# 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.

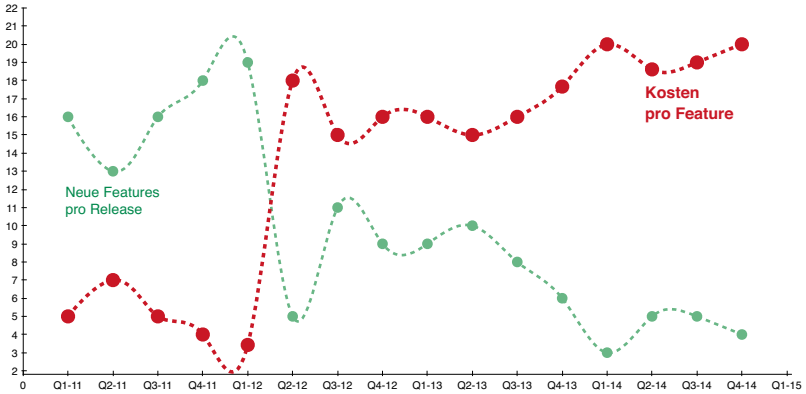
**Dennoch ...**

... 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 eine 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.

**Die Vierte ...**

... Auflage geht noch einen Schritt weiter – und rückt neben Clean Architecture auch die bewährte (aber leider oft vernachlässigte) Modularität in Form des hilfreichen Modularity Maturity Index (MMI) in den Fokus aktueller Architekturdiskussion. Zusätzlich lernen wir in der neuen Auflage einiges Nützliches über Domain-Driven Transformation, einen eleganten Weg zur systematischen Verbesserung bestehender Systeme. Mir persönlich hat Carolas Buch viele Denkanstöße für strukturelle Architekturentscheidungen geben können – danke dafür.

*Gernot Starke*
Köln, Februar 2024
*www.arc42.de*

# Vorwort zur 4. Auflage[1]

Liebe Leserinnen und Leser, in den letzten viereinhalb Jahren ist in unser aller Leben viel passiert, sowohl beruflich als auch gesamtgesellschaftlich. Einen Großteil meiner Arbeit als Architekturberaterin und Referentin habe ich in dieser Zeit virtuell am Bildschirm erledigt. Das war einerseits notwendig und hat mich andererseits manchmal gefreut, weil viel Reisezeit eingespart wurde und wird. Gleichzeitig sind virtuelle Begegnungen deutlich unpersönlicher und lassen kaum einen direkten Austausch zu. Umso mehr freue ich mich, dass wir inzwischen wieder unbeschwert zu Konferenzen fahren und Workshops und Beratung vor Ort durchführen können. Dabei hatte ich insbesondere in 2023 endlich wieder Gelegenheit, persönlich mit einigen von Ihnen über dieses Buch zu sprechen, und ich habe festgestellt, dass es nach wie vor seinen Platz auf der Bücherliste von Softwareentwicklern und Architekten hat. Das freut mich sehr und hat mir die Energie gegeben, mich an die 4. Auflage zu setzen. ☺

In dieser neuen Auflage habe ich insbesondere ein Thema deutlich erweitert: die Beschreibung des Modularity Maturity Index (MMI). In der Vorauflage war der MMI ein Unterkapitel des vierten Kapitels zu Architekturanalyse und -verbesserung. Nun hat der MMI ein eigenes Kapitel, das Kapitel 11, bekommen, in dem die Berechnung des MMI im Detail beschrieben ist. Ich hoffe, dass ich Sie damit in die Lage versetze, Ihre eigenen Systeme selbst zu vermessen und zu bewerten. Sollten Sie noch Inhalte zum MMI vermissen, dann bitte ich um Feedback.

1. Das Vorwort zur 2. und 3. Auflage dieses Buches finden Sie auf der Website des Buches unter: *www.langlebige-softwarearchitektur.de*.

Des Weiteren sind in diese Auflage auch Erkenntnisse aus meinem neusten Buch »Domain-Driven Transformation – Monolithen und Microservices zukunftsfähig machen« eingeflossen, das ich zusammen mit meinem Kollegen Henning Schwentner in den letzten zwei Jahren geschrieben habe. Die beiden Themen »Langlebige Softwarearchitekturen« und »Domain-Driven Design« sind eng miteinander verbunden und ergänzen sich.

*Carola Lilienthal*
Hamburg, Januar 2024
*@cairolali*
*www.langlebige-softwarearchitektur.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 Zepter, 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*

# 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?

*Langlebigkeit von Softwarearchitekturen*

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.

## 1.1 Softwarearchitektur

*50 Architekturdefinitionen*

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:

**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 wichtigen 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.

*Entscheidung schafft Leitplanken.*

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.

**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.

*Leitplanken für die Entwicklung*

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.

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

## 1.2 Langlebigkeit

*Kurzlebige Software*

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]

*Das Jahr-2000-Problem*

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

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 Erfahrungen 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.

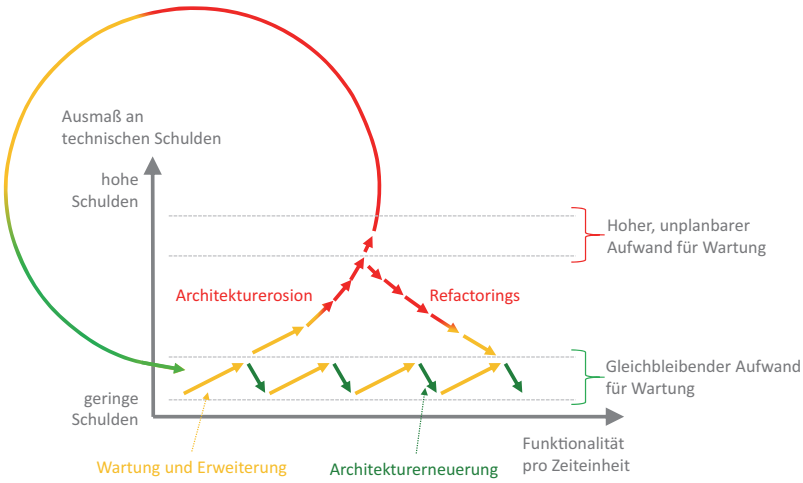



***Abb. 1–1***
*Technische Schulden und Architekturerosion*

*Ein qualitätsbewusstes Team*

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 chaotisches 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 Soft-

waresystem 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.

*Ursachen von technischen Schulden*

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

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 Naturwissenschaften[4].

*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 Haltung Entwicklungsteams zusammenstellt, werden wir weiterhin in schöner Regel-

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

mäßigkeit 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!«

*Start mit technischen Schulden*

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.

*Große Refactorings*

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.

### 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 *probleminhärenten* und der *lösungsabhängigen* Komplexität. Je höher die probleminhärente Komplexität ist, desto höher wird auch die lösungsabhängige Komplexität ausfallen müssen[5].

*Probleminhärente Komplexität*

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 Beginn

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

des Projekts nicht erfasst werden; und so wird die Komplexität der Lösung um ein Vielfaches unterschätzt[6].

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 ausfallen[7]. 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<br>■ 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

6. s. [Booch 2004] und [McBride 2007].
7. s. [Fenton & Pfleegler 1997] und [Henderson-Sellers 1996].

Komplexität kann sowohl aus Missverständnissen bei der Analyse der 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.

*Software ist komplex.*

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 Code[8] 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.

*Architektur reduziert Komplexität.*

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.

Das Thema Komplexität haben mein Kollege Henning Schwentner und ich in einem weiteren Buch mit dem Titel »Domain-Driven Transformation – Monolithen und Microservices zukunftsfähig machen« wieder aufgenommen (s. [Lilienthal & Schwentner 2023]).

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

### 1.3.3 Die Architekturerosion steigt unbemerkt

*Ein schleichender Prozess*

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 herausschälenden Anforderungen doch nicht gerecht wird. Die Komplexität des Problems und der Lösung wurde unterschätzt 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 Patterns« 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.

- **Immobility** *Unbeweglichkeit*
  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** *Zähigkeit*
  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.

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!

*Architektur kostet Extrageld.*

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 ausgebe?« 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.

*Individualsoftware-entwicklung = unplanbarer Prozess*

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!

*Software ≠ industriell herstellbares Gut*

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. 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 -ver-

besserung, 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.

## 1.4 Was ich mir alles anschauen durfte

Nach dem Ende meines Informatikstudiums 1995 habe ich als Softwareentwicklerin, später Softwarearchitektin, Projektleiterin und Beraterin gearbeitet. Seit 2002 durfte ich immer wieder Softwaresysteme auf ihre Qualität hin untersuchen. Zu Anfang noch allein durch Draufschauen auf den Sourcecode, seit 2004 werkzeuggestützt. Mit der Möglichkeit, ein Werkzeug über den Sourcecode laufen zu lassen und ihn nach bestimmten Kriterien insgesamt zu überprüfen, hat sich die Architekturanalyse und -verbesserung erst richtig entfalten können. In Kapitel 2 und 4 sehen Sie, wie solche Analysen und Verbesserungen technisch und organisatorisch ablaufen.

*Größen und Sprachen*

Im Laufe der Zeit durfte ich mir Systeme in Java (130), C++ (30), C# (70), ABAP (5) und PHP (20) und PLSQL (10) ansehen. Diese Liste wird bald um TypeScript und JavaScript erweitert werden. Jede dieser Programmiersprachen hat ihre Eigenarten, wie wir in Kapitel 3 sehen werden. Auch die Größe der Systeme (s. Abb. 1–2) hat Einfluss darauf, wie die Softwarearchitektur gestaltet ist oder sein müsste.

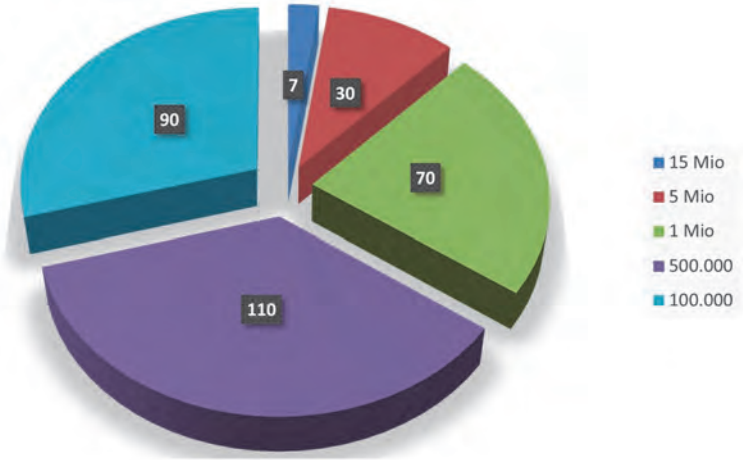



***Abb. 1–2***
*Größenordnung der untersuchten Systeme in Lines of Code (LOC)*

*Lines of Code*

Die Angabe Lines of Code (LOC) in Abbildung 1–2 beinhaltet sowohl die ausführbaren Zeilen Code als auch die Leerzeilen und Kommentarzeilen. Will man die Kommentar- und Leerzeilen herausrechnen, so muss man im Mittel 50 % des LOC abziehen. Typischerweise liegt das Ver-

hältnis zwischen ausführbarem und nicht ausführbarem Code zwischen 40 und 60 %. Je nachdem, ob das Entwicklungsteam die Beginn- und Ende-Markierungen für Blöcke (z. B. {}) in eigene Zeilen geschrieben hat, oder nicht. Die Größen von Systemen in diesem Buch sind immer Zahlen für Java-/C#-, C++- und PHP-Systeme. Diese Sprachen haben ungefähr eine ähnliche »Satzlänge«. ABAP-Programmierung ist sehr viel gesprächiger. Hier muss man mit ca. dem 2- bis 3-Fachen an Sourcecode rechnen.

All diese Analysen haben mein Verständnis von Softwarearchitektur und meine Erwartungen daran, wie ein Softwaresystem aufgebaut sein sollte, geschärft und vertieft.

## 1.5 Wer sollte dieses Buch lesen?

*Programmieren*

Dieses Buch ist für Architekten und Entwickler geschrieben, die in ihrer täglichen Arbeit mit Sourcecode arbeiten. Sie werden von diesem Buch am meisten profitieren, weil es auf potenzielle Probleme in großen und kleinen Systemen hinweist und Lösungen anbietet.

*Verbessern*

Berater mit Entwicklungserfahrung, praktizierende Architekten und Entwicklungsteams, die bestehende Softwarelösungen methodisch verbessern wollen, werden in diesem Buch viele Hinweise auf große und kleine Verbesserungen finden.

*Für die Zukunft lernen*

Unerfahrene Entwickler werden an manchen Stellen wahrscheinlich Probleme haben, die Inhalte zu verstehen, da sie die angesprochenen Probleme schlecht nachvollziehen können. Das grundlegende Verständnis für den Bau von langlebigen Softwarearchitekturen können aber auch sie aus diesem Buch mitnehmen.

## 1.6 Wegweiser durch das Buch

Das Buch besteht aus dreizehn Kapiteln, die zum Teil aufeinander aufbauen, aber auch getrennt voneinander gelesen werden können.

*Inhaltsschwerpunkte*

Abbildung 1–3 zeigt die Kapitel in unterschiedlichen Farben: Die beiden weißen Kapitel 1 und 13 geben dem Buch einen Rahmen. Das heißt, der Leser wird in das Thema eingeführt und am Ende werden die Erkenntnisse zusammengefasst. Die türkisfarbenen Kapitel sind die Anteile des Buches, in denen Konzepte vorgestellt werden, die Sie in den anderen Kapiteln brauchen. Die dunkelblauen Kapitel befassen sich damit, wie man bei eine Architekturanalyse vorgeht und Ergebnisse erzielt. Die hellblauen Kapitel enthalten meine Erfahrungen aus Architekturanalysen und -beratungen, die ich mit vielen kleinen und großen Praxisbeispielen verdeutliche.

*Verschiedene Wege*

Aus meiner Sicht wäre es natürlich am sinnvollsten, alle Kapitel des Buches von vorne nach hinten durchzulesen. Steht diese Zeit nicht zur Verfügung, dann ist es auf jeden Fall zu empfehlen, zuerst die Kapitel 1 und 2 zu lesen. Diese Kapitel schaffen die Basis. Im Anschluss kann man über Kapitel 5 und/oder 6 zu den Kapiteln 7, 8, 9 oder 10 übergehen. Wobei der Clou dieses Buches in Kapitel 5 zur kognitiven Psychologie zu finden ist – dieses Kapitel öffnet jedem, der Software entwickelt, die Augen, warum gute Strukturen im Sourcecode so wichtig sind. Man kann auch von Kapitel 2 direkt zu Kapitel 4 oder 12 springen und sich in das Vorgehen bei der Architekturanalyse oder in die Geschichten aus der Praxis vertiefen.

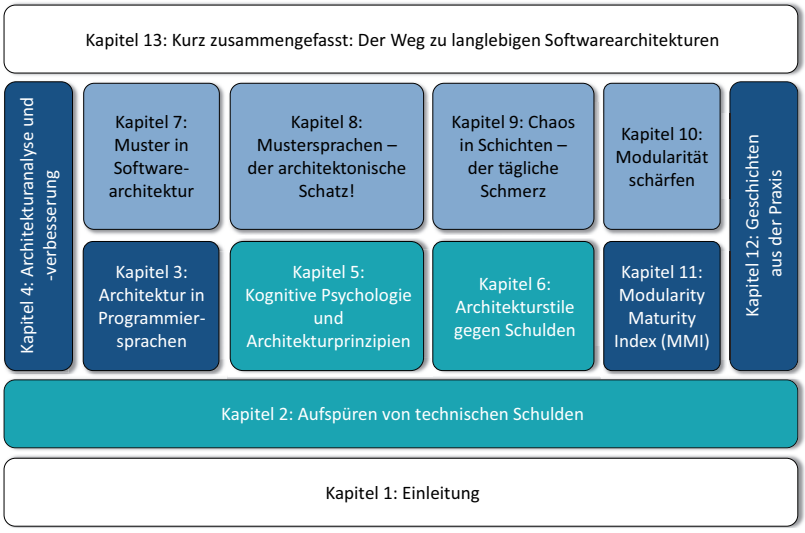



***Abb. 1–3***
*Aufbau des Buches*

Kapitel 1 legt die Basis für das Verständnis von langlebigen Architekturen und technischen Schulden.

Kapitel 2 zeigt am Beispiel, wie man technische Schulden in Architekturen findet und reduzieren kann.

In Kapitel 3 werden die Spezialitäten von Programmiersprachen bei der Architekturanalyse erläutert.

Kapitel 4 macht deutlich, welche Rollen bei der Architekturanalyse und -verbesserung wie zusammenarbeiten müssen, um zu einem wertvollen Ergebnis zu kommen, und zeigt, wie man seine Architektur dauerhaft gegen technische Schulden schützen kann.

Kapitel 5 setzt sich mit der Frage auseinander, wie große Strukturen geartet sein müssen, damit Menschen sich schnell darin zurechtfinden. Die kognitive Psychologie gibt uns Anhaltspunkte, welche Vorgaben zu Architekturen führen, die schnell zu überblicken und zu durchschauen sind.

In Kapitel 6 werden die heute üblichen Architekturstile vorgestellt. Mit ihren Regeln geben sie Leitplanken für Softwarearchitekturen vor.

Kapitel 7, 8, 9 und 10 beschreiben die Erkenntnisse aus den verschiedenen Analysen und Beratungen in der Praxis.

In Kapitel 11 werden die Ergebnisse aus den Kapiteln 7 bis 10 zusammengeführt und Architekturen mit dem Modularity Maturity Index (MMI) vergleichbar gemacht.

In Kapitel 12 finden sich schließlich beispielhafte Fallstudien von sieben aus meiner Sicht spannenden Analysen. Die Systeme in den Fallstudien sind so weit anonymisiert, dass die Systeme und die sie betreibenden Firmen nicht mehr zu erkennen sind.

Den Abschluss bildet Kapitel 13 mit einer kurzen Zusammenfassung, wie Architekten, Entwicklungsteam und Management vorgehen sollten, um die Qualität ihrer Architektur zu verbessern.

Im Anhang werden einige Analysewerkzeuge vorgestellt, die ich in meiner täglichen Praxis benutzen durfte.

# 7 Muster in Softwarearchitekturen

In der Praxis findet man viele Varianten von Architekturen und Architekturvorstellungen vor. In diesem Kapitel zu Muster in Softwarearchitekturen gehen wir den Fragen nach: Wie sollten sich die Architekturstile aus dem letzten Kapitel in der Sourcecode-Basis darstellen? Auf welche konkreten Ausprägungen stößt man in der Praxis?

## 7.1 Abbildung der Soll-Architektur auf die Ist-Architektur

*Leider nur technisch*

Meine erste Frage nach der Architektur des Systems beantworten die Architekten und Entwickler von kleineren Systemen unter 250.000 LOC in der Regel mit: »Unser System hat die folgenden Schichten: GUI, Businesslogik, Datenbank-Mapping.« Mich überrascht diese Antwort nicht. Für Systeme bis zu dieser Größe ist diese grobe technische Schichtung typisch. Aber lieber wäre mir eine andere Antwort! Nämlich eine fachliche Schichtung, wie wir im Weiteren sehen werden. Sind die Systeme größer, werden auch die Strukturierungsfragen drängender und es kommen Antworten wie: »Unser System ist in XY Komponenten aufgeteilt und hat Schichten.« Hier ist die technische Schichtung um eine fachliche Aufteilung ergänzt.

*Architektur als Muster*

Die Architekten und Entwickler machen mit diesen Aussagen deutlich, dass sie ein Muster für ihre Architektur haben. Ein Muster, das ihnen helfen könnte, sich in dem Softwaresystem zurechtzufinden – wenn das Muster denn gut eingesetzt würde.

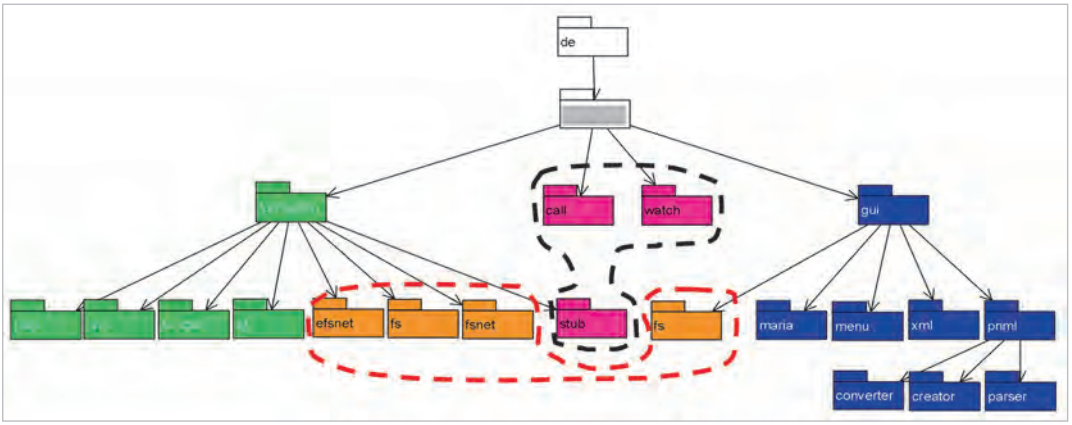
*Architektur im Sourcecode*

Normalerweise verwenden die Architekten und Entwickler das in der Sprache vorhandene Paketkonzept oberhalb von Klassen, um diese Soll-Architektur aus technischen Schichten abzubilden. Gibt es oberhalb der Pakete noch Build-Artefakte, werden auch diese für die Abbildung der Architektur verwendet. Das gelingt mal besser und mal schlechter.

*Eine schlechte Abbildung*

Ein Beispiel für einen noch verbesserungswürdigen Versuch findet sich in Abbildung 7–1. Dort sieht man einen Java-Package-Baum, der unterhalb des zweiten Wurzelknotens vier Subpackages hat. Die Pfeile gehen jeweils vom übergeordneten Paket zu seinen Kindern. Zwei dieser Subpackages – das blaue und das grüne jeweils außen – haben wiederum eine Reihe von Subpackages vorzuweisen. Der erste Schritt bei der Modellierung der Architektur bestand darin, zu klären, welche Packages in dem Package-Baum zusammengehören. Die jeweils zusammengehörenden Packages haben in Abbildung 7–1 die gleiche Farbe. Die Farbgebung offenbart sofort, dass dieser Package-Baum nur zum Teil mit der Architektur übereinstimmt. Insbesondere gibt es zwei Komponenten (lila und orange), deren Packages sich auf verschiedenen Ebenen im Package-Baum befinden. Als zusätzliche Schwierigkeit sind sie zum Teil unterhalb von Package-Knoten angesiedelt, die wiederum den Wurzelknoten anderer Komponenten (blau und grün) bilden. In Abbildung 7–1 sind diese beiden Komponenten durch die rote und schwarze gestrichelte Linie gekennzeichnet.

***Abb. 7–1***
*Soll-Architektur in einem Java-Package-Baum*




Das Muster der Architektur ist in diesem Beispiel also nur schwer im Package-Baum wiederzufinden. Die Entwickler müssen diese Anomalie bei der Navigation im Package-Baum immer im Kopf behalten. In vielen Fällen führt ein solches Nichtzusammenpassen zwischen Soll-Architektur und Package-Baum dazu, dass das Verständnis der Entwickler von der Architektur über die Zeit verschwimmt oder nur sehr vage bleibt. Hier muss also dringend der Package-Baum angepasst werden.

*Je größer, desto besser*

Auffällig ist, dass alle Systeme, die aus mehr als 1,5 Millionen LOC bestehen, eine gute Paketierung haben. Bei kleineren Systemen herrscht in der Regel mehr Chaos. Die Modellierung der Architektur ist bei kleineren Systemen infolgedessen im Verhältnis aufwendiger: Die Schichten und Komponenten setzten sich aus verschiedenen Teilbäumen zusam-

men, die zum Teil auf verschiedenen Ebenen des Package-Baums liegen. Oder es gibt Package-Knoten, die nach Aussage der Architekten Klassen aus verschiedenen Komponenten enthalten.

*Große Systeme brauchen Ordnung.*

Die Architekten der großen Systeme hingegen haben irgendwo, auf dem Weg über die 1-Millionen-Grenze, Ordnung in die Paketierung gebracht. Zu diesem Zeitpunkt mussten die Build-Artefakte oder der Package-/Directory-Baum bereinigt werden, sonst wäre das Chaos nicht mehr beherrschbar gewesen. Außerdem gaben die Architekten ihren Teams feste Regeln, wie der Baum gestaltet sein sollte. Mit diesen Regeln legten sie auch fest, wo welche Klassen untergebracht werden sollen. Eine wichtige Aufgabe sahen die Architekten dieser großen Systeme darin, diese Regeln ständig und am besten automatisiert zu überprüfen.

Wenn bei der Architekturanalyse deutlich wurde, dass die Paketierung nicht der geplanten Strukturierung entsprach, sind die folgenden Fragen zu klären: Was hindert Sie und das Entwicklungsteam daran, die Paketierung an Veränderungen in der Soll-Architektur anzupassen?

*Gründe für Chaos*

Die folgenden Argumente werden typischerweise angeführt:

- *Fehlendes Budget*

  *Keine Zeit + kein Geld*

  Eine grundlegende Restrukturierung des Systems braucht etwas Zeit. Für eine solche Veränderung, die die Funktionalität des Softwaresystems nicht erweitert, sondern »nur« seine Struktur verbessert, waren häufig keine Zeit und kein Geld vorhanden (s. Abschnitt 1.3.4).

- *Testaufwände*

  *Großes Refactoring = hohe Testaufwände*

  Für viele Java-/C#-/C++-Architekten erscheint eine Restrukturierung des Systems als ein großes Refactoring. Insbesondere dann, wenn Klassen oder Paket-Bäume zwischen Build-Artefakten verschoben werden müssen oder ganz neue Build-Artefakte geschaffen werden sollen. Diese Umstrukturierungen müssen mit einer Reihe von Komponenten-, Integrations- und Anwendertests abgesichert werden. Diesen Aufwand scheuen die Architekten oft.

  In der ABAP-Welt existiert dieses Argument nicht: Da alle Elemente im systemweiten Dictionary verzeichnet sind, hat die Paketzuordnung keine Auswirkung im Laufzeitsystem.

- *Widerspenstige Technik*

  *Veraltete Technologie*

  Teams, die restriktive und heute zum Glück veraltete Sourcecode-Management-Systeme, wie CVS, einsetzen, haben wenig Lust, Klassen zu verschieben. Solche Sourcecode-Management-Systeme löschen die Klassen beim Verschieben an ihrem Ursprungsort und fügen sie am Zielort als neue Klasse wieder hinzu. Durch das Löschen und Neuanlegen geht die Historie der verschobenen Klasse mit allen Änderungen verloren. Aus diesem Grund wurden in einigen Fallstudien

ungern Klassen in der Paketstruktur verschoben und ein Architekt ging sogar so weit zu sagen, dass der einmal entstandene Package-Baum nicht geändert werden dürfe. Zum Glück setzen die meisten Teams heute fortschrittlichere Sourcecode-Management-Systeme ein.

*Sourcecode-Struktur = Soll-Architektur*

**Soll-Architektur im Sourcecode**

Stärken Sie die Architekturmuster für Ihre Entwickler und leiten Sie sie durch den Sourcecode, indem Sie die Strukturierung in Paketen oder Projekten mit der Soll-Architektur regelmäßig in Übereinstimmung bringen. Verwenden Sie dieselben Namen und Hierarchien.

Wenn wir nun aber Geld, ein gutes Entwicklungsteam und passende technische Unterstützung haben, um die ideale Struktur in unserem System umzusetzen, wie sollte sie aussehen?

## 7.2 Die ideale Struktur: fachlich oder technisch?

*Statik und Deployment*

Für die Abbildung der Architektur schafft sich jedes Entwicklungsteam sein eigenes Schema. Alle Architekten nutzen dafür die vorhandenen Strukturierungsmechanismen ihrer Programmiersprache und Entwicklungsumgebung: also die Paketstrukturen und die sie umschließenden Build-Artefakte. Mit diesen beiden Mitteln wollen Entwickler und Architekten einerseits die statische Architektur ausdrücken – also welche Bausteine gibt es und wie sollen sie miteinander in Beziehung stehen. Andererseits bereiten sie mit der Aufteilung in Build-Artefakte auch das Deployment von einzelnen Artefakten auf verschiedene Clients und Server vor. Wenden wir uns als Erstes der statischen Architektur zu, um zu klären, ob Technik oder Fachlichkeit den Vorrang haben sollten. Im Anschluss integrieren wir dann noch die Frage des Deployments für verschiedene Arten von Architekturen (Rich Client, Webanwendung etc.).

*Das erste Kriterium*

Geht man davon aus, dass die Architekten eine technische Schichtung und eine fachliche Aufteilung in Module ausdrücken wollen, so haben wir grundsätzlich zwei Möglichkeiten: Entweder wird als erste Strukturierungsebene die technische Schichtung herangezogen und darunter die fachlichen Strukturen gebildet oder die umgekehrte Sortierung wird gewählt. Abbildung 7–2 illustriert diese beiden Varianten mit einem Java-Package-Baum.

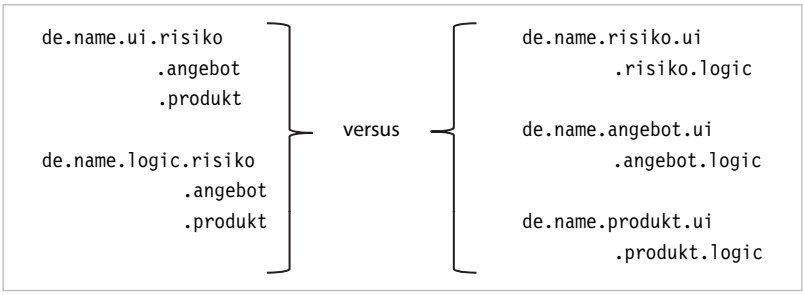



***Abb. 7–2***
*Package-Baum bei Schichtenarchitekturen*

Eine Verwendung der Build-Artefakte für diese Strukturierung ist genauso denkbar. Wir könnten in diesem Fall also entweder zwei technische Build-Artefakte UI und Logic haben oder drei fachliche Build-Artefakte Risiko, Angebot und Produkt.

*Klein = technisch*

Bei den meisten kleineren Projekten, die eine fachlich begrenzte Funktionalität umsetzen, konnte man beobachten, dass die technische Schichtung als erste Strukturierung gewählt wurde. Je größer die Systeme werden, desto mehr steigen die Architekten auf die fachliche Aufteilung als erstes Strukturierungsmittel um.

*Fachlichkeit zuerst*

Die meisten Entwickler denken schon aufgrund ihrer technischen Ausbildung in technischen Schichten. Die fachliche Aufteilung macht ihnen mehr Schwierigkeiten, weil hierfür das Anwendungsgebiet durchdrungen werden muss. Unser Projektziel ist aber eine gute Modellierung der Fachlichkeit. Ohne gutes fachliches Design wird unser System die Anforderungen der Anwender schon im ersten Wurf nicht erfüllen. Ganz zu schweigen von der Erweiterung und der Fehlerbehebung in einem fachlich schlecht modellierten System. Hat man sein System gar in technische Build-Artefakte aufgeteilt und es existieren einzelne Projekte für die Oberfläche, die Businesslogik etc., dann leidet die inhaltliche Übersichtlichkeit. Fachliche Zusammenhänge sind in einer solchen Aufteilung nur bei kleinen Systemen und sehr guter Namensgebung zu erkennen.

Die fachliche Modularisierung von Systemen sollte gestärkt werden, deshalb gilt folgende Empfehlung:

**Fachlichkeit vor Technik**

Wählen Sie die fachliche Aufteilung des Systems als wichtigstes Kriterium für die Paket- oder Artefakt-Strukturierung Ihres Systems.