

DAVID FARLEY

MODERNES SOFTWARE ENGINEERING

Bessere Software schneller
und effektiver entwickeln



Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Liebe Leserinnen und Leser,

dieses E-Book, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Mit dem Kauf räumen wir Ihnen das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. Jede Verwertung außerhalb dieser Grenzen ist ohne unsere Zustimmung unzulässig und strafbar. Das gilt besonders für Vervielfältigungen, Übersetzungen sowie Einspeicherung und Verarbeitung in elektronischen Systemen.

Je nachdem wo Sie Ihr E-Book gekauft haben, kann dieser Shop das E-Book vor Missbrauch durch ein digitales Rechtemanagement schützen. Häufig erfolgt dies in Form eines nicht sichtbaren digitalen Wasserzeichens, das dann individuell pro Nutzer signiert ist. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Beim Kauf des E-Books in unserem Verlagsshop ist Ihr E-Book DRM-frei.

Viele Grüße und viel Spaß beim Lesen,

Ihr mitp-Verlagsteam





Stimmen zum Buch

»*Modern Software Engineering* macht es richtig und beschreibt die Art und Weise, wie erfahrene Praktiker heute tatsächlich Software entwickeln. Die von Farley vorgestellten Techniken sind nicht starr, präskriptiv oder linear, sondern genauso diszipliniert, wie es die Software-Entwicklung erfordert: empirisch, iterativ, feedback-getrieben, wirtschaftlich und auf ausführbaren Code ausgerichtet.«

– *Glenn Vanderburg, Leiter der Entwicklungsabteilung bei Nubank*

»Es gibt viele Bücher, die Ihnen sagen, wie Sie ein bestimmtes Software Engineering-Verfahren anwenden sollen; dieses Buch ist anders. Was Dave hier tut, ist, das Wesentliche dessen darzulegen, was Software Engineering ausmacht und wie es sich vom einfachen Handwerk unterscheidet. Er erklärt, warum und wie man ein Meister des Lernens und des Umgangs mit Komplexität werden muss, um Software Engineering zu beherrschen, wie bereits existierende Verfahren dies unterstützen und wie man andere Konzepte nach ihren Vorzügen im Sinne des Software Engineerings beurteilt. Dies ist ein Buch für jeden, der die Software-Entwicklung ernsthaft als echte Ingenieursdisziplin behandeln möchte, egal ob man gerade erst anfängt oder schon seit Jahrzehnten Software entwickelt.«

– *Dave Hounslow, Software Engineer*

»Dies sind wichtige Themen und es ist großartig, ein Kompendium zu haben, das sie zu einem Paket zusammenschnürt.«

– *Michael Nygard, Autor von Release IT, professioneller Programmierer und Software-Architekt*

»Ich habe das Rezensionsexemplar von Dave Farleys Buch gelesen, und es ist genau das, was wir brauchen. Es sollte Pflichtlektüre für jeden sein, der Software-Entwickler werden will oder das Handwerk beherrschen möchte. Pragmatische, praktische Ratschläge für professionelles Engineering. Es sollte Pflichtlektüre an Universitäten und in Bootcamps sein.«

– *Bryan Finster, angesehener Ingenieur und Value-Stream-Architekt bei Platform One der USAF (United States Air Force)*

David Farley

Modernes Software Engineering

**Bessere Software schneller
und effektiver entwickeln**

Übersetzung aus dem Englischen von
Uwe Schirmer



Bibliografische Information der Deutschen Nationalbibliothek

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

ISBN 978-3-7475-0635-6

1. Auflage 2023

www.mitp.de

E-Mail: mitp-verlag@sigloch.de

Telefon: +49 7953 / 7189 - 079

Telefax: +49 7953 / 7189 - 082

© 2023 mitp Verlags GmbH & Co. KG, Frechen

Authorized translation from the English language edition, entitled MODERN SOFTWARE ENGINEERING 1st Edition by DAVID FARLEY, published by Pearson Education, Inc, publishing as Addison Wesley, Copyright © 2023 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

GERMAN language edition published by MITP VERLAGS GMBH & CO. KG, Copyright © 2023.

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Lektorat: Janina Bahlmann, Nicole Winkel

Sprachkorrektorat: Jürgen Benvenuti

Covergestaltung: Christian Kalkert

Coverbild: © [spainter_vfx](https://www.spainter_vfx.com) / [stock.adobe.com](https://www.stock.adobe.com)

Satz: III-satz, Kiel, www.drei-satz.de

Inhaltsverzeichnis

Vorwort	13
Einleitung	17
Danksagungen	21
Über den Autor	23
Teil I Was ist Software Engineering?	25
1 Einführung	27
1.1 Engineering – Die praktische Anwendung von Wissenschaft	27
1.2 Was ist Software Engineering?	28
1.3 Die Rückeroberung des »Software Engineering«	30
1.4 Wie man Fortschritte macht	30
1.5 Die Geburt des Software Engineering	31
1.6 Paradigmenwechsel	33
1.7 Zusammenfassung	34
2 Was ist Engineering?	35
2.1 Die Produktion ist nicht unser Problem	35
2.2 Konstruktionsingenieurwesen, nicht Produktionstechnik	36
2.3 Eine Arbeitsdefinition von Engineering	42
2.4 Engineering != Code	43
2.5 Warum ist Engineering so wichtig?	45
2.6 Die Grenzen von »Handwerk«	46
2.7 Präzision und Skalierbarkeit	47
2.8 Komplexität handhaben	48
2.9 Reproduzierbarkeit und Messgenauigkeit	49
2.10 Engineering, Kreativität und Handwerk	51
2.11 Warum das, was wir tun, kein Software Engineering ist	53
2.12 Kompromisse	54
2.13 Die Illusion des Fortschritts	55
2.14 Der Weg vom Handwerk zum Engineering	56
2.15 Handwerk ist nicht genug	57

2.16	Zeit für ein Umdenken?	58
2.17	Zusammenfassung	59
3	Grundlagen eines Engineering-Ansatzes.	61
3.1	Eine Branche im Wandel?	61
3.2	Die Bedeutung von Messungen	62
3.3	Anwendung von Stabilität und Durchsatz	65
3.4	Die Grundlagen einer Ingenieursdisziplin für die Software- Entwicklung	67
3.5	Experten im Lernen	67
3.6	Experten im Umgang mit Komplexität	68
3.7	Zusammenfassung	70
Teil II	Für das Lernen optimieren	71
4	Iterativ arbeiten	73
4.1	Praktische Vorteile iterativen Arbeitens	75
4.2	Iteration als defensive Design-Strategie	77
4.3	Der Reiz des Plans	79
4.4	Praktische Aspekte des iterativen Arbeitens	86
4.5	Zusammenfassung	88
5	Feedback.	89
5.1	Ein praktisches Beispiel für die Wichtigkeit von Feedback	89
5.2	Feedback bei der Entwicklung	93
5.3	Feedback bei der Integration	94
5.4	Feedback beim Design	96
5.5	Feedback zur Architektur	99
5.6	Frühzeitiges Feedback bevorzugen	101
5.7	Feedback beim Produktdesign	102
5.8	Feedback in Unternehmen und Kultur	103
5.9	Zusammenfassung	106
6	Inkrementalismus	107
6.1	Die Bedeutung von Modularität	108
6.2	Inkrementalismus im Unternehmen	110
6.3	Werkzeuge für den Inkrementalismus	112
6.4	Die Auswirkungen von Änderungen begrenzen	113
6.5	Inkrementelles Design	115
6.6	Zusammenfassung	118

7	Empirismus	119
7.1	In der Realität verankert	120
7.2	Trennung von Epirismus und Experiment	120
7.3	»Ich kenne diesen Bug!«	121
7.4	Selbsttäuschung vermeiden	123
7.5	Eine Realität erfinden, die zu unserer Argumentation passt	124
7.6	Von der Realität geleitet	128
7.7	Zusammenfassung	129
8	Experimentell vorgehen	131
8.1	Was bedeutet »experimentell vorgehen«?	132
8.2	Feedback	133
8.3	Hypothese	135
8.4	Messung	136
8.5	Kontrolle der Variablen	137
8.6	Automatisierte Tests als Experimente	138
8.7	Einordnung der Versuchsergebnisse der Tests in den Kontext	140
8.8	Der Umfang eines Experiments	142
8.9	Zusammenfassung	143

Teil III	Optimieren für den Umgang mit Komplexität	145
-----------------	--	------------

9	Modularität	147
9.1	Merkmale von Modularität	148
9.2	Die Bedeutung von gutem Design wird unterschätzt	149
9.3	Die Bedeutung von Testbarkeit	151
9.4	Für Testbarkeit zu designen verbessert die Modularität	152
9.5	Services und Modularität	159
9.6	Deploybarkeit und Modularität	161
9.7	Modularität auf verschiedenen Ebenen	163
9.8	Modularität menschlicher Systeme	164
9.9	Zusammenfassung	166
10	Kohäsion	167
10.1	Modularität und Kohäsion: Grundlagen des Designs	167
10.2	Der grundlegende Abbau von Kohäsion	168
10.3	Kontext ist wichtig	171
10.4	High-Performance-Software	175
10.5	Verbindung zur Kopplung	176
10.6	Hohe Kohäsion durch TDD	177

10.7	Wie erreicht man gute Kohäsion bei Software?	177
10.8	Kosten von schlechter Kohäsion	180
10.9	Kohäsion in menschlichen Systemen	181
10.10	Zusammenfassung	182
11	Trennung von Zuständigkeiten	183
11.1	Dependency Injection	187
11.2	Trennung von wesentlicher und zufälliger Komplexität.	188
11.3	Bedeutung von DDD	192
11.4	Testbarkeit	194
11.5	Ports & Adapters	195
11.6	Wann sollte »Ports & Adapters« eingesetzt werden?	198
11.7	Was ist eine API?	199
11.8	Verwendung von TDD zur Förderung der Trennung von Zuständigkeiten	201
11.9	Zusammenfassung	202
12	Information Hiding und Abstraktion.	203
12.1	Abstraktion oder Information Hiding	203
12.2	Was verursacht den »Big Ball of Mud«?	204
12.3	Unternehmerische und unternehmenskulturelle Probleme	204
12.4	Technische Probleme und Probleme des Designprozesses	207
12.5	Furcht vor Over-Engineering	211
12.6	Verbesserung der Abstraktion durch Testen	214
12.7	Die Macht der Abstraktion	215
12.8	Undichte Abstraktionen	217
12.9	Geeignete Abstraktionen auswählen	218
12.10	Abstraktionen in der Anwendungsdomäne	221
12.11	Abstrakte zufällige Komplexität	222
12.12	Systeme und Code von Drittanbietern isolieren	225
12.13	Immer das Verbergen von Informationen bevorzugen	226
12.14	Zusammenfassung	228
13	Kopplung handhaben	229
13.1	Kosten von Kopplung	229
13.2	Hochskalieren	230
13.3	Microservices	231
13.4	Entkopplung kann mehr Code bedeuten	233
13.5	Lose Kopplung ist nicht das Einzige, was wichtig ist	235

13.6	Lose Kopplung bevorzugen.	236
13.7	Wie unterscheidet sich dies von der Trennung von Zuständigkeiten?	238
13.8	DRY ist zu simpel	239
13.9	Asynchronität als Werkzeug für lose Kopplung	241
13.10	Für lose Kopplung designen.	243
13.11	Lose Kopplung in menschlichen Systemen.	243
13.12	Zusammenfassung	245

Teil IV Werkzeuge zur Unterstützung von Engineering in der Software-Entwicklung 247

14	Die Werkzeuge einer Ingenieursdisziplin.	249
14.1	Was ist Software-Entwicklung?	249
14.2	Testbarkeit als Werkzeug	252
14.3	Messpunkte	255
14.4	Schwierigkeiten, die Testbarkeit zu erreichen.	256
14.5	Wie man die Testbarkeit verbessert.	260
14.6	Deploybarkeit.	261
14.7	Geschwindigkeit	263
14.8	Die Variablen kontrollieren	264
14.9	Continuous Delivery	266
14.10	Allgemeine Werkzeuge zur Unterstützung von Engineering	267
14.11	Zusammenfassung	268
15	Der moderne Software Engineer	269
15.1	Engineering als menschlicher Prozess	271
15.2	Digital disruptive Unternehmen	272
15.3	Ergebnisse vs. Mechanismen	274
15.4	Langlebigkeit und allgemeine Anwendbarkeit	277
15.5	Grundlagen einer Ingenieursdisziplin.	280
15.6	Zusammenfassung	281
	Stichwortverzeichnis	283



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 Karrierbibliothek 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 Erfolgchancen, 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 »Ingenieurwissenschaften« 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.



Danksagungen

Ein Buch wie dieses zu schreiben, erfordert viel Zeit, eine Menge Arbeit und die Erkundung zahlreicher Ideen. Die Menschen, die mir während dieses Prozesses geholfen haben, taten dies auf ganz unterschiedliche Weise. Manchmal stimmten sie mir zu und bestärkten mich in meinen Überzeugungen, manchmal waren sie anderer Meinung und zwangen mich, meine Argumente zu verstärken oder meine Meinung zu ändern.

Zunächst möchte ich mich bei meiner Frau Kate bedanken, die mir auf vielfältige Weise geholfen hat. Obwohl Kate kein Software-Profi ist, hat sie große Teile dieses Buchs gelesen und mir dabei geholfen, meine Grammatik zu korrigieren und meine Botschaft zu verfeinern.

Ich möchte mich bei meinem Schwager Bernard McCarty bedanken, der mich mit Ideen zum Thema Wissenschaft versorgt und mich dazu gebracht hat, intensiver darüber nachzudenken, warum ich über Experimente und Empirie und viele andere Dinge sprechen wollte.

Ich möchte Trisha Gee dafür danken, dass sie nicht nur ein so nettes Vorwort geschrieben hat, sondern auch von diesem Buch begeistert war, als ich neuen Antrieb brauchte.

Ich möchte Martin Thompson dafür danken, dass er immer für mich da ist, wenn es darum geht, Meinungen über Informatik auszutauschen, und dass er meist innerhalb von Minuten auf meine ziemlich zufälligen Gedanken antwortet.

Ich möchte mich bei Martin Fowler bedanken, der mir, obwohl er mit anderen Projekten ausgelastet ist, Ratschläge gegeben hat, die zur Stärkung dieses Buchs beigetragen haben.

Viele weitere meiner Freunde haben im Laufe der Jahre indirekt dazu beigetragen, mein Denken zu diesen Themen zu formen, und vieles mehr: Dave Hounslow, Steve Smith, Chris Smith, Mark Price, Andy Stewart, Mark Crowther, Mike Barker und viele andere.

Ich möchte dem Team von Pearson¹ für seine Hilfe und Unterstützung während des Veröffentlichungsprozesses dieses Buchs danken.

¹ Anm. zur Übersetzung: Die englischsprachige Originalausgabe dieses Buchs ist bei Pearson erschienen.

Ich möchte auch einer ganzen Reihe von Menschen danken – die ich nicht alle kenne –, die mich unterstützt, mit mir argumentiert, mich herausgefordert und nachdenklich gemacht haben. Ich habe viele dieser Ideen schon seit einigen Jahren auf Twitter und auf meinem YouTube-Kanal geteilt und bin dadurch in einige großartige Gespräche verwickelt worden. Ich danke euch!



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>

Teil I

Was ist Software Engineering?

In diesem Teil:

- **Kapitel 1**
Einführung 27
- **Kapitel 2**
Was ist Engineering? 35
- **Kapitel 3**
Grundlagen eines Engineering-Ansatzes 61

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.