieur-Weisheit »Was nicht kaputt ist, muss man auch nicht reparieren«?

Ich bin hier zugegebenermaßen etwas voreingenommen, denn der Berater war ich. Das Projekt scheiterte sechs Monate später, vor allem, weil der Code zu komplex war, um ihn zu debuggen oder ihn so anzupassen, dass er eine akzeptable Leistung lieferte.

Dann wurde Kent Beck als Berater engagiert, der das Projekt wieder auf die Beine stellen sollte – eine Aufgabe, die es erforderlich machte, fast das gesamte System von Grund auf neu zu schreiben. Er machte einiges anders, aber die wichtigste Änderung war, dass er darauf bestand, den Code durch Refactorings kontinuierlich aufzuräumen. Die erhöhte Effektivität des Teams und die Rolle, die das Refactoring dabei einnahm, inspirierten mich dazu, die erste Ausgabe dieses Buchs zu verfassen – um das von Kent und anderen erworbene Wissen weiterzugeben, das sie sich durch die Anwendung von Refactorings zur Verbesserung der Softwarequalität angeeignet hatten.

Seitdem gehört »Refactoring« in der Programmierung zum gängigen Wortschatz. Tatsächlich hat sich das ursprüngliche Buch auch ziemlich gut behaupten können. Allerdings sind achtzehn Jahre ein hohes Alter für ein Buch über Programmierung. Daher dachte ich mir, es sei an der Zeit, es zu überarbeiten. Dabei habe ich praktisch jede Seite des Buchs neu geschrieben. In gewisser Hinsicht hat sich jedoch nur sehr wenig geändert. Die Essenz des Buches ist unverändert; die meisten der wichtigsten Refactorings sind grundsätzlich die gleichen. Ich hoffe jedoch, dass die Neuformulierungen mehr Lesern dabei helfen, zu erlernen, wie das Refactoring effektiv durchgeführt wird.

Was ist Refactoring?

Refactoring ist der Prozess, ein Softwaresystem so zu modifizieren, dass sich das externe Verhalten des Codes nicht ändert, aber dennoch die interne Struktur des Codes zu verbessern. Es handelt sich um eine Vorgehensweise zum Aufräumen von Code, die Disziplin erfordert und die Wahrscheinlichkeit, Bugs zu verursachen, minimiert. Beim Refactoring verbessern Sie im Wesentlichen das Design des Codes, nachdem er geschrieben wurde.

»Verbessern des Designs des Codes, nachdem er geschrieben wurde.« Das ist schon eine seltsame Ausdrucksweise. Seit es Softwareentwicklung gibt, dachten die meisten Leute, dass zunächst das Design entwickelt wird, und dass die Programmierung erst dann erfolgt, wenn es abgeschlossen ist. Der Code wird im Laufe der Zeit modifiziert, und die Integrität des Systems – die dem ursprünglichen Design entsprechende Struktur – geht allmählich verloren. Der Code wird nicht mehr sorgfältig entwickelt, und die Programmierung wird allmählich zum »Hacken«.

Refactoring ist das Gegenteil dieser Vorgehensweise. Mit Refactoring können wir ein schlechtes oder sogar chaotisches Design in sinnvoll strukturierten Code umwandeln. Die einzelnen Schritte sind einfach – geradezu simpel. Ich verschiebe ein Attribut von einer Klasse in eine andere, entnehme einer Methode etwas Code, um daraus eine eigene Methode zu machen oder verschiebe Teile des Codes in der Hierarchie nach oben oder unten. Dennoch kann die Gesamtheit dieser kleinen Änderungen das Design drastisch verbessern. Hierbei handelt es sich um die genaue Umkehrung dessen, was man als Softwarezerfall bezeichnen könnte.

Beim Refactoring verschiebt sich die Gewichtung der Tätigkeiten. Das Design wird nicht nur ganz am Anfang festgelegt, sondern entsteht kontinuierlich während der Entwicklung. Während sich das System entwickelt, finde ich heraus, wie sich das Design verbessern lässt.

Diese Interaktion führt zu einem Programm, dessen Design auch bei fortschreitender Entwicklung gut bleibt.

Worum geht es in diesem Buch?

Dieses Buch ist ein Leitfaden für das Refactoring und wendet sich an den fortgeschrittenen Entwickler. Ich möchte Ihnen zeigen, wie Sie Refactorings auf kontrollierte und effiziente Weise durchführen können. Sie erfahren, wie Sie ein Refactoring vornehmen, ohne Bugs im Code zu verursachen, und gleichzeitig die Struktur des Codes systematisch verbessern.

Traditionell beginnt ein Buch mit einer Einführung. Das halte ich im Prinzip für richtig, finde es jedoch schwierig, Refactoring anhand allgemeiner Erklärungen oder Definitionen einzuführen, deshalb folgt zunächst ein Beispiel. In Kapitel 1 wird ein kleines Programm mit einigen typischen Designfehlern durch Refactorings so umstrukturiert, dass es anschließend besser verständlich und leichter modifizierbar ist. Sie lernen dabei den generellen Refactoring-Prozess sowie eine Reihe nützlicher Refactorings kennen. Hierbei handelt es sich um das wichtigste Kapitel, das Sie lesen sollten, wenn Sie verstehen möchten, worum es beim Refactoring eigentlich geht.

In Kapitel 2 erläutere ich weitere allgemeine Prinzipien des Refactorings und stelle einige Definitionen sowie die Gründe für ein Refactoring vor. Anschließend umreiße ich einige der Herausforderungen, die das Refactoring mit sich bringt. In Kapitel 3 unterstützt mich Kent Beck dabei, zu beschreiben, wie man »Code-Smells« identifiziert und wie man sie durch Refactorings bereinigen kann. Auch das Testen spielt beim Refactoring eine sehr wichtige Rolle, deshalb beschreibt Kapitel 4, wie Sie Tests in Ihren Code integrieren können.

Der Hauptgegenstand dieses Buchs – der Refactoring-Katalog – nimmt die verbleibenden Seiten ein. Der Katalog ist zwar alles andere als vollständig, enthält jedoch die wichtigsten Refactorings, die wahrscheinlich von den meisten Entwicklern benötigt werden. Der Katalog ist aus den Notizen entstanden, die ich mir gemacht habe, als ich mich Ende der 1990er Jahre mit Refactoring befasste. Ich verwende diese Notizen auch heute noch, weil ich sie mir nicht alle merken kann. Wenn ich ein Refactoring durchführen möchte, wie etwa *Phase aufteilen* (Split Phase, Abschnitt 6.11), kann ich im Katalog nachsehen, wie ich das auf sichere Weise Schritt für Schritt tun kann. Ich kann mir nur wünschen, dass Sie diesen Teil des Buchs regelmäßig aufschlagen werden.

Schwerpunkt Internet

Das Internet hat enorme Auswirkungen auf unsere Gesellschaft, insbesondere auf die Art und Weise, wie wir uns Informationen beschaffen. Als ich die erste Ausgabe dieses Buchs verfasste, wurde das Wissen über Softwareentwicklung hauptsächlich durch Druckerzeugnisse vermittelt. Inzwischen beschaffe ich mir die meisten Informationen online. Das stellt für Autoren wie mich eine Herausforderung dar: Haben Bücher noch eine Daseinsberechtigung, und wie sollten sie gestaltet sein?

Ich glaube, Bücher wie dieses haben noch immer ihren Platz – sie müssen sich jedoch ändern. Der Wert eines Buchs besteht in einer großen zusammenhängenden Wissensmenge. Beim Schreiben dieses Buchs habe ich versucht, eine Vielzahl verschiedener Refactorings abzuhandeln und sie in sinnvoller und gegliederter Weise zu organisieren.

Aber im Großen und Ganzen handelt es sich dabei um ein abstraktes literarisches Werk, herkömmlicherweise in Form eines auf Papier gedruckten Buchs, das zukünftig nicht mehr nötig sein wird. Die meisten Verlage bieten noch immer vornehmlich auf Papier gedruckte Bücher an. Zwar wurden eBooks mit Begeisterung angenommen, diese stellen jedoch nur elektronische Versionen der ursprünglichen Werke dar, die auf der Struktur eines auf Papier gedruckten Buchs beruhen.

Mit diesem Buch möchte ich einen anderen Ansatz verfolgen. Die Standardform dieses Buchs ist die dazugehörige englischsprachige Website, die ich als *Webbook* bezeichne (`https://refactoring.com/catalog`). Das auf Papier gedruckte Buch stellt eine Auswahl des auf der Website enthaltenen Materials dar, die so zusammengestellt wurde, dass es sinnvoll ist, sie auf Papier zu drucken. Das Buch soll auch gar nicht alle auf der Website vorhandenen Refactorings enthalten, zumal ich zukünftig vermutlich weitere Refactorings zu dem Webbook hinzufügen werde. Auf ähnliche Weise stellt auch das eBook eine bestimmte Auswahl des Webbooks dar.

Während ich das hier schreibe, weiß ich natürlich nicht, ob Sie diese Zeilen auf der Website, in einem eBook, in einem auf Papier gedruckten Buch oder in einer anderen Form lesen, die ich mir noch gar nicht vorstellen kann. Jedenfalls bemühe ich mich, ein nützliches Werk zu erstellen, unabhängig davon, in welcher Form es Ihnen vorliegt.

Codebeispiele in JavaScript

Wie in den meisten technisch orientierten Bereichen der Softwareentwicklung sind Codebeispiele zur Veranschaulichung der Konzepte sehr wichtig. Die Refactorings sind sich in verschiedenen Programmiersprachen allerdings sehr ähnlich. Hin und wieder gibt es gewisse Dinge, die in einer bestimmten Programmiersprache zu beachten sind, die grundlegenden Elemente der Refactorings ändern sich jedoch nicht.

Zur Veranschaulichung der Refactorings habe ich mich für JavaScript entschieden, weil ich den Eindruck habe, dass die meisten Leser diese Programmiersprache verstehen dürften. Es sollte Ihnen aber auch nicht schwerfallen, die Refactorings auf beliebige andere Programmiersprachen zu übertragen. Ich habe mich bemüht, keine der komplizierteren Features der Sprache zu verwenden, damit Sie die Refactorings auch mit nur oberflächlichen JavaScript-Kenntnissen verstehen können. Dass ich JavaScript verwende, ist jedenfalls sicher nicht als uneingeschränkte Befürwortung der Sprache zu verstehen.

Ich verwende für meine Beispiele zwar JavaScript, das heißt jedoch nicht, dass die im Buch vorgestellten Verfahren auf JavaScript beschränkt sind. In der ersten Ausgabe dieses Buchs wurde Java verwendet, und viele Programmierer fanden es nützlich, obwohl sie noch nie zuvor auch nur eine einzige Java-Klasse geschrieben hatten. Ich habe mit dem Gedanken gespielt, diese Allgemeingültigkeit durch die Verwendung von einem Dutzend verschiedener Programmiersprachen für die Beispiele zu verdeutlichen, hatte jedoch den Eindruck, dass dieses Vorgehen für den Leser zu verwirrend wäre. Dessen ungeachtet richtet sich das Buch an Programmierer, die beliebige Sprachen verwenden. Außerhalb der Abschnitte mit Codebeispielen treffe ich keine Annahmen über die verwendete Programmiersprache. Ich gehe davon aus, dass die Leser meine allgemeinen Kommentare zur Kenntnis nehmen und sie auf die von ihnen verwendete Programmiersprache übertragen. Tatsächlich erwarte ich sogar, dass Leser die JavaScript-Beispiele an die von ihnen verwendete Programmiersprache anpassen.

Das bedeutet, dass ich, abseits der Diskussion von bestimmten Beispielen, Begriffe wie »Klasse«, »Modul«, »Funktion« usw. im allgemeinen Kontext der Programmierung verwende und nicht im Kontext des Sprachmodells von JavaScript.

Dass ich für die Codebeispiele JavaScript verwende, bedeutet darüber hinaus auch, dass ich mich bemühe, einen für JavaScript typischen Programmierstil zu vermeiden, der Programmierern, die normalerweise kein JavaScript verwenden, weniger vertraut ist. Es geht in diesem Buch nicht um »Refactoring in JavaScript«, sondern um Refactoring im Allgemeinen, wobei zufälligerweise JavaScript für die Beispiele verwendet wird. Es gibt eine Vielzahl interessanter Refactorings, die spezifisch für JavaScript sind (wie etwa das Refactoring von Callbacks zu Promises oder zu async/await). Diese Refactorings gehen jedoch über den Rahmen dieses Buchs hinaus.

Wer sollte dieses Buch lesen?

Ich richte mich mit diesem Buch an fortgeschrittene Programmierer – an Menschen, die mit dem Schreiben von Software ihren Lebensunterhalt verdienen. Die Beispiele und die Erläuterungen umfassen eine Menge Code, den es zu lesen und zu verstehen gilt. Die Beispiele sind in JavaScript verfasst, sollten aber auf die meisten Programmiersprachen übertragen werden können. Ich erwarte, dass Leser genügend Erfahrung besitzen, um zu verstehen, was in diesem Buch beschrieben wird, umfassende Kenntnisse werden jedoch nicht vorausgesetzt.

Das Buch wendet sich zwar insbesondere an Entwickler, die das Refactoring erlernen möchten, ist aber auch nützlich für jemanden, der bereits Erfahrungen mit dem Refactoring gesammelt hat – es kann also auch als Lernhilfe verwendet werden. Ich habe große Anstrengungen unternommen, um die Funktionsweise der verschiedenen Refactorings zu erklären, damit erfahrene Entwickler dieses Material zur Unterstützung und Beratung ihrer Kollegen nutzen können.

Das Refactoring konzentriert sich zwar vornehmlich auf den Code, hat aber auch erhebliche Auswirkungen auf das Design eines Systems. Für leitende Designer und Softwarearchitekten ist von entscheidender Bedeutung, die Grundlagen des Refactorings zu verstehen und sie in ihren Projekten zu verwenden. Ein Refactoring wird idealerweise durch einen angesehenen und erfahrenen Entwickler eingeleitet. Ein solcher Entwickler kann die dem Refactoring zugrunde liegenden Prinzipien am besten verstehen und sie auf eine bestimmte Situation übertragen. Das trifft insbesondere dann zu, wenn Sie eine andere Sprache als JavaScript verwenden, weil die von mir gezeigten Beispiele an diese andere Programmiersprache angepasst werden müssen.

So können Sie das Buch am besten ausschöpfen, ohne es vollständig zu lesen:

- **Wenn Sie wissen möchten, was Refactoring eigentlich ist**, sollten Sie Kapitel 1 lesen – das Beispiel sollte die Vorgehensweise verdeutlichen.
- **Wenn Sie wissen möchten, welche Gründe für ein Refactoring sprechen**, sollten Sie die ersten beiden Kapitel lesen. Sie erläutern, was Refactoring eigentlich ist und warum es durchgeführt werden sollte.
- **Wenn Sie wissen möchten, wo ein Refactoring angebracht ist**, sollten Sie Kapitel 3 lesen. Darin werden die Anzeichen dafür erläutert, dass ein Refactoring notwendig wäre.

- **Wenn Sie das Refactoring tatsächlich durchführen möchten**, sollten Sie die ersten vier Kapitel vollständig lesen und anschließend den Katalog überfliegen. Lesen Sie so viel, dass Sie in groben Zügen wissen, was der Katalog enthält. Sie brauchen die ganzen Details nicht zu kennen. Lesen Sie ausführlich nach, wenn Sie das Refactoring tatsächlich durchführen, und lassen Sie sich von dem Beispiel unterstützen. Der Katalog ist zum Nachschlagen gedacht, deshalb werden Sie ihn vermutlich nicht in einem Zug durchlesen wollen.

Beim Schreiben des Buchs war die Benennung der verschiedenen Refactorings ein wichtiger Teil. Die Terminologie erleichtert uns die Kommunikation: Wenn ein Entwickler einen anderen darum bittet, einen Codeabschnitt in eine Funktion auszulagern oder eine Berechnung in separate Phasen aufzuteilen, ist beiden klar, was mit den Bezeichnungen *Funktion extrahieren* (Extract Function, Abschnitt 6.1) und *Phase aufteilen* (Split Phase, Abschnitt 6.11) gemeint ist. Diese Bezeichnungen sind auch bei der Auswahl automatisierter Refactorings hilfreich.

Von anderen geschaffene Grundlagen

Ich möchte gleich am Anfang klarstellen, dass ich mit diesem Buch eine große Verpflichtung eingegangen bin – eine Verpflichtung denen gegenüber, die in den 1990er Jahren das Fachgebiet Refactoring entwickelt haben. Ich habe aus ihren Erfahrungen gelernt, die es mir ermöglichten und mich dazu inspirierten, die erste Ausgabe dieses Buchs zu verfassen. Auch wenn seitdem viele Jahre vergangen sind, halte ich es für wichtig, die Grundlagen zu würdigen, die hier geschaffen wurden. Idealerweise hätte einer der Schöpfer des Refactorings diese erste Ausgabe verfasst, aber letzten Endes war ich es, der die Zeit und die Kraft fand, diese Aufgabe zu bewältigen.

Ward Cunningham und **Kent Beck** gehörten zu den führenden ersten Verfechtern des Refactorings. Sie nutzten es schon früh als Grundlage für ihre Projekte und passten ihren Entwicklungsprozess so an, dass sie vom Refactoring profitieren konnten. Insbesondere die Zusammenarbeit mit Kent hat mir immer wieder die große Bedeutung des Refactorings verdeutlicht – eine Erkenntnis, die unmittelbar zum Entstehen dieses Buchs führte.

Ralph Johnson leitet an der University of Illinois in Urbana-Champaign eine Forschungsgruppe, die für ihre praktischen Beiträge zu objektorientierten Technologien bekannt ist. Ralph war lange ein Meister des Refactorings, und einige seiner Studenten haben auf diesem Gebiet unverzichtbare Beiträge geleistet. **Bill Opdyke** hat mit seiner Doktorarbeit das erste ausführliche schriftliche Werk über Refactoring geschaffen. **John Brant** und **Don Roberts** beließen es nicht beim Schreiben von Wörtern – sie schufen das erste automatisierte Werkzeug für Refactoring, den Refactoring-Browser zur Durchführung des Refactorings von Smalltalk-Programmen.

Seit der Veröffentlichung der ersten Ausgabe dieses Buchs haben viele Leute das Fachgebiet Refactoring fortentwickelt. Insbesondere die Arbeit derer, die das automatisierte Refactoring bei der Entwicklung ermöglichten, hat enorm dazu beigetragen, das Leben der Programmierer zu erleichtern. Für mich ist es praktisch selbstverständlich, dass ich eine häufig verwendete Funktion durch eine einfache Tastenkombination umbenennen kann – aber diese Einfachheit beruht auf den Anstrengungen der Programmierer integrierter Entwicklungsumgebungen, deren Arbeit für uns alle von Nutzen ist.

Danksagungen

Auch wenn ich auf all diesen Grundlagen aufbauen konnte, benötigte ich beim Schreiben des Buchs dennoch eine Menge Hilfe. Die erste Ausgabe beruhte in hohem Maß auf der Erfahrung und Unterstützung von **Kent Beck**. Durch ihn kam ich erstmals mit Refactoring in Berührung. Er inspirierte mich dazu, Notizen zu den Refactorings zu erstellen, und half dabei, sie in lesbare Form zu bringen. Er kam auf die Idee, Refactorings in Form von Gerüchen (»Code-Smells«) zu beschreiben. Ich denke manchmal, er hätte eine bessere Version der ersten Ausgabe geschrieben als ich – wenn er stattdessen nicht damit beschäftigt gewesen wäre, das wegweisende Buch zum Thema Extreme Programming zu verfassen.

Alle mir bekannten Buchautoren weisen darauf hin, wie viel sie ihren technischen Gutachtern verdanken. Wir alle haben Arbeiten mit groben Fehlern verfasst, die nur erkannt wurden, weil unsere Kollegen als Gutachter tätig waren. Ich persönlich beschäftige mich nur wenig mit der Begutachtung technischer Arbeiten, nicht zuletzt, weil ich denke, dass ich das nicht besonders gut kann, deshalb bewundere ich die Kollegen, die sich dieser Aufgabe annehmen. Mit der Begutachtung von Büchern ist noch nicht einmal ein Blumentopf zu gewinnen, deshalb betrachte ich das als einen Akt wahrer Großzügigkeit.

Als ich damit anfing, ernsthaft am Buch zu arbeiten, habe ich eine Mailingliste mit Beratern erstellt, um Feedback zu erhalten. Wenn ich Fortschritte gemacht hatte, stellte ich der Gruppe Entwürfe der neuen Inhalte zur Verfügung und bat um Rückmeldung. Ich möchte mich bei den folgenden Personen für ihr Feedback per Mailingliste bedanken: **Arlo Belshee**, **Avdi Grimm**, **Beth Anders-Beck**, **Bill Wake**, **Brian Guthrie**, **Brian Marick**, **Chad Wathington**, **Dave Farley**, **David Rice**, **Don Roberts**, **Fred George**, **Giles Alexander**, **Greg Doench**, **Hugo Corbucci**, **Ivan Moore**, **James Shore**, **Jay Fields**, **Jessica Kerr**, **Joshua Kerievsky**, **Kevlin Henney**, **Luciano Ramalho**, **Marcos Brizeno**, **Michael Feathers**, **Patrick Kua**, **Pete Hodgson**, **Rebecca Parsons** und **Trisha Gee**.

Aus dieser Gruppe gebührt **Beth Anders-Beck**, **James Shore** und **Pete Hodgson** für ihre Hilfe bezüglich JavaScript besonderer Dank.

Nachdem ich einen mehr oder weniger vollständigen Entwurf fertiggestellt hatte, bat ich weitere Personen um Rückmeldung, weil ich gerne wissen wollte, wie das Buch als Ganzes beim Leser ankommt. **William Chargin** und **Michael Hunger** haben mir unglaublich detailliertes Feedback gegeben, und auch **Bob Martin** und **Scott Davis** lieferten viele nützliche Kommentare. **Bill Wake** hat neben seinen Beiträgen zur Mailingliste zudem den ersten kompletten Entwurf vollständig begutachtet.

Meine Kollegen bei ThoughtWorks liefern mir kontinuierlich Ideen und Feedback zu meiner Arbeit. In das Buch sind unzählige Fragen, Kommentare und Anmerkungen eingeflossen. Dass ich als Angestellter von ThoughtWorks einen beträchtlichen Teil meiner Arbeitszeit mit dem Schreiben verbringen kann, ist wirklich toll. Ich schätze insbesondere die regelmäßigen Gespräche mit **Rebecca Parsons**, unserer technischen Direktorin, und ihre Ideen.

Der Redakteur **Greg Doench** ist bei Pearson dafür verantwortlich, die vielen bei der Veröffentlichung eines Buchs auftretenden Probleme zu lösen. **Julie Nahil** ist für die Produktion zuständig. Und es war mir wieder ein Vergnügen, mit **Dmitry Kirsanov** (Redigieren) und **Alina Kirsanov** (Zusammenstellung und Verschlagwortung) zusammenzuarbeiten.