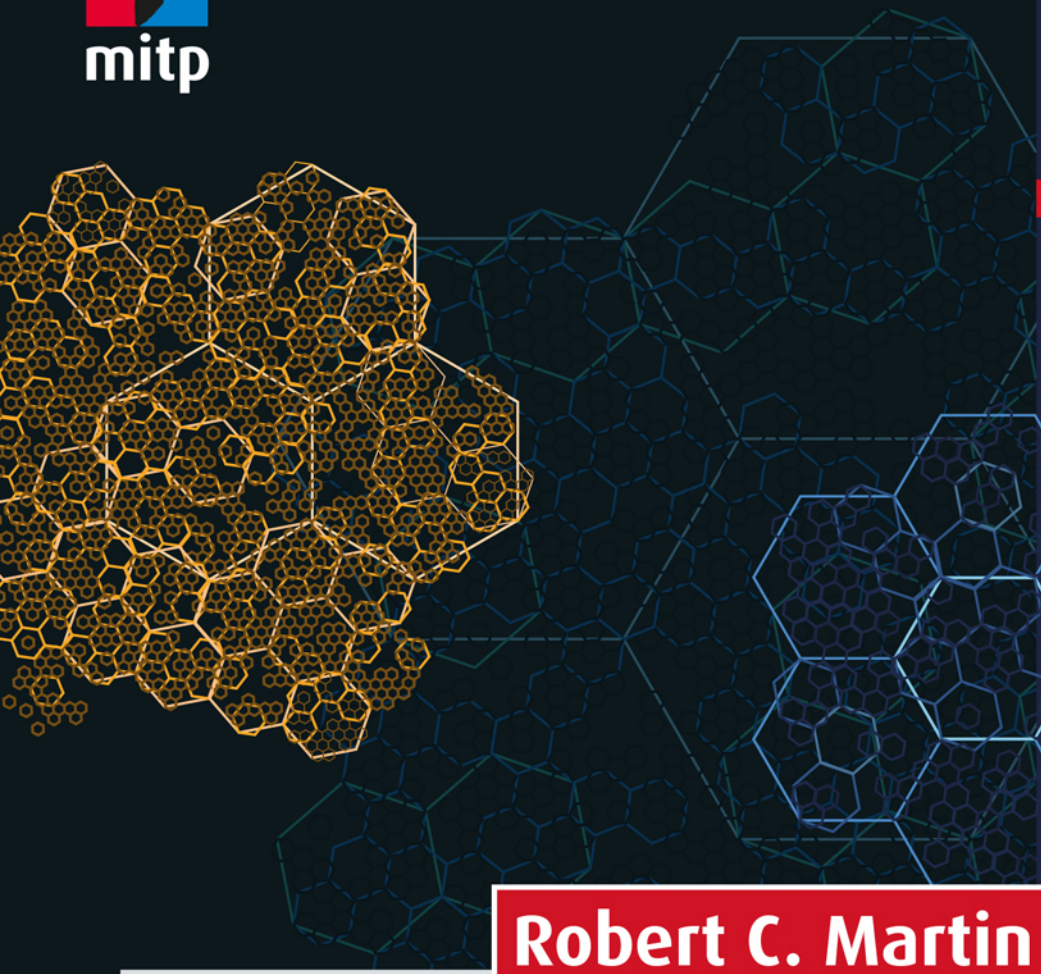
mitp

Mark Seemann

Vorwort von Robert C. Martin

Deutsche Ausgabe



Robert C. Martin Series

Code That Fits in Your Head

Heuristiken für die Softwareentwicklung

Komplexität reduzieren | Legacy Code beherrschen | Performance optimieren

Inhaltsverzeichnis

»Die Zukunft ist schon hier – sie ist nur nicht besonders gleichmäßig verteilt.«

– William Gibson

Vorwort des Herausgebers der englischen Ausgabe

Mein Enkel lernt jetzt, zu programmieren.

Ja, Sie haben richtig gelesen. Mein 18-jähriger Enkel lernt, Computer zu programmieren. Wer unterrichtet ihn? Seine Tante, meine jüngste Tochter, die 1986 geboren wurde und sich vor 16 Monaten entschlossen hat, doch nicht Chemikerin, sondern Programmiererin zu werden. Und für wen arbeiten die beiden? Mein ältester Sohn ist gerade im Begriff, zusammen mit meinem jüngsten Sohn ein zweites Software-Beratungsunternehmen zu gründen.

Ja, in unsere Familie spielt Software eine große Rolle. Und ja, ich programmiere schon sehr, sehr lange.

Wie dem auch sei, meine Tochter hat mich gebeten, dass ich eine Stunde mit meinem Enkel verbringen möge und ihm die Grundlagen und Anfänge der Programmierung von Computern erläutere. Wir setzten uns also zusammen, und ich erklärte ihm, was Computer eigentlich sind, wie alles begann, wie die ersten Computer aussahen und … na ja, Sie wissen schon.

Als die Lektion sich dem Ende näherte, programmierte ich einen Algorithmus zum Multiplizieren zweier binärer Ganzzahlen in der PDP-8-Assembler-Sprache. Falls Ihnen das nichts sagt: Bei der PDP-8 gab es keine Multiplikationsanweisung. Man musste selbst einen Algorithmus schreiben, um Zahlen miteinander zu multiplizieren. Tatsächlich gab es noch nicht einmal eine Subtraktionsanweisung. Man musste das Zweierkomplement verwenden und eine scheinbar negative Zahl addieren.

Nachdem ich das Codebeispiel fertiggestellt hatte, wurde mir klar, dass ich meinen Enkel wohl zu Tode erschreckt hatte. Als ich 18 Jahre alt war, haben mich solche abgefahrenen Details fasziniert, aber für einen 18-Jährigen, dessen Tante versucht, ihm beizubringen, wie man einfache Clojure-Programme schreibt (Clojure ist ein moderner Lisp-Dialekt), ist das vielleicht nicht so interessant.

Jedenfalls musste ich wieder daran denken, wie schwierig es tatsächlich ist, zu programmieren. Und es ist schwierig, wirklich schwierig. Vielleicht ist es das Schwierigste, was Menschen jemals versucht haben.

Ich will damit nicht sagen, dass es schwierig ist, den Code zu schreiben, um ein paar Primzahlen oder eine Fibonacci-Folge zu berechnen oder ein einfaches Bubble-Sort zu programmieren. Das ist nicht allzu schwierig. Aber ein System für die Luftverkehrskontrolle? Ein System zur Reisegepäckverfolgung? Eine Warenbestandsliste? Angry Birds? Das ist schwierig. Wirklich richtig schwierig.

Ich kenne Mark Seemann nun schon seit einigen Jahren. Ich kann mich aber nicht daran erinnern, ihn jemals getroffen zu haben. Gut möglich, dass wir niemals im selben Raum gewesen sind. Aber wir beide stehen durchaus durch Newsgroups und soziale Medien in regelmäßigem Kontakt. Er gehört zu den Leuten, denen ich am liebsten widerspreche.

Wir beide sind uns bei den unterschiedlichsten Dingen uneinig. Wir haben zur statischen bzw. dynamischen Typisierung, zu Betriebssystemen und Plattformen und zu Programmiersprachen verschiedene Meinungen. Wir sind uns bei vielen intellektuell herausfordernden Dingen nicht einig. Aber wenn man Mark widerspricht, muss man sehr vorsichtig sein, denn die Logik seiner Argumente ist bestechend.

Als ich dieses Buch in Händen hielt, stellte ich mir vor, wie viel Vergnügen ich dabei haben würde, es durchzulesen und anderer Ansicht zu sein. Und genauso war es auch. Ich war bei einigen Punkten anderer Ansicht und hatte meinen Spaß dabei, einen Weg zu finden, dass meine Logik die seine aussticht. Ich denke, in ein oder zwei Fällen ist mir das vielleicht sogar gelungen.

Aber das ist nicht der Punkt. Das Entscheidende ist, dass Softwareentwicklung schwierig ist und in den letzten sieben Jahrzehnten viel Zeit mit dem Versuch verbracht wurde, sie etwas einfacher zu machen. In diesem Buch hat Mark die besten Ideen aus diesen sieben Jahrzehnten zusammengetragen.

Darüber hinaus hat er sie in Form von Heuristiken und Verfahren organisiert und sie in der Reihenfolge sortiert, in der man sie anwenden würde. Die Heuristiken und Verfahren bauen aufeinander auf und helfen Ihnen so, bei der Entwicklung eines Softwareprojekts schrittweise voranzukommen.

Tatsächlich entwickelt Mark im Verlauf des Buchs ein Softwareprojekt und erklärt bei jeder Stufe, wie die Heuristiken und Verfahren funktionieren, von der Sie in dieser Stufe profitieren.

Mark verwendet C# (hier hätte ich mich anders entschieden), aber das ist nicht relevant. Der Code ist einfach, und die Heuristiken und Verfahren lassen sich auch auf irgendeine andere Sprache anwenden, die Sie vielleicht verwenden.

Er behandelt Themen wie Checklisten, TDD, Trennung von Befehlen und Anfragen, Git, zyklomatische Komplexität, referenzielle Transparenz, Vertical Slicing, die Überarbeitung von veraltetem Code und Outside-in-Entwicklung, um nur einige zu nennen.

Zudem finden sich buchstäblich überall im Buch verstreut kleine Schätze. Sie lesen so vor sich hin, und plötzlich schreibt er etwas wie »Wenn Sie sich vorstellen, dass Sie den Code um 90 Grad drehen, sollte der Umfang in etwa dem Umfang des dazugehörigen *Act*-Abschnitts entsprechen« oder »Das Ziel ist nicht, Code möglichst schnell zu schreiben. Das Ziel ist nachhaltige Software« oder »Übergeben Sie das Datenbankschema dem Git-Repository«.

Einige dieser kleinen Schätze sind tiefgründig, einige nur beiläufige Erwähnungen und wiederum andere sind Spekulationen, aber alle sind Beispiele für die weitreichenden Erkenntnisse, die Mark im Laufe der Jahre gewonnen hat.

Lesen Sie dieses Buch. Lesen Sie es sorgfältig. Durchdenken Sie Marks bestechende Logik. Verinnerlichen Sie die Heuristiken und Verfahren. Halten Sie inne, und machen Sie sich über die aufschlussreichen kleinen Schätze Gedanken, wenn sie auftauchen. Und wenn die Zeit dafür gekommen ist, dass Sie Ihren Enkeln etwas beibringen möchten, dann werden Sie sie vielleicht nicht zu Tode erschrecken.

— Robert C. Martin

Einleitung

In der zweiten Hälfte der 2000er-Jahre habe ich angefangen, fachliche Gutachten für einen Verlag zu erstellen. Nachdem ich eine Handvoll Bücher beurteilt hatte, kontaktierte mich der Herausgeber wegen eines Buchs über Dependency Injection.

Das Angebot war ein wenig merkwürdig. Wenn der Verlag mich wegen eines Buchs kontaktierte, hatte es für gewöhnlich schon einen Autor und ein Inhaltsverzeichnis. Dieses Mal war das nicht der Fall. Der Herausgeber bat mich lediglich um einen Anruf, um zu besprechen, ob das Thema des Buchs tauglich sei.

Ich dachte ein paar Tage darüber nach und fand das Thema anregend. Andererseits konnte ich nicht erkennen, warum dafür ein ganzes Buch erforderlich sein sollte. Schließlich war das Wissen schon vorhanden: in Blogbeiträgen, in der Dokumentation von Bibliotheken, in Fachmagazinartikeln, und es gab sogar einige Bücher, die das Thema streiften.

Als ich darüber nachdachte, fiel mir auf, dass die Informationen zwar vorhanden waren, sie waren aber weit verstreut, und es wurde eine uneinheitliche und teilweise widersprüchliche Terminologie verwendet. Es wäre durchaus sinnvoll, dieses Wissen zusammenzutragen und in einer einheitlichen Sprache zu präsentieren.

Zwei Jahre später war ich der stolze Autor eines veröffentlichten Buchs.

Nachdem ein paar Jahre vergangen waren, begann ich darüber nachzudenken, ein weiteres Buch zu verfassen. Nicht dieses, sondern ein Buch über ein anderes Thema. Dann hatte ich eine dritte Idee und eine vierte, aber nicht die Idee zu diesem Buch.

Ein Jahrzehnt verging, und mir wurde klar, dass ich, wenn ich Teams anleitete, besseren Code zu schreiben, Verfahren vorschlug, die ich von anderen gelernt hatte, die klüger waren als ich. Und wieder fiel mir auf, dass der größte Teil dieses Wissens schon vorhanden, aber weit verstreut war, und nur wenige hatten einen Zusammenhang hergestellt, um eine einheitliche Beschreibung zu geben, wie man Software entwickelt.

Aufgrund meiner Erfahrung mit dem ersten Buch wusste ich, dass es durchaus sinnvoll ist, die verschiedenen Informationen zusammenzutragen und sie auf einheitliche Weise darzustellen. Dieses Buch ist mein Versuch, genau das zu tun.

Wer dieses Buch lesen sollte

Dieses Buch richtet sich an Programmierer, die bereits einige Jahre Berufserfahrung haben. Ich gehe davon aus, dass Leser einige schlechte Softwareentwicklungsprozesse erleben mussten und Erfahrung mit nicht wartbarem Code haben. Zudem erwarte ich, dass die Leser das verbessern wollen.

Zur Zielgruppe gehören vor allem Entwickler, die in Unternehmen tätig sind, insbesondere Backend-Entwickler. Ich war in meiner Laufbahn meistens in diesem Bereich tätig, daher spiegelt das nur meine eigene Kompetenz wider. Wenn Sie Frontend-Entwickler, Spieleprogrammierer oder Programmierer von Entwicklungstools oder etwas ganz anderes sind, gehe ich davon aus, dass die Lektüre dieses Buchs Ihnen dennoch eine Menge bringen wird.

Sie sollten damit vertraut sein, den Code einer kompilierten objektorientierten Sprache aus der C-Familie zu lesen. Ich war in meiner Laufbahn zwar die meiste Zeit C#-Programmierer, habe aber auch eine Menge aus Büchern mit Codebeispielen in C++ oder Java gelernt.[1] Bei diesem Buch sind die Rollen vertauscht. Die Codebeispiele sind zwar in C# geschrieben, aber ich hoffe, dass Entwickler, die Java, TypeScript oder C++ verwenden, sie ebenfalls nützlich finden.

Voraussetzungen

Dieses Buch ist nicht für Anfänger gedacht. Es wird zwar erläutert, wie man Quellcode organisiert und strukturiert, aber die meisten Grundlagen kommen nicht zur Sprache. Ich gehe davon aus, dass Sie bereits wissen, warum Einrückungen nützlich und lange Funktionen problematisch sind, dass globale Variablen schlecht sind und so weiter.

Ein Hinweis für Softwarearchitekten

Der Begriff »Architekt« hat für verschiedene Menschen unterschiedliche Bedeutung, sogar wenn der Kontext auf Softwareentwicklung eingeschränkt ist. Manche Architekten konzentrieren sich auf das Gesamtbild; sie helfen einer ganzen Organisation dabei, dass ihre Anstrengungen erfolgreich sind. Andere Architekten befassen sich intensiv mit dem Code und sind vor allem an der Nachhaltigkeit einer bestimmten Codebasis interessiert.

Sofern man mich als Softwarearchitekt bezeichnen kann, gehöre ich zu Letzteren. Meine Kompetenz besteht darin, den Quellcode so zu organisieren, dass langfristige geschäftliche Ziele erreicht werden. Ich schreibe über das, was ich weiß. Sofern dieses Buch für Architekten nützlich ist, dann für Architekten der zweiten Gruppe.

1 Sehen Sie sich die Bibliografie an, wenn Sie neugierig sind, welche Bücher ich meine.

Zu Themen wie *Architecture Tradeoff Analysis Method* (ATAM), *Failure Mode and Effects Analysis* (FMEA), Diensterkennung und so weiter werden Sie nichts lesen, denn sie gehen über den Rahmen dieses Buchs hinaus.

Aufbau des Buchs

In diesem Buch geht es um Methodologien, deshalb orientiert es sich an einem Codebeispiel, das im Buch durchgängig verwendet wird. Ich habe mich dafür entschieden, damit die Lektüre überzeugender ist als das Lesen eines typischen Musterkatalogs. Diese Entscheidung hat zur Folge, dass ich Verfahren und Heuristiken dann vorstelle, wenn sie zu den geschilderten Themen passen. In dieser Reihenfolge stelle ich die Verfahren normalerweise auch vor, wenn ich Teams berate.

Die Schilderung beruht auf einer Beispiel-Codebasis, die ein Reservierungssystem für ein Restaurant implementiert. Der Quellcode für dieses Beispiel ist unter `www.mitp.de/0514` verfügbar.

Wenn Sie das Buch als Nachschlagewerk verwenden möchten, finden Sie im Anhang eine Liste aller Verfahren mit der Angabe, wo Sie im Buch weitere Informationen finden.

Über den Codestil

Der Beispielcode ist in C# geschrieben, einer Sprache, die sich in den letzten Jahren rasant weiterentwickelt hat. Die Sprache übernimmt immer mehr Syntaxelemente aus der funktionalen Programmierung. Beispielsweise wurden *unveränderliche Record-Datentypen* veröffentlicht, während ich dieses Buch geschrieben habe. Ich habe mich dafür entschieden, einige der neuen Sprachfeatures zu ignorieren.

Früher einmal hatte Java-Code große Ähnlichkeit mit C#-Code. Moderner C#-Code hat kaum noch Ähnlichkeit mit Java.

Ich möchte, dass der Code für so viele Leser wie möglich verständlich ist. Ebenso wie ich viel aus Büchern mit Java-Beispielen gelernt habe, möchte ich, dass die Leser in der Lage sind, das Buch zu verwenden, ohne die neueste C#-Syntax zu kennen. Deshalb versuche ich, nur eine Teilmenge von C# zu verwenden, die auch für Programmierer anderer Sprachen verständlich sein sollte.

Das ändert nichts an den im Buch vorgestellten Konzepten. Ja, in manchen Fällen sind kompaktere C#-spezifische Alternativen möglich, aber das bedeutet nur, dass zusätzliche Verbesserungen möglich sind.

var verwenden oder var nicht verwenden

Das Schlüsselwort `var` wurde 2007 in C# eingeführt. Es ermöglicht, eine Variable zu deklarieren, ohne ausdrücklich ihren Typ anzugeben. Stattdessen ermittelt der Compiler den Typ anhand des Kontexts. Der Deutlichkeit halber: Mit `var` deklarierte Variablen sind ebenso statisch typisiert wie Variablen, die explizit mit einem Typ deklariert werden.

Die Verwendung dieses Schlüsselworts war lange umstritten, aber die meisten verwenden es *inzwischen*. Ich auch, aber gelegentlich stoße ich dabei auf Widerstand. Beruflich verwende ich zwar `var`, aber den Code für ein Buch zu schreiben, ist ein etwas anderer Kontext. Unter normalen Umständen steht eine IDE zur Verfügung. Eine moderne Entwicklungsumgebung kann Ihnen schnell den Typ einer implizit typisierten Variable anzeigen, aber ein Buch kann das nicht.

Aus diesem Grund verwende ich gelegentlich explizit typisierte Variablen. Der meiste Beispielcode verwendet das Schlüsselwort `var`, weil der Code dadurch kürzer wird und die Zeilenbreite in einem gedruckten Buch begrenzt ist. In einigen wenigen Fällen habe ich mich allerdings bewusst dafür entschieden, den Typ einer Variablen explizit zu deklarieren, in der Hoffnung, dass der Code besser verständlich ist, wenn man ihn in einem Buch liest.

Code-Listings

Die meisten Code-Listings sind derselben Codebasis entnommen. Dabei handelt es sich um ein Git-Repository und die Codebeispiele entstammen verschiedenen Stufen der Entwicklung. Bei den Code-Listings ist der relative Pfad zur entsprechenden Datei angegeben. Ein Teil davon ist eine Git-Commit-ID.

Bei Listing 2.1 ist beispielsweise der relative Pfad *Restaurant/f729ed9/Restaurant.RestApi/Program.cs* angegeben. Das bedeutet, dass das Beispiel der Commit-ID `f729ed9` entstammt, und die Datei ist `Restaurant.RestApi/Program.cs`. Mit anderen Worten: Wenn Sie diese Version der Datei anzeigen möchten, müssen Sie dieses Commit auschecken:

```
$ git checkout f729ed9
```

Wenn Sie das erledigt haben, können Sie die Datei `Restaurant.RestApi/Program.cs` in ihrem vollständigen ausführbaren Kontext anzeigen.

Hinweis zur Bibliografie

In der Bibliografie finden Sie verschiedene Ressourcen: Bücher, Blogbeiträge und Videoaufzeichnungen. Viele der Quellen sind online, daher habe ich natürlich

URLs angegeben. Ich habe mich bemüht, vor allem Ressourcen anzugeben, bei denen ich Grund zu der Annahme habe, dass sie im Internet dauerhaft verfügbar sind.

Aber die Dinge ändern sich. Wenn Sie dieses Buch in der Zukunft lesen und eine URL ungültig geworden ist, können Sie versuchen, auf einen Internetarchivdienst zurückzugreifen. Derzeit ist `https://archive.org` der beste Kandidat, aber diese Website könnte es in der Zukunft auch nicht mehr geben.

Eigene Zitate

Neben den anderen Ressourcen enthält die Bibliografie auch eine Liste meiner eigenen Arbeiten. Mir ist klar, dass eigene Zitate an sich keine stichhaltigen Argumente sind.

Es soll aber kein Taschenspielertrick sein, wenn ich auf meine eigene Arbeit verweise. Ich stelle diese Ressourcen vielmehr für Leser zur Verfügung, die möglicherweise an weiteren Details interessiert sind. Wenn ich mich selbst zitiere, mache ich das, weil Sie in den Ressourcen, auf die ich verweise, möglicherweise ausführlichere Argumente oder detailliertere Codebeispiele finden können.

Danksagung

Ich danke meiner Frau Cecilie für ihre Liebe und Unterstützung in all den Jahren, die wir zusammen sind, und meinen Kindern Linea und Jarl, dass sie uns keinen Kummer bereiten.

Neben meiner Familie gilt mein Dank meinem langjährigen guten Freund Karsten Strøbæk , der nicht nur seit 25 Jahren mein Dasein toleriert, sondern auch der erste Leser dieses Buchs war. Zudem hat er mit verschiedenen Tipps und Tricks für LaTeX geholfen und dem Stichwortverzeichnis der englischen Ausgabe viele Einträge hinzugefügt.

Ich möchte auch Adam Tornhill für sein Feedback zum Abschnitt über seine Arbeit danken.

Außerdem bin ich Dan North dafür zu Dank verpflichtet, dass er den Ausdruck *Code That Fits in your Head* in mein Unterbewusstsein »eingepflanzt« hat. Das war wohl schon 2011 [72].

Downloads zum Buch

Den Programmcode der im Buch verwendeten Codebeispiele finden Sie unter `www.mitp.de/0514` im Reiter »Downloads« unterhalb des Buchcovers.

Über den Autor

Mark Seemann ist ein miserabler Volkswirtschaftler, dem eine zweite Karriere als Programmierer gelang. Er hat seit Ende der 1990er-Jahre als Webentwickler gearbeitet und war in verschiedenen Unternehmen als Programmierer tätig. Als junger Mann wollte Mark eigentlich Rockstar werden, aber dafür fehlten ihm leider sowohl das Talent als auch das Aussehen. Später wurde er dann allerdings zertifizierter Rockstar-Entwickler. Darüber hinaus hat er ein Buch über Dependency Injection geschrieben, das mit einem Jolt Award ausgezeichnet wurde. Er hat auf internationalen Konferenzen mehr als hundert Vorträge gehalten und ist Verfasser von Videokursen für *Pluralsight* und *Clean Coders*. Seit 2006 veröffentlicht er regelmäßig Blogbeiträge. Er lebt mit seiner Frau und zwei Kindern in Kopenhagen.

Kapitel 1

Kunst oder Wissenschaft?

Sind Sie Wissenschaftler oder Künstler? Ingenieur oder Kunsthandwerker? Gärtner oder Chefkoch? Dichter oder Architekt?

Sind Sie Programmierer oder Softwareentwickler? Wenn ja, was sind Sie dann?

Meine Antwort auf diese Frage lautet: *Nichts von alldem.*

Ich würde mich selbst zwar als Programmierer bezeichnen, fühle mich aber allen genannten Gruppen ein wenig zugehörig – und doch auch wieder nicht.

Fragen wie diese sind wichtig. Die Branche der Softwareentwicklung ist rund 70 Jahre alt, und wir grübeln immer noch darüber nach. Ein hartnäckiges Problem ist, wie wir darüber *denken*. Deshalb stellen sich diese Fragen. Gleicht die Softwareentwicklung dem Bau eines Hauses? Oder dem Verfassen eines Gedichts?

Im Laufe der Jahrzehnte haben wir diverse Metaphern ausprobiert, doch sie lösen sich alle in Wohlgefallen auf. Softwareentwicklung gleicht dem Bau eines Hauses, es sei denn, das ist nicht der Fall. Softwareentwicklung gleicht dem Kultivieren eines Gartens, es sei denn, das ist nicht der Fall. Letztendlich passt keine der Metaphern.

Ich bin überzeugt, dass die Art, wie wir über Softwareentwicklung denken, unsere Arbeitsweise beeinflusst.

Wenn Sie glauben, dass Softwareentwicklung dem Bau eines Hauses gleicht, werden Sie Fehler begehen.

1.1 Bau eines Hauses

Seit Jahrzehnten wird die Entwicklung von Software mit dem Bau eines Hauses verglichen. Kent Beck formuliert das so:

»Leider wurde das Design von Software von Metaphern physischer Designaktivitäten in Ketten gelegt.« [5]

Dabei handelt es sich um eine der weitverbreitetsten, verführerischsten und hinderlichsten Metaphern für Softwareentwicklung.

1.1.1 Das Problem mit Projekten

Wenn Sie glauben, dass die Entwicklung von Software dem Bau eines Hauses gleicht, werden Sie als Erstes den Fehler begehen, die Sache als Projekt zu betrachten. Ein Projekt hat einen Anfang und ein Ende. Sobald Sie am Ende angekommen sind, ist die Arbeit erledigt.

Nur misslungene Software kommt zu einem Ende. Erfolgreiche Software hat Bestand. Wenn Sie das Glück haben, eine erfolgreiche Software zu entwickeln, werden Sie sich nach Fertigstellung einer Version mit der Entwicklung der nächsten befassen. Das kann jahrelang so weitergehen. Manche erfolgreiche Software gibt es seit Jahrzehnten.[1]

Sobald Sie ein Haus erbaut haben, können die Bewohner einziehen. Sie müssen Wartungsarbeiten durchführen, aber die Kosten dafür sind nur ein Bruchteil der ursprünglichen Baukosten. Zugegebenermaßen gibt es auch Software, für die das zutrifft. Insbesondere im Unternehmensbereich ist die Entwicklung einer internen geschäftlichen Anwendung nach der Fertigstellung abgeschlossen, und die Benutzer sind daran gebunden. Sobald das Projekt abgeschlossen ist, erreicht die Software den Wartungsmodus.

Auf die meiste Software trifft dies jedoch nicht zu. Software, die im Wettbewerb mit anderer Software steht, ist niemals fertig. Wenn Sie bei der Hausbau-Metapher bleiben möchten, können Sie sich eine Reihe von Projekten vorstellen. Vielleicht planen Sie, die nächste Version Ihres Produkts in neun Monaten zu veröffentlichen, müssen aber zu Ihrem Schrecken feststellen, dass Ihr Wettbewerber alle drei Monate verbesserte Versionen veröffentlicht.

Sie arbeiten also hart daran, die Dauer der »Projekte« zu verkürzen. Und wenn Sie schließlich in der Lage sind, alle drei Monate eine neue Version auszuliefern, hat Ihr Wettbewerber einen einmonatigen Veröffentlichungszyklus eingeführt. Sie können sich wohl denken, wo das hinführt, oder?

Es führt zu Continuos Delivery (CD) (kontinuierliche Auslieferung). Ansonsten werden Sie früher oder später das Geschäft aufgeben müssen. Das Buch *Accelerate* [29] legt anhand von Studien überzeugend dar, dass die entscheidende Fähigkeit, die sehr leistungsfähige Teams von nur wenig leistungsfähigen unterscheidet, darin besteht, auf der Stelle eine neue Version veröffentlichen zu können.

Wenn Sie auf diese Weise vorgehen können, ergibt die Vorstellung eines Softwareentwicklungs-*Projekts* gar keinen Sinn mehr.

1 Ich verfasse die Originalausgabe dieses Buchs mit LaTeX – ein Programm, das 1984 erstmals veröffentlicht wurde!

1.1.2 Das Problem mit Phasen

Ein weiteres Missverständnis, das oft mit der Hausbau-Metapher einhergeht, ist, dass die Softwareentwicklung in verschiedenen *Phasen* erfolgen sollte. Beim Hausbau zeichnet der Architekt zunächst verschiedene Pläne. Anschließend wird die Logistik vorbereitet und das Baumaterial geliefert. Mit dem Bauen können Sie erst anfangen, wenn das erledigt ist.

Wenn diese Metapher auf Softwareentwicklung angewendet wird, ernennen Sie einen Softwarearchitekten, der dafür verantwortlich ist, einen Plan zu erstellen. Die Entwicklung sollte erst dann beginnen, wenn der Plan fertig ist. Diese Sichtweise der Softwareentwicklung betrachtet die Planung als diejenige Phase, in der geistige Arbeit geleistet wird. Der Metapher zufolge entspricht die Programmierung der eigentlichen Bauphase eines Hauses. Entwickler werden als austauschbare Bauarbeiter betrachtet, die kaum mehr als bessere Schreibkräfte sind.[2]

Nichts könnte weiter von der Wahrheit entfernt sein. Jack Reeves wies 1992 [87] darauf hin, dass das Kompilieren des Quellcodes die eigentliche Bauphase der Softwareentwicklung darstellt. Im Gegensatz zum Hausbau fällt dabei praktisch kein Aufwand an. Die gesamte Arbeit wird in der Designphase erledigt. Kevlin Henney hat das sehr eloquent folgendermaßen formuliert:

»Die Beschreibung eines Programms durch eindeutige Details und die Programmierung sind ein und dasselbe.« [42]

Bei der Softwareentwicklung gibt es also keine nennenswerte Bauphase. Das bedeutet nicht, dass eine Planung nicht nützlich wäre, ist jedoch ein Hinweis darauf, dass die Hausbau-Metapher bestenfalls nicht hilfreich ist.

1.1.3 Abhängigkeiten

Beim Hausbau gibt es Einschränkungen durch die physischen Gegebenheiten. Sie müssen zuerst ein Fundament errichten und anschließend die Mauern hochziehen. Das Dach kann erst aufgesetzt werden, wenn das erledigt ist. Mit anderen Worten: Das Dach ist von den Mauern abhängig, die wiederum vom Fundament abhängig sind.

Die Metapher lässt manche Leute irrtümlich glauben, dass sie die Abhängigkeiten managen müssen. Ich kenne Projektmanager, die ausgefeilte Gantt-Diagramme erstellen haben, um ein Projekt zu planen.

Ich habe mit zahlreichen Teams zusammengearbeitet. Die meisten starten ein neues Entwicklungsprojekt mit dem Design eines Relationenschemas einer Datenbank. Die Datenbank bildet die Grundlage für die meisten Onlinedienste,

2 Ich habe nichts gegen Bauarbeiter; mein geliebter Vater war Maurer.

und die Teams können sich offenbar nicht an den Gedanken gewöhnen, eine Benutzeroberfläche zu entwickeln, bevor die Datenbank vorhanden ist.

Einigen Teams gelingt es nie, eine funktionierende Software zu erstellen. Nachdem sie die Datenbank designt haben, denken sie, dass sie ein *Framework* benötigen, um sie zu verwenden. Sie fahren also damit fort, die objektrelationale Abbildung (engl. *object-relational mapping*, ORM, siehe Neward, Ted, *The Vietnam of Computer Science* [70]) noch einmal neu zu erfinden.

Die Hausbau-Metapher ist gefährlich, weil sie zu einer bestimmten Denkweise über Softwareentwicklung führt. Ihnen entgehen Möglichkeiten, die Sie nicht erkennen, weil Ihre Sichtweise nicht an der Realität ausgerichtet ist. Tatsächlich verhält es sich in der Softwareentwicklung so, dass man – metaphorisch gesagt – sehr wohl mit dem Dach anfangen kann. Ein Beispiel hierfür folgt später im Buch.

1.2 Kultivieren eines Gartens

Die Hausbau-Metapher ist falsch, aber vielleicht funktionieren andere Metaphern besser. Seit 2010 hat die Gartenbau-Metapher an Beliebtheit gewonnen. Es ist kein Zufall, dass Nat Pryce und Steve Freeman ihrem ausgezeichneten Buch den Titel *Growing Object-Oriented Software, Guided by Tests* [36] verliehen haben.

Diese Sichtweise auf Softwareentwicklung betrachtet Software als lebenden Organismus, der gehegt, gepflegt und zurechtgeschnitten werden muss. Es handelt sich um eine weitere überzeugende Metapher. Hatten Sie schon einmal das Gefühl, dass eine Codebasis ein Eigenleben führt?

Es kann durchaus aufschlussreich sein, Softwareentwicklung in diesem Licht zu betrachten. Zumindest kann es eine Änderung der Sichtweise erzwingen, die Ihren Glauben ins Wanken bringt, dass Softwareentwicklung dem Bau eines Hauses gleicht.

Betrachtet man Software als lebenden Organismus, dann legt die Gartenbau-Metapher einen Schwerpunkt auf das Zurechtschneiden. Überlässt man einen Garten sich selbst, wird er verwildern. Um Nutzen aus einem Garten zu ziehen, muss ein Gärtner Unkraut entfernen und die erwünschten Pflanzen hegen und pflegen. Auf die Softwareentwicklung übertragen, bedeutet das, dass es hilfreich ist, eine Überalterung des Codes (»Code Rot«) zu *verhindern*, etwa durch Refactoring und das Löschen von totem Code.

Ich halte diese Metapher für nicht annähernd so problematisch wie die Hausbau-Metapher, denke aber nach wie vor, dass sie den Sachverhalt nicht vollständig erfasst.

1.2.1 Was lässt einen Garten gedeihen?

Mir gefällt, dass die Gartenbau-Metapher Wert darauf legt, Unordnung zu bekämpfen. Einen Garten müssen Sie zurechtschneiden, und Sie müssen Unkraut jäten. Bei einer Codebasis müssen Sie Refactorings durchführen und technische Schulden tilgen.

Die Gartenbau-Metapher sagt andererseits kaum etwas dazu aus, woher der Code stammt. In einem Garten wachsen die Pflanzen von allein. Sie benötigen lediglich Nährstoffe, Wasser und Licht. Software hingegen entwickelt sich nicht von allein. Wenn Sie einen Computer, Chips und Getränke in einen dunklen Raum abstellen, dürfen Sie nicht erwarten, dass dadurch Software entsteht. Ihnen fehlt ein wichtiger Bestandteil: Programmierer.

Der Code wird von irgendjemandem geschrieben. Dabei handelt es sich um einen aktiven Prozess, und die Gartenbau-Metapher macht dazu kaum eine Aussage. Wie entscheiden Sie, was programmiert wird und was nicht? Wie entscheiden Sie, *wie* ein Codeabschnitt strukturiert werden soll?

Wenn wir die Branche der Softwareentwicklung verbessern möchten, müssen wir diese Fragen beantworten.

1.3 Der Weg zur Ingenieurswissenschaft

Es gibt noch einige weitere Metaphern für Softwareentwicklung. Ich habe beispielsweise schon den Begriff *technische Schulden* genannt, der die Sichtweise eines Buchhalters beschreibt. Zudem habe ich kurz den Prozess des *Schreibens* von Code erwähnt, der es nahelegt, dass es Ähnlichkeiten zu anderen Arten des Verfassens von Inhalten gibt. Nur wenige Metaphern sind komplett falsch, aber es gibt auch keine, die vollständig richtig ist.

Es gibt Gründe dafür, dass ich insbesondere die Hausbau-Metapher ins Visier genommen habe. Beispielsweise weil sie so weitverbreitet ist. Ein weiterer Grund ist, dass sie so falsch zu sein scheint, dass sie dadurch unhaltbar wird.

1.3.1 Software als Kunsthandwerk

Ich bin schon vor vielen Jahren zu dem Schluss gelangt, dass die Hausbau-Metapher gefährlich ist. Aber sobald man eine Sichtweise verwirft, sucht man sich typischerweise eine neue. Ich habe mich für *Software Craftsmanship* (»Software Handwerkskunst«) entschieden.

Es erscheint verlockend, Softwareentwicklung als Kunsthandwerk zu betrachten, also als fachgerechte Arbeit. Man kann sich zwar zum Informatiker ausbilden lassen, das ist aber nicht unbedingt erforderlich. Ich habe das nicht gemacht.[3]

Die Fähigkeiten, die man braucht, um als professioneller Softwareentwickler zu arbeiten, sind meist situationsabhängig. Lernen Sie, wie die fragliche Codebasis strukturiert ist. Lernen Sie, wie das dazugehörige Framework verwendet wird. Durchleben Sie das Martyrium, drei Tage damit zu verschwenden, einen Bug im Produktionscode zu beheben. Solche Sachen.

Je mehr Sie sich damit befassen, desto professioneller werden Sie. Wenn Sie in einem bestimmten Unternehmen bleiben und jahrelang dieselbe Codebasis verwenden, werden Sie zu einem spezialisierten Experten, aber wird sich das als nützlich erweisen, wenn Sie sich entschließen, woanders zu arbeiten?

Sie können schneller lernen, wenn Sie von einer Codebasis zur nächsten wechseln. Probieren Sie aus, ein Backend zu entwickeln. Arbeiten Sie auch an der Entwicklung eines Frontends. Vielleicht versuchen Sie sich an der Entwicklung von Spielen, oder Sie befassen sich mit Machine Learning. Auf diese Weise lernen Sie ein breites Aufgabenspektrum kennen und sammeln Erfahrung.

Die Sache weist eine bemerkenswerte Ähnlichkeit zur alten europäischen Tradition der *Wanderjahre* (engl. *journeymans years*) auf. Handwerksgesellen, wie Schreiner oder Dachdecker, reisen quer durch Europa, arbeiten eine Weile an einem Ort und ziehen dann weiter. Auf diese Weise lernen sie für ihre Aufgaben verschiedene Lösungen kennen, was ihre handwerklichen Fähigkeiten verbessert.

Es ist verlockend, auch Softwareentwickler so zu betrachten. Das Buch *The Pragmatic Programmer* trägt sogar den Untertitel *From Journeyman to Master* [50].

Wenn das alles stimmt, folgt daraus, dass wir unsere Branche dementsprechend strukturieren sollten. Auszubildende sollten zusammen mit Meistern arbeiten. Wir könnten sogar die Zünfte organisieren.

Das heißt, *wenn* das alles stimmt.

Software Craftsmanship ist eine weitere Metapher. Ich halte sie für aufschlussreich, aber wenn man ein Objekt ins Rampenlicht zerrt, erzeugt man auch Schatten. Je heller das Licht ist, desto dunkler ist der Schatten, wie in Abbildung 1.1 dargestellt.

3 Ich verfüge zwar über einen Universitätsabschluss, allerdings in Wirtschaftswissenschaft. Aber abgesehen von einer kurzen Dienstzeit beim dänischen Wirtschaftsministerium war ich nie auf diesem Gebiet tätig.

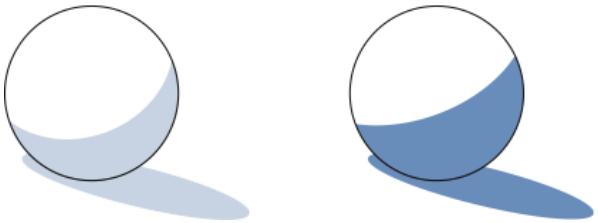


Abb. 1.1: Je heller das Licht ist, das auf ein Objekt fällt, desto dunkler erscheint der Schatten.

Dem Gesamtbild fehlt noch immer ein Teil.

1.3.2 Heuristik

Die Jahre, in denen ich mich mit Software Craftsmanship befasst habe, waren eine Zeit völliger Ernüchterung. Ich betrachtete Fähigkeiten als nichts weiter als angesammelte Erfahrung. Ich hatte den Eindruck, dass es keine Methodologie der Softwareentwicklung gab. Dass alles von den Umständen abhing. Dass es keinen richtigen oder falschen Weg gab, die Dinge anzugehen.

Dass Programmieren im Grunde genommen eine Kunst war.

Das gefiel mir gut. Kunst mochte ich schon immer. Als ich klein war, wollte ich ein Künstler werden.[4]

Diese Sichtweise hat allerdings den Haken, dass sie offenbar nicht skaliert. Um neue Programmierer hervorzubringen, musste man Auszubildende einstellen und abwarten, bis sie genug gelernt hatten, um Gesellen zu werden. Die Meisterhaftigkeit ist dann aber immer noch Jahre entfernt.

Programmierung als Kunst oder Kunsthandwerk zu betrachten, bringt ein weiteres Problem mit sich, nämlich dass es auch nicht zur Realität passt. Um 2010 wurde mir allmählich klar [106], dass ich beim Programmieren Heuristiken verwendete – Faustregeln und Richtlinien, die man unterrichten kann.

Zunächst kümmerte mich das wenig. Im Laufe der Jahre stellte ich jedoch fest, dass ich regelmäßig Aufgaben übernahm, bei denen ich andere Entwickler betreute. Dabei formulierte ich häufig Gründe, den Code auf bestimmte Art und Weise zu schreiben.

Mir wurde allmählich klar, dass ich mit meinem Nihilismus vermutlich falsch lag. Dass Richtlinien vielleicht der Schlüssel dazu sein könnten, aus der Programmierung ein planbares Fachgebiet zu machen.

4 Zuallererst wollte ich ein traditioneller europäischer Comiczeichner werden. Später, im Teenageralter, lernte ich, Gitarre zu spielen und träumte davon, ein Rockstar zu werden. Das Zeichnen und das Spielen der Gitarre bereiteten mir zwar viel Freude, aber wie sich herausstellte, war ich nicht besonders talentiert.

1.3.3 Ältere Vorstellungen von Software Engineering

Der Begriff Software Engineering (Softwaretechnik) geht auf die späten 1960er-Jahre zurück.[5] Er steht im Zusammenhang mit der damaligen Softwarekrise, die allmählich deutlich machte, dass Programmierung *schwierig* ist.

Die Programmierer damals hatten eine ziemlich gute Vorstellung von dem, was sie taten. Viele bekannte Persönlichkeiten unserer Branche waren in jenen Tagen aktiv: Edsger Dijkstra, Tony Hoare, Donald Knuth, Alan Kay. Wenn Sie die Leute damals gefragt hätten, ob sie glauben, dass Programmierung in den 2020er-Jahren eine Ingenieurswissenschaft ist, hätten sie wahrscheinlich mit »Ja« geantwortet.

Vielleicht ist Ihnen aufgefallen, dass ich den Begriff Software Engineering nicht als alltägliche Tatsache der Softwareentwicklung, sondern als ambitioniertes Ziel darstelle. Es ist durchaus möglich, dass es auf der Welt vereinzelte Gruppen gibt, die tatsächlich Software Engineering betreiben[6], aber meiner Erfahrung nach wird die meiste Softwareentwicklung auf andere Art und Weise durchgeführt.

Ich bin nicht der Einzige, der den Eindruck hat, dass Software Engineering noch ein zukünftiges Ziel ist. Adam Barr hat das sehr schön formuliert:

»Wenn es Ihnen so geht wie mir, dann träumen Sie von dem Tag, an dem Software Engineering auf wohlüberlegte, methodische Weise untersucht wird und sich die Richtlinien für Programmierer auf einem Fundament der Ergebnisse von Experimenten befinden und nicht im Treibsand individueller Erfahrung.« [4]

Er erklärt, dass Software Engineering auf einem guten Weg war, doch dann geschah etwas, das es zu Fall brachte. Was war passiert? Barr zufolge war es der Personal Computer, der eine Generation von Programmierern hervorbrachte, die sich zu Hause das Programmieren selbst beibrachten. Da sie ungestört an ihren Computern herumbasteln konnten, ignorierten sie das bereits vorhandene Fachwissen weitestgehend.

Das ist offenbar bis heute der Stand der Dinge. Alan Kay bezeichnet das als Popkultur:

»Aber die Popkultur verachtet die Geschichte. Bei der Popkulturgeht es vor allem um Identität und das Gefühl, teilzuhaben. Zusammenarbeit, die Zukunft oder die Vergangenheit spielen keine Rolle – alles dreht sich um die Gegenwart. Ich denke, das gilt auch für die meisten Menschen, die für das Schreiben von Code bezahlt werden. Sie haben keine Ahnung, woher ihre Kultur stammt.« [52]

5 Der Begriff könnte auch älter sein. Er ist mir nicht vollständig klar, und ich habe damals noch nicht gelebt, daher kann ich mich an nichts erinnern. Es ist jedoch offenbar unstrittig, dass zwei NATO-Konferenzen, die 1968 und 1969 stattfanden, den Begriff »software engineering« bekannt gemacht haben.

6 Die NASA scheint ein guter Kandidat hierfür zu sein.

Wir haben womöglich 50 Jahre vergeudet, in denen wir kaum Fortschritte beim Software Engineering erzielt haben, ich denke aber, dass wir auf anderem Wege Fortschritte gemacht haben.

1.3.4 Fortschritte beim Software Engineering

Was sind die Aufgaben eines Ingenieurs? Ingenieure designen und beaufsichtigen den Bau von Objekten. Dazu gehören große Konstruktionen, wie etwa Brücken, Tunnel, Wolkenkratzer und Kraftwerke, aber auch sehr kleine Objekte, wie Mikroprozessoren.[7] Sie helfen dabei, physische Objekte zu erstellen.

Abb. 1.2: Die Königin-Alexandrine-Brücke ist auch unter dem Namen Mønbroen (Mønbrücke) bekannt. Sie wurde 1943 fertiggestellt und verbindet die dänische Insel Seeland und die kleinere Insel Møn.

Programmierer tun das nicht. Software ist immateriell. Jack Reeves wies darauf hin [87], dass das Erstellen von Software praktisch keine Kosten verursacht, weil kein physisches Objekt produziert wird. Softwareentwicklung ist prinzipiell eine Entwurfstätigkeit. Wenn wir Code in einen Editor eingeben, handelt es sich um das Pendant zu einem Ingenieur, der einen Plan zeichnet, nicht zu den Arbeitern, die Objekte bauen.

»Richtige« Ingenieure nutzen eine Methodologie, die für gewöhnlich zu einem gelungenen Ergebnis führt. So würden wir Programmierer auch gerne vorgehen,

7 Ein Bekannter von mir ist studierter Chemie-Ingenieur. Nach Abschluss der Universität wurde er Brauer bei Carlsberg. Ingenieure brauen also auch Bier.

müssen dabei aber darauf achtgeben, nur solche Tätigkeiten zu übernehmen, die in unserem Kontext einen Sinn ergeben. Wenn sie physische Objekte *entwerfen*, ist die Umsetzung des Baus mit hohen Kosten verbunden. Sie können nicht mal einfach so eine Brücke bauen, einige Experimente anstellen, nur um zu entscheiden, dass sie nichts taugt, sie wieder abreißen und von vorn anfangen. Weil echtes Bauen teuer ist, nutzen Ingenieure Berechnungen und Simulationen. Es erfordert weniger Zeit und Material, die Stabilität einer Brücke zu berechnen, als sie tatsächlich zu bauen.

Es gibt ein eigenes Fachgebiet, das sich nur mit Logistik befasst. Die Planung erfolgt mit äußerster Sorgfalt, weil das die sicherste und preiswerteste Methode ist, physische Objekte zu bauen.

Hierbei handelt es sich um den Teil des Engineerings, den wir *nicht* übernehmen müssen.

Es gibt jedoch eine Vielzahl anderer Methodologien, die wir uns zum Vorbild nehmen können. Ingenieure leisten durchaus auch kreative Arbeit, sind dabei aber oft durch bestimmte Rahmenbedingungen eingeschränkt. Bestimmten Tätigkeiten sollten bestimmte andere Tätigkeiten folgen. Man begutachtet sich gegenseitig und gibt die Arbeit des anderen frei. Dazu werden Checklisten [40] verwendet.

Auf diese Weise können Sie ebenfalls vorgehen.

Darum geht es in diesem Buch. Es ist ein Leitfaden für die Verwendung von Heuristiken, die mir nützlich waren. Leider ähnelt er wohl eher dem, was Andrew Barr als *Treibsand der individuellen Erfahrung*bezeichnet, als einer Reihe wissenschaftlich begründeter Gesetze.

Ich denke, das spiegelt den gegenwärtigen Zustand unserer Branche wider. Alle, die glauben, dass es für irgendetwas handfeste wissenschaftliche Beweise gibt, sollten *The Leprechauns of Software Engineering* [13] lesen.

1.4 Fazit

Wenn Sie über die Geschichte der Softwareentwicklung nachdenken, stellen Sie sich vermutlich Fortschritte im Rahmen von mehreren Größenordnungen vor. Viele dieser Fortschritte sind jedoch der Hardware zu verdanken, nicht der Software. Dennoch haben wir in den letzten 50 Jahren enorme Fortschritte in der Softwareentwicklung beobachten können. Heutzutage gibt es sehr viel fortgeschrittenere Programmiersprachen als vor 50 Jahren. Wir können auf das Internet zugreifen (das in Form von Stack Overflow eine quasi sofortige Onlinehilfe bietet), uns stehen objektorientierte und funktionale Programmierung, automatisierte Test-Frameworks, Git, integrierte Entwicklungsumgebungen und so weiter zur Verfügung.

Andererseits haben wir noch immer mit der Softwarekrise zu kämpfen, wenngleich es umstritten ist, ob man tatsächlich von einer Krise sprechen kann, wenn sich der Zustand seit einem halben Jahrhundert nicht geändert hat.

Trotz ernsthafter Anstrengungen hat die Branche der Softwareentwicklung noch immer keine Ähnlichkeit mit einer Ingenieurswissenschaft. Es gibt einige fundamentale Unterschiede zwischen Engineering und Programmierung. Wenn wir das nicht begreifen, können wir keine Fortschritte erzielen.

Die gute Nachricht lautet, dass Sie viele der Dinge, die Ingenieure nutzen, übernehmen können. Es gibt eine bestimmte Geisteshaltung, die Sie einnehmen und eine Reihe von Verfahren, die Sie übernehmen können.

Der Science-Fiction-Autor William Gibson hat einmal gesagt:

»Die Zukunft ist schon hier – sie ist nur nicht besonders gleichmäßig verteilt.«[8]

Wie im Buch *Accelerate* dargestellt, nutzen einige Organisationen fortschrittliche Verfahren, andere hingegen bleiben zurück. Die Zukunft ist in der Tat ungleichmäßig verteilt. Die gute Nachricht lautet, dass für die fortschrittlichen Ideen keine Kosten anfallen. Es liegt in Ihrem Ermessen, sie auch umzusetzen.

In Kapitel 2 erhalten Sie einen ersten Eindruck davon, wie Sie konkret in Aktion treten können.

8 Hier handelt es sich um eines dieser Zitate, deren Ursprung im Dunkeln liegt. Es scheint unstrittig zu sein, dass die Idee und die endgültige Formulierung von Gibson stammen, aber wann genau er es erstmals gesagt hat, ist unklar. [76]

Stichwortverzeichnis

D

E

F

Q

R

S