DAVID FARLEY

MODERNES SOFTWARE ENGINEERING

Bessere Software schneller und effektiver entwickeln

mitp

Inhaltsverzeichnis

Vorwort

Ich habe an der Universität Informatik studiert und natürlich mehrere Module absolviert, die »Software Engineering« oder Variationen davon im Titel führten.

Als ich mein Studium begann, war das Programmieren nicht neu für mich und ich hatte bereits ein voll funktionsfähiges Inventarsystem für die Job- und Karrierebliothek meiner High School implementiert. Ich erinnere mich, dass mich der Begriff »Software Engineering« hochgradig verwirrte. Das alles schien, als ob es nur dazu diente, dem eigentlichen Schreiben von Code und der Auslieferung einer Anwendung in die Quere zu kommen.

Als ich in den ersten Jahren dieses Jahrhunderts meinen Abschluss machte, arbeitete ich in der IT-Abteilung eines großen Automobilkonzerns. Wie nicht anders zu erwarten, stand dort die Software-Entwicklung ganz weit oben auf der Liste. Hier habe ich mein erstes (aber sicher nicht mein letztes!) Gantt-Diagramm gesehen und die Wasserfall-Entwicklung miterlebt. Das heißt, ich sah, wie Software-Teams viel Zeit und Mühe in die Anforderungserhebung und die Entwurfsphase steckten, und viel weniger Zeit in die Implementierung (das Programmieren), was natürlich auch für die Testzeit galt, und für das Testen ... nun ja, dafür war nicht mehr viel Zeit.

Es schien, als ob das, was uns als »Software Engineering« verkauft wurde, der Entwicklung von qualitativ hochwertigen Anwendungen, die für unsere Kunden nützlich sind, im Weg stand.

Wie viele Entwickler dachte ich, dass es einen besseren Weg geben muss.

Ich habe etwas über Extreme Programming und Scrum gelesen. Ich wollte in einem agilen Team arbeiten und habe ein paarmal den Job gewechselt, um eines zu finden. Viele sagten, sie seien agil, aber oft lief es darauf hinaus, dass man Anforderungen oder Aufgaben auf Karteikarten schrieb, sie an die Wand klebte, eine Woche als *Sprint* bezeichnete und dann vom Entwicklungsteam verlangte, in jedem Sprint »x« Karten zu liefern, um eine willkürliche Frist einzuhalten. Die Abschaffung des traditionellen »Software Engineering«-Ansatzes schien auch nicht zu funktionieren.

Zehn Jahre nach Beginn meiner Karriere als Entwickler bewarb ich mich bei einer Finanzhandelsplattform in London. Der Leiter der Software-Abteilung erzählte mir, dass sie Extreme Programming anwendeten, einschließlich TDD und Pair

Programming. Er sagte, dass sie etwas praktizierten, das sie *Continuous Delivery* nannten, was etwa der Continuous Integration entsprach, sich aber bis in die Produktion erstreckte.

Ich hatte für große Investmentbanken gearbeitet, bei denen die Bereitstellung mindestens drei Stunden dauerte und mithilfe eines 12-seitigen Dokuments mit manuellen Schritten und Befehlen, die man eingeben musste, »automatisiert« wurde. Continuous Delivery schien eine schöne Idee zu sein, war aber sicher nicht möglich.

Der Leiter der Software-Abteilung war Dave Farley, und er war gerade dabei, sein Buch über *Continuous Delivery* zu schreiben, als ich in das Unternehmen kam.

Ich habe dort vier Jahre lang mit ihm zusammengearbeitet, Jahre, die mein Leben und meine Karriere verändert haben. Wir haben tatsächlich Pair Programming, TDD und Continuous Delivery angewendet. Außerdem lernte ich etwas über verhaltensgesteuerte Entwicklung (Behavior-driven Development), automatisierte Akzeptanztests, domänengetriebenes Design (Domain-driven Design), Trennung von Zuständigkeiten, Anti-Corruption-Layer, Mechanical Sympathy und Ebenen der Indirektion.

Ich lernte, wie man in Java hochperformante Anwendungen mit geringer Latenzzeit erstellt. Endlich verstand ich, was die O-Notation wirklich bedeutet und wie sie in der realen Welt der Programmierung eingesetzt werden kann. Kurz gesagt, all das, was ich an der Universität gelernt und in Büchern gelesen hatte, wurde tatsächlich angewendet.

Es wurde so eingesetzt, dass es Sinn hatte, funktionierte und eine extrem hochwertige, leistungsstarke Anwendung lieferte, die etwas bot, was es vorher nicht gab. Mehr noch, wir waren glücklich in unserem Job und zufrieden als Entwickler. Wir machten keine Überstunden, wir hatten keine Engpässe kurz vor der Veröffentlichung, der Code wurde in diesen Jahren nicht unübersichtlicher und unwartbarer und wir lieferten durchgehend und regelmäßig neue Funktionen und »Geschäftswert«.

Wie haben wir das erreicht? Indem wir den Verfahren folgten, die Dave in diesem Buch beschreibt. Es war nicht so formalisiert, und Dave hat eindeutig seine Erfahrungen aus vielen anderen Unternehmen eingebracht, um die spezifischen Konzepte herauszuarbeiten, die für ein breiteres Spektrum von Teams und Geschäftsbereichen anwendbar sind.

Was für zwei oder drei Teams an einer hochperformanten Finanzhandelsplattform funktioniert, wird nicht auf genau dieselbe Weise für ein großes Unternehmensprojekt in einem Produktionsbetrieb oder für ein schnelllebiges Start-up-Unternehmen gelten.

In meiner derzeitigen Position als Developer Advocate spreche ich mit Hunderten von Entwicklern aus den unterschiedlichsten Unternehmen und Geschäftsbereichen und höre mir ihre Probleme (von denen viele meinen eigenen Erfahrungen vor 20 Jahren nicht unähnlich sind) und ihre Erfolgsgeschichten an. Die Konzepte, die Dave in diesem Buch behandelt, sind allgemein genug, um in all diesen Umgebungen zu funktionieren, und spezifisch genug, um in der Anwendung hilfreich zu sein.

Komischerweise begann ich mich mit der Bezeichnung *Software Engineer* unwohl zu fühlen, nachdem ich Daves Team verlassen hatte. Ich dachte nicht, dass das, was wir als Entwickler tun, Engineering ist; ich dachte nicht, dass es das Engineering war, das das Team erfolgreich gemacht hatte. Ich dachte, dass Engineering eine zu strukturierte Disziplin für das ist, was wir tun, wenn wir komplexe Systeme entwickeln. Mir gefällt die Idee, dass es sich dabei um ein »Handwerk« handelt, da dies sowohl Kreativität als auch Produktivität beinhaltet, auch wenn es die Teamarbeit nicht ausreichend betont, die für die Arbeit an Software-Problemen in großem Maßstab erforderlich ist. Die Lektüre dieses Buchs hat meine Meinung geändert.

Dave erklärt deutlich, warum wir falsche Vorstellungen davon haben, was »echtes« Engineering ist. Er zeigt, dass Engineering eine wissenschaftsbasierte Disziplin ist, die aber nicht starr sein muss. Er erklärt Schritt für Schritt, wie wissenschaftliche Prinzipien und Engineering-Techniken auf die Software-Entwicklung angewendet werden können und warum die produktionsbezogenen Techniken, von denen wir dachten, sie seien ingenieurwissenschaftlich, für die Software-Entwicklung nicht geeignet sind.

Was ich an Daves Buch so toll finde, ist, dass er Konzepte nimmt, die abstrakt und schwierig auf echten Code anzuwenden scheinen, mit dem wir in unserem Job arbeiten müssen, und zeigt, wie wir sie als Tools einsetzen, um unsere konkreten Probleme anzugehen.

Das Buch begegnet der chaotischen Realität der Code-Entwicklung – oder sollte ich sagen, der Software-Entwicklung –, mit offenen Armen: Es gibt nicht die eine, einzig richtige Lösung. Die Dinge werden sich ändern. Was zu einem bestimmten Zeitpunkt richtig war, ist manchmal schon kurze Zeit später völlig falsch.

Die erste Hälfte des Buchs bietet praktische Lösungen, um in dieser Realität nicht nur zu überleben, sondern zu gedeihen. In der zweiten Hälfte werden Themen aufgegriffen, die von manchen als abstrakt oder akademisch angesehen werden könnten, und es wird gezeigt, wie man sie anwendet, um besseren (z.B. robusteren oder besser wartbaren oder in anderen Merkmalen »besseren«) Code zu designen.

Dabei bedeutet Design nicht unbedingt seitenlange Designdokumente oder UML-Diagramme, sondern kann so einfach sein wie »über den Code nachzudenken,

bevor oder während man ihn schreibt«. (Als ich mit Dave zusammen Pair Programming betrieben habe, ist mir unter anderem aufgefallen, wie wenig Zeit er mit dem eigentlichen Schreiben des Codes verbringt. Es stellte sich heraus, dass das Nachdenken über das, was wir schreiben, bevor wir es schreiben, uns eine Menge Zeit und Mühe ersparen kann).

Dave versucht nicht, Widersprüche bei der gemeinsamen Anwendung der Verfahren zu vermeiden oder die mögliche Verwirrung, die durch ein einzelnes Verfahren verursacht werden kann, aufzuklären. Weil er sich die Zeit nimmt, über die Kompromisse und die häufig auftretenden Unklarheiten zu sprechen, habe ich stattdessen zum ersten Mal verstanden, dass es genau die Balance und die Spannung zwischen diesen Dingen ist, die »bessere« Systeme schafft. Es geht darum zu verstehen, dass diese Dinge Richtlinien sind, ihre Vor- und Nachteile zu sehen und sie als Linsen zu verstehen, durch die man den Code/das Design/die Architektur betrachten kann, und gelegentlich als Stellschrauben, an denen man drehen kann, anstatt als binäre, schwarz-weiße, Richtig-oder-falsch-Regeln.

Durch die Lektüre dieses Buchs habe ich verstanden, warum wir in der Zeit, in der ich mit Dave gearbeitet habe, als »Software Engineers« so erfolgreich und zufrieden waren. Ich hoffe, dass Sie durch die Lektüre dieses Buchs von Daves Erfahrung und Ratschlägen profitieren können, ohne einen Dave Farley für Ihr Team einstellen zu müssen.

Viel Spaß beim Entwickeln!

– Trisha Gee, Developer Advocate und Java Champion

Einleitung

Dieses Buch bringt das *Engineering* zurück ins *Software Engineering*[1]. Ich beschreibe hier einen praktischen Ansatz für die Software-Entwicklung, bei dem ein bewusst rationaler, wissenschaftlicher Denkstil zur Lösung von Problemen eingesetzt wird. Diese Idee gründet auf der konsequenten Anwendung dessen, was wir in den letzten Jahrzehnten über Software-Entwicklung gelernt haben.

Mein Anliegen mit diesem Buch ist es, Sie davon zu überzeugen, dass Engineering vielleicht nicht das ist, wofür Sie es halten, und dass es völlig angemessen und effektiv ist, es im Rahmen der Software-Entwicklung anzuwenden. Danach werde ich die Grundlagen eines solchen Engineering-Ansatzes für Software-Entwicklung beschreiben und wie und warum er funktioniert.

Dabei geht es nicht um die neuesten Modetrends bei Prozessen oder Technologien, sondern um bewährte, praktische Ansätze, für die wir Daten haben, die uns zeigen, was funktioniert und was nicht.

Iteratives Arbeiten in kleinen Schritten funktioniert besser, als darauf zu verzichten. Wenn wir unsere Arbeit in eine Reihe kleiner, formloser Experimente aufteilen und Feedback einholen, um daraus zu lernen, können wir überlegter vorgehen und die Problem- und Lösungsräume erkunden, in denen wir uns bewegen. Indem wir unsere Arbeit so aufteilen, dass jeder Teil fokussiert, klar und verständlich ist, können wir unsere Systeme sicher und wohlüberlegt weiterentwickeln, auch wenn wir das Ziel nicht kennen, bevor wir beginnen.

Dieser Ansatz gibt uns eine Orientierung und zeigt uns, worauf wir uns konzentrieren müssen, selbst wenn wir die Antworten nicht kennen. Das verbessert unsere Erfolgschancen, egal wie die Herausforderung aussieht, vor der wir stehen.

In diesem Buch beschreibe ich ein Modell der Selbstorganisation, um großartige Software effizient und in jeder Skalierung zu entwickeln, sowohl für wirklich komplexe als auch für einfachere Systeme.

Es hat schon immer Gruppen von Menschen gegeben, die hervorragende Arbeit geleistet haben. Wir haben von innovativen Pionieren profitiert, die uns gezeigt

1 Anmerkung des Übersetzers: Software Engineering wird im Deutschen auch als »Software-Entwicklung« bezeichnet. Engineering als alleinstehender Begriff ist im Sinne von »Ingenieurswissenschaften« zu verstehen. Wir haben beide Begriffe im englischen Original belassen, um den Bezug zu erhalten.

haben, was möglich ist. In den letzten Jahren hat unsere Branche jedoch gelernt, besser zu erklären, was wirklich funktioniert. Wir verstehen jetzt besser, welche Ideen allgemeiner sind und breiter angewendet werden können, und uns liegen Daten vor, die diese Erkenntnisse bestätigen.

Wir können Software zuverlässiger, besser und schneller entwickeln, und wir haben Daten, die das belegen. Wir können extrem anspruchsvolle Probleme lösen, und wir haben Erfahrung mit vielen erfolgreichen Projekten und Unternehmen, die diese Behauptung bestätigen.

Der hier beschriebene Ansatz bildet eine Sammlung wichtiger grundlegender Konzepte und baut auf der Arbeit meiner Vorgänger auf. Dabei werden keine neuen Verfahren eingeführt, doch er führt wichtige Konzepte und Verfahren zu einem kohärenten Ganzen zusammen und gibt uns Prinzipien an die Hand, auf denen eine Disziplin der Software-Entwicklung aufgebaut werden kann.

Dies ist keine willkürliche Ansammlung unterschiedlicher Konzepte. Die Konzepte sind eng miteinander verwoben und verstärken sich gegenseitig. Wenn sie zusammenkommen und konsequent darauf angewandt werden, wie wir unsere Arbeit verstehen, sie organisieren und durchführen, haben sie einen erheblichen Einfluss auf die Effizienz und die Qualität dieser Arbeit. Dies ist eine grundlegend andere Art, unsere Arbeit zu sehen, auch wenn jeder einzelne Gedanke für sich genommen vertraut sein mag. Wenn diese Dinge zusammenkommen und als Leitprinzipien für die Entscheidungsfindung in der Software-Entwicklung angewendet werden, stellt dies ein neues Paradigma für die Entwicklung dar.

Wir lernen gerade, was Software Engineering wirklich bedeutet, und es ist nicht immer das, was wir erwartet haben.

Beim Engineering geht es um die Einführung eines wissenschaftlichen, rationalistischen Ansatzes zur Lösung praktischer Probleme im Rahmen wirtschaftlicher Zwänge, aber das bedeutet nicht, dass ein solcher Ansatz theoretisch oder bürokratisch ist. Engineering ist schon fast per Definition pragmatisch.

Bei früheren Versuchen, *Software Engineering* zu definieren, wurde der Fehler gemacht, sich zu sehr auf bestimmte Tools oder Technologien zu beschränken. Software Engineering ist mehr als der Code, den wir schreiben, und die Tools, die wir benutzen. Software Engineering ist nicht irgendeine Form von Produktionstechnik; das ist nicht unsere Baustelle. Wenn Sie bei dem Wort *Engineering* an Bürokratie denken, dann lesen Sie dieses Buch und denken Sie noch einmal darüber nach.

Software Engineering ist nicht dasselbe wie Informatik, auch wenn die beiden oft verwechselt werden. Wir brauchen sowohl Software Engineers als auch Informatiker. In diesem Buch geht es um die Disziplin, den Prozess und die Konzepte, die

wir anwenden müssen, um zuverlässig und reproduzierbar bessere Software zu entwickeln.

Von einer Ingenieursdisziplin für die Software-Entwicklung, die diesen Namen auch verdient, erwarten wir, dass sie uns hilft, die Probleme, mit denen wir konfrontiert sind, mit höherer Qualität und mehr Effizienz zu lösen.

Ein solcher Engineering-Ansatz würde uns auch helfen, Probleme zu lösen, an die wir noch nicht gedacht haben, und zwar mithilfe von Technologien, die noch nicht erfunden wurden. Die Leitgedanken einer solchen Disziplin wären allgemein, beständig und allgegenwärtig.

Dieses Buch ist ein Versuch, eine Sammlung solcher eng verwandten, miteinander verbundenen Konzepte abzustecken. Mein Ziel ist es, sie zu einem kohärenten Ansatz zusammenzufassen, den wir als Grundlage für fast alle Entscheidungen heranziehen können, die wir als Software-Entwickler und Software-Entwicklungs-Teams treffen.

Software Engineering als Konzept muss uns, wenn es überhaupt eine Bedeutung haben soll, einen Vorteil verschaffen und nicht nur die Möglichkeit bieten, neue Tools einzusetzen.

Nicht alle Konzepte sind gleich. Es gibt gute und schlechte. Wie können wir also den Unterschied erkennen? Welche Prinzipien können wir anwenden, die es uns ermöglichen, jede neue Idee im Bereich Software und Software-Entwicklung zu bewerten und zu entscheiden, ob sie voraussichtlich gut oder schlecht ist?

Alles, was mit Fug und Recht als Engineering-Ansatz zur Lösung von Software-Problemen bezeichnet werden kann, ist allgemein anwendbar und von grundlegendem Charakter. In diesem Buch geht es um diese Konzepte. Nach welchen Kriterien sollten Sie Ihre Tools auswählen? Wie sollten Sie Ihre Arbeit organisieren? Wie sollten Sie die Systeme, die Sie bauen, und den Code, den Sie schreiben, organisieren, um Ihre Erfolgsaussichten während des Entstehungsprozesses zu erhöhen?

Eine Definition von Software Engineering?

In diesem Buch stelle ich die Behauptung auf, dass wir Software Engineering in folgendem Sinne denken sollten:

> ***Software Engineering*** *ist die Anwendung eines empirischen, wissenschaftlichen Ansatzes, um effiziente, wirtschaftliche Lösungen für praktische Probleme in der Software-Entwicklung zu finden.*

Mein Ziel ist ehrgeizig. Ich möchte ein Konzept, eine Struktur, einen Ansatz vorschlagen, den wir als eine echte Ingenieursdisziplin für die Software-Entwicklung betrachten können. Im Grunde genommen basiert dies auf drei Schlüsselideen:

- Die Wissenschaft und ihre praktische Anwendung, das »Engineering«, sind unverzichtbare Tools, um in technischen Disziplinen effektiv voranzukommen.
- In unserem Fachgebiet geht es im Wesentlichen um Lernen und Entdecken. Um erfolgreich zu sein, müssen wir daher **Experten im Lernen** werden, und mit Wissenschaft und Technik lernen wir am effektivsten.
- Schließlich sind die Systeme, die wir bauen, oft komplex und werden immer komplexer. Um ihre Entwicklung zu bewältigen, müssen wir also **Experten im Umgang mit dieser Komplexität** werden.

Der Inhalt des Buchs

In Teil I, »Was ist Software Engineering?«, geht es zunächst darum, was Engineering im Zusammenhang mit Software wirklich bedeutet. Hier geht es um die Prinzipien und die Philosophie des Engineerings und wie wir diese Konzepte auf die Software-Entwicklung anwenden können. Dies ist eine technische Philosophie für die Software-Entwicklung.

In Teil II, »Für das Lernen optimieren«, geht es darum, wie wir unsere Arbeit so organisieren, dass es uns möglich ist, in kleinen Schritten Fortschritte zu machen. Wie können wir beurteilen, ob wir gute Fortschritte machen oder heute nur das Altsystem von morgen schaffen?

Teil III, »Optimieren für den Umgang mit Komplexität«, befasst sich mit den Prinzipien und Methoden, die für den Umgang mit Komplexität notwendig sind. Hier wird jedes dieser Prinzipien näher beleuchtet und ihre Bedeutung und Anwendbarkeit für die Erstellung hochwertiger Software, egal welcher Art, erläutert.

Der letzte Abschnitt, Teil IV, »Werkzeuge zur Unterstützung von Engineering in der Software-Entwicklung«, beschreibt Konzepte und Herangehensweisen, die unsere Lernmöglichkeiten maximieren und es uns erleichtern, in kleinen Schritten Fortschritte zu machen und mit der Komplexität unserer Systeme umzugehen, während sie wachsen.

Über das ganze Buch verstreut finden sich Überlegungen zur Geschichte und Philosophie des Software Engineerings und wie sich das Denken weiterentwickelt hat. Diese Einschübe bieten einen hilfreichen Kontext für viele der Ideen in diesem Buch.

Über den Autor

David Farley ist ein Pionier der Continuous Delivery sowie Vordenker und Experte für Continuous Delivery, DevOps, TDD und Software-Entwicklung im Allgemeinen.

Dave ist seit vielen Jahren Programmierer, Software Engineer, Systemarchitekt und Leiter erfolgreicher Teams. Seit den Anfängen der modernen Computertechnik hat er die grundlegenden Prinzipien der Funktionsweise von Computern und Software aufgegriffen und bahnbrechende, innovative Ansätze entwickelt, die unsere Herangehensweise an die moderne Software-Entwicklung verändert haben. Er hat konventionelles Denken in Frage gestellt und Teams dazu gebracht, Software von Weltklasse zu entwickeln.

Dave ist Mitautor des mit dem Jolt-Award ausgezeichneten Buchs *Continuous Delivery*, ein gefragter Speaker und betreibt den äußerst erfolgreichen und beliebten YouTube-Kanal »Continuous Delivery« zum Thema Software Engineering. Er hat eine der schnellsten Finanzhandelsplattform der Welt aufgebaut und ist ein Pionier von BDD, Autor des Reactive Manifesto und Gewinner des Duke Award für Open-Source-Software mit dem LMAX Disruptor.

Daves Leidenschaft ist es, Entwicklungsteams auf der ganzen Welt dabei zu helfen, das Design, die Qualität und die Zuverlässigkeit ihrer Software zu verbessern, indem er sein Fachwissen durch seine Beratung, seinen YouTube-Kanal und seine Schulungen weitergibt.

Twitter: @davefarley77

YouTube Channel: `https://bit.ly/CDonYT`

Blog: `http://www.davefarley.net`

Company Website: `https://www.continuous-delivery.co.uk`

Kapitel 1

Einführung

1.1 Engineering – Die praktische Anwendung von Wissenschaft

Software-Entwicklung ist ein Prozess des Entdeckens und Erkundens; um darin erfolgreich zu sein, müssen Software-Entwickler Experten **im Lernen** werden.

Der beste Ansatz zum Lernen ist die Wissenschaft, also müssen wir die Techniken und Strategien der Wissenschaft übernehmen und sie auf unsere Probleme anwenden. Dies wird oft dahin gehend missverstanden, dass wir zu Physikern werden müssen, die alles mit einer für Software unangemessenen Präzision messen. Engineering ist pragmatischer als das.

Wenn ich sage, dass wir die Techniken und Strategien der Wissenschaft anwenden sollten, meine ich, dass wir einige ziemlich grundlegende, aber dennoch äußerst wichtige Konzepte anwenden sollten.

Die wissenschaftliche Methode, die die meisten von uns in der Schule gelernt haben, wird von Wikipedia wie folgt beschrieben:

- **Charakterisieren:** Beobachte den aktuellen Zustand.
- **Hypothesen aufstellen:** Erstelle eine Beschreibung, eine Theorie, die die Beobachtung erklären könnte.
- **Vorhersagen:** Triff eine Vorhersage auf Grundlage der Hypothese.
- **Experimentieren:** Teste die Vorhersage.

Wenn wir unser Denken auf diese Weise organisieren und beginnen, Fortschritte auf der Grundlage vieler kleiner, formloser Experimente zu machen, verringern wir das Risiko, vorschnell falsche Schlüsse zu ziehen, und leisten am Ende bessere Arbeit.

Wenn wir damit beginnen, die Variablen in unseren Experimenten zu kontrollieren, um mehr Konsistenz und Zuverlässigkeit in unseren Ergebnissen zu erreichen, führt uns das in Richtung deterministischerer Systeme und Code. Wenn wir beginnen, unseren Ideen gegenüber skeptisch zu sein und zu untersuchen, wie wir sie widerlegen könnten, können wir schlechte Ideen schneller erkennen und eliminieren sowie viel schneller Fortschritte machen.

Dieses Buch basiert auf einem praktischen, pragmatischen Ansatz zur Lösung von Softwareproblemen, basierend auf einer formlosen Adaption grundlegender wissenschaftlicher Prinzipien, mit anderen Worten: **Engineering**!

1.2 Was ist Software Engineering?

Die Definition von Software Engineering, die meinen Überlegungen in diesem Buch zugrunde liegt, lautet wie folgt:

> ***Software Engineering*** *ist die Anwendung eines empirischen, wissenschaftlichen Ansatzes, um effiziente, wirtschaftliche Lösungen für praktische Probleme in der Softwareentwicklung zu finden.*

Die Anwendung eines Engineering-Ansatzes bei der Software-Entwicklung ist vor allem aus zwei Gründen wichtig. Erstens ist Software-Entwicklung immer eine Praxis des Entdeckens und Lernens, und zweitens muss unsere Fähigkeit zu lernen nachhaltig sein, wenn es unser Ziel ist, »effizient« und »wirtschaftlich« zu sein.

Das bedeutet, dass wir die Komplexität der von uns geschaffenen Systeme so steuern müssen, dass unsere Fähigkeit, Neues zu lernen und uns anzupassen, erhalten bleibt.

Wir müssen also **Experten im Lernen und Experten im Umgang mit Komplexität** werden.

Es gibt fünf Techniken, die die Grundlage für diese Fokussierung auf das Lernen bilden. Um **Experten im Lernen** zu werden, brauchen wir insbesondere Folgendes:

- Iteration
- Feedback
- Inkrementalismus
- Experimente
- Empirismus

Dies ist ein evolutionärer Ansatz für die Entwicklung komplexer Systeme. Komplexe Systeme entspringen nicht bereits vollständig ausgearbeitet unserer Vorstellungskraft. Sie sind das Ergebnis vieler kleiner Schritte, in denen wir unsere Ideen ausprobieren und dabei auf Erfolge und Misserfolge reagieren. Dies sind die Werkzeuge, die uns Erkunden und Entdecken ermöglichen.

Diese Arbeitsweise bringt einige Einschränkungen in Bezug darauf mit sich, wie wir sicher voranschreiten können. Wir müssen in der Lage sein, so zu arbeiten, dass die Entdeckungsreise, die das Herzstück eines jeden Software-Projekts ist, erleichtert wird.

Folglich müssen wir neben der Konzentration auf das Lernen so arbeiten, dass wir Fortschritte machen können, wenn die Antworten und manchmal sogar die Richtung unsicher sind.

Dazu müssen wir **Experten im Umgang mit Komplexität** werden. Unabhängig von der Art der Probleme, die wir lösen, oder der Technologien, die wir zu ihrer Lösung einsetzen, ist der Umgang mit der Komplexität der Probleme, mit denen wir konfrontiert sind, und den Lösungen, die wir dafür anwenden, ein zentrales Unterscheidungsmerkmal zwischen schlechten und guten Systemen.

Um **Experten im Umgang mit Komplexität** zu werden, brauchen wir Folgendes:

- Modularität
- Kohäsion
- Trennung von Zuständigkeiten (engl. Separation of Concerns)
- Abstraktion
- Lose Kopplung

Es ist leicht, diese Konzepte zu betrachten und sie als bekannt abzutun. Ja, Sie sind fast sicher mit allen von ihnen vertraut. Ziel dieses Buchs ist es, sie zu ordnen und in eine zusammenhängende Strategie für die Entwicklung von Softwaresystemen zu überführen, die Ihnen hilft, Ihr Potenzial bestmöglich auszuschöpfen.

Dieses Buch beschreibt, wie diese zehn Konzepte als Werkzeuge zur Steuerung der Software-Entwicklung eingesetzt werden können. Es beschreibt anschließend eine Reihe von Konzepten, die als praktische Werkzeuge dienen, um eine effektive Strategie für jede Software-Entwicklung voranzutreiben. Zu diesen Konzepten gehören die folgenden:

- Testbarkeit
- Deploybarkeit
- Geschwindigkeit
- Kontrolle der Variablen[1]
- Continuous Delivery (dt. Kontinuierliche Bereitstellung)

Wenn wir diese Denkweise anwenden, sind die Ergebnisse tiefgreifend. Wir erstellen Software von höherer Qualität, wir produzieren schneller, und die Mitarbeiter der Teams, die diese Prinzipien anwenden, berichten, dass sie mehr Spaß an ihrer Arbeit haben, weniger Stress empfinden und eine bessere Work-Life-Balance haben.[2]

Das sind gewagte Behauptungen, aber auch sie werden durch Daten gestützt.

1 Anmerkung des Übersetzers: Im englischen Original heißt es »Controlling the variables«. Damit sind hier die Umgebungsparameter gemeint, nicht die Variablen in einem Programm.

2 Basierend auf den Erkenntnissen aus den »State-of-DevOps«-Berichten sowie den Berichten von Microsoft und Google.

1.3 Die Rückeroberung des »Software Engineering«

Ich habe mit dem Titel dieses Buchs gerungen, nicht weil ich nicht wusste, wie ich es nennen wollte, sondern weil unsere Branche die Bedeutung von *Engineering* im Zusammenhang mit Software so neu definiert hat, dass der Begriff abgewertet wurde.

In der Software-Branche wird er oft einfach als Synonym für »Code« angesehen oder als etwas, das die Leute abschreckt, da es als übermäßig bürokratisch und verfahrenstechnisch gilt. Für echtes Engineering könnte nichts weiter von der Wahrheit entfernt sein.

In anderen Disziplinen bedeutet *Engineering* einfach »Dinge, die funktionieren«. Es ist der Prozess bzw. das Verfahren, das angewendet wird, um die Chance, gute Arbeit zu leisten, zu erhöhen.

Wenn unsere Verfahren im »Software Engineering« es uns nicht ermöglichen, bessere Software schneller zu entwickeln, dann sind sie nicht wirklich im Sinne des Engineerings, und wir sollten sie ändern!

Das ist der Grundgedanke dieses Buchs, und sein Ziel ist es, ein nachvollziehbar konsistentes Modell zu beschreiben, das einige Grundprinzipien zusammenfasst, die die Grundlage jeder guten Software-Entwicklung bilden.

Es gibt nie eine Erfolgsgarantie, aber wenn Sie sich diese gedanklichen Hilfsmittel und organisatorischen Grundsätze zu eigen machen und sie auf Ihre Arbeit anwenden, werden Sie Ihre Erfolgschancen sicherlich erhöhen.

1.4 Wie man Fortschritte macht

Software-Entwicklung ist eine komplexe, anspruchsvolle Tätigkeit. Sie ist in gewisser Weise eine der komplexesten Tätigkeiten, die unsere Spezies ausübt. Es ist lächerlich zu erwarten, dass jeder Einzelne oder sogar jedes Team die Bewältigung dieser Aufgabe jedes Mal von Grund auf neu erfinden kann oder sollte, wenn wir ein neues Projekt beginnen.

Wir haben gelernt und lernen weiterhin, was funktioniert und was nicht. Wie können wir also, sowohl als gesamte Branche als auch als Teams, Fortschritte machen und auf den Schultern von Giganten aufbauen, wie Isaac Newton einmal sagte, wenn jeder bei allem ein Veto einlegen kann? Wir brauchen einige vereinbarte Grundsätze und eine Disziplin, die unsere Aktivitäten leitet.

Die Gefahr bei dieser Denkweise besteht darin, dass sie bei falscher Anwendung zu drakonischem, übermäßig direktivem »Entscheidung-von-oben«-Denken führen kann.

Wir werden auf frühere schlechte Denkmuster zurückfallen, nämlich dass die Aufgabe von Managern und Führungskräften darin besteht, allen anderen zu sagen, was sie zu tun haben und wie sie es tun sollen.

Das große Problem bei »präskriptivem« oder übermäßig »direktivem« Vorgehen ist, was wir tun sollen, wenn einige unserer Ansichten falsch oder unvollständig sind. Das wird unvermeidlich sein. Wie können wir also schlechte, eingefahrene Konzepte hinterfragen und neue, potenziell großartige, jedoch unerprobte Konzepte bewerten?

Es gibt eine sehr gute Herangehensweise dafür, diese Probleme zu lösen. Es ist ein Ansatz, der uns die geistige Freiheit gibt, Dogmen zu hinterfragen und zu widerlegen und zwischen modischen, jedoch schlechten, und großartigen Konzepten zu unterscheiden, unabhängig davon, aus welcher Quelle sie stammen. Er ermöglicht es uns, die schlechten Ideen durch bessere zu ersetzen und gute Konzepte zu verbessern. Im Grunde brauchen wir eine Struktur, die es uns ermöglicht zu wachsen und verbesserte Ansätze, Strategien, Prozesse, Technologien und Lösungen zu entwickeln. Wir nennen diese gute Herangehensweise *Wissenschaft*!

Wenn wir diese Art des Denkens auf die Lösung praktischer Probleme anwenden, nennen wir es *Engineering*!

In diesem Buch geht es darum, was es bedeutet, wissenschaftliches Denken auf unser Fachgebiet anzuwenden und so etwas zu erreichen, das wir wirklich und präziser Weise als *Software Engineering* bezeichnen können.

1.5 Die Geburt des Software Engineering

Das Konzept des Software Engineering wurde Ende der 1960er-Jahre entwickelt. Der Begriff wurde erstmals von Margaret Hamilton verwendet, die später Direktorin der Software Engineering Division des MIT Instrumentation Lab wurde. Hamilton war federführend bei der Entwicklung der Flugkontroll-Software für das Apollo-Raumfahrtprogramm.

Zur gleichen Zeit berief die North Atlantic Treaty Organization (NATO) eine Konferenz in Garmisch-Partenkirchen (Deutschland) ein, um den Begriff zu definieren. Dies war die erste **Software-Engineering**-Konferenz.

Die ersten Computer wurden durch das Umlegen von Schaltern oder sogar als Teil ihrer Konstruktion fest programmiert. Den Pionieren wurde schnell klar, dass dies langsam und unflexibel war, und die Idee des »gespeicherten Programms« war geboren. Dieser Ansatz unterschied zum ersten Mal klar zwischen Software und Hardware.

Ende der 1960er-Jahre waren die Computerprogramme so komplex geworden, dass es schwierig wurde, sie zu erstellen und zu warten. Sie waren an der Lösung

immer komplexerer Probleme beteiligt und wurden schnell zu dem Auslöser, der es überhaupt erst ermöglichte, bestimmte Arten von Problemen zu lösen.

Es entstand eine erhebliche Diskrepanz zwischen der Geschwindigkeit des Fortschritts bei der Hardware-Entwicklung gegenüber dem Fortschritt bei der Software-Entwicklung. Dies wurde seinerzeit als *Software-Krise* bezeichnet.

Die NATO-Konferenz wurde unter anderem als Reaktion auf diese Krise einberufen.

Wenn man heute die Aufzeichnungen der Konferenz liest, findet man viele Konzepte, die offensichtlich beständig sind. Sie haben sich im Laufe der Zeit bewährt und sind heute noch so aktuell wie 1968. Diese Tatsache sollte uns interessieren, wenn wir danach streben, einige grundlegende Merkmale zu bestimmen, die unsere Disziplin ausmachen.

Ein paar Jahre später verglich der Turing-Preisträger Fred Brooks rückblickend die Fortschritte der Software-Entwicklung mit denen der Hardware-Entwicklung:

> *Es gibt keine einzige Entwicklung, weder bei Technologien noch bei den Führungstechniken, die für sich genommen innerhalb eines Jahrzehnts auch nur annähernd eine Verbesserung in ähnlicher Größenordnung bezüglich Produktivität, Zuverlässigkeit und Einfachheit aufwies.*[3]

Brooks verglich dies mit dem berühmten Moore'schen Gesetz[4], dem die Hardware-Entwicklung für viele Jahre gefolgt ist.

Dies ist eine interessante Beobachtung, die, wie ich glaube, viele überraschen wird, aber im Grunde genommen hat sie schon immer zugetroffen.

Brooks führt weiter aus, dass dies nicht so sehr eine Frage der Software-Entwicklung ist, sondern viel mehr eine Folge der einzigartigen, erstaunlichen Verbesserung der Leistungsfähigkeit von Hardware:

> *Wir müssen feststellen, dass die Anomalie nicht darin besteht, dass die Software-Entwicklung so langsam ist, sondern dass die Entwicklung der Computer-Hardware so schnell ist. Keine andere Technologie seit Beginn der Zivilisation hat innerhalb von 30 Jahren einen Preis-/Leistungszuwachs von sechs Größenordnungen erzielt.*

3 Quelle: Fred Brooks' 1986 veröffentlichtes Paper mit dem Titel »No Silver Bullet«. Siehe `http://worrydream.com/refs/Brooks-NoSilverBullet.pdf`.

4 1965 sagte Gordon Moore voraus, dass sich die Transistordichte (nicht die Leistung) im nächsten Jahrzehnt (bis 1975) jedes Jahr verdoppeln würde, was später zu alle zwei Jahre korrigiert wurde. Diese Vorhersage setzten sich die Produzenten von Halbleitern zum Ziel und übertrafen Moores Erwartungen bei Weitem, die noch jahrzehntelang Gültigkeit behielten. Einige Beobachter sind der Ansicht, dass wir das Ende dieses explosiven Kapazitätswachstums aufgrund der Grenzen bestehender Ansätze und dem Auftreten von Quanteneffekten inzwischen erreicht haben. Aber zum Zeitpunkt der Entstehung dieses Buchs folgt die Entwicklung von Halbleitern mit hoher Dichte weiterhin dem Moore'schen Gesetz.

Er schrieb dies 1986, was wir heute als den Beginn des Computerzeitalters bezeichnen würden. Die Hardware-Entwicklung hat sich seither in diesem Tempo fortgesetzt, und die Computer, die Brooks so leistungsfähig erschienen, sehen im Vergleich zur Kapazität und Leistung moderner Systeme wie Spielzeug aus. Und doch ... seine Beobachtungen bezüglich der Geschwindigkeit der Verbesserungen in der Software-Entwicklung bleiben korrekt.

1.6 Paradigmenwechsel

Das Konzept des *Paradigmenwechsels* wurde von dem Physiker Thomas Kuhn entwickelt.

Meistens ist Lernen eine Art Anhäufung. Wir bauen Schichten des Verstehens auf, wobei jede Schicht von der vorherigen untermauert wird.

Aber Lernen ist nicht immer so. Manchmal ändern wir unsere Perspektive auf etwas grundlegend und das ermöglicht es uns, Neues zu lernen, aber das bedeutet auch, dass wir Gelerntes wieder verwerfen müssen.

Im 18. Jahrhundert glaubten angesehene Biologen (damals wurden sie noch nicht so bezeichnet), dass sich einige Tiere spontan selbst erzeugen. Als Darwin Mitte des 19. Jahrhunderts auftauchte und den Prozess der natürlichen Selektion beschrieb, wurde damit das Konzept der spontanen Entstehung völlig über den Haufen geworfen.

Dieses Umdenken führte schließlich zu unserem modernen Verständnis der Genetik und zu unserer Fähigkeit, das Leben auf einer grundlegenderen Ebene zu verstehen, Technologien zu entwickeln, die es uns ermöglichen, diese Gene zu manipulieren und COVID-19-Impfstoffe und Gentherapien zu entwickeln.

In ähnlicher Weise stellten Kepler, Kopernikus und Galilei die damals vorherrschende Überzeugung infrage, dass die Erde im Zentrum des Universums lag. Sie schlugen stattdessen ein heliozentrisches Modell für das Sonnensystem vor. Dieses führte schließlich dazu, dass Newton die Gesetze der Gravitation und Einstein die allgemeine Relativitätstheorie entwickeln konnten. Es ermöglichte uns, in den Weltraum zu reisen und Technologien wie GPS zu entwickeln.

Das Konzept des Paradigmenwechsels beinhaltet implizit die Vorstellung, dass wir bei einem solchen Wechsel als Teil des Prozesses einige andere Konzepte verwerfen, von denen wir jetzt wissen, dass sie nicht länger richtig sind.

Software-Entwicklung als echte Ingenieursdisziplin zu behandeln, die ihre Wurzeln in der Philosophie der wissenschaftlichen Methode und des wissenschaftlichen Realismus' hat, hat tiefgreifende Folgen. Sie sind nicht nur wegen ihrer

Wirkung und Effektivität, die so wortgewandt in dem Buch *Accelerate*[5] beschrieben werden, tiefgreifend, sondern auch wegen der essenziellen Notwendigkeit, die Konzepte zu verwerfen, die durch einen neuen Ansatz ersetzt werden.

Dies gibt uns einen Ansatz, um effektiver zu lernen und schlechte Konzepte effizienter zu verwerfen.

Ich glaube, dass die von mir in diesem Buch beschriebene Herangehensweise an die Software-Entwicklung einen solchen Paradigmenwechsel darstellt. Sie bietet uns eine neue Perspektive auf das, was wir tun und wie wir es tun.

1.7 Zusammenfassung

Die Anwendung dieser ingenieurstechnischen Sichtweise auf Software muss nicht schwerfällig oder übermäßig komplex sein. Der Paradigmenwechsel, der darin besteht, anders darüber zu denken, was wir bei der Software-Erstellung tun und wie wir es tun, sollte uns helfen, den Wald vor lauter Bäumen zu sehen und Software einfacher, zuverlässiger und effizienter machen.

Es geht nicht um mehr Bürokratie, sondern darum, unsere Fähigkeit zu verbessern, qualitativ hochwertige Software nachhaltiger und zuverlässiger zu erstellen.

5 Die Leute, die hinter den »State-of-DevOps«-Berichten stehen, DORA, beschrieben das Prognosemodell, das sie auf der Grundlage ihrer Untersuchungen erstellt haben. Quelle: *Accelerate: The Science of Lean Software and DevOps* von Nicole Fosgren, Jez Humble, und Gene Kim (2018).

Stichwortverzeichnis