Sauberer Code

Sie lesen das Buch aus zwei Gründen. Erstens: Sie sind Programmierer. Zweitens: Sie wollen ein besserer Programmierer werden. Gut. Wir brauchen bessere Programmierer.

Dieses Buch handelt von guter Programmierung. Es ist voller Code. Wir werden uns Code aus allen möglichen Richtungen anschauen. Wir werden ihn von oben und unten und von außen und innen betrachten. Wenn wir fertig sind, werden Sie sehr viel über Code wissen. Darüber hinaus werden Sie guten Code von schlechtem Code unterscheiden können. Sie werden wissen, wie Sie guten Code schreiben können. Und Sie werden wissen, wie Sie schlechten Code in guten Code transformieren können.

1.1 Code, Code und nochmals Code

Vielleicht könnte man einwenden, ein Buch über Code wäre doch etwas altmodisch – Code wäre doch längst kein Thema mehr; stattdessen sollte man sich mit Modellen und Anforderungen befassen. Tatsächlich vertreten einige Leute die Auffassung, die Ära des Codes ginge zu Ende. Bald werde aller Code nicht mehr geschrieben, sondern generiert werden. Programmierer würden einfach überflüssig werden, weil Geschäftsentwickler Programme einfach aus Spezifikationen generieren würden.

Unsinn! Wir werden niemals ohne Code arbeiten können, weil der Code die Details der Anforderungen repräsentiert. Auf einer gewissen Ebene können diese Details nicht ignoriert oder abstrahiert werden; sie müssen spezifiziert werden. Und die Spezifikation von Anforderungen in einer Detailgenauigkeit, dass sie von einer Maschine ausgeführt werden können, ist *Programmierung*. Und eine solche Spezifikation ist *Code*.

Ich rechne damit, dass die Abstraktionsebene unserer Sprachen noch höher gehen wird. Ich erwarte auch, dass die Anzahl der domänenspezifischen Sprachen weiterhin wachsen wird. Diese Entwicklung bringt Vorteile mit sich, aber sie wird den Code nicht eliminieren. Tatsächlich werden alle Spezifikationen, die auf diesen höheren Ebenen und in den domänenspezifischen Sprachen geschrieben werden, Code *sein*! Sie müssen immer noch stringent, genau und so formal und detailliert sein, dass sie von einer Maschine verstanden und ausgeführt werden können.

Leute, die denken, Code werde eines Tages verschwinden, ähneln Mathematikern, die hoffen, eines Tages eine Mathematik zu entdecken, die nicht formal sein muss. Sie hoffen, dass wir eines Tages eine Methode entdecken werden, Maschinen zu erschaffen, die tun, was wir wollen, und nicht, was wir sagen. Diese Maschinen müssen in der Lage sein, uns so gut zu verstehen, dass sie unsere unscharf formulierten Bedürfnisse in perfekt ausgeführte Programme übersetzen können, die genau diese Bedürfnisse erfüllen.

Dies wird nie passieren. Nicht einmal Menschen mit all ihrer Intuition und Kreativität sind bis jetzt in der Lage gewesen, aus den unscharfen Gefühlen ihrer Kunden erfolgreiche Systeme abzuleiten. Wenn wir überhaupt etwas aus der Disziplin der Anforderungsspezifikation gelernt haben, ist es Folgendes: Wohlspezifizierte Anforderungen sind genauso formal wie Code und können als ausführbare Tests dieses Codes verwendet werden!

Vergessen Sie nicht, dass Code letztlich die Sprache ist, in der wir die Anforderungen ausdrücken. Wir können Sprachen konstruieren, die näher bei den Anforderungen angesiedelt sind. Wir können Werkzeuge schaffen, die uns helfen, diese Anforderungen zu parsen und zu formalen Strukturen zusammenzusetzen. Aber wir werden niemals die erforderliche Präzision eliminieren können – und deshalb wird es immer Code geben.

1.2 Schlechter Code

Abb. 1.1: Kent Beck

Neulich las ich das Vorwort zu dem Buch *Implementation Patterns* von Kent Beck [Beck07]. Darin schreibt er: »... dieses Buch basiert auf einer recht fragilen Prämisse: dass guter Code eine Rolle spiele ...«. Eine *fragile* Prämisse? Dem kann ich nicht zustimmen! Ich glaube, dass diese Prämisse zu den robustesten, am besten unterstützten und meistdiskutierten Prämissen unserer Zunft gehört (und ich glaube, das weiß Kent Beck auch). Wir wissen, dass guter Code eine Rolle spielt, weil wir uns so lange mit seiner mangelnden Qualität auseinandersetzen mussten.

Ich kenne ein Unternehmen, das in den späten 80er-Jahren eine *Killer*-Applikation herausbrachte. Sie war sehr beliebt, und zahlreiche professionelle Anwender kauften und nutzten sie. Aber dann wurden die Release-Zyklen immer länger. Bugs wurden von einem Release zum nächsten nicht mehr repariert. Die Startzeiten wurden länger und die Abstürze häufiger. Ich erinnere mich an den Tag, an dem ich das Produkt frustriert abschaltete und niemals wieder benutzte. Kurz danach verschwand das Unternehmen vom Markt.

Zwei Jahrzehnte später traf ich einen früheren Mitarbeiter dieses Unternehmens und fragte ihn, was damals passiert wäre. Die Antwort bestätigte meine Befürchtungen. Das Unternehmen hatte das Produkt zu schnell auf den Markt gebracht und im Code ein riesiges Chaos angerichtet. Je mehr Funktionen zu dem Code hinzugefügt wurden, desto schlechter wurde er, bis das Unternehmen ihn einfach nicht mehr verwalten konnte. *Es war der schlechte Code, der das Unternehmen in den Abgrund trieb.*

Sind *Sie* jemals erheblich von schlechtem Code beeinträchtigt worden? Wenn Sie als Programmierer auch nur ein bisschen Erfahrung haben, dann haben Sie eine solche Behinderung viele Male erlebt. Tatsächlich haben wir eine spezielle Bezeich-

nung dafür: *Wading* (Waten). Wir waten durch schlechten Code. Wir kämpfen uns durch einen Morast verschlungener Schlingpflanzen und verborgener Fallgruben. Wir mühen uns ab, den richtigen Weg zu finden, und hoffen auf irgendwelche Hinweise, die uns zeigen, was passiert; aber alles, was wir sehen, ist ein schier endloses Meer von sinnlosem Code.

Natürlich sind Sie von schlechtem Code behindert worden. Also – warum haben Sie ihn geschrieben?

Haben Sie zu schnell gearbeitet? Waren Sie unter Druck? Wahrscheinlich. Vielleicht hatten Sie das Gefühl, keine Zeit für gute Arbeit zu haben, meinten, Ihr Chef würde ärgerlich werden, wenn Sie sich die Zeit nehmen würden, Ihren Code aufzuräumen. Vielleicht waren Sie es einfach leid, an diesem Programm zu arbeiten, und wollten endlich damit fertig werden. Oder vielleicht haben Sie Ihren Stapel unerledigter Arbeit angeschaut, die Sie längst hätten erledigen müssen, und sind zu dem Schluss gekommen, Sie müssen dieses Modul zusammenschustern, um mit dem nächsten weitermachen zu können. Wir alle kennen diese Erfahrung.

Wir alle haben uns das Chaos angeschaut, das wir gerade angerichtet hatten, und dann beschlossen, es an einem anderen Tag zu beseitigen. Wir alle haben die Erleichterung gefühlt, zu sehen, dass unser chaotisches Programm lief, und beschlossen, dass ein laufendes Chaos besser wäre als nichts. Wir alle haben uns vorgenommen, später zurückzukommen und das Chaos zu beseitigen. Natürlich kannten wir damals das Gesetz von LeBlanc nicht: *Später gleich niemals*.

1.3 Die Lebenszykluskosten eines Chaos

Wenn Sie schon länger als zwei bis drei Jahre programmieren, haben Sie wahrscheinlich die Erfahrung gemacht, dass Ihre Arbeit von dem chaotischen Code eines anderen Entwicklers erheblich verlangsamt worden ist. Die Verlangsamung kann beträchtlich sein. Im Laufe von einem oder zwei Jahren kann es passieren, dass Teams, die am Anfang eines Projekts sehr schnell vorangekommen waren, sich plötzlich nur noch im Schneckentempo vorwärtsbewegen. Jede Änderung des Codes führt zur Defekten an zwei oder drei anderen Stellen des Codes. Keine Änderung ist trivial. Für jede zusätzliche Funktion oder Modifikation des Systems müssen alle Verzweigungen, Varianten und Knoten »verstanden« werden, damit weitere Verzweigungen, Varianten und Knoten hinzugefügt werden können. Im Laufe der Zeit wird das Chaos so groß und so verfilzt, dass Sie es nicht mehr bereinigen können. Sie sind am Ende Ihres Weges angelangt.

Während das Chaos immer größer wird, nimmt die Produktivität des Teams laufend ab und geht asymptotisch gegen null. Während die Produktivität sinkt, tut das Management das Einzige, was es kann: Es weist dem Projekt mehr Personal zu in der Hoffnung, die Produktivität zu steigern. Aber das neue Personal versteht das Design des Systems nicht. Es kennt nicht den Unterschied zwischen einer Ände-

rung, die zum Zweck des Designs passt, und einer Änderung, die dem zuwiderläuft. Darüber hinaus stehen Sie und die anderen Teammitglieder unter schrecklichem Druck, die Produktivität zu verbessern. Deshalb produzieren alle immer mehr Chaos und senken damit die Produktivität immer weiter gegen null (siehe Abbildung 1.2).

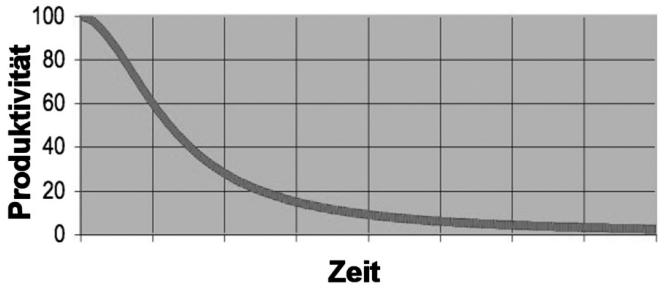

Abb. 1.2: Produktivität und Zeit

Das große Redesign in den Wolken

Schließlich rebelliert das Team. Das Management wird darüber informiert, dass man mit dieser zweifelhaften Code-Basis nicht weiterarbeiten könne. Es wird ein Redesign gefordert. Das Management will aber nicht die Ressourcen für ein komplett neues Redesign des Projekts aufwenden, kann sich aber auch nicht der Erkenntnis verschließen, dass die Produktivität nicht akzeptabel ist. Schließlich beugt es sich den Forderungen der Entwickler und autorisiert das große Redesign in den Wolken.

Es wird ein neues Tiger-Team zusammengestellt. Jeder möchte zu diesem Team gehören, weil es ein frisches neues Projekt ist. Man darf neu anfangen und etwas wirklich Schönes erstellen. Aber nur die Besten und Hellsten werden für das Tiger-Team ausgewählt. Alle anderen müssen sich um die Wartung des gegenwärtigen Systems kümmern.

Jetzt gibt es ein Wettrennen zwischen den beiden Teams. Das Tiger-Team muss ein neues System erstellen, das alle Funktionen des alten erfüllt. Und nicht nur das: Es muss auch mit den Änderungen Schritt halten, die laufend an dem alten System vorgenommen werden. Das Management wird das alte System nicht ersetzen, bevor nicht das neue alle Funktionen des alten erfüllt.

Dieser Wettlauf kann sich sehr lange hinziehen. Ich habe Zeitspannen von bis zu zehn Jahren erlebt. Und wenn er beendet wird, sind die ursprünglichen Mitglieder des Tiger-Teams längst nicht mehr da, und die gegenwärtigen Mitglieder verlangen nach einem Redesign des neuen Systems, weil es ein solches Chaos sei.

Wenn nur ein kleiner Teil dieser Geschichte Ihrer Erfahrung entspricht, dann wissen Sie bereits, dass die Zeit, die Sie für das Sauberhalten Ihres Codes verwenden, nicht nur kosteneffizient, sondern eine Frage des beruflichen Überlebens ist.

Einstellung

Sind Sie jemals durch einen Morast gewatet, der so dicht war, dass es Wochen dauerte, um zu tun, was nur einige Stunden hätte dauern sollen? Haben Sie erlebt, dass eine Änderung, die nur eine Zeile hätte erfordern sollen, in Hunderten verschiedener Module durchgeführt werden musste? Diese Symptome kommen leider allzu oft vor.

Warum passiert das mit Code? Warum verrottet guter Code so schnell zu schlechtem Code? Dafür haben wir viele Erklärungen. Wir beklagen uns, dass die Anforderungen in einer Weise geändert wurden, die dem ursprünglichen Design zuwiderläuft. Sie jammern, dass der Zeitplan zu eng bemessen war, um die Aufgaben richtig zu erledigen. Geben dummen Managern und den toleranten Kunden und nutzlosen Marketing-Typen und einem unzureichenden Telefon-Support die Schuld. Aber der Fehler, lieber Dilbert, liegt nicht in unseren Sternen, sondern in uns selbst. Wir sind unprofessionell.

Diese Pille zu schlucken, mag etwas bitter sein. Wie könnte dieses Chaos *unsere* Schuld sein? Was ist mit den Anforderungen? Was ist mit dem Zeitplan? Gibt es etwa keine dummen Manager und nutzlose Marketing-Typen? Tragen sie nicht einen Teil der Schuld?

Nein. Die Manager und Marketing-Leute fragen *uns* nach den Informationen, die sie benötigen, um Versprechungen und Zusagen zu machen; und selbst wenn sie uns nicht fragen, sollten wir keine Hemmungen haben, ihnen zu sagen, was wir denken. Die Benutzer wenden sich an uns, damit wir ihnen zeigen, wie das System ihre Anforderungen erfüllt. Die Projektmanager benutzen unsere Informationen, um ihre Zeitpläne aufzustellen. Wir sind eng in die Planung des Projekts eingebunden und tragen einen großen Teil der Verantwortung für auftretende Fehler, insbesondere wenn diese Fehler mit schlechtem Code zu tun haben!

»Doch halt!«, sagen Sie. »Wenn ich nicht tue, was mein Manager sagt, werde ich gefeuert.« Wahrscheinlich nicht. Die meisten Manager wollen die Wahrheit wissen, selbst wenn sie sich nicht immer entsprechend verhalten. Die meisten Manager wollen guten Code haben, selbst wenn sie von ihrem Zeitplan besessen sind. Vielleicht verteidigen sie leidenschaftlich den Zeitplan und die Anforderungen; aber das ist ihr Job. Dagegen ist es *Ihr* Job, den Code mit gleicher Leidenschaft zu verteidigen.

Betrachten wir eine Analogie: Was würden Sie als Arzt machen, wenn ein Patient Sie auffordern würde, dieses blödsinnige Händewaschen bei der Vorbereitung auf einen chirurgischen Eingriff zu lassen, weil es zu viel Zeit kostet? (Als das Händewaschen 1847 den Ärzten erstmals von Ignaz Semmelweis empfohlen wurde, wurde es mit der Begründung zurückgewiesen, die Ärzte wären zu beschäftigt und

hätten keine Zeit, sich die Hände zwischen ihren Patientenbesuchen zu waschen.) Natürlich hat der Patient Vorrang. Dennoch sollte der Arzt in diesem Fall die Forderung kompromisslos zurückweisen. Warum? Weil der Arzt mehr über die Risiken einer Erkrankung und Infektion weiß als der Patient. Es wäre unprofessionell (und in diesem Fall sogar kriminell), wenn der Arzt der Forderung des Patienten nachgeben würde.

Deshalb ist es auch unprofessional, dass sich Programmierer dem Willen von Managern beugen, die die Risiken nicht verstehen, die mit dem Erzeugen von Chaos im Code verbunden sind.

Das grundlegende Problem

Programmierer werden mit einem grundlegenden Wertekonflikt konfrontiert. Erfahrene Entwickler wissen, dass ihre Arbeit durch alten chaotischen Code erheblich behindert wird. Dennoch fühlen alle Entwickler den Druck, chaotischen Code zu schreiben, um Termine einzuhalten. Kurz gesagt: Sie nehmen sich nicht die Zeit, es richtig zu machen!

Echte Profis wissen, dass der zweite Teil dieses Konflikts falsch ist. Man erfüllt einen Termin eben *nicht*, indem man chaotischen Code produziert. Tatsächlich verlangsamt chaotischer Code Ihre Arbeit sofort und führt dazu, dass Sie Ihren Termin nicht einhalten können. Die *einzige* Methode, den Termin einzuhalten, besteht darin, den Code jederzeit so sauber wie möglich zu halten.

Sauberen Code schreiben – eine Kunst?

Angenommen, Sie glaubten, chaotischer Code wäre eine beträchtliche Behinderung Ihrer Arbeit. Wenn Sie jetzt akzeptieren, dass die einzige Möglichkeit, schneller zu arbeiten, darin besteht, den eigenen Code sauber zu halten, müssen Sie sich zwangsläufig fragen: »Wie schreibe ich sauberen Code?« Es hat keinen Sinn, zu versuchen, sauberen Code zu schreiben, wenn Sie nicht wissen, wie sauberer Code aussieht!

Leider haben wir hier eine schlechte Nachricht: Sauberen Code zu schreiben, hat sehr viel mit dem Malen eines Bildes zu tun. Die meisten können erkennen, wann ein Bild gut oder schlecht gemalt ist. Aber dies erkennen zu können, bedeutet nicht, dass wir auch malen können. Wenn Sie also in der Lage sind, sauberen von schlechtem Code zu unterscheiden, bedeutet dies nicht automatisch, dass Sie sauberen Code schreiben können!

Sauberen Code zu schreiben, erfordert den disziplinierten Einsatz zahlreicher kleiner Techniken, die mit einem sorgfältig erworbenen Gefühl für »Sauberkeit« angewendet werden. Dieses »Gefühl für den Code« ist der Schlüssel. Einige sind damit geboren. Einige müssen es sich mehr oder weniger mühsam erarbeiten. Dieses Gefühl für den Code hilft uns nicht nur, guten von schlechtem Code zu unterschei-

den; sondern zeigt uns die Strategie, wie wir unser Arsenal von erworbenen Techniken anwenden müssen, um schlechten Code in guten zu transformieren.

Ein Programmierer ohne dieses »Gefühl für den Code« kann sich ein chaotisches Modul anschauen und das Chaos erkennen, hat aber absolut keine Vorstellung davon, was er dagegen tun könnte. Ein Programmierer *mit* »Gefühl für den Code« schaut sich das chaotische Modul an und erkennt seine Optionen und Änderungsmöglichkeiten. Sein »Gefühl für den Code« hilft ihm dabei, die beste Option auszuwählen und eine Reihe von Änderungsschritten festzulegen, die ihn zum Ziel bringen und zugleich nach jedem Teilschritt die volle Funktionsfähigkeit des Codes erhalten.

Kurz gesagt: Ein Programmierer, der sauberen Code schreibt, ist ein Künstler, der einen leeren Bildschirm mit einer Reihe von Transformationen in ein elegant codiertes System umwandelt.

Was ist sauberer Code?

Es gibt wahrscheinlich so viele Definitionen wie Programmierer. Deshalb fragte ich einige sehr bekannte und sehr erfahrene Programmierer nach ihrer Meinung.

Bjarne Stroustrup

Abb. 1.3: Bjarne Stroustrup

Bjarne Stroustrup, Erfinder von C++ und Autor von *The C++ Programming Language*

> *Mein Code sollte möglichst elegant und effizient sein. Die Logik sollte gradlinig sein, damit sich Bugs nur schwer verstecken können, die Abhängigkeiten sollten minimal sein, um die Wartung zu vereinfachen, das Fehler-Handling sollte vollständig gemäß einer vordefinierten Strategie erfolgen, und das Leistungsverhalten sollte dem Optimum so nah wie möglich kommen, damit der Entwickler nicht versucht ist, den Code durch Ad-hoc-Optimierungen zu verunstalten. Sauberer Code erledigt eine Aufgabe gut.*

Bjarne verwendet das Wort »elegant«. Was für ein Wort! Im Wörterbuch findet man auch folgende Synonyme: *ansprechende Anmut und Kunstfertigkeit in Aussehen oder Verhalten; ansprechende Sinnlichkeit und Einfachheit*. Wichtig ist dabei die Betonung von »ansprechend«. Offensichtlich glaubt Bjarne, dass sauberer Code *angenehm* zu lesen sein soll. Solchen Code zu lesen, sollte einen Ausdruck des Wohlgefallens auf Ihr Gesicht zaubern, den Sie auch vom Anschauen eines rassigen Automobils kennen.

Bjarne erwähnt auch die Effizienz des Codes. Vielleicht sollte uns dies bei dem Erfinder von C++ weniger überraschen; aber ich glaube, dass er damit mehr als seinen Wunsch nach einer Geschwindigkeit meint. Verschwendete Zyklen sind unelegant, sie vermitteln kein angenehmes Gefühl. Und jetzt beachten Sie, wie Bjarne die Folgen dieser Uneleganz beschreibt. Er verwendet den Ausdruck »versucht sein«. Darin ist eine tiefe Weisheit verborgen. Schlechter Code *verleitet dazu*, das Chaos zu vergrößern! Wenn andere schlechten Code ändern, neigen sie dazu, ihn noch schlechter zu machen.

Die Pragmatischen Programmierer Dave Thomas und Andy Hunt drückten dasselbe auf andere Weise aus. Sie verwendeten die Metapher der zerbrochenen Fenster (`http://www.pragprog.com/the-pragmatic-programmer/extracts/software-entropy`). Ein Gebäude mit zerbrochenen Fenstern sieht so aus, als würde sich niemand darum kümmern. Deshalb kümmern sich auch andere Entwickler nicht um den Code. Sie lassen es gewissermaßen zu, dass weitere Fenster zerbrochen werden. Schließlich helfen sie aktiv dabei. Sie verschmieren die Vorderfront mit Graffiti und lassen zu, dass sich Müll ansammelt. Der Prozess des Zerfalls beginnt mit einem zerbrochenen Fenster.

Bjarne erwähnt auch, dass das Fehler-Handling vollständig sein sollte. Dies gehört zur Disziplin, aufmerksam in den Details zu sein. Ein verkürztes Fehler-Handling ist nur eine Methode, wie Programmierer Details vernachlässigen. Speicherlecks und Race-Bedingungen sind weitere Beispiele dafür. Eine inkonsistente Namensgebung zählt ebenfalls zu. Das Fazit ist: Sauberer Code bedeutet auch große Sorgfalt im Detail.

Bjarne schließt mit der Zusicherung, dass sauberer Code eine Aufgabe gut erledigt. Es ist kein Zufall, dass so viele Prinzipien des Software-Designs auf diese einfache Mahnung zurückgeführt werden können. Autor für Autor hat versucht, diesen einen Gedanken zu kommunizieren. Schlechter Code tut zu viel; seine Absicht ist nicht klar zu erkennen und er versucht, mehrere Zwecke auf einmal zu erfüllen. Sauberer Code ist *fokussiert*. Jede Funktion, jede Klasse, jedes Modul ist eindeutig auf einen einzigen Zweck ausgerichtet und lässt sich von den umgebenden Details weder ablenken noch verunreinigen.

Grady Booch

Abb. 1.4: Grady Booch

Grady Booch, Autor von *Object-Oriented Analysis and Design with Applications*

> *Sauberer Code ist einfach und direkt. Sauberer Code liest sich wie wohlgeschriebene Prosa. Sauberer Code verdunkelt niemals die Absicht des Designers, sondern ist voller griffiger (engl. crisp) Abstraktionen und geradliniger Kontrollstrukturen.*

Einige Punkte von Grady decken sich mit denen von Bjarne, aber er betrachtet das Ganze aus der Perspektive der *Lesbarkeit*. Mir gefällt besonders seine Auffassung, dass sich sauberer Code wie wohlgeschriebene Prosa lesen lassen soll. Denken Sie zurück an ein wirklich gutes Buch, das Sie gelesen haben. Erinnern Sie sich, wie die Wörter verschwanden und durch Bilder ersetzt wurden? Es war, als würden Sie einen Film sehen, nicht wahr? Besser! Sie sahen die Zeichen, Sie hörten die Geräusche, Sie erlebten die Leidenschaften und den Humor.

Sauberen Code zu lesen, wird natürlich niemals dasselbe sein, wie *Der Herr der Ringe* zu lesen. Dennoch ist die literarische Metapher brauchbar. Wie ein guter Roman sollte sauberer Code die Spannung in den zu lösenden Problemen sauber herausarbeiten. Diese Spannung sollte einem Höhepunkt zutreiben und dann dem Leser dieses »Aha! Ja natürlich!« vermitteln, wenn die Probleme und Spannungen bei der Enthüllung einer offensichtlichen Lösung aufgelöst werden.

Für mich ist der Ausdruck »griffige Abstraktion« (im Original: »crisp abstraction«, wörtl. »knackige oder frische Abstraktion«, unübersetzbar) ein faszinierendes Oxymoron (ein Widerspruch in sich)! Schließlich bedeutet das Wort »griffig« eher etwas Handgreifliches, Konkretes, praktisch Nutzbares. Trotz dieses scheinbaren Widerspruchs der Wörter vermitteln sie eine klare Botschaft. Unser Code sollte nicht spekulativ, sondern nüchtern und sachbezogen sein. Es sollte nur das Erforderliche enthalten. Unsere Leser sollten unsere Bestimmtheit erkennen können.

Dave Thomas

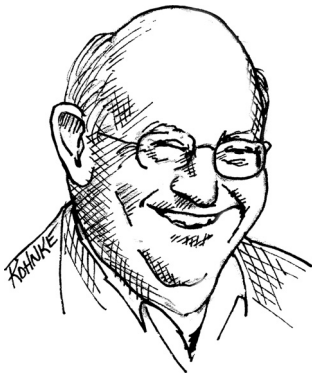

Abb. 1.5: Dave Thomas

»Big« Dave Thomas, Gründer der OTI, der Pate (Godfather) der Eclipse-Strategie

> *Sauberer Code kann von anderen Entwicklern gelesen und verbessert werden. Er verfügt über Unit- und Acceptance-Tests. Er enthält bedeutungsvolle Namen. Er stellt zur Lösung einer Aufgabe nicht mehrere, sondern eine Lösung zur Verfügung. Er enthält minimale Abhängigkeiten, die ausdrücklich definiert sind, und stellte ein klares und minimales API zur Verfügung. Code sollte »literate« sein, da je nach Sprache nicht alle erforderlichen Informationen allein im Code klar ausgedrückt werden können. (A.d.Ü.: »Literate Programming« ist eine Unterströmung, bei der die Einheit von Kommentaren und Code betont und gefördert wird.)*

Auch Big Dave strebt wie Grady Lesbarkeit an, fordert aber eine wichtige Ergänzung. Für Dave ist es wichtig, dass sauberer Code es *anderen* Entwicklern erleichtert, ihn zu verbessern. Dies mag offensichtlich scheinen, aber es kann nicht genug betont werden. Schließlich gibt es einen Unterschied zwischen Code, der einfach zu lesen ist, und Code, der einfach zu ändern ist.

Dave macht Sauberkeit von Tests abhängig! Vor zehn Jahren hätten viele bei dieser Forderung die Stirn gerunzelt. Aber die Disziplin der Test Driven Development (TDD) hat einen wesentlichen Einfluss auf die Software-Branche gehabt und hat sich zu einer der grundlegenden Disziplinen entwickelt. Dave hat recht. Code ohne Tests ist nicht sauber. Egal wie elegant er ist, egal wie lesbar und änderungsfreundlich er ist, ohne Tests ist er unsauber.

Dave verwendet das Wort *minimal* zweimal. Offensichtlich schätzt er Code, der einen geringen Umfang hat. Tatsächlich hat sich im Laufe der letzten Jahre ein Konsens in der Software-Literatur gebildet: Kleiner ist besser.

Dave sagt auch, Code solle *literate* sein. Dies ist ein sanfter Hinweis auf das *Literate Programming* von Donald Knuth [Knuth92]. Die Quintessenz lautet: Code sollte so aufbereitet werden, dass er von Menschen gelesen werden kann.

Michael Feathers

Abb. 1.6: Michael Feathers

Michael Feathers, Autor von *Working Effectively with Legacy Code*

> *Ich könnte alle Eigenschaften auflisten, die mir bei sauberem Code auffallen; aber es gibt eine übergreifende Qualität, die alle anderen überragt: Sauberer Code sieht immer so aus, als wäre er von jemandem geschrieben worden, dem dies wirklich wichtig war. Es fällt nichts ins Auge, wie man den Code verbessern könnte. Alle diese Dinge hat der Autor des Codes bereits selbst durchdacht; und wenn Sie versuchen, sich Verbesserungen vorzustellen, landen Sie wieder an der Stelle, an der Sie gerade sind: Sie sitzen einfach da und bewundern den Code, den Ihnen jemand hinterlassen hat – jemand, der sein ganzes Können in sorgfältige Arbeit gesteckt hat.*

Ein Wort: Sorgfalt. Das ist das eigentliche Thema dieses Buches. Vielleicht hätte ein passender Untertitel gelautet: *Wie man seinem Code Sorgfalt angedeihen lässt.*

Michael trifft den Nagel auf den Kopf. Sauberer Code ist Code, der sorgfältig erstellt worden ist. Jemand hat sich die Zeit genommen, den Code sauber zu strukturieren und zu schreiben. Jemand hat den Details die erforderliche Sorgfalt angedeihen lassen. Jemand war es nicht egal, wie sein Arbeitsergebnis aussah.

Ron Jeffries

Abb. 1.7: Ron Jeffries

Ron Jeffries, Autor von *Extreme Programming Installed* und *Extreme Programming Adventures in C#*

Ron begann seine Karriere als Programmierer in Fortran bei dem Strategic Air Command und hat Code in fast jeder Sprache und auf fast jeder Maschine geschrieben. Es zahlt sich aus, seine Worte sorgfältig zu überdenken.

> *In den letzten Jahren beginne ich (und ende ich fast immer) mit den Regeln von Beck für einfachen Code. In der Reihenfolge der Priorität erfüllt einfacher Code die folgenden Bedingungen:*
>
> - *Er besteht alle Tests.*
> - *Er enthält keine Duplizierung.*
> - *Er drückt alle Design-Ideen aus, die in dem System enthalten sind.*
> - *Er minimiert die Anzahl der Entities, also der Klassen, Methoden, Funktionen usw.*
>
> *Unter diesen Bedingungen konzentriere ich mich hauptsächlich auf die Duplizierung. Wenn dieselbe Sache immer wieder gemacht wird, ist dies ein Zeichen dafür, dass wir einen bestimmten Gedanken in unserem Code nicht gut genug repräsentiert haben. Dann versuche ich herauszufinden, diesen Gedanken zu fassen und klarer auszudrücken.*
>
> *Ausdruckskraft umfasst für mich bedeutungsvolle Namen. Häufig ändere ich die Namen der Dinge mehrfach, bis ich die endgültige Version gefunden habe. Mit modernen Entwicklungswerkzeugen wie etwa Eclipse ist die Umbenennung recht einfach, weshalb ich mir darüber keine Gedanken mache. Doch die Ausdruckskraft geht über Namen hinaus. Ich schaue mir auch an, ob ein Objekt oder eine Methode mehr als eine Aufgabe erfüllt. Ist dies der Fall, zer-*

lege ich ein Objekt in zwei oder mehr kleinere Objekte oder führe ein Refactoring einer Methode mit der Extract-Methode so durch, dass die neue Methode klarer zum Ausdruck bringt, was sie tut, und einige Untermethoden sagen, wie dies getan wird.

Duplizierungen zu eliminieren und die Ausdruckskraft zu steigern, bringen mich meinem Ideal von sauberem Code schon sehr viel näher. Chaotischen Code allein unter zwei dieser Gesichtspunkte zuvor zu verbessern, kann bereits zu einem erheblich besseren Ergebnis führen. Es gibt jedoch noch einen anderen Aspekt meiner Bemühungen, der etwas schwerer zu erklären ist.

Aufgrund meiner langen Erfahrung beim Programmieren scheint es mir, dass alle Programme aus sehr ähnlichen Elementen aufgebaut sind. Ein Beispiel: »Suche Dinge in einer Collection.« Egal was wir durchsuchen wollen, eine Datenbank mit Mitarbeiter-Datensätzen, eine Hash-Map mit Schlüsseln und Werten oder ein Array mit Elementen bestimmter Art, immer wollen wir ein spezielles Element aus dieser Collection abrufen. Wenn ich eine solche Aufgabe identifiziere, verpacke ich oft ihre spezielle Implementierung in eine abstraktere Methode oder Klasse. Dadurch erziele ich eine Reihe interessanter Vorteile.

Ich kann die Funktionalität jetzt mit etwas Einfachem, etwa einer Hash-Map, implementieren. Doch da jetzt alle Referenzen auf diese Suche in meiner kleinen Abstraktion eingekapselt sind, kann ich die Implementierung jederzeit ändern. Ich komme schneller voran und bewahre mir trotzdem meine Fähigkeit, den Code später zu ändern.

Außerdem lenkt die Collection-Abstraktion oft meine Aufmerksamkeit auf das, was »wirklich« passiert, und hält mich davon ab, Implementierungspfade zu verfolgen, auf denen ich alle möglichen Collection-Verhaltensweisen realisiere, auch wenn ich wirklich nur eine ziemlich einfache Methode brauche, um etwas Gesuchtes zu finden.

Reduzierung der Duplizierung, Steigerung der Ausdruckskraft und frühe Formulierung einfacher Abstraktionen machen für mich sauberen Code aus.

Hier hat Ron in einigen kurzen Absätzen den Inhalt dieses Buches zusammengefasst: keine Duplizierung, eine Aufgabe, Ausdruckskraft, kleine Abstraktionen. Es ist alles da.

Ward Cunningham

Ward Cunningham, Erfinder des Wiki, Erfinder von Fit, Miterfinder des eXtreme Programming. Treibende Kraft hinter den Design Patterns. Smalltalk- und OO-Vordenker. Der Pate aller Leute, denen ihr Code nicht egal ist.

Abb. 1.8: Ward Cunningham

> *Sie wissen, dass Sie an sauberem Code arbeiten, wenn jede Routine, die Sie lesen, ziemlich genau so funktioniert, wie Sie es erwartet haben. Sie können den Code auch »schön« nennen, wenn er die Sprache so aussehen lässt, als wäre sie für das Problem geschaffen worden.*

Solche Aussagen sind für Ward charakteristisch. Sie lesen sie, nicken mit dem Kopf und wenden sich dann dem nächsten Thema zu. Die Aussage hört sich so vernünftig, so offensichtlich an, dass sie kaum als etwas Grundlegendes wahrgenommen wird. Sie glauben, sie drücke recht genau das aus, was Sie erwartet haben. Doch schauen wir etwas genauer hin.

»... ziemlich genau so, wie Sie es erwartet haben.« Wann haben Sie zum letzten Mal ein Modul gesehen, das ziemlich genau dem entsprach, was Sie erwartet haben? Ist es nicht wahrscheinlicher, dass die Module, die Sie sich anschauen, rätselhaft, kompliziert und verworren aussehen? Ist es nicht die Regel, dass Sie in die falsche Richtung gelockt werden? Sind Sie es nicht gewohnt, verzweifelt und frustriert die Denkfäden aufzuspüren und durch das ganze System zu verfolgen, aus denen letztlich das Modul gewebt ist? Wann haben Sie zum letzten Mal Code gelesen und dabei mit dem Kopf genickt, wie Sie eben die Aussage von Ward aufgenommen haben?

Ward erwartet, dass Sie beim Lesen von sauberem Code überhaupt keine Überraschungen erleben. Tatsächlich sollte das Ganze fast mühelos sein. Sie lesen den Code, und er entspricht ziemlich genau dem, was Sie erwartet haben. Er ist offensichtlich, einfach und überzeugend. Jedes Modul ist eine Stufe für das nächste. Jedes gibt vor, wie das nächste geschrieben sein wird. Programme, die *so* sauber sind, sind so außerordentlich gut geschrieben, dass Sie es gar nicht mal bemerken. Der Designer lässt das Problem lächerlich einfach aussehen – ein herausragendes Merkmal aller außerordentlichen Designs.

Und was ist mit Wards Vorstellung von Schönheit? Wir haben alle schon darüber geschimpft, dass unsere Sprachen nicht für unsere Probleme konzipiert worden wären. Aber Wards Aussage gibt uns den Schwarzen Peter zurück. Er sagt, schöner Code *lasse die Sprache aussehen, als wäre sie für das Problem gemacht worden*! Es liegt

also in *unserer* Verantwortung, die Sprache einfach aussehen zu lassen! Dies sollte Spracheiferern jeder Couleur zu denken geben! Es ist nicht die Sprache, die ein Programm einfach aussehen lässt. Es ist der Programmierer, der die Sprache einfach aussehen lässt!

1.4 Denkschulen

Abb. 1.9: Uncle Bob (Robert C. Martin)

Was ist mit mir (Uncle Bob)? Was ist für mich sauberer Code? Dieses Buch wird Ihnen bis ins kleinste Detail sagen, was meine Mitstreiter und ich über sauberen Code denken. Wir werden Ihnen sagen, was unserer Meinung nach einen sauberen Variablennamen, eine saubere Funktion, eine saubere Klasse usw. ausmacht. Wir werden diese Meinungen als Absoluta formulieren, und wir werden uns nicht für unsere Schärfe entschuldigen. Für uns sind sie, an diesem Punkt unserer Karriere, absolute Postulate. Sie sind *unsere Denkschule* für sauberen Code.

Kampfsportkünstler sind sich über die beste Kampfsportart oder die beste Technik innerhalb einer Kampfsportart überhaupt nicht einig. Oft gründen Meister einer Kampfsportart ihre eigene Denkschule und sammeln Schüler um sich, denen sie ihren speziellen Kampfstil vermitteln. So gibt es etwa ein *Gracie Jiu Jistu*, das von der Gracie-Familie in Brasilien begründet wurde und gelehrt wird. Es gibt ein *Hakkoryu Jiu Jistu*, das von Okuyama Ryuho in Tokyo begründet wurde und gelehrt wird. Es gibt ein *Jeet Kune Do*, das von Bruce Lee in den Vereinigten Staaten begründet wurde und von seinen Nachfolgern gelehrt wird.

Schüler dieser Ansätze unterziehen sich einem intensiven Studium der Lehren der Gründer. Sie widmen ihre Zeit dem Erlernen des speziellen Kampfstils des jeweiligen Meisters. Oft blenden sie dabei die Lehren aller anderen Meister aus. Später, wenn die Schüler in ihrem Kampfstil eine gewisse Reife erreicht haben, können sie auch bei einem anderen Meister studieren, um ihr Wissen und ihre Kampftechniken auf eine breitere Basis zu stellen. Einige entwickeln und verfeinern ihre Fähig-

keiten schließlich so weit, dass sie neue Techniken entdecken und eigene Schulen gründen.

Keine dieser verschiedenen Schulen ist die absolut *richtige*. Doch innerhalb einer speziellen Schule *verhalten* wir uns so, als *wären* die Lehren und Techniken richtig. Denn schließlich gibt es eine korrekte Methode, Hakkoryu Jiu Jitsu oder Jeet Kune Do auszuüben. Aber diese Selbstgerechtigkeit innerhalb einer Schule entwertet nicht die Lehren einer anderen Schule.

Betrachten Sie dieses Buch als eine Beschreibung der *Object Mentor School of Clean Code* (Object-Mentor-Schule des sauberen Codes). Die Techniken und Lehren in diesem Buch sind die Methoden, wie *wir unsere* Kunst praktizieren. Wir gehen so weit zu behaupten, dass Sie, wenn Sie diese Lehren befolgen, die Vorteile erlangen werden, die wir erlangt haben, und dass Sie lernen werden, sauberen und professionellen Code zu schreiben. Sie sollten aber nicht den Fehler machen zu glauben, dass wir in irgendeinem absoluten Sinne »recht« hätten. Es gibt andere Schulen und andere Meister, die mit demselben Nachdruck Professionalität für sich in Anspruch nehmen wie wir. Und auch von ihnen zu lernen, würde Ihrem Können nur zugutekommen.

Tatsächlich werden viele Empfehlungen in diesem Buch kontrovers diskutiert. Sie werden wahrscheinlich nicht mit allem einverstanden sein. Vielleicht werden Sie einige sogar heftig ablehnen. Das ist in Ordnung. Wir können keinen Anspruch auf die endgültige Autorität erheben. Andererseits sind die Empfehlungen in diesem Buch Dinge, über die wir lange und gründlich nachgedacht haben. Sie basieren auf jahrzehntelanger Erfahrung und haben sich in wiederholten Versuch-und-Irrtum-Zyklen herauskristallisiert. Also: Egal, ob Sie mit uns übereinstimmen oder nicht, es wäre eine Schande, wenn Sie unseren Standpunkt nicht kennen lernen und respektieren würden.

1.5 Wir sind Autoren

Das `@author`-Feld einer Javadoc sagt, wer wir sind. Wir sind Autoren. Ein Merkmal von Autoren ist es, dass sie Leser haben. Tatsächlich sind Autoren dafür *verantwortlich*, erfolgreich mit ihren Lesern zu kommunizieren. Wenn Sie Ihre nächste Codezeile schreiben, sollten Sie daran denken, dass Sie ein Autor sind, der für Leser schreibt, die Ihre Anstrengung beurteilen.

Vielleicht fragen Sie, wie viel Code wirklich gelesen wird. Steckt der größte Aufwand nicht darin, den Code zu schreiben?

Haben Sie jemals ein Play-back einer Edit-Sitzung durchgeführt? In den 80ern und 90ern hatten wir Editoren wie Emacs, die jeden Tastenanschlag speichern konnten. Sie konnten eine Stunde lang arbeiten und dann ein Play-back Ihrer gesamten Edit-Sitzung wie im Zeitraffer ablaufen lassen. Als ich dies tat, waren die Ergebnisse faszinierend.

Der bei Weitem größte Anteil des Play-backs bestand darin, dass ich herumscrollte und zu anderen Modulen navigierte!

Bob kommt zu dem Modul.

Er scrollt zu der Funktion herunter, die geändert werden muss.

Er macht eine Pause und denkt über seine Optionen nach.

Oh, er scrollt wieder nach oben an den Anfang des Moduls, um die Initialisierung einer Variablen zu überprüfen.

Jetzt scrollt er zurück nach unten und beginnt zu tippen.

Ups, er löscht, was er getippt hat!

Er tippt erneut.

Er löscht erneut!

Er tippt die Hälfte von etwas anderem, aber löscht es dann wieder!

Er scrollt runter zu einer anderen Funktion, die die Funktion aufruft, die er ändert, um zu sehen, wie sie aufgerufen wird.

Er scrollt zurück und tippt denselben Code ein, den er gerade gelöscht hat.

Er macht eine Pause.

Er löscht den Code wieder!

Er öffnet ein anderes Fenster und schaut sich eine Unterklasse an. Wird die Funktion überschrieben?

...

Sie verstehen, was abgeht. Tatsächlich beträgt das Verhältnis der Zeit, die mit Lesen verbracht wird, zu der Zeit, die mit Schreiben verbracht wird, weit über 10:1. Wir lesen *permanent* alten Code als Teil unseres Bemühens, neuen Code zu schreiben.

Weil dieses Verhältnis so hoch ist, sollte das Lesen von Code leicht sein, selbst wenn dadurch das Schreiben schwerer wird. Natürlich ist es unmöglich, Code zu schreiben, ohne ihn zu lesen. Deshalb ist das Bemühen, *das Lesen von Code zu erleichtern,* zugleich ein Bemühen, *das Schreiben von Code leichter zu machen*.

Dieser Logik kann man sich nicht entziehen. Man kann keinen Code schreiben, wenn man den umgebenden Code nicht lesen kann. Der Code, den Sie heute zu schreiben versuchen, wird schwer oder leicht zu schreiben sein, und zwar abhängig davon, wie schwer oder leicht Sie den umgebenden Code lesen können. Wenn Sie also schnell vorwärtskommen wollen, wenn Sie schnell fertig werden wollen, wenn Sie Ihren Code leicht schreiben wollen, machen Sie es leicht, ihn zu lesen.

1.6 Die Pfadfinder-Regel

Es reicht nicht aus, guten Code zu schreiben. Der Code muss auch im Zeitablauf *sauber gehalten* werden. Wir haben alle erlebt, wie Code im Laufe der Zeit verrottet und immer schlechter geworden ist. Deshalb müssen wir aktiv tätig werden, um diesem schleichenden Verfall vorzubeugen.

Bei den Pfadfindern gibt es eine einfache Regel, die wir auch auf unseren Beruf anwenden können. (Sie wurde aus der Abschiedsbotschaft, *Versuche die Welt ein wenig besser zu hinterlassen, als du sie gefunden hast, ...*, von Robert Stephenson Smyth Baden-Powell an die Pfadfinder, abgeleitet.) Die Botschaft lautet:

> *Hinterlasse den Campingplatz sauberer, als du ihn gefunden hast.*

Wenn wir alle unseren Code ein wenig sauberer einchecken, als wir ihn ausgecheckt haben, kann der Code einfach nicht verrotten. Es muss kein Großreinemachen sein. Verbessern Sie hier einen Variablennamen, zerlegen Sie dort eine etwas zu große Funktion, eliminieren Sie doppelte Codezeilen oder bauen Sie eine zusammengesetzte `if`-Anweisung um.

Können Sie sich vorstellen, an einem Projekt zu arbeiten, bei dem der Code im Laufe der Zeit *einfach besser wurde*? Glauben Sie, dass ein anderes Verhalten professionell wäre? Oder ist die laufende Verbesserung vielleicht ein wesentliches Merkmal von Professionalität?

1.7 Vorläufer und Prinzipien

In vieler Hinsicht ist dieses Buch ein »Vorläufer« zu meinem Buch *Agile Software Development: Principles, Patterns, and Practices* (PPP) aus dem Jahre 2002. Das PPP-Buch behandelt die Prinzipien des Objekt-orientierten Design (OOD) und viele Techniken, die von professionellen Entwicklern eingesetzt werden. Falls Sie PPP nicht gelesen haben, werden Sie meinen, dass es eine Fortsetzung der Geschichte ist, die in diesem Buch erzählt wird. Wenn Sie es bereits gelesen haben, werden Sie in dem vorliegenden Buch einen Widerhall vieler Aussagen aus dem PPP-Buch finden, und zwar diesmal auf der Ebene des Codes.

In diesem Buch werden gelegentlich verschiedene Design-Prinzipien referenziert: das Single-Responsibility-Prinzip (SRP), das Open-Closed-Prinzip (OCP) und das Dependency-Inversion-Prinzip (DIP) und andere. Diese Prinzipien werden in PPP ausführlich beschrieben.

1.8 Zusammenfassung

Bücher über Kunst versprechen Ihnen nicht, aus Ihnen einen Künstler zu machen. Sie können Ihnen nur einige Werkzeuge, Techniken und Gedankenprozesse ver-

mitteln, die von anderen Künstlern verwendet worden sind. Deshalb verspricht Ihnen dieses Buch nicht, aus Ihnen einen guten Programmierer zu machen. Es kann Ihnen nicht versprechen, Ihnen ein Gefühl für den Code zu vermitteln. Es kann Ihnen nur die Gedankenprozesse von guten Programmierern aufzeigen und die Tricks, Techniken und Werkzeuge mitteilen, die sie verwenden.

Ähnlich wie ein Kunstbuch enthält dieses Buch zahlreiche Details. Es enthält umfangreichen Code. Sie sehen guten Code, und Sie sehen schlechten Code. Es wird demonstriert, wie schlechter Code in guten Code transformiert werden kann. Sie finden Listen mit Heuristiken, Prinzipien und Techniken. Und es wird Ihnen ein Beispiel nach dem anderen gezeigt. Danach sind Sie auf sich gestellt.

Kennen Sie die Geschichte von dem Konzert-Violinenspieler, der sich auf dem Weg zur Vorführung verlaufen hatte? Er fragte einen alten Mann an der Straßenecke nach dem Weg zur Carnegie Hall. Der alte Mann schaute den Violinenspieler an, sah die Geige unter seinem Arm und sagte: »Übung, mein Sohn. Übung!«