



3.

Auflage



Carola Lilienthal

Langlebige Software- Architekturen

Technische Schulden analysieren,
begrenzen und abbauen



Dr. Carola Lilienthal ist Geschäftsführerin der WPS Workplace Solutions GmbH in Hamburg und verantwortet dort den Bereich Softwarearchitektur. Sie hat 1995 an der Universität Hamburg ihr Diplom in Informatik gemacht und dort zum Thema »Komplexität von Softwarearchitekturen« promoviert. Seit 2003 analysiert sie international im Auftrag ihrer Kunden die Architektur von Softwaresystemen und berät Entwicklungsteams, wie sie die Langlebigkeit ihrer Softwaresysteme verbessern können. Insgesamt hat sie mit ihrem Team über dreihundert Softwaresysteme zwischen 30000 und 15 Mio. LOC in Java, C++, C#, ABAP, PHP und TypeScript untersucht. Besonders am Herzen liegt ihr die Ausbildung von Softwarearchitekten und Entwicklern, weshalb sie aktives Mitglied bei iSAQB, International Software Architecture Qualification Board, ist und ihr Wissen aus über 25 Jahren Softwareentwicklung regelmäßig auf Konferenzen, in Artikeln und bei Schulungen weitergibt.

Carola Lilienthal

Langlebige Softwarearchitekturen

**Technische Schulden analysieren, begrenzen
und abbauen**

3., überarbeitete und erweiterte Auflage



dpunkt.verlag

Carola Lilienthal
Carola.Lilienthal@wps.de
www.lisa.de
www.langlebige-softwarearchitektur.de

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Nadine Thiele, Frank Heidt
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

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:
Print 978-3-86490-729-6
PDF 978-3-96088-896-3
ePub 978-3-96088-897-0
mobi 978-3-96088-898-7

3., überarbeitete und erweiterte Auflage 2020
Copyright © 2020 dpunkt.verlag GmbH
Wiebinger Weg 17
69123 Heidelberg

Hinweis:

Dieses Buch wurde auf PEFC-zertifiziertem Papier aus nachhaltiger
Waldwirtschaft gedruckt. Der Umwelt zuliebe verzichten wir
zusätzlich auf die Einschweißfolie.



Schreiben Sie uns:

Falls Sie Anregungen, Wünsche und Kommentare haben, lassen Sie es uns wissen: hallo@dpunkt.de.

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

Geleitwort

Eigentlich ...

... wissen Softwareentwickler(innen) und -architekt(inn)en ganz genau, worauf sie bei Entwicklung und Änderung von Software achten sollten: Einsatz etablierter Architektur- und Entwurfsmuster, saubere Modularisierung, lose Kopplung, hohe Kohäsion und Durchgängigkeit (Konsistenz und innere Ordnung), dazu eine große Portion sinnvoller weiterer Entwurfsprinzipien. Haben wir alle gelernt, geübt und erfolgreich genutzt.

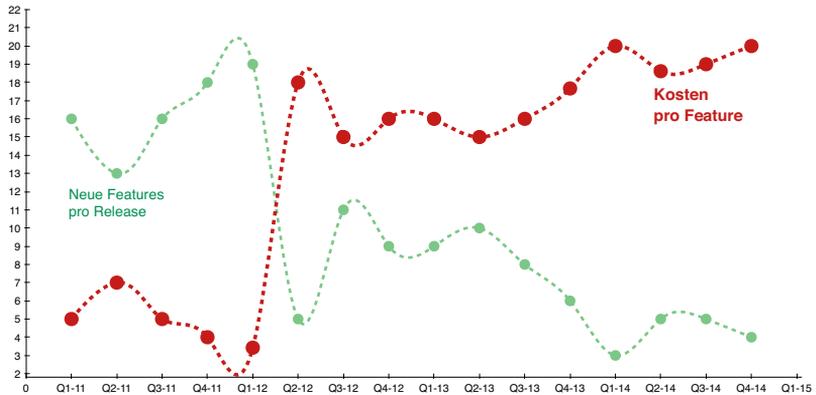
Dennoch ...

... geht in den Tücken der Praxis so einiges schief: Viele Softwaresysteme erkranken über kurz oder lang an der IT-Seuche Nr. 1 – der »generellen Verrottung«: Folgen dieser Malaise:

- Wartungs- und Änderungskosten steigen unaufhaltsam auf ein schier unerträgliches Maß an.
- Intransparenz wohin man nur schaut. Kaum jemand überblickt noch die Konsequenzen von Änderungen. Selbst kleine Erweiterungen werden zum technischen Vabanquespiel.
- Arbeit an solchen Systemen wird zur Qual – obwohl Softwareentwicklung an sich zu den interessantesten Tätigkeiten überhaupt gehört. ☹

Der folgenden Abbildung liegt kein reales System zugrunde, spiegelt aber meine Erfahrung in Dutzenden mittlerer und großer Systeme aus unterschiedlichen Branchen und Technologien wider:

Abb. 1
Steigende Pflegekosten



Endlich ...

... bricht Carola Lilienthal die Lanze für die beiden (in der Praxis allzu oft vergessenen) Qualitätsmerkmale Langlebigkeit und Wartbarkeit. Diese beiden bilden eine wesentliche Grundlage der inneren Qualität von Software. Niemand kann sie einer Software von außen ansehen – aber alle Stakeholder von Systemen möchten sie haben:

- Auftraggeber von Systemen freuen sich, weil sich die hohen Investitionen in Erstellung und Wartung lohnen und weitere Änderungen *kostengünstig* sind.
- Anwender und Fachabteilungen freuen sich, weil Änderungen am System schnell und mit hoher Zuverlässigkeit erfolgen können.
- Entwickler und Softwarearchitekten freuen sich, weil Arbeit an *sauberen* Systemen viel produktiver ist. Niemand braucht mehr Angst vor bösen Nebenwirkungen einer kleinen Änderung zu haben.

Prima ...

... dass Carola dieses Buch geschrieben hat: Ihre langjährigen Erfahrungen auf dem Gebiet der Architekturanalyse von Systemen unterschiedlicher Technologien sind einzigartig. Dadurch stellt sie in jedem Winkel dieses Buches den nötigen Praxisbezug her.

Ich durfte als einer der Ersten das Manuskript lesen – und habe sehr davon profitiert. Freuen Sie sich auf spannende und lehrreiche Unterhaltung!

Die Dritte ...

... Auflage geht noch weiter – und bringt Ihnen Clean-, Onion- und hexagonale Architekturen näher, sehr vorteilhaft für Wartbarkeit und Änderbarkeit Ihrer Systeme. Mir selbst hat Carolas Buch oftmals hilfreiche Anregungen für strukturelle Architekturentscheidungen geben können – danke dafür.

Gernot Starke
Köln, September 2019
www.arc42.de

Vorwort zur 3. Auflage

Liebe Leserinnen und Leser, nun ist nach weiteren zwei Jahren auch schon die 2. Auflage verkauft. Ich danke Ihnen sehr herzlich für Ihr Interesse und vor allem für das viele Feedback, das ich bekomme!

Wie in der 2. Auflage habe ich auch in dieser Auflage Fehler behoben und drei neue Abschnitte in das Buch aufgenommen. Neu hinzugekommen ist der Abschnitt 3.6 zu TypeScript-Systemen. In unserer eigenen Entwicklung in der WPS, aber auch bei unseren Kunden sehen wir immer mehr Systeme, deren Frontend in TypeScript implementiert ist. Hier blind zu fliegen, weil wir keine Möglichkeiten haben, die Architektur zu überprüfen, hat uns keine Ruhe gelassen. Da das von uns hauptsächlich eingesetzte Analysewerkzeug, der Sotograph, eine offene Schnittstelle hat, haben wir uns einen passenden Parser für TypeScript entwickeln lassen. Drei Systeme haben wir in TypeScript schon analysiert, sind aber noch nicht so weit, dass wir allgemeingültige Aussagen machen wollen. Bis zur vierten Auflage werden wir das aber bestimmt schaffen.

Der zweite neue Abschnitt ist 6.4 zu hexagonalen Architekturen, Onion Architecture und Clean Architecture. Zu diesem Abschnitt hat mich ein Kommentar eines Rezensenten angeregt. Sicherlich könnte hier noch sehr viel mehr stehen, aber mir war es erst einmal wichtig, diese Architekturstile in den Gesamtzusammenhang des Buches zu stellen. Außerdem habe ich den Abschnitt zu Microservices und DDD (Abschnitt 7.5) um einige neue Erkenntnisse erweitert. Über Ihre Meinung und weitere Kommentare zu diesen Themen würde ich mich sehr freuen.

Carola Lilienthal
Hamburg, August 2019

@caiolali
www.langlebige-softwarearchitektur.de

Vorwort zur 2. Auflage

Liebe Leserinnen und Leser, als der Buchbestand der ersten Auflage zu Ende ging, habe ich die Gelegenheit genutzt, für die zweite Auflage einige Anpassungen und Erweiterungen zu machen. In den letzten zwei Jahren habe ich viele weitere Systeme analysiert, ausführliche Diskussionen mit anderen Architekten geführt und eine Reihe von Vorträgen und Schulungen zum Thema Langlebigkeit und Architektur gehalten. Schließlich habe ich noch mit meinem Kollegen Henning Schwentner das Buch »Domain-Driven Design Distilled« von Vaughn Vernon übersetzt¹. Bei all diesen Aktivitäten habe ich Neues gelernt, Dinge aus einer anderen Richtung betrachtet und auch meine Meinung geändert.

Die folgenden Veränderungen habe ich in der zweiten Ausgabe vorgenommen: In Kapitel 4 habe ich den Modularity Maturity Index hinzugefügt (Abschnitt 4.5), mit dem wir seit eineinhalb Jahren die Ergebnisse unserer Analysen vergleichbar machen. Der direkt daran anschließende Abschnitt 4.6 zu technischen Schulden im Lebenszyklus ist um verschiedene Möglichkeiten ergänzt, die Langlebigkeit im Entwicklungsprozess zu erhöhen. In Kapitel 6 habe ich den Abschnitt zu Microservices (Abschnitt 6.5) komplett überarbeitet und den Abschnitt zur Mustersprache des Domain-Driven Design (Abschnitt 6.6.2) verbessert. In den letzten zwei Jahren wurde ich immer wieder darum gebeten, ein System daraufhin zu untersuchen, ob und wie es in Microservices zerlegt werden kann. Diese Erfahrungen habe ich zusammengefasst und an einem Beispiel illustriert (Abschnitt 7.5). Ich würde mich freuen, wenn dieser neue Abschnitt zu einer Reihe von kontroversen Diskussionen führen würde, denn das Zerlegen von Monolithen in Microservices wird die Aufgabe der nächsten Jahre sein.

1. Vernon, V.: *Domain-Driven Design kompakt*, dpunkt.verlag, Heidelberg, 2017.

Selbstverständlich wurden in dieser Ausgabe auch Fehler behoben, die trotz der wunderbaren Arbeit des dpunkt.verlags durchgerutscht sind. Mein Dank gilt allen Lesern, die mich auf diese Fehler aufmerksam gemacht haben – das war eine große Hilfe!

Carola Lilienthal
Hamburg, April 2017
@caiolali
www.llsa.de

Vorwort zur 1. Auflage

Liebe Leser und Leserinnen, ich begrüße Sie ganz herzlich in diesem Buch zu langlebiger Softwarearchitektur. In den nun folgenden Kapiteln möchte ich Sie in das Innere von Softwaresystemen entführen und Ihnen die Schönheiten und Grausamkeiten zeigen, die man dort finden kann.

In den vergangenen Jahren hatte ich das Glück, in Softwaresysteme hineinschauen zu dürfen. Dabei habe ich mir viele Gedanken gemacht und mit vielen Architekten diskutiert, welche Strukturen warum langlebiger sind als andere. Sie finden in diesem Buch also viele Empfehlungen zu allem, was die Entwicklung langlebiger Systeme ausmacht; viele Geschichten aus der Praxis, mit denen ich versuche, die Empfehlungen lebendig werden zu lassen; viele Bilder aus echten Systemen, damit die Empfehlungen plastisch werden; und schließlich auch ein wenig Theorie, um zu erklären, was Menschen schneller erfassen und im Kopf behalten können.

Ich würde mich freuen, wenn Sie mir Feedback zu meinen Erfahrungen geben. Vielleicht haben Sie Ähnliches gesehen. Vielleicht schauen Sie aber auch ganz anders auf Softwarearchitektur. Ich bin gespannt und hoffe, dass Sie auf den nächsten Seiten Interessantes, Informatives, Diskussionswürdiges und manchmal vielleicht auch Amüsantes finden.

Zum Schluss dieses Vorworts möchte ich mich bei allen bedanken, die mich im letzten Jahr beim Schreiben dieses Buches begleitet haben: Dank an meine Familie, die mir immer Rückhalt gibt bei all meinen Vorhaben. Auch bei einer so verrückten Idee, wie ein Buch schreiben zu wollen. Ihr seid wunderbar! Merci beaucoup!

Dank an alle, die Texte reviewt haben: Eberhard Wolff, Gernot Starke, Johannes Rost, Stefan Sarstedt, Stefan Tilkov, Stefan Zörner, Tobias Zepfer, Ulf Fildebrandt und meine anonymen Reviewer. Eure Kommentare haben mich zum Nachdenken, Umdenken und Weiterdenken gebracht – danke!

Ganz herzlichen Dank an Gernot Starke für das tolle Geleitwort! Es macht mir sehr viel Freude, mit Dir über Architektur zu diskutieren und mein Wissen und mein Verständnis mit Deiner Hilfe weiter zu schärfen.

Dank an all meine Kollegen in der WPS Workplace-Solution für die vielen Diskussionen um Architektur, ohne Euch hätte dieses Buch niemals entstehen können: Holger Breitling, Martin Fahl, Guido Gryczan, Stefan Hofer, Bettina Koch, Jörn Koch, Michael Kowalczyk, Tobias Rathjen, Kai Rüstmann, Arne Scharping, Lasse Schneider, Henning Schwentner und Heinz Züllighoven. Dank an alle, die mir den Rücken freigehalten haben, sodass ich in Ruhe schreiben konnte, insbesondere: Martina Bracht-Kopp, Inge Fontaine, Petra Gramß und Doris Nied.

Vielen Dank an Thomas Schoen und Heinrich Rust, die mich schon so lange in meiner Arbeit beim Architekturreview begleiten. Es ist toll, mit Euch zusammenarbeiten zu dürfen!

Herzlichen Dank an die Mitarbeiter des dpunkt.verlags, die mich so freundlich und konstruktiv durch das letzte Jahr begleitet haben.

Und zu guter Letzt vielen Dank an alle Kunden, die mir erlaubt haben, von ihren Systemen zu erzählen. Sie haben einen wertvollen Beitrag für dieses Buch geleistet!

Carola Lilienthal

Hamburg, Oktober 2015

www.langlebige-softwarearchitektur.de

www.llsa.de

Inhaltsverzeichnis

1	Einleitung	1
1.1	Softwarearchitektur	1
1.2	Langlebigkeit	3
1.3	Technische Schulden	4
1.3.1	»Programmieren kann jeder!«	8
1.3.2	Komplexität und Größe	9
1.3.3	Die Architekturerosion steigt unbemerkt	11
1.3.4	Für Qualität bezahlen wir nicht extra!	13
1.3.5	Arten von technischen Schulden	14
1.4	Was ich mir alles anschauen durfte	15
1.5	Wer sollte dieses Buch lesen?	16
1.6	Wegweiser durch das Buch	16
2	Aufspüren von technischen Schulden	19
2.1	Begriffsbildung für Bausteine	19
2.2	Soll- und Ist-Architektur	21
2.3	Verbesserung am lebenden System	25
2.4	False Positives und generierter Code	43
2.5	Spickzettel zum Sotographen	45
3	Architektur in Programmiersprachen	47
3.1	Java-Systeme	47
3.2	C#-Systeme	52
3.3	C++-Systeme	54
3.4	ABAP-Systeme	56
3.5	PHP-Systeme	57
3.6	TypeScript -Systeme	59

4	Architekturanalyse und -verbesserung	61
4.1	Entwickler und Architektur	61
4.2	Architekturarbeit ist eine Holschuld	62
4.3	Live-Workshop zur Architekturverbesserung	63
4.4	Der Umgang mit den Vätern und Müttern	65
4.5	Modularity Maturity Index (MMI)	66
4.6	Technische Schulden im Lebenszyklus	68
5	Kognitive Psychologie und Architekturprinzipien	71
5.1	Modularität	72
5.1.1	Chunking	72
5.1.2	Übertragung auf Entwurfsprinzipien	74
5.1.2.1	Einheiten	75
5.1.2.2	Schnittstellen	77
5.1.2.3	Kopplung	78
5.2	Musterkonsistenz	79
5.2.1	Aufbau von Schemata	80
5.2.2	Übertragung auf Entwurfsprinzipien	82
5.3	Hierarchisierung	86
5.3.1	Bildung von Hierarchien	86
5.3.2	Übertragung auf Entwurfsprinzipien	88
5.4	Zyklen = misslungene Modularität + Muster	90
5.5	Konsequenzen für die Architekturanalyse	91
6	Architekturstile gegen technische Schulden	93
6.1	Regeln von Architekturstilen	93
6.2	Trennung von fachlichen und technischen Bausteinen	94
6.3	Schichtenarchitekturen	96
6.3.1	Technische Schichtung	97
6.3.2	Fachliche Schichtung	98
6.3.3	Infrastrukturschicht	100
6.3.4	Integration von fachlichen Schichten	102
6.4	Hexagonal, Onion und Clean Architecture	103
6.5	Microservices und Domain-Driven Design	105
6.6	Mustersprachen	108
6.6.1	WAM-Mustersprache	110
6.6.2	DDD-Mustersprache	112
6.6.3	Typische Framework-Muster	114
6.7	Langlebigkeit und Architekturstile	116

7	Muster in Softwarearchitekturen	117
7.1	Abbildung der Soll-Architektur auf die Ist-Architektur	117
7.2	Die ideale Struktur: fachlich oder technisch?	120
7.3	Schnittstellen von Bausteinen	125
7.4	Interfaces – das architektonische Allheilmittel?	130
	7.4.1 Die Basistherapie	130
	7.4.2 Die Nebenwirkungen	132
	7.4.3 Feldstudien am lebenden Patienten	135
	7.4.4 Der Kampf mit dem Monolithen	138
7.5	Der Wunsch nach Microservices	140
	7.5.1 Früh übt sich	141
	7.5.2 Der Knackpunkt: das Domänenmodell	143
8	Mustersprachen – der architektonische Schatz!	147
8.1	Die Schatzsuche	147
8.2	Die Ausgrabungsarbeiten	149
8.3	Aus der Schatztruhe	151
8.4	Den Goldanteil bestimmen	155
8.5	Jahresringe	156
8.6	Unklare Muster führen zu Zyklen	157
9	Chaos in Schichten – der tägliche Schmerz	161
9.1	Bewertung des Durcheinanders	163
	9.1.1 Ausmaß der Unordnung	164
	9.1.1.1 Architekturstile und Zyklen	166
	9.1.1.2 Programmzeilen in Zyklen	167
	9.1.1.3 Dependency Injection und Zyklen	169
	9.1.2 Umfang und Verflochtenheit	169
	9.1.3 Reichweite in der Architektur	172
9.2	Das große Wirrwarr	176
	9.2.1 Der Schwarze-Loch-Effekt	178
	9.2.2 Der Befreiungsschlag	180
	9.2.3 Technische Schichtung als Waffe	182
	9.2.4 Mustersprache als Leuchtturm	184
9.3	Uneven Modules	187
10	Modularität schärfen	191
10.1	Kohäsion von Bausteinen	192
10.2	Größen von Bausteinen	196
10.3	Größen von Klassen	196

10.4	Größe und Komplexität von Methoden	202
10.5	Lose Kopplung	205
10.6	Kopplung und Größe von Klassen	211
10.7	Wie modular sind Sie?	213
11	Geschichten aus der Praxis	215
11.1	Das Java-System Alpha	216
11.2	Das C#-System Gamma	224
11.3	Das C++-System Beta	232
11.4	Das Java-System Delta	241
11.5	Das Java-System Epsilon mit C#-Satelliten	248
	11.5.1 Java-Epsilon	248
	11.5.2 C#-Epsilon 1	256
	11.5.3 C#-Epsilon 2	259
11.6	Das ABAP-System Lambda	264
12	Fazit: der Weg zu langlebigen Architekturen	271

Anhang

A	Analysewerkzeuge	277
A.1	Lattix	279
A.2	Sonargraph Architect	280
A.3	Sotograph und SotoArc	282
A.4	Structure101	283
	Literatur	287
	Index	295

1 Einleitung

Softwaresysteme gehören mit Sicherheit zu den komplexesten Konstruktionen, die Menschen erdacht und erbaut haben. Insofern ist es nicht verwunderlich, dass Softwareprojekte scheitern und Altsysteme – aus Angst, dass sie ihren Dienst einstellen – nicht mehr angerührt werden. Trotzdem begegnen mir immer wieder Projektteams, die ihre Softwaresysteme unabhängig von ihrem Anwendungsgebiet, ihrer Größe und ihrem Alter im Griff haben. Erweiterungen und Fehlerbehebung sind in akzeptabler Zeit machbar. Neue Mitarbeiter können mit vertretbarem Aufwand eingearbeitet werden. Was machen diese Projektteams anders? Wie schaffen sie es, mit ihren Softwarearchitekturen langfristig und gut zu leben?

Die Frage nach den Ursachen für das langfristige Gelingen oder Scheitern von Softwareentwicklung und Softwarewartung lässt sich auf vielen Ebenen beantworten: dem Anwendungsgebiet und der involvierten Organisation, der eingesetzten Technologie, der fachlichen Qualität des Softwaresystems oder auch der Qualifikation der Anwender und Entwickler. In diesem Buch lege ich den Fokus auf die *Langlebigkeit von Softwarearchitekturen*. Ich zeige Ihnen, welche Faktoren ausschlaggebend sind, um eine Softwarearchitektur mit gleichbleibendem Aufwand über viele Jahre zu warten und zu erweitern.

*Langlebigkeit von
Softwarearchitekturen*

1.1 Softwarearchitektur

Bis heute hat sich die Informatik nicht auf eine einzige Definition von Softwarearchitektur festlegen können. Vielmehr gibt es mehr als 50 verschiedene Definitionen, die jeweils bestimmte Aspekte von Architektur hervorheben. Für dieses Buch werden wir uns an zwei der prominentesten Definitionen halten:

50 Architekturdefinitionen

Definition 1:

»Softwarearchitektur ist die Struktur eines Software-Produkts. Diese umfasst Elemente, die extern wahrnehmbaren Eigenschaften der Elemente und die Beziehungen zwischen den Elementen.« [Bass et al. 2012]

Sichten auf Architektur

Diese Definition spricht mit Absicht sehr allgemein von Elementen und Beziehungen. Denn mit diesen beiden Grundstoffen können ganz verschiedenste Sichten auf Architektur beschrieben werden. Die statische Sicht (oder auch Bausteinsicht) enthält als Elemente: Klassen, Packages, Namespaces, Directories, Projekte – also alles, was man in der jeweiligen Programmiersprache an Behältern für Programmzeilen hat. In der Verteilungssicht findet man als Elemente: Archive (JARs, WARs, Assemblies), Rechner, Prozesse, Kommunikationsprotokolle und -kanäle etc. In der dynamischen Sicht (oder auch Laufzeitsicht) interessiert man sich für die Objekte zur Laufzeit und ihre Interaktion. In diesem Buch werden wir uns mit Strukturen in der Bausteinsicht (statischen Sicht)¹ beschäftigen und sehen, warum manche Strukturen langlebiger sind als andere.

Die zweite Definition für Softwarearchitektur, die mir sehr wichtig ist, definiert Architektur nicht über ihre Struktur, sondern über Entscheidungen.

Definition 2:

»Softwarearchitektur = \sum aller wichtige Entscheidungen

Wichtige Entscheidungen sind alle Entscheidungen, die im Verlauf der weiteren Entwicklung nur schwer zu ändern sind.« [Kruchten 2004]

Struktur vs. Entscheidungen

Die beiden Definitionen sind sehr verschieden. Die erste legt auf sehr abstraktem Niveau fest, woraus die Struktur eines Softwaresystems besteht. Die zweite Definition bezieht sich auf Entscheidungen, die die Entwickler oder Architekten für das System getroffen haben. Die zweite Definition öffnet damit den Raum für alle übergreifenden Aspekte von Architektur, wie Technologieauswahl, Auswahl des Architekturstils, Integration, Sicherheit, Performance und vieles, vieles mehr. Diese Aspekte sind genauso wichtig für die Architektur, wie die gewählte Struktur, sind aber nicht das Thema dieses Buches.

1. Die Strukturen der Bausteinsicht haben in der Regel auch Einfluss auf die Verteilungssicht. Abschnitt 7.2 enthält einen Vorschlag für die Abbildung der Verteilungssicht in der Bausteinsicht.

In diesem Buch geht es um die Entscheidungen, die die Struktur eines Softwaresystems beeinflussen. Hat ein Entwicklungsteam zusammen mit seinen Architekten entschieden, welche Struktur das Softwaresystem haben soll, so legen sie *Leitplanken für die Architektur* fest.

Entscheidung schafft Leitplanken.

Leitplanken für die Architektur

Schaffen Sie eine Architektur, die den Designraum bei der Entwicklung des Softwaresystems einschränkt und Ihnen dadurch bei Ihrer Arbeit die Richtung weist.

Mit Leitplanken können sich die Entwickler und Architekten orientieren. Die Entscheidungen werden alle in eine einheitliche Richtung gelenkt und lassen sich mithilfe der Leitplanken verstehen und nachvollziehen. Das Softwaresystem bekommt auf diese Weise eine gleichförmige Struktur. Bei der Lösung von Wartungsaufgaben dirigieren die Leitplanken alle Beteiligten in eine einheitliche Richtung und die Anpassung oder Erweiterung des Softwaresystems führt zu schnelleren und einheitlicheren Ergebnissen.

Leitplanken für die Entwicklung

Dieses Buch wird die Fragen beantworten, welche Leitplanken zu langlebigen Architekturen führen und damit die Lebensdauer des Softwaresystems verlängern.

1.2 Langlebigkeit

Bei Software, die nur kurzzeitig im Einsatz ist, ist es nicht so wichtig, ob die Architektur auf Langlebigkeit ausgelegt ist. Ein Beispiel für ein solches Stück Software ist ein Programm zur Migration von Daten aus einem Altsystem in die Datenbank für die neue Anwendung. Diese Software wird einmal gebraucht und dann hoffentlich weggeworfen. Hier steht »hoffentlich«, weil man in vielen Softwaresystemen solche nicht mehr verwendeten Programmteile findet. Sie werden nicht weggeworfen, weil die Entwickler davon ausgehen, dass sie sie später noch einmal brauchen könnten. Außerdem ist das Löschen von Codezeilen, die man mit viel Nachdenken erschaffen hat und die schließlich funktioniert haben, eine schwierige Angelegenheit. Es gibt kaum einen Entwickler oder Architekten, der das gerne tut.²

Kurzlebige Software

Die meiste Software, die wir heute programmieren, lebt wesentlich länger. Noch dazu wird sie häufig angefasst und angepasst. In vielen

Das Jahr-2000-Problem

2. Um das Wegwerfen von Software als etwas Positives wahrzunehmen, hat die Clean-Code-Bewegung Workshops mit Namen »Code Kata« erfunden, bei denen dasselbe Problem mehrfach gelöst wird und der Code nach jedem Schritt weggeworfen wird.

Fällen wird Software über viel mehr Jahre verwendet, als man es sich in seinen kühnsten Träumen hat vorstellen können. Denken wir z.B. an die Cobol-Entwickler, die in den 1960er/70er-Jahren die ersten größeren Cobol-Systeme in Banken und Versicherungen geschrieben haben. Damals war Speicherplatz teuer. Deshalb hat man bei jedem Feld, das man auf die Datenbank gespeichert hat, überlegt, wie man Speicherplatz sparen könnte. Für die Cobol-Entwickler damals war es eine sehr sinnvolle Entscheidung, die Jahreszahlen nur als zweistellige Felder anzulegen. Niemand konnte sich damals vorstellen, dass diese Cobol-Programme auch im Jahr 2000 noch existieren würden. In den Jahren vor der Jahrtausendwende musste daher einiges an Aufwand getrieben werden, um all die alten Programme umzustellen. Wären die Cobol-Entwickler in den 1960/70er-Jahren davon ausgegangen, dass ihre Software so lange leben würde, hätten sie sicherlich vierstellige Felder für Jahreszahlen verwendet.

Unsere Software wird alt.

Für eine Reihe von Softwaresystemen, die wir heute bauen, ist eine so lange Lebensdauer durchaus realistisch. Schon allein aus dem Grund, dass viele Unternehmen die Investition in eine Neuentwicklung scheuen. Neuentwicklung erzeugt hohe Kosten – meistens höhere als geplant. Der Ausgang der Neuentwicklung ist ungewiss und die Anwender müssen mitgenommen werden. Zusätzlich wird die Organisation für die Zeit der Neuentwicklung ausgebremst und es entsteht ein Investitionsstau bei dringend benötigten Erweiterungen. Da bleibt man doch lieber bei der Software, die man hat, und erweitert sie, wenn es nötig wird. Vielleicht ist ein neues Frontend, das auf der alten Serversoftware arbeitet, ja auch schon ausreichend.

Alt und kostengünstig?

Diese Erwartung, dass sich die Investition in Software möglichst lange rentieren soll, liegt diesem Buch zugrunde: Unsere Software soll im Laufe ihres Lebens möglichst geringe Wartungs- und Erweiterungskosten verursachen, d.h., die technischen Schulden müssen so gering wie möglich gehalten werden.

1.3 Technische Schulden

Der Begriff »Technische Schulden« ist eine Metapher, die Ward Cunningham 1992 geprägt hat [Cunningham 1992]. Technische Schulden entstehen, wenn bewusst oder unbewusst falsche oder suboptimale technische Entscheidungen getroffen werden. Diese falschen oder suboptimalen Entscheidungen führen zu einem späteren Zeitpunkt zu Mehraufwand, der Wartung und Erweiterung verzögert.

Gute Vorsätze

Waren zu Beginn eines Softwareentwicklungsprojekts fähige Entwickler und Architekten im Team, so werden sie ihre besten Erfahrun-

gen und ihr gesammeltes Know-how eingebracht haben, um eine langlebige Architektur ohne technische Schulden zu entwerfen. Dieses Ziel lässt sich aber nicht zu Beginn des Projekts abhaken. So nach dem Motto: Wir entwerfen am Anfang eine langlebige Architektur und nun ist und bleibt alles gut.

Vielmehr kann man eine langlebige Architektur nur erreichen, wenn man die technischen Schulden ständig im Auge behält. In Abbildung 1-1 sehen Sie, was passiert, wenn man die technischen Schulden im Laufe der Zeit anwachsen lässt oder aber, wenn sie regelmäßig reduziert werden.

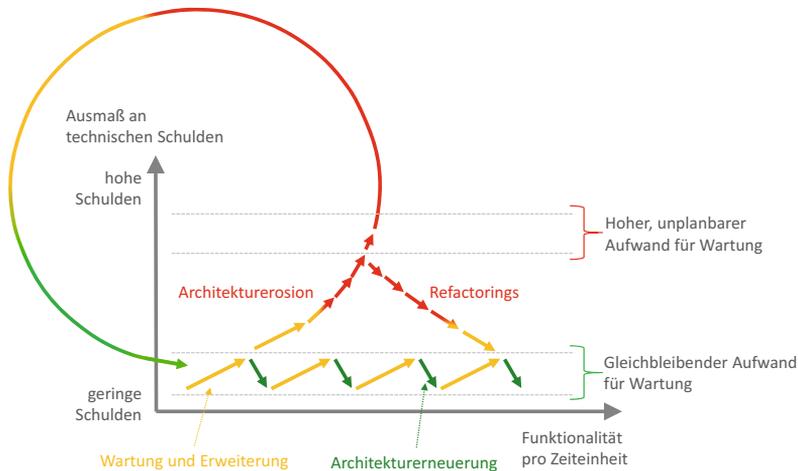


Abb. 1-1

Technische Schulden und Architekturerosion

Stellen wir uns ein Team vor, das ein System in Releases oder Iterationen immer weiter entwickelt. Haben wir ein qualitätsbewusstes Team im Einsatz, so wird sich das Team darüber im Klaren sein, dass es mit jeder Erweiterung einige neue technische Schulden aufnimmt (gelbe Pfeile in Abb. 1-1). Im Zuge der Erweiterung wird dieses Team also bereits darüber nachdenken, was an der Architektur zu verbessern ist. Währenddessen oder im Anschluss an die Erweiterung wird die technische Schuld dann wieder reduziert (grüne Pfeile in Abb. 1-1). Eine stetige Folge von Erweiterung und Verbesserung entsteht. Geht das Team so vor, dann bleibt das System in einem Korridor von geringen technischen Schulden mit einem gleichbleibenden Aufwand für die Wartung (s. grüne Klammer in Abb. 1-1).

Ein qualitätsbewusstes Team

Arbeitet man nicht auf diese Weise an einem stetigen Erhalt der Architektur, so geht die Architektur des Systems langsam verloren und die Wartbarkeit verschlechtert sich. Über kurz oder lang verlässt das

Ein chaotisches Team

Softwaresystem den Korridor geringer technischer Schulden (s. rote aufsteigende Pfeile in Abb. 1–1).

Architekturerosion

Die Architektur erodiert mehr und mehr. Wartung und Erweiterung der Software werden immer teurer bis zu dem Punkt, an dem jede Änderung zu einer schmerzhaften Anstrengung wird. In Abbildung 1–1 wird dieser Umstand dadurch deutlich gemacht, dass die roten Pfeile immer kürzer werden. Pro Zeiteinheit kann man bei steigender Architekturerosion immer weniger Funktionalität umsetzen, Bugs fixen und weniger Adaptionen an anderen Qualitätsanforderungen erreichen. Das Entwicklungsteam ist entsprechend frustriert und demotiviert und sendet verzweifelte Signale an Projektleitung und Management. Aber bis diese Warnungen erhört werden, ist es meistens viel zu spät.

Zu teuer!

Befindet man sich auf dem aufsteigenden Ast der roten Pfeile, so nimmt die Zukunftsfähigkeit des Softwaresystems kontinuierlich ab. Das Softwaresystem wird fehleranfällig. Das Entwicklungsteam kommt in den Ruf, es sei schwerfällig. Änderungen, die früher einmal in zwei Personentagen möglich waren, dauern jetzt doppelt bis dreimal so lange. Insgesamt geht alles zu langsam. »Langsam« ist in der IT-Branche ein Synonym für »zu teuer«. Genau! Technische Schulden sind angehäuft und bei jeder Änderung muss man neben der Erweiterung auch noch die Zinsen auf die technischen Schulden entrichten.

*Rückkehr zu guter
Qualität*

Der Weg, der aus diesem Dilemma der technischen Schulden herausführt, ist, die Architekturqualität rückwirkend zu verbessern. Dadurch kann das System Schritt für Schritt wieder in den Korridor geringer technischer Schulden zurückgebracht werden (s. rote absteigende Pfeile in Abb. 1–1). Dieser Weg ist anstrengend und kostet Geld – ist aber eine sinnvolle Investition in die Zukunft. Schließlich sorgt man dafür, dass die Wartung in Zukunft weniger anstrengend und billiger wird. Bei Softwaresystemen, die einmal eine gute Architektur hatten, führt dieses Vorgehen in der Regel schnell zum Erfolg.

CRAP-Cycle

Anders sieht es aus, wenn man im Korridor hoher technischer Schulden angelangt ist und der Aufwand für die Wartung unverhältnismäßig hoch und unplanbar wird (s. rote Klammer in Abb. 1–1). Ich werde oft gefragt, was es heißt, dass die Wartung unverhältnismäßig hoch ist. Eine generelle Antwort, die für alle Projekte gilt, ist selbstverständlich schwer zu geben. Allerdings habe ich bei verschiedenen Systemen mit guter Architektur beobachten können, dass pro 500.000 LOC die Kapazität von einem bis zwei Vollzeitentwicklern für die Wartung gebraucht wird. Das heißt, im Umfang von 40–80 Std. pro Woche pro 500.000 LOC werden Fehler behoben und kleine Anpassungen gemacht. Soll neue Funktionalität in das System eingebaut werden, dann muss natürlich mehr Kapazität eingeplant werden.

Komme ich zu einer Firma, um die Architektur zu bewerten, frage ich als Erstes nach der oder den Systemgrößen und als Zweites nach der Größe und Leistungsfähigkeit der Entwicklungsabteilung. Wenn die Antwort ist: »Für unser Java-System von 3 Millionen LOC beschäftigen wir 30 Entwickler, aber die sind alle nur noch mit Wartung beschäftigt und wir bekommen kaum noch neue Features eingebaut ...«, dann gehe ich davon aus, dass ich ein deutlich verschuldetes System vorfinden werde. Selbstverständlich ist dieses Maß sehr grob, aber als erster Hinweis war es bisher immer hilfreich.

Hat das System zu viele Schulden, um noch wartbar und erweiterbar zu sein, dann entscheiden sich Unternehmen immer wieder dafür, das System durch ein neues zu ersetzen (s. farbiger Kreis in Abb. 1–1). 2015 hat Peter Vogel den typischen Lebenszyklus eines Systems mit technischen Schulden zu meiner großen Freude als CRAP-Cycle bezeichnet. Das Akronym CRAP steht für C(reate) R(epair) A(bandon) (re)P(lace)³. Wenn das Reparieren des Systems nicht mehr hilft oder zu teuer erscheint, wird das System erst sich selbst überlassen und schließlich ersetzt.

Dieser letzte Schritt ist allerdings mit Vorsicht zu genießen. Bereits einmal Anfang 2000 wurden viele Altsysteme in COBOL und PL1 für nicht mehr wartbar erklärt und durch Java-Systeme ersetzt. Mit dieser neuen Programmiersprache sollte alles besser werden! Das versprach man damals den Managern, die das Geld für die Neuimplementierung zur Verfügung stellen sollten. Heute ist eine Reihe dieser mit viel Elan gebauten Java-Systeme voller technischer Schulden und verursacht immense Wartungskosten. An dieser traurigen Entwicklung sieht man sehr deutlich, dass sich das Ausmaß an technischen Schulden nicht durch die Wahl einer Programmiersprache oder spezieller Technologien begrenzen lässt, sondern nur durch Entwicklungsteams mit umfassendem Architektur-Know-how.

In meinem bisherigen Berufsleben sind mir immer wieder vier Ursachen für technische Schulden begegnet:

Ursachen von technischen Schulden

1. Das Phänomen »Programmieren-kann-jeder«
2. Die Architekturerosion steigt unbemerkt
3. Komplexität und Größe von Softwaresystemen
4. Das Unverständnis des Managements und der Kunden für Individualsoftwareentwicklung

Üblicherweise treten diese vier Punkte in Kombination auf und beeinflussen sich häufig auch gegenseitig.

3. <https://visualstudiomagazine.com/articles/2015/07/01/domain-driven-design.aspx>

1.3.1 »Programmieren kann jeder!«

Zu Besuch bei Physikern

Jedes Mal, wenn mich jemand fragt, warum ich mich so intensiv mit Softwarearchitektur und der Wartung von Softwaresystemen beschäftige, kommt mir ein Erlebnis in den Sinn: Im Jahre 1994 gegen Ende meines Informatikstudiums nahm ich an einer Besichtigung des DESY teil. Das DESY ist das in Hamburg ansässige »Deutsche Elektronen-Synchrotron«. Im Anschluss an einen einführenden Vortrag wurden wir von einem Physikstudenten über das Gelände geführt. Dabei durften wir verschiedene technische Anlagen mit einer Vielzahl von fürs DESY eigens entwickelten Geräten besichtigen. Uns wurden Labore gezeigt, in denen Physiker Experimente machen, und wir durften den Leitstand betreten, über den die gesamte Anlage überwacht und gesteuert wird.

*Programmierer =
Informatiker?*

Zum Abschluss passierten wir einen Raum, in dem Menschen vor Bildschirmen saßen und offensichtlich programmierten. Mein freudiger Ausruf: »Oh! Hier sitzen also die Informatiker!«, wurde von unserem Führer mit einem erstaunten Blick und den Worten quittiert: »Nein! Das sind alle Physiker! Programmieren können wir auch!«

Ich ging überrascht und verwirrt nach Hause. Später fragte ich mich aber noch oft, was der Physikstudent wohl dazu gesagt hätte, wenn ich ihm geantwortet hätte, dass ich als Informatikerin dann wohl seine Anlage genauso gut steuern könnte wie die Physiker. Schließlich gehörte die Informatik damals zu den Naturwissenschaften⁴.

*Programmieren ≠
Softwarearchitektur*

Dieses »Programmieren kann jeder!«-Phänomen verfolgt mich seit damals durch die unterschiedlichen Stationen meines Arbeitslebens. Nicht nur Physiker sagen und glauben diesen Satz, sondern auch Mathematiker, Ingenieure, Wirtschaftswissenschaftler, Wirtschaftsinformatiker und viele Informatiker, die kaum Softwarearchitektur in ihrem Studium gehört haben, schätzen sich so ein.

Und sie haben in der Regel alle recht: Programmieren können sie! Das heißt aber leider nicht, dass sie wissen oder gelernt haben, wie man Softwarearchitekturen oder gar Systemlandschaften aufbaut. Dieses Wissen kann man sich an einigen deutschen Universitäten im Masterstudiengang »Softwarearchitektur« aneignen. Manchmal hat man auch die Chance, von erfahrenen Softwarearchitekten in der täglichen Arbeit zu lernen. Oder man nimmt an einer guten Fortbildung zum Thema »Softwarearchitektur« teil.

Unbrauchbare Software

Solange aber »Programmieren können wir auch!« bei der Softwareentwicklung vorherrscht und das Management mit dieser Hal-

4. Heute wird die Informatik meistens als eine Disziplin neben Mathematik, Naturwissenschaften und Technik geführt (die sogenannten MINT-Fächer).

tung Entwicklungsteams zusammenstellt, werden wir weiterhin in schöner Regelmäßigkeit wartungsintensive Softwaresysteme zu Gesicht bekommen. Die Architektur dieser Systeme entsteht im Laufe der Zeit ohne Plan. Jeder Entwickler verwirklicht sich mit seinen lokalen Architektur- oder Entwurfsideen in seinem Teil der Software selbst. Häufig hört man dann: »Das ist historisch gewachsen!«

Technische Schulden werden in diesem Fall gleich zu Beginn der Entwicklung aufgenommen und kontinuierlich erhöht. Über solche Softwaresysteme kann man wohl sagen: Sie sind unter schlechten Bedingungen aufgewachsen. Solche Softwaresysteme sind häufig schon nach nur drei Jahren Entwicklungszeit und Einsatz nicht mehr zu warten.

Um diese Systeme überhaupt in die Nähe des Korridors geringer technischer Schulden zu bringen, müssen als Erstes die Architektur- und Entwurfsideen der Architekten und Entwickler auf ihre Qualität hin hinterfragt und vereinheitlicht werden. Das ist insgesamt deutlich aufwendiger, als ein System mit ehemals guter Architektur zurück auf den Pfad der Tugend zu führen. Aber auch solche großen Qualitäts-Refactorings lassen sich in beherrschbare Teilschritte zerlegen. Bereits nach den ersten kleineren Verbesserungen (Quick Wins) wird der Qualitätsgewinn in schnellerer Wartung spürbar. Häufig verursachen solche qualitätsverbessernden Arbeiten weniger Kosten als eine Neuimplementierung – auch wenn vielen Entwicklungsteams eine Neuentwicklung verständlicherweise sehr viel mehr Spaß macht. Diese positive Einstellung zum Neumachen geht oft damit einher, dass die Komplexität dieser Aufgabe unterschätzt wird.

Start mit technischen Schulden

Große Refactorings

1.3.2 Komplexität und Größe

Die Komplexität eines Softwaresystems speist sich aus zwei verschiedenen Quellen: dem Anwendungsproblem, für das das Softwaresystem gebaut wurde, und der Lösung aus Programmtext, Datenbank usw.

Für das Problem in der Fachdomäne muss eine angemessene Lösung gefunden werden. Eine Lösung, die dem Anwender erlaubt, mit dem Softwaresystem die geplanten Geschäftsprozesse durchzuführen. Man spricht hier von der *problemabhängigen* und der *lösungsabhängigen* Komplexität. Je höher die problemabhängige Komplexität ist, desto höher wird auch die lösungsabhängige Komplexität ausfallen müssen⁵.

Dieser Zusammenhang ist die Ursache dafür, dass Vorhersagen über die Kosten oder die Dauer einer Softwareentwicklung häufig zu gering ausfallen. Die eigentliche Komplexität des Problems kann zu

Problemabhängige Komplexität

5. s. [Ebert 1995], [Glass 2002] und [Woodfield 1979].

Beginn des Projekts nicht erfasst werden; und so wird die Komplexität der Lösung um ein Vielfaches unterschätzt⁶.

An dieser Stelle setzen agile Methoden an. Agile Methoden versuchen, am Ende jeder Iteration nur die als Nächstes zu implementierende Funktionalität zu schätzen. So werden die probleminhärente Komplexität und die daraus resultierende Komplexität der Lösung immer wieder überprüft.

Lösungsabhängige Komplexität

Nicht nur die probleminhärente Komplexität ist schwer zu bestimmen, auch die Lösung trägt zur Komplexität bei: Je nach Erfahrung und Methodenfestigkeit der Entwickler wird der Entwurf und die Implementierung zu einem Problem unterschiedlich komplex ausfallen⁷. Im Idealfall würde man sich wünschen, dass die Lösung eine für das Problem angemessene Komplexität aufweist. In diesem Fall kann man davon sprechen, dass es sich um eine gute Lösung handelt.

Essenziell oder akzidentell?

Weist die Lösung mehr Komplexität auf als das eigentliche Problem, so ist die Lösung nicht gut gelungen und ein entsprechendes Redesign ist erforderlich. Dieser Unterschied zwischen besseren und schlechteren Lösungen wird mit der *essenziellen* und *akzidentellen* Komplexität bezeichnet. Tabelle 1-1 fasst den Zusammenhang zwischen diesen vier Komplexitätsbegriffen zusammen.

Tab. 1-1 Komplexität

	Essenziell	Akzidentell
Probleminhärent	■ Komplexität der Fachdomäne	■ Missverständnisse über die Fachdomäne
Lösungsabhängig	■ Komplexität der Technologie und der Architektur	■ Missverständnisse über die Technologie ■ Überflüssige Lösungsanteile

Essenziell = unvermeidlich

Essenzielle Komplexität nennt man die Art von Komplexität, die im Wesen einer Sache liegt, also Teil seiner Essenz ist. Bei der Analyse der Fachdomäne versuchen die Entwickler, die essenzielle Komplexität des Problems zu identifizieren. Die einer Fachdomäne innewohnende essenzielle Komplexität führt zu einer entsprechend komplexen Lösung und lässt sich niemals auflösen oder durch einen besonders guten Entwurf vermeiden. Die essenzielle probleminhärente Komplexität ist also zur essenziellen Komplexität in der Lösung geworden.

Akzidentell = überflüssig

Im Gegensatz dazu wird der Begriff *akzidentelle Komplexität* verwendet, um auf Komplexitätsanteile hinzuweisen, die nicht notwendig sind und somit beseitigt bzw. verringert werden können. Akzidentelle Komplexität kann sowohl aus Missverständnissen bei der Analyse der

6. s. [Booch 2004] und [McBride 2007].

7. s. [Fenton & Pfleegler 1997] und [Henderson-Sellers 1996].

Fachdomäne als auch bei der Implementierung durch das Entwicklungsteam entstehen.

Wird bei der Entwicklung aus Unkenntnis oder mangelndem Überblick keine einfache Lösung gefunden, so ist das Softwaresystem überflüssigerweise komplex. Beispiele hierfür sind: Mehrfachimplementierungen, Einbau nicht benötigter Funktionalität und das Nichtbeachten softwaretechnischer Entwurfsprinzipien. Akzidentelle Komplexität kann von Entwicklern aber auch billigend in Kauf genommen werden, wenn sie z.B. gern neue und für das zu bauende Softwaresystem überflüssige Technologie ausprobieren wollen.

Selbst wenn ein Team es hinbekommen sollte, nur essenzielle Komplexität in seine Software einzubauen, ist Software aufgrund der immensen Anzahl ihrer Elemente ein für Menschen schwer zu beherrschendes Konstrukt. Ein intelligenter Entwickler ist nach meiner Erfahrung bei einer Änderung in der Lage, ca. 30.000 Zeilen Code⁸ zu überblicken und die Auswirkungen seiner Änderung an einer Stelle in den anderen Teilen des Codes vorauszuahnen. In der Regel sind die Softwaresysteme, die heute produktiv eingesetzt werden, sehr viel größer. Sie bewegten sich eher in der Größenordnung zwischen 200T und 100 Millionen Zeilen Code.

Software ist komplex.

All diese Argumente machen deutlich: Entwickler brauchen eine Softwarearchitektur, die ihnen den größtmöglichen Überblick bietet. Nur dann können sie sich in der vorhandenen Komplexität zurechtfinden. Haben die Entwickler den Überblick, so wird die Wahrscheinlichkeit höher, dass sie Änderungen an der Software korrekt durchführen. Bei ihren Änderungen können sie alle betroffenen Stellen berücksichtigen und die Funktionalität der nicht geänderten Codezeilen unangetastet lassen. Selbstverständlich sind weitere Techniken sehr hilfreich, wie automatisierte Tests und eine hohe Testabdeckung, Architekturausbildung und -weiterbildung sowie eine unterstützende Projekt- und Unternehmensorganisation.

Architektur reduziert Komplexität.

1.3.3 Die Architekturerosion steigt unbemerkt

Selbst bei einem fähigen Entwicklungsteam steigt die Architekturerosion unbemerkt. Wie kann es dazu kommen? Nun, häufig ist das ein schleichender Vorgang: Während der Implementierung weichen die Entwickler mehr und mehr von den Vorgaben der Architektur ab. In manchen Fällen tun sie das bewusst, weil die geplante Architektur den sich immer klarer herauschälenden Anforderungen doch nicht gerecht wird. Die Komplexität des Problems und der Lösung wurde unter-

Ein schleichender Prozess

8. Die Zahlen in diesem Absatz beziehen sich auf Java-Systeme.

schätzt und macht Änderungen in der Architektur notwendig. Es fehlt aber die Zeit, diese Änderungen konsequent im gesamten System nachzuziehen. In anderen Fällen müssen Probleme aus Zeit- und Kostendruck so schnell gelöst werden, dass keine Zeit bleibt, ein passendes Design zu entwickeln und die Architektur zu überdenken. Manchen Entwicklern ist die geplante Architektur auch gar nicht präsent, sodass sie ungewollt und unbemerkt dagegen verstoßen. Beispielsweise werden Beziehungen zwischen Komponenten eingebaut, die Schnittstellen missachten oder der Schichtung des Softwaresystems zuwiderlaufen. Oder Programmtext wird kopiert, anstatt über Abstraktionen und Wiederverwendung nachzudenken. Wenn man diesen schleichenden Verfall schließlich bemerkt, ist es höchste Zeit, einzugreifen!

*Symptome von starker
Architekturerosion*

Hat man den Tiefpunkt der Architekturerosion erreicht, so wird jede Veränderung zur Qual. Niemand möchte mehr an diesem System weiterarbeiten. Robert C. Martin hat in seinem Artikel »Design Principles and Design Pattern« diese Symptome eines verrotteten Systems gut auf den Punkt gebracht [Martin 2000]:

Starrheit

■ **Rigidity**

Das System ist unflexibel gegenüber Änderungen. Jede Änderung führt zu einer Kaskade von weiteren Anpassungen in abhängigen Modulen. Entwickler sind sich an vielen Stellen über Abläufe im System im Unklaren, es besteht eine Unbehaglichkeit gegenüber Änderungen. Was als eine kleine Anpassung oder ein kleines Refactoring beginnt, führt zu einem ständig länger werdenden Marathon von Reparaturen in immer weiteren Modulen. Die Entwickler jagen den Effekten ihrer Änderungen im Sourcecode hinterher und hoffen bei jeder Erkenntnis, das Ende der Kette erreicht zu haben.

Zerbrechlichkeit

■ **Fragility**

Änderungen am System führen zu Fehlern, die keinen offensichtlichen Bezug zu den Änderungen haben. Jede Änderung erhöht die Wahrscheinlichkeit neuer Folgefehler an überraschenden Orten. Die Scheu vor Änderungen wächst und der Eindruck entsteht, dass die Entwickler die Software nicht mehr unter Kontrolle haben.

Unbeweglichkeit

■ **Immobility**

Es gibt Entwurfs- und Konstruktionseinheiten, die eine ähnliche Aufgabe bereits lösen, wie die, die gerade neu implementiert werden soll. Diese Lösungen können aber nicht wiederverwendet werden, da zu viel »Gepäck« an dieser Einheit hängt. Eine generische Implementierung oder das Herauslösen ist ebenfalls nicht möglich, weil der Umbau der alten Einheiten zu aufwendig und fehleranfällig ist. Meist wird der benötigte Code kopiert, weil das weniger Aufwand erzeugt.

■ Viscosity

Sollen Entwickler eine Anpassung machen, gibt es in der Regel mehrere Möglichkeiten. Einige dieser Möglichkeiten erhalten das Design, andere machen das Design kaputt. Wenn diese »Hacks« leichter zu implementieren sind als die designerhaltende Lösung, dann ist das System zäh.

Zähigkeit

Gegen die Ursachen dieser Symptome müssen Entwicklungsteams stetig ankämpfen, damit ihr System langlebig bleibt und das Anpassen und Warten auf Dauer Spaß macht. Wenn da nur nicht die Kosten wären ...

1.3.4 Für Qualität bezahlen wir nicht extra!

Viele Kunden sind überrascht, wenn ihre Dienstleister – ob extern oder im Hause – ihnen sagen, dass sie Geld brauchen, um die Architektur und damit die Qualität des Softwaresystems zu verbessern. Häufig sagen die Kunden Sätze wie: »Es läuft doch! Was habe ich davon, wenn ich für Qualität Geld ausbebe?« oder »Ihr habt am Anfang den Vertrag bekommen, weil ihr uns versprochen habt, dass ihr gute Qualität liefert! Da könnt ihr doch jetzt kein Geld für Qualität fordern!«. Das sind sehr unangenehme Situationen. Denn als Softwareentwickler und -architekten ist unser erklärtes Ziel, dass wir Software mit einer langlebigen Architektur und hoher Qualität schreiben. An dieser Stelle deutlich zu machen, dass eine sich weiter entwickelnde Architektur eine Investition in die Zukunft ist und auf lange Sicht Geld spart, ist gar nicht so einfach.

*Architektur kostet
Extrageld.*

Diese Situationen entstehen, weil der Kunde oder das Management nicht erkennen oder erkennen wollen, dass Individualsoftwareentwicklung ein unplanbarer Prozess ist. Wird eine neue, so noch nie da gewesene Software entwickelt, so ist die essenzielle Komplexität schwer zu beherrschen. Die Software selbst, ihre Nutzung und ihre Integration in einen Kontext von Arbeitsorganisation und sich ändernden Geschäftsprozessen sind nicht vorhersehbar. Mögliche Erweiterungen oder neue Nutzungsformen lassen sich nicht vorausahnen. Das sind wesentliche Charakteristika jeder Individualsoftwareentwicklung!

*Individualsoftware-
entwicklung =
unplanbarer Prozess*

Heute ist eigentlich jedes Softwaresystem eine Individualsoftwareentwicklung: Die Integration in die IT-Landschaft des Kunden ist jedes Mal anders. Die technologischen und wirtschaftlichen Entwicklungen sind so rasant, dass ein Softwaresystem, was heute die exakt richtige Lösung mit der perfekten Architektur ist, morgen schon an seine Grenzen stößt. All diese Randbedingungen führen zu dem Schluss, dass Softwareentwicklung kein industriell herstellbares Gut ist.

*Software ≠ industriell
herstellbares Gut*

Sondern eine individuelle und zu einem Zeitpunkt sinnvolle Lösung mit einer hoffentlich langlebigen Architektur, die sich ständig weiter entwickeln muss. Zu dieser Weiterentwicklung gehören sowohl funktionale als auch nichtfunktionale Aspekte, wie innere und äußere Qualität.

Zum Glück können mehr und mehr Kunden die Begriffe »technische Schulden« und »Langlebigkeit« nachvollziehen.

1.3.5 Arten von technischen Schulden

In der Diskussion um technische Schulden werden viele Arten und Varianten aufgeführt. Für dieses Buch sind vier Arten von technischen Schulden relevant:

Code-Smells

■ **Implementationsschulden**

Im Sourcecode finden sich sogenannte Code-Smells, wie lange Methoden, Codeduplikate etc.

Struktur-Smells

■ **Design- und Architekturschulden**

Das Design der Klassen, Pakete, Subsysteme, Schichten und Module ist uneinheitlich, komplex und passt nicht mit der geplanten Architektur zusammen.

Unit Tests

■ **Testschulden**

Es fehlen Tests bzw. nur der Gut-Fall wird getestet. Die Testabdeckung mit automatisierten Unit Tests ist gering.

■ **Dokumentationsschulden**

Es gibt keine, wenig oder veraltete Dokumentation. Der Überblick über die Architektur wird nicht durch Dokumente unterstützt. Entwurfsentscheidungen sind nicht dokumentiert.

*Basisanforderung:
geringe Testschulden*

Die meisten Hinweise, Anregungen sowie gute und schlechte Beispiele finden Sie in diesem Buch zu den ersten beiden Punkten: Implementations- sowie Design- und Architekturschuld. Sie werden sehen, wie solche Schulden entstehen und reduziert werden können. Diese Schulden lassen sich aber nur gefahrlos verringern, wenn die Testschulden gering sind oder gleichzeitig reduziert werden. Insofern sind geringe Testschulden eine Basisanforderung. Eine Dokumentation der Architektur ist einerseits eine gute Grundlage für die Architekturanalysen und -verbesserung, um die es in diesem Buch geht. Das heißt, geringe Dokumentationsschulden helfen bei der Analyse. Andererseits entsteht bei der Architekturanalyse auch Dokumentation für das analysierte System, sodass die Dokumentationsschulden verringert werden.