

# 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 alledem.*

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. Gleichet 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.<sup>1</sup>

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 Continuous 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.<sup>2</sup>

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 Ingenieurwissenschaft

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.<sup>3</sup>

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.

---

<sup>3</sup> 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.

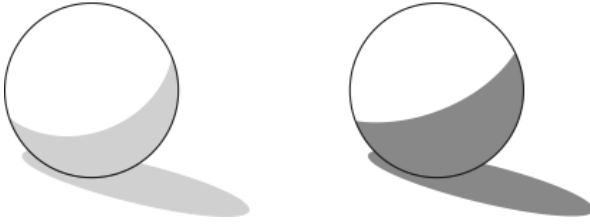


Abb. 1.1: Je heller das Licht ist, das auf ein Objekt fällt, desto dunkler erscheint der Schatten.

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.<sup>4</sup>

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.<sup>5</sup> 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 Ingenieurwissenschaft 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 vereinzelt Gruppen gibt, die tatsächlich Software Engineering betreiben<sup>6</sup>, 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 Popkultur geht 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.<sup>7</sup> 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,

---

<sup>7</sup> 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 Ingenieurwissenschaft. 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.«<sup>8</sup>*

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.

---

<sup>8</sup> 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]

Diese Leseprobe haben Sie beim  
 **edv-buchversand.de** heruntergeladen.  
Das Buch können Sie online in unserem  
Shop bestellen.

[Hier zum Shop](#)