mitp

Robert C. Martin

Mit Beiträgen von James Grenning und Simon Brown

Vorwort von Kevlin Henney

Nachwort von Jason Gorman

Deutsche Ausgabe

Clean Architecture

Das Praxis-Handbuch für professionelles Softwaredesign

Regeln und Paradigmen für effiziente Softwarestrukturierung

Hinweis des Verlages zum Urheberrecht und Digitalen Rechtemanagement (DRM)

Der Verlag räumt Ihnen mit dem Kauf des ebooks das Recht ein, die Inhalte im Rahmen des geltenden Urheberrechts zu nutzen. 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 Einspeicherung und Verarbeitung in elektronischen Systemen.

Der Verlag schützt seine ebooks vor Missbrauch des Urheberrechts durch ein digitales Rechtemanagement. Bei Kauf im Webshop des Verlages werden die ebooks mit einem nicht sichtbaren digitalen Wasserzeichen individuell pro Nutzer signiert.

Bei Kauf in anderen ebook-Webshops erfolgt die Signatur durch die Shopbetreiber. Angaben zu diesem DRM finden Sie auf den Seiten der jeweiligen Anbieter.

Dieses Buch ist meiner wunderbaren Frau,
meinen vier großartigen Kindern und deren Familien
sowie der Schar meiner fünf Enkelkinder gewidmet
– sie sind die Freude meines Lebens.

Robert C. Martin

Clean Architecture

Das Praxis-Handbuch für professionelles Softwaredesign

Regeln und Paradigmen für effiziente Softwarestrukturen

Übersetzung aus dem Amerikanischen
von Maren Feilen und Knut Lorenzen

mitp

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 978-3-95845-725-6
1. Auflage 2018

www.mitp.de
E-Mail: mitp-verlag@sigloch.de
Telefon: +49 7953 / 7189 - 079
Telefax: +49 7953 / 7189 - 082

© 2018 mitp Verlags GmbH & Co. KG, Frechen

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.

Authorized translation from the English language edition, CLEAN ARCHITECTURE: A CRAFTSMAN'S GUIDE TO SOFTWARE STRUCTURE AND DESIGN, 1st Edition by ROBERT MARTIN, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2018 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 © 2018 mitp Verlags GmbH & Co. KG, Frechen.

Lektorat: Sabine Schulz
Sprachkorrektorat: Petra Heubach-Erdmann
Coverbild: © Vadim Sadovski/ShutterStock
Satz: III-satz, Kiel, www.drei-satz.de

Inhaltsverzeichnis

Vorwort

Worüber reden wir, wenn wir uns über »Architektur« im Allgemeinen unterhalten?

Wie bei allen metaphorischen Annäherungen kann auch die Betrachtung einer Software unter architektonischen Gesichtspunkten gleichermaßen viel verhüllen wie offenbaren. Ebenso kann sie mehr versprechen, als sie zu liefern imstande ist, oder mehr liefern, als sie ursprünglich zu versprechen schien.

Der offenkundige Reiz der Architektur besteht in ihrer Struktur, deshalb steht Letztere auch im Bereich der Softwareentwicklung im Fokus sämtlicher Paradigmen und Überlegungen – Komponenten, Klassen, Funktionen, Module, Layer und Services, egal, ob Mikro oder Makro. Allerdings erweist sich die Gesamtstruktur vieler Softwaresysteme nicht selten als fragwürdig oder widersinnig – etwa wie die sowjetischen Kollektivbetriebe, die als Vermächtnis für das Volk bestimmt waren, undenkbare Jenga-Türme, die bis zum Himmel hinaufragen, oder wundersame, unter riesigen Schlammlawinen begrabene archäologische Fundschichten. Dass Softwarestrukturen in der gleichen Art und Weise unserer Intuition folgen, wie dies auch bei Gebäudestrukturen der Fall ist, lässt sich häufig nur schwer erkennen.

Gebäude besitzen dagegen eine unübersehbare physische Struktur – ob in Stein oder Beton verwurzelt, ob hoch nach oben ragend oder weit in die Breite verlaufend, ob groß oder klein, ob atemberaubend oder schlicht und banal. Bei Bauwerken gibt es kaum eine Alternative, als sie nach der Physik der Schwerkraft und den verwendeten Materialien auszurichten. Im Gegensatz dazu hat Software – außer im Sinne der Realitätstreue – wenig für die Schwerkraft übrig. Und woraus besteht Software? Anders als Gebäude, die aus Ziegeln, Beton, Holz, Stahl und Glas gefertigt werden, ist Software eben nur aus Software gemacht. Große Softwarekonstrukte setzen sich aus kleineren Softwarekomponenten zusammen, die wiederum aus noch kleineren Softwarekomponenten bestehen und so weiter, und so fort. Oder, wie es Stephen W. Hawking in seinem Buch »Eine kurze Geschichte der Zeit« ausdrückt:

> *Da stehen lauter Schildkröten aufeinander.*[1]

Wenn wir über Softwarearchitektur im Speziellen reden, geht es darum, dass Software in ihrer Beschaffenheit rekursiv und fraktal, im Code prägnant und richtung-

1 Stephen W. Hawking, *Eine kurze Geschichte der Zeit*, Rowohlt Verlag, 2004.

weisend ist. Einfach alles hat mit Details zu tun. Zwar spielen ineinandergreifende Detailebenen auch bei der Architektur von Gebäuden eine Rolle, in Bezug auf Software ist es allerdings kaum sinnvoll, über physische Maßstäbe nachzudenken. Software besitzt eine Struktur – eigentlich jede Menge und zahlreiche Arten von Strukturen –, deren Vielfalt das Spektrum der physischen Gebäudestrukturen problemlos in den Schatten stellt. Man kann durchaus behaupten, dass dem Design in der Softwareentwicklung mehr Einsatz und Aufmerksamkeit zuteilwird, als dies in der Gebäudearchitektur der Fall ist – und insofern ist es auch nicht unbedingt abwegig, die Softwarearchitektur als architektonischer zu betrachten als die Gebäudearchitektur!

Aber physische Maßstäbe sind für den menschlichen Verstand besser zu begreifen. Sie bieten uns Orientierung in der Welt, deshalb halten wir stets Ausschau danach. Und sicherlich sind die einzelnen Kästen in schematischen PowerPoint-Darstellungen ansprechend und vermitteln einen klaren visuellen Eindruck, trotzdem geben sie nicht den vollen und lückenlosen Umfang der Architektur eines Softwaresystems wider. Zweifellos bieten sie eine bestimmte Sicht auf einen architektonischen Aufbau; die Kästen allerdings fälschlicherweise mit dem großen Ganzen – also der Architektur selbst – zu verwechseln, heißt definitiv, das große Ganze – und somit die Architektur als solche – aus den Augen zu verlieren: Softwarearchitektur sieht nicht irgendwie besonders aus. Eine spezifische Visualisierung ist eine Entscheidungsfrage, keine Gegebenheit. Vielmehr handelt es sich um eine Entscheidung, die auf einer weiteren Auswahl von Möglichkeiten basiert, nämlich: was enthalten sein soll, was durch Form oder Farbgebung hervorgehoben werden soll, was durch Gleichförmigkeit oder Auslassung heruntergespielt werden soll. Eine Sichtweise hat gegenüber einer anderen nichts Natürliches oder Intrinsisches an sich.

Nun mag es nicht viel Sinn machen, sich im Kontext der Softwarearchitektur mit physikalischen Gesetzmäßigkeiten und physischen Maßstäben auseinanderzusetzen, dennoch müssen wir auch in diesem Bereich bestimmte physische Einschränkungen bedenken und entsprechend berücksichtigen. Prozessorgeschwindigkeiten und Netzwerkbandbreiten können ein hartes Urteil über die Performance eines Systems fällen. RAM- und Datenspeicherkapazitäten können den Ambitionen jeder Codebasis Grenzen setzen. Software mag einer dieser Stoffe sein, aus denen Träume gemacht sind, sie wird aber trotz allem in einer physischen Welt betrieben.

> *Das ist das Ungheuerliche in der Liebe, dass der Wille unendlich ist und die Ausführung beschränkt, dass das Verlangen grenzenlos ist und die Tat ein Sklave der Beschränkung.*
>
> – William Shakespeare[2]

2 Shakespeare, *Troilus und Cressida*, um 1601, Erstdruck 1610, erste deutsche Übers. von Johann Joachim Eschenburg, 1777. Hier übersetzt von Wolf Graf Baudissin, Georg Andreas Reimer, Berlin, 1832.

Es ist die physische Welt, in der wir ebenso wie unsere Unternehmen und unsere Volkswirtschaften existieren. Und diese Tatsache liefert uns einen weiteren Kalibrierungsgesichtspunkt, anhand dessen wir die Softwarearchitektur verstehen können – andere, weniger physische Kräfte und Größen, auf deren Grundlage wir uns verständigen und argumentieren können.

> *Architektur repräsentiert die signifikanten Designentscheidungen, die ein System formen und gestalten, wobei die Signifikanz an den Kosten von Änderungen bemessen wird.*
>
> – Grady Booch

Zeit, Geld und Aufwand geben uns einen Maßstab vor, um das Große und das Kleine, das Wesentliche und das weniger Wesentliche zu sortieren und die architektonischen Aspekte von dem Rest zu unterscheiden. Dieser Maßstab sagt uns auch, wie wir feststellen können, ob eine Architektur gut ist oder nicht: Eine gute Softwarearchitektur erfüllt die Bedürfnisse aller User, Entwickler und Product Owner (Produkteigentümer), und zwar nicht nur zu einem gegebenen Zeitpunkt, sondern langfristig.

> *Wenn Sie denken, eine gute Architektur sei teuer, dann probieren Sie es mal mit einer schlechten.*
>
> – Brian Foote und Joseph Yoder

Die Modifikationen und Anpassungen, die im Rahmen einer Systementwicklung typischerweise vorzunehmen sind, sollten nicht von der Art sein, dass sie kostspielig und schwer zu realisieren sind und eine jeweils eigene Projektverwaltung erforderlich machen, sondern in die täglichen und wöchentlichen Arbeitsabläufe eingebunden werden können.

Und das bringt uns zu einem nicht unerheblichen Physik-orientierten Problem: der Zeitreise. Wie wissen wir, welcher Art diese typischen Modifikationen bzw. Anpassungen sein werden, sodass wir die damit einhergehenden signifikanten Entscheidungen darauf ausrichten können? Wie reduzieren wir den zukünftigen Entwicklungsaufwand und die Kosten, ohne Kristallkugeln und Zeitmaschinen zu Hilfe zu nehmen?

> *Architektur ist die Menge der Entscheidungen, von denen Sie wünschten, dass Sie sie bereits frühzeitig in einem Projekt richtig treffen, bei denen die Wahrscheinlichkeit, sie auch tatsächlich richtig zu fällen, aber nicht unbedingt höher ist als bei allen anderen Entscheidungen auch.*
>
> – Ralph Johnson

Das Vergangene zu verstehen, ist schon schwierig genug. Unser Verständnis von dem Gegenwärtigen ist bestenfalls vage – und die Vorhersage des Zukünftigen ist alles andere als trivial.

An diesem Punkt verzweigt der Weg in viele Richtungen.

Auf dem dunkelsten Pfad wartet die Vorstellung, dass starke und stabile Architekturen direkte Abkömmlinge von Autorität und Starrheit sind. Ist eine Modifikation kostenintensiv, wird sie verworfen, die ursächlichen Beweggründe werden kleingeredet oder vollständig in bürokratischen Abgründen versenkt. Das Mandat des Architekten ist total und totalitär – und als Folge davon verkümmert die Architektur zu einer Dystopie für ihre Entwickler und eine ständige Quelle der Frustration für alle anderen.

Entlang eines anderen Abzweigs richtet sich das Interesse hingegen auf die sauberste Variante. Hier wird der »Weichheit« der Software Rechnung getragen und darauf hingearbeitet, sie als vorrangigste Eigenschaft des Systems zu bewahren. Ebenso wird nicht nur die Tatsache berücksichtigt, dass wir auf der Grundlage unvollständigen Wissens arbeiten, sondern auch anerkannt, dass dies für uns menschliche Wesen zudem etwas ist, worin wir gut sind. Diese Vorgehensweise kommt unseren Stärken mehr entgegen als unseren Schwächen. Wir erschaffen Dinge und wir entdecken Dinge. Wir stellen Fragen und führen Experimente durch. Eine gute Architektur kommt genau dann zustande, wenn wir sie als Reise und nicht als Ziel verstehen, mehr als einen fortlaufenden Prozess des Untersuchens denn als unumstößliches Artefakt.

> *Architektur ist eine Hypothese, die durch Implementierung und Bewertung bewiesen werden muss.*
>
> – Tom Gilb

Das Beschreiten dieses Pfades erfordert Sorgfalt und Aufmerksamkeit, Überlegungen und Beobachtung, Praxis und Prinzipien. Auf den ersten Blick mag sich dies nach einem schleppenden Prozess anhören, im Endeffekt hängt jedoch alles davon ab, wie Sie den Weg beschreiten.

> *Der einzige Weg, um schnell voranzukommen, ist gut voranzukommen.*
>
> – Robert C. Martin

Genießen Sie die Reise.

Kevlin Henney, Mai 2017

Einleitung

Der Titel dieses Buches lautet *Clean Architecture*, zu Deutsch also »Saubere Architektur«. Zugegeben, das ist eine recht selbstbewusst erscheinende Überschrift. Warum also habe ich mich für diesen Titel entschieden – und wieso habe ich dieses Buch überhaupt geschrieben?

Meine erste Programmzeile tippte ich 1964 im Alter von zwölf Jahren. Mit dem Schreiben dieses Buches begann ich im Jahr 2016 – ich programmiere inzwischen also bereits seit mehr als einem halben Jahrhundert. In all diesen Jahren blieb es natürlich nicht aus, dass ich ein paar Dinge über das Strukturieren von Softwaresystemen gelernt habe – Dinge, von denen ich glaube, dass sie auch für andere nützlich und wertvoll sind.

Dass ich mir dieses Wissen aneignen konnte, ist der Tatsache zu verdanken, dass ich in der Vergangenheit sehr viele Systeme entwickelt habe, sowohl große als auch kleine. Ich errichtete kleine eingebettete Systeme (Embedded Systems) ebenso wie große Stapelverarbeitungssysteme. Außerdem Echtzeit- und Websysteme. Und nicht zu vergessen Konsolen-Apps, GUI-Anwendungen, Prozesssteuerungsapplikationen, Spiele, Kontierungssysteme, Telekommunikationssysteme, Designtools, Zeichenanwendungen sowie viele, viele andere Systeme.

Dasselbe gilt für Singlethread-Anwendungen, Multithread-Anwendungen, Anwendungen mit wenigen schwergewichtigen Prozessen, Anwendungen mit vielen leichtgewichtigen Prozessen, Multiprozessoranwendungen, Datenbankanwendungen, mathematische Anwendungen, algorithmische Geometrieanwendungen sowie viele, viele andere Anwendungen.

Ich habe wirklich jede Menge Systeme entwickelt und Anwendungen programmiert. Und all diese Projekte brachten mich ausnahmslos zu einer erstaunlichen Erkenntnis:

Die Regeln der Architektur sind stets dieselben!

Erstaunlich ist dies vor allem insofern, als sich die Systeme, mit denen ich im Laufe der Jahre befasst war, radikal voneinander unterscheiden. Wie also kommt es, dass sie auf ähnlichen Architekturregeln basieren, wenn sie doch so verschieden sind? Meine Schlussfolgerung bezüglich dieser Frage lautet: *Weil die Regeln der Softwarearchitektur von allen anderen Variablen unabhängig sind.*

Bedenkt man dann noch den Wandel, der sich in den letzten 50 Jahren im Bereich der Hardware vollzogen hat, ist diese Erkenntnis umso bemerkenswerter. Meine ersten Programmierversuche machte ich auf Maschinen von der Größe eines Haushaltskühlschranks, mit einer Taktzeit von einem halben Megahertz, 4 KB Kernspeicher, 32 KB Festplattenspeicher und einer Teletype-Schnittstelle, die gerade mal zehn Zeichen pro Sekunde zuließ. Und nun sitze ich hier im Bus auf einer Rundreise durch Südafrika und schreibe dieses Geleitwort auf einem MacBook mit vier i7-Prozessorkernen, von denen jeder einzelne mit 2,8 Gigahertz läuft. Dieses Gerät verfügt über 16 GB RAM-Speicher, eine SSD-Festplatte mit einer Kapazität von einem Terabyte und ein Retina-Display mit einer Auflösung von 2.880x1.800, das in der Lage ist, extrem hochauflösende Videos abzuspielen. Die in den letzten Jahrzehnten in puncto Rechenpower erzielten Fortschritte sind geradezu atemberaubend. Jede halbwegs vernünftige Analyse wird bestätigen, dass mein MacBook mindestens 10^{22} Mal leistungsfähiger ist als die ersten Computer, mit denen ich seinerzeit vor einem halben Jahrhundert angefangen habe.

Eine 22-fache Zehnerpotenz ist schon eine geradezu unfassbare Größenordnung. Das entspricht der Anzahl der Angströms von der Erde bis nach Alpha Centauri. Oder auch der Anzahl der in dem Kleingeld in Ihrer Hosentasche oder Geldbörse enthaltenen Elektronen. Und doch beschreibt diese Zahl – *mindestens* – den inzwischen erzielten Anstieg der Rechenleistung, wie ich ihn in meiner bisherigen Lebenszeit erlebt habe.

Legt man nun diese gigantischen Fortschritte in Bezug auf die Rechnerperformance zugrunde, wie hat sich das dann auf die Software ausgewirkt, die ich schreibe? Es steht außer Frage, dass sie umfangreicher geworden ist: Früher hielt ich bereits ein aus 2.000 Zeilen bestehendes Programm für gewaltig – immerhin entsprach das einer etwa fünf Kilo schweren Kiste voller Lochkarten. Im Vergleich dazu gelten heutzutage erst Programme ab 100.000 Zeilen als groß.

Abgesehen davon ist die Software aber auch erheblich performanter geworden. Wir können inzwischen Dinge damit bewerkstelligen, die wir uns in den 1960er-Jahren nicht mal im Traum hätten vorstellen können. Die Science-Fiction-Bücher bzw. -Filme *Colossus, Revolte auf Luna* und *2001: Odyssee im Weltraum* waren allesamt Versuche, Zukunftsszenarien von unserer aktuellen Gegenwart abzubilden, die jedoch reichlich am Ziel vorbeigeschossen sind. In all diesen Fiktionen herrschte die Vorstellung von riesigen Maschinen mit Empfindungsvermögen oder einem Bewusstsein vor – was wir jedoch stattdessen vorweisen können, sind unglaublich winzige Maschinen, die trotzdem immer noch nur Maschinen sind.

Und es gibt noch eine Sache, die auffällt, wenn man die Software, die uns heute zur Verfügung steht, mit der von damals vergleicht: *Sie enthält immer noch dieselben Bestandteile*. Sie besteht wie gehabt aus `if`-Anweisungen, Zuweisungskommandos und `while`-Schleifen.

Nun mögen Sie einwenden, dass wir mittlerweile auf viel bessere Programmiersprachen und überlegenere Paradigmen zurückgreifen können – immerhin programmieren wir inzwischen in Java oder C# oder Ruby. Und wir verwenden objektorientiertes Design. Das ist zwar alles richtig, dennoch ist der Code an sich nach wie vor bloß eine Ansammlung von *Sequenzen* (Abfolgen), *Selektionen* (Verzweigungen) und *Iterationen* (Schleifen) – ganz genau so, wie es schon in den 1950er- und 1960er-Jahren der Fall war.

Wirft man einen genauen Blick auf die Praxis der Computerprogrammierung, stellt man fest, dass sich in den letzten 50 Jahren tatsächlich nur sehr wenig geändert hat. Sicher, die Sprachen sind ein bisschen besser geworden. Und die Tools sind sogar in exorbitantem Maße besser geworden. Die grundlegenden Bausteine eines Computerprogramms selbst haben sich allerdings nicht verändert.

Hätte ich 1966 eine Computerprogrammiererin[1] per Zeitreise in das Jahr 2016 befördert, sie an mein MacBook mit der *IntelliJ-IDE* (**I**ntegrated **D**evelopment **E**nvironment, integrierte Entwicklungsumgebung) gesetzt und ihr Java gezeigt, hätte sie sicherlich 24 Stunden gebraucht, um sich von dem Schock zu erholen. Aber unmittelbar danach wäre sie bereits in der Lage gewesen, Code zu schreiben – bei genauerer Betrachtung unterscheidet sich Java nämlich gar nicht so sehr von C oder auch von Fortran.

Im umgekehrten Fall, wenn ich Sie in das Jahr 1966 zurückbeamen und Ihnen zeigen würde, wie Sie PDP-8-Code schreiben und bearbeiten können, indem Sie mithilfe einer Teletype-Tastatur, die lediglich zehn Zeichen pro Sekunde schafft, einen Lochstreifen erstellen, müssten Sie sich vermutlich erst einmal 24 Stunden von der Enttäuschung erholen – doch dann wären Sie ebenfalls in der Lage, Code zu schreiben, denn: Der Code als solcher hat sich schlicht und ergreifend nicht allzu sehr verändert.

Und genau das ist das Geheimnis: Diese Unveränderlichkeit des Codes ist der Grund dafür, dass die Regeln der Softwarearchitektur über die verschiedenen Systemtypen und -variationen hinweg so konstant sind. Die Regeln, denen die Softwarearchitektur unterliegt, sind Regeln hinsichtlich der Anordnung und Zusammensetzung der einzelnen Bausteine von Programmen. Und da diese Bausteine allgemeingültig sind und sich nicht verändert haben, sind auch die Regeln für deren Anordnung gleichermaßen allgemeingültig und unveränderlich.

Nun mögen Programmierer der jüngeren Generation diese Aussage schlichtweg für Unsinn halten. Möglicherweise beharren sie sogar darauf, dass heutzutage alles neu und anders sei und die Regeln der Vergangenheit passé seien. Diejenigen, die so denken, täuschen sich jedoch. Die Regeln haben sich *nicht* geändert. Trotz all der neuen Programmiersprachen, all der neuen Frameworks und all der

1 Mit hoher Wahrscheinlichkeit hätte es sich hierbei um eine Frau gehandelt, denn damals war die Zunft der Programmierer zum überwiegenden Teil weiblich.

Paradigmen sind sie auch heute noch genau dieselben wie 1946, als Alan Turing den ersten Maschinencode schrieb.

Eines hat sich jedoch in der Tat geändert: Damals wussten wir noch nicht, wie diese Regeln genau lauteten. Und deshalb haben wir auch wieder und wieder dagegen verstoßen. Jetzt allerdings, nachdem wir ein halbes Jahrhundert lang Erfahrungen sammeln konnten, verstehen wir diese Regeln.

Und einzig und allein um genau diese Regeln – diese zeitlosen, unveränderlichen Regeln – geht es in diesem Buch.

Hinweis

Sollte es Updates, Korrekturen oder Ergänzungen zum Buch geben, finden Sie diese auf der Webseite des mitp-Verlags unter `www.mitp.de/724`, sobald sie verfügbar sind.

Über den Autor

Robert C. Martin (auch bekannt als »Uncle Bob«) ist seit 1970 als Softwareprogrammierer tätig. Er ist Mitbegründer der Organisation *cleancoders.com*, die Online-Videotrainings für Softwareentwickler bereitstellt. Des Weiteren gründete er das Unternehmen *Uncle Bob Consulting LLC*, das Dienstleistungen in den Bereichen IT-Beratung, Training und Skill Development für große Firmen weltweit anbietet. Außerdem war er als Master Craftsman in dem in Chicago ansässigen IT-Consulting-Unternehmen *8th Light Inc*. beschäftigt. Martin veröffentlichte Dutzende von Artikeln in diversen Handelsmagazinen und tritt regelmäßig als Redner bei internationalen Konferenzen und Kongressen auf. Er war drei Jahre lang Chefredakteur des Computermagazins *C++ Report* und ist zudem Erster Vorsitzender der *Agile Alliance*, einer gemeinnützigen Organisation zur Förderung der im *Agile Manifesto* festgelegten Konzepte der Agilen Softwareentwicklung.

Martin hat zahlreiche Bücher geschrieben sowie redaktionell bearbeitet, darunter Titel wie *Clean Coder: Verhaltensregeln für professionelle Programmierer* (mitp, 2014), *Clean Code – Refactoring, Patterns, Testen und Techniken für sauberen Code* (mitp, 2009), *UML for Java Programmers* (Pearson, 2003), *Agile Software Development* (Prentice Hall International, 2013), *Extreme Programming in Practice* (Addison-Wesley, 2001), *More C++ Gems* (Cambridge University Press, 2000), *Pattern Languages of Program Design 3* (Addison-Wesley, 1997) sowie *Designing Object Oriented C++ Aplications Using the Booch Method* (Prentice Hall, 1995).

Danksagung

Die folgenden Personen waren maßgeblich an der Entstehung dieses Buches beteiligt (in keiner bestimmten Reihenfolge):

Chris Guzikowski	Kent Beck
Chris Zahn	Martin Fowler
Matt Heuser	Alistair Cockburn
Jeff Overbey	James O. Coplien
Micah Martin	Tim Conrad
Justin Martin	Richard Lloyd
Carl Hickman	Ken Finder
James Grenning	Kris Iyer (CK)
Simon Brown	Mike Carew
Kevlin Henney	Jerry Fitzpatrick
Jason Gorman	Jim Newkirk
Doug Bradbury	Ed Thelen
Colin Jones	Joe Mabel
Grady Booch	Bill Degnan

Darüber hinaus gab es noch zahlreiche weitere Mitwirkende, deren namentliche Erwähnung den Rahmen dieser kurzen Danksagung jedoch sprengen würde.

Die finale Überarbeitung des Kapitels *Schreiende Architektur* rief mir Jim Weirichs strahlendes Lächeln und sein melodisches Lachen in Erinnerung. Mach's gut, Jim!

Teil I

Einführung

In diesem Teil:

Um ein Programm zum Laufen zu bringen, bedarf es nicht notwendigerweise Unmengen an Wissen und Fertigkeiten – schon Schüler sind dazu in der Lage. Nicht selten gründen junge Männer und Frauen noch während ihrer College- bzw. Hochschulzeit Milliarden-Dollar-Unternehmen, die lediglich auf ein paar Zeilen PHP oder Ruby basieren. Horden von Nachwuchsprogrammierern in Großraumbüros rund um den Globus rackern sich durch seitenschwere Anforderungsdokumentationen, die in riesigen Issue-Tracking-Systemen (auch »Fallbearbeitungssysteme« genannt) vorgehalten werden, um ihre eigenen Softwareentwicklungen durch schiere unerschütterliche *Willenskraft* zum Funktionieren zu bringen. Der Code, den sie dabei schreiben, mag nicht unbedingt hübsch anzusehen sein, aber er funktioniert, weil es in der Tat nicht allzu schwierig ist, ein Programm – zumindest einmalig – zum Laufen zu bringen.

Eine völlig andere Sache ist es allerdings, das Ganze *richtig* anzufangen. Eine wirklich gute Software auf die Beine zu stellen, das ist ein *schwieriges* Unterfangen. Es erfordert in der Tat ein gewisses Maß an Wissen und Fertigkeiten, das die meisten jungen Programmierer noch nicht erworben haben. Ebenso sind Überlegungen und Einblicke notwendig, die zu erlangen die meisten Softwareentwickler sich nicht die Zeit nehmen. Darüber hinaus erfordert gutes Programmieren auch einiges an Disziplin und Hingabe, wie es sich die meisten von uns nicht mal im Traum vorgestellt hätten. Vor allem aber erfordert es echte Leidenschaft für das Handwerk selbst und den unbedingten Wunsch, dieser Arbeit professionell nachzugehen.

Doch wenn Sie die Software richtig hinbekommen, dann passiert etwas Magisches: Sie brauchen keine Armada von Programmierern, um sie funktionsfähig zu halten. Sie brauchen keine seitenschweren Anforderungsdokumentationen und keine riesigen Issue-Tracking-Systeme. Ebenso wenig brauchen Sie Großraumbüros, und Sie müssen auch nicht an sieben Tagen in der Woche rund um die Uhr programmieren.

Wenn Software in der richtigen Art und Weise entwickelt wird, erfordert ihre Erstellung und Instandhaltung lediglich einen Bruchteil der vorgenannten menschli-

chen Ressourcen. Vielmehr lassen sich Modifikationen und Anpassungen schnell und einfach umsetzen. Mängel und Fehler treten nur hin und wieder mal in Erscheinung. Der Aufwand ist minimal, und das bei maximaler Funktionalität und Flexibilität.

Zugegeben, diese Vision klingt ein wenig nach Utopie, aber ich weiß, dass es funktioniert – denn ich habe es selbst erlebt. Ich habe an Projekten mitgewirkt, deren Design und Architektur sowohl das Schreiben als auch die Instandhaltung des jeweiligen Systemcodes leichtfallen ließen. Ich habe auch Projekte erlebt, die lediglich einen Bruchteil der ursprünglich angenommenen menschlichen Ressourcen erforderten. Und ich habe an Systemen gearbeitet, die extrem niedrige Fehlerraten aufwiesen. Ich selbst war Zeuge des außergewöhnlichen Effekts, den eine gute Softwarearchitektur auf ein System, ein Projekt und ein Team haben kann. Kurz: Ich war in diesem Gelobten Land.

Aber natürlich sollten Sie sich nicht nur auf mein Wort verlassen. Blicken Sie doch einmal auf Ihre eigenen Erfahrungen zurück: Haben Sie das Gegenteil erlebt? Haben Sie an Systemen gearbeitet, die so ineinander verwoben und derart kompliziert verknüpft waren, dass sich jede noch so simple Modifikation bzw. Anpassung wochenlang hinzog und mit enormen Risiken einherging? Haben Sie die durch schlecht geschriebenen Code und miserables Design verursachten Hemmnisse am eigenen Leib erfahren? Haben Sie erlebt, dass sich das Design der Systeme, an denen Sie gearbeitet haben, in massiver Weise negativ auf die Moral des Teams, das Vertrauen der Kunden und die Geduld der Managementebene ausgewirkt hat? Haben Sie miterlebt, dass Teams, Abteilungen und vielleicht sogar ganze Unternehmen von der armseligen Struktur Ihrer Software zugrunde gerichtet wurden? Mit anderen Worten: Waren Sie schon mal in der Programmierhölle?

Ich für meinen Teil war es – so wie die meisten unserer Zunft. Denn leider kommt es bedeutend häufiger vor, dass man sich durch grauenvolle Softwaredesigns hindurchkämpfen muss, als dass man das große Vergnügen hat, mit einem wirklich guten Design zu arbeiten.

Kapitel 1

Was bedeuten »Design« und »Architektur«?

Im Laufe der Jahre sorgte die Verwendung der Begriffe »Design« und »Architektur« immer wieder für Verwirrung. Was ist Design? Was ist Architektur? Und wo liegt der Unterschied?

Eine der Zielsetzungen dieses Buches besteht darin, Ihnen einen Weg durch das verworrene Dickicht der Begriffsauslegungen aufzuzeigen und ein für alle Mal zu definieren, was genau unter Design und Architektur zu verstehen ist. Für den Anfang soll an dieser Stelle zunächst einmal festgehalten werden, dass es eigentlich keinen Unterschied gibt. *Überhaupt keinen Unterschied.*

Das Wort »Architektur« wird häufig im Zusammenhang mit Dingen auf einer übergeordneten Ebene verwendet, die sich von den Details auf einer untergeordneten Ebene unterscheiden. Mit »Design« scheinen dagegen oftmals Strukturen und Entscheidungen auf einer untergeordneten Ebene bezeichnet zu werden. Betrachtet man die Arbeit eines echten Architekten, ist diese Begriffsverwendung jedoch geradezu absurd.