



**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



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, Husby, www.drei-satz.de

Inhaltsverzeichnis

	Vorwort	15
	Einleitung	19
	Über den Autor	23
	Danksagung	24
Teil I	Einführung	25
<hr/>		
1	Was bedeuten »Design« und »Architektur«?	29
1.1	Das Ziel?	30
1.2	Fallstudie	31
1.2.1	Die Signatur des Chaos	32
1.2.2	Die Perspektive der Unternehmensleitung	33
1.2.3	Was ist schiefgelaufen?	34
1.3	Fazit	37
2	Die Geschichte zweier Werte	39
2.1	Verhalten	39
2.2	Architektur	40
2.3	Der größere Wert	41
2.4	Das Eisenhower-Prinzip	42
2.5	Der Kampf für die Architektur	43
Teil II	Die ersten Bausteine setzen: Programmierparadigmen	45
<hr/>		
3	Die Paradigmen	47
3.1	Strukturierte Programmierung	47
3.2	Objektorientierte Programmierung	48
3.3	Funktionale Programmierung	48
3.4	Denkanstöße	49
3.5	Fazit	49

4	Strukturierte Programmierung	51
4.1	Die Beweisführung	52
4.2	Eine »schädliche« Proklamation	54
4.3	Funktionale Dekomposition	55
4.4	Keine formalen Beweise	55
4.5	Wissenschaft als Rettung	55
4.6	Tests	56
4.7	Fazit	57
5	Objektorientierte Programmierung	59
5.1	Datenkapselung [?]	60
5.2	Vererbung [?]	63
5.3	Polymorphie	65
	5.3.1 Die Macht der Polymorphie	67
	5.3.2 Abhängigkeitsumkehr	68
5.4	Fazit	71
6	Funktionale Programmierung	73
6.1	Quadrierung von Integern	74
6.2	Unveränderbarkeit und Architektur	75
6.3	Unterteilung der Veränderbarkeit	76
6.4	Event Sourcing	77
6.5	Fazit	79
Teil III Designprinzipien		81
7	SRP: Das Single-Responsibility-Prinzip	85
7.1	Symptom 1: Versehentliche Duplizierung	86
7.2	Symptom 2: Merges	88
7.3	Lösungen	89
7.4	Fazit	90
8	OCP: Das Open-Closed-Prinzip	91
8.1	Ein Gedankenexperiment	92
8.2	Richtungssteuerung	95
8.3	Information Hiding	95
8.4	Fazit	96
9	LSP: Das Liskov'sche Substitutionsprinzip	97
9.1	Gesteuerte Nutzung der Vererbung	98

9.2	Das Quadrat-Rechteck-Problem	98
9.3	Das LSP und die Softwarearchitektur	99
9.4	Beispiel für einen Verstoß gegen das LSP	99
9.5	Fazit	101
10	ISP: Das Interface-Segregation-Prinzip	103
10.1	Das ISP und die Programmiersprachen	104
10.2	Das ISP und die Softwarearchitektur	105
10.3	Fazit	105
11	DIP: Das Dependency-Inversion-Prinzip	107
11.1	Stabile Abstraktionen	108
11.2	Factories	109
11.3	Konkrete Komponenten	110
11.4	Fazit	110
Teil IV Komponentenprinzipien		111
12	Komponenten	113
12.1	Eine kurze Historie der Komponenten	114
12.2	Relokatierbarkeit	116
12.3	Linker	117
12.4	Fazit	119
13	Komponentenkohäsion	121
13.1	REP: Das Reuse-Release-Equivalence-Prinzip	121
13.2	CCP: Das Common-Closure-Prinzip	123
13.2.1	Ähnlichkeiten mit dem SRP	124
13.3	CRP: Das Common-Reuse-Prinzip	124
13.3.1	Relation zum ISP	125
13.4	Das Spannungsdiagramm für die Komponentenkohäsion	125
13.5	Fazit	127
14	Komponentenkopplung	129
14.1	ADP: Das Acyclic-Dependencies-Prinzip	129
14.1.1	Der wöchentliche Build	130
14.1.2	Abhängigkeitszyklen abschaffen	131
14.1.3	Auswirkung eines Zyklus in einem Komponenten- abhängigkeitsgraphen	133
14.1.4	Den Zyklus durchbrechen	134
14.1.5	Jitters (Fluktuationen)	135

14.2	Top-down-Design	136
14.3	SDP: Das Stable-Dependencies-Prinzip	137
14.3.1	Stabilität	137
14.3.2	Stabilitätsmetriken	139
14.3.3	Nicht alle Komponenten sollten stabil sein	141
14.3.4	Abstrakte Komponenten	143
14.4	SAP: Das Stable-Abstractions-Prinzip	143
14.4.1	Wo werden die übergeordneten Richtlinien hinterlegt?	143
14.4.2	Einführung in das SAP (Stable-Abstractions-Prinzip)	143
14.4.3	Bemessung der Abstraktion	144
14.4.4	Die Hauptreihe	144
14.4.5	Die »Zone of Pain«.	145
14.4.6	Die »Zone of Uselessness«.	146
14.4.7	Die Ausschlusszonen vermeiden	147
14.4.8	Abstand von der Hauptreihe	147
14.5	Fazit	149
Teil V Softwarearchitektur		151
15	Was ist Softwarearchitektur?	153
15.1	Entwicklung	155
15.2	Deployment	155
15.3	Betrieb	156
15.4	Instandhaltung	157
15.5	Optionen offenhalten	157
15.6	Geräteunabhängigkeit	159
15.7	Junk Mail	161
15.8	Physische Adressierung	162
15.9	Fazit	163
16	Unabhängigkeit	165
16.1	Use Cases	165
16.2	Betrieb	166
16.3	Entwicklung	167
16.4	Deployment	167
16.5	Optionen offenhalten	168
16.6	Layer entkoppeln	168
16.7	Use Cases entkoppeln	169
16.8	Entkopplungsmodi	170

16.9	Unabhängige Entwickelbarkeit	171
16.10	Unabhängige Deploybarkeit	171
16.11	Duplizierung	171
16.12	Entkopplungsmodi (zum Zweiten)	172
16.12.1	Welcher Modus ist am besten geeignet?	173
16.13	Fazit	174
17	Grenzen: Linien ziehen	175
17.1	Ein paar traurige Geschichten	176
17.2	FitNesse	179
17.3	Welche Grenzen sollten Sie ziehen – und wann?	181
17.4	Wie verhält es sich mit der Ein- und Ausgabe?	184
17.5	Plug-in-Architektur	185
17.6	Das Plug-in-Argument	186
17.7	Fazit	187
18	Anatomie der Grenzen	189
18.1	Grenzüberschreitungen	189
18.2	Der gefürchtete Monolith	190
18.3	Deployment-Komponenten	192
18.4	Threads	192
18.5	Lokale Prozesse	193
18.6	Services	193
18.7	Fazit	194
19	Richtlinien und Ebenen	195
19.1	Ebene	196
19.2	Fazit	199
20	Geschäftsregeln	201
20.1	Entitäten	202
20.2	Use Cases	203
20.3	Request-and-Response-Modelle	205
20.4	Fazit	206
21	Die schreiende Softwarearchitektur	207
21.1	Das Thema einer Architektur	208
21.2	Der Zweck einer Softwarearchitektur	208
21.3	Aber was ist mit dem Web?	209
21.4	Frameworks sind Tools, keine Lebenseinstellung	209
21.5	Testfähige Architekturen	210
21.6	Fazit	210

22	Die saubere Architektur	211
22.1	Die Abhängigkeitsregel (Dependency Rule)	213
22.1.1	Entitäten	213
22.1.2	Use Cases	213
22.1.3	Schnittstellenadapter	214
22.1.4	Frameworks und Treiber	214
22.1.5	Nur vier Kreise?	215
22.1.6	Grenzen überschreiten	215
22.1.7	Welche Daten überqueren die Grenzlinien?	216
22.2	Ein typisches Beispiel	216
22.3	Fazit	217
23	Presenters und »Humble Objects«	219
23.1	Das Pattern »Humble Object«	219
23.2	Presenters und Views	220
23.3	Das Testen und die Softwarearchitektur	221
23.4	Datenbank-Gateways	221
23.5	Data Mappers	221
23.6	Service Listeners	222
23.7	Fazit	222
24	Partielle Grenzen	223
24.1	Den letzten Schritt weglassen	224
24.2	Eindimensionale Grenzen	224
24.3	Fassaden	225
24.4	Fazit	226
25	Layer und Grenzen	227
25.1	Hunt the Wumpus	228
25.2	Saubere Architektur?	229
25.3	Datenstromüberschreitungen	231
25.4	Datenströme teilen	232
25.5	Fazit	234
26	Die Komponente Main	235
26.1	Das ultimative Detail	235
26.2	Fazit	239
27	Services – große und kleine	241
27.1	Servicearchitektur?	241

27.2	Vorteile der Services?	242
27.2.1	Denkfalle: Entkopplung	242
27.2.2	Denkfalle: Unabhängige Entwickel- und Deploybarkeit	243
27.3	Das Kätzchen-Problem	243
27.4	Objekte als Rettung	245
27.5	Komponentenbasierte Services	247
27.6	Cross-Cutting Concerns	248
27.7	Fazit	248
28	Die Testgrenze	249
28.1	Tests als Systemkomponenten	249
28.2	Design für Testfähigkeit	250
28.3	Die Test-API	251
28.3.1	Strukturelle Kopplung	251
28.3.2	Sicherheit	252
28.4	Fazit	252
29	Saubere eingebettete Architektur	253
29.1	App-Eignungstest	256
29.2	Der Flaschenhals der Zielhardware	259
29.2.1	Eine saubere eingebettete Architektur ist eine testfähige eingebettete Architektur.	260
29.2.2	Offenbaren Sie dem HAL-User keine Hardwaredetails	263
29.3	Fazit	269
Teil VI	Details	271
30	Die Datenbank ist ein Detail	273
30.1	Relationale Datenbanken	274
30.2	Warum sind Datenbanksysteme so weit verbreitet?	274
30.3	Was wäre, wenn es keine Festplatten gäbe?	275
30.4	Details	276
30.5	Und was ist mit der Performance?	276
30.6	Anekdote	277
30.7	Fazit	278
31	Das Web ist ein Detail	279
31.1	Der immerwährende Pendelausschlag	280
31.2	Quintessenz	281
31.3	Fazit	282

32	Ein Framework ist ein Detail	283
32.1	Framework-Autoren	284
32.2	Asymmetrische Ehe	284
32.3	Die Risiken	285
32.4	Die Lösung	285
32.5	Hiermit erkläre ich euch zu	286
32.6	Fazit	286
33	Fallstudie: Software für den Verkauf von Videos	287
33.1	Das Produkt	287
33.2	Use-Case-Analyse	288
33.3	Komponentenarchitektur	289
33.4	Abhängigkeitsmanagement	291
33.5	Fazit	291
34	Das fehlende Kapitel	293
34.1	Package by Layer	294
34.2	Package by Feature	295
34.3	Ports and Adapters	297
34.4	Package by Component	299
34.5	Der Teufel steckt in den Implementierungsdetails	304
34.6	Organisation vs. Kapselung	305
34.7	Andere Entkopplungsmodi	308
34.8	Fazit: Der fehlende Ratschlag	309
A	Architekturarchäologie	311
A.1	Das Buchhaltungssystem für die Gewerkschaft	312
A.2	Zurechtschneiden mit dem Laser	317
A.3	Monitoring von Aluminiumspritzguss	321
A.4	4-TEL	322
	A.4.1 Service Area Computer	326
	A.4.2 Ermittlung des Wartungsbedarfs	327
	A.4.3 Architektur	327
	A.4.4 Die große Neugestaltung	329
	A.4.5 Europa	330
	A.4.6 SAC: Fazit	330
A.5	Die Programmiersprache C	331
	A.5.1 C	332
A.6	BOSS	332

A.7	Projekt CCU	333
	A.7.1 Denkfalle: Die Planung	334
A.8	DLU/DRU	335
	A.8.1 Architektur	336
A.9	VRS	337
	A.9.1 Der Name	338
	A.9.2 Architektur	338
	A.9.3 VRS: Fazit	339
A.10	Der Elektronische Rezeptionist	340
	A.10.1 Der Untergang des ER	341
A.11	Craft Dispatch System	342
A.12	Clear Communications	344
	A.12.1 Die Gegebenheiten	345
	A.12.2 Uncle Bob	346
	A.12.3 Das Telefongespräch	346
A.13	ROSE	347
	A.13.1 Fortsetzung der Debatten	348
	A.13.2 ... unter anderem Namen	348
A.14	Prüfung zum eingetragenen Architekten	349
A.15	Fazit	352
B	Nachwort	353
	Stichwortverzeichnis	357

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 hinauffragen, oder wunderbare, 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. Prozessgeschwindigkeiten 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 Shakespeare²

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 kleineredert 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 1022 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*, *Revolt 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 Computerprogrammiererin¹ per Zeitreise in das Jahr 2016 befördert, sie an mein MacBook mit der *IntelliJ-IDE* (Integrated Development Environment, 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++ Applications 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:

- **Kapitel 1**
Was bedeuten »Design« und »Architektur«? 29
- **Kapitel 2**
Die Geschichte zweier Werte. 39

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.

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.