Carola Lilienthal

Langlebige Softwarearchitekturen

Technische Schulden analysieren, begrenzen und abbauen

2., überarbeitete und erweiterte Auflage

dpunkt.verlag

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

Lektorat: Christa Preisendanz
Copy-Editing: Ursula Zimpfer, Herrenberg
Satz: Nadine Thiele
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie 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 978-3-86490-494-3
PDF 978-3-96088-244-2
ePub 978-3-96088-245-9
mobi 978-3-96088-246-6

2., überarbeitete und erweiterte Auflage 2017
Copyright © 2017 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 Heidelberg

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

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 übersetzt[1]. 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 6.4) komplett überarbeitet und den Abschnitt zur Mustersprache des Domain-Driven Design (Abschnitt 6.5.2) 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

@cairolali
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 55

3.5 PHP-Systeme 57

4 Architekturanalyse und -verbesserung 59

4.1 Entwickler und Architektur . . . 59
4.2 Architekturarbeit ist eine Holschuld . . . 60
4.3 Live-Workshop zur Architekturverbesserung . . . 61
4.4 Der Umgang mit den Vätern und Müttern . . . 63
4.5 Modularity Maturity Index (MMI) . . . 64
4.6 Technische Schulden im Lebenszyklus . . . 66

5 Kognitive Psychologie und Architekturprinzipien 69

5.1 Modularität . . . 70
5.1.1 Chunking . . . 70
5.1.2 Übertragung auf Entwurfsprinzipien . . . 72
5.1.2.1 Einheiten . . . 73
5.1.2.2 Schnittstellen . . . 75
5.1.2.3 Kopplung . . . 76
5.2 Musterkonsistenz . . . 77
5.2.1 Aufbau von Schemata . . . 78
5.2.2 Übertragung auf Entwurfsprinzipien . . . 80
5.3 Hierarchisierung . . . 84
5.3.1 Bildung von Hierarchien . . . 84
5.3.2 Übertragung auf Entwurfsprinzipien . . . 86
5.4 Zyklen = misslungene Modularität + Muster . . . 88
5.5 Konsequenzen für die Architekturanalyse . . . 89

6 Architekturstile gegen technische Schulden 91

6.1 Regeln von Architekturstilen . . . 91
6.2 Trennung von fachlichen und technischen Bausteinen . . . 92
6.3 Schichtenarchitekturen . . . 94
6.3.1 Technische Schichtung . . . 95
6.3.2 Fachliche Schichtung . . . 96
6.3.3 Infrastrukturschicht . . . 98
6.3.4 Integration von fachlichen Schichten . . . 100
6.4 Microservices und Domain-Driven Design . . . 101
6.5 Mustersprachen . . . 105
6.5.1 WAM-Mustersprache . . . 107
6.5.2 DDD-Mustersprache . . . 109
6.5.3 Typische Framework-Muster . . . 111
6.6 Langlebigkeit und Architekturstile . . . 112

7 Muster in Softwarearchitekturen 113
7.1 Abbildung der Soll-Architektur auf die Ist-Architektur 113
7.2 Die ideale Struktur: fachlich oder technisch? 116
7.3 Schnittstellen von Bausteinen 121
7.4 Interfaces – das architektonische Allheilmittel? 126
7.4.1 Die Basistherapie 126
7.4.2 Die Nebenwirkungen 128
7.4.3 Feldstudien am lebenden Patienten 131
7.4.4 Der Kampf mit dem Monolithen 134
7.5 Der Wunsch nach Microservices 136

8 Mustersprachen – der architektonische Schatz! 139
8.1 Die Schatzsuche 139
8.2 Die Ausgrabungsarbeiten 141
8.3 Aus der Schatztruhe 143
8.4 Den Goldanteil bestimmen 147
8.5 Jahresringe 148
8.6 Unklare Muster führen zu Zyklen 149

9 Chaos in Schichten – der tägliche Schmerz 153
9.1 Bewertung des Durcheinanders 155
9.1.1 Ausmaß der Unordnung 156
9.1.1.1 Architekturstile und Zyklen 158
9.1.1.2 Programmzeilen in Zyklen 159
9.1.1.3 Dependency Injection und Zyklen 161
9.1.2 Umfang und Verflochtenheit 161
9.1.3 Reichweite in der Architektur 164
9.2 Das große Wirrwarr 168
9.2.1 Der Schwarze-Loch-Effekt 170
9.2.2 Der Befreiungsschlag 172
9.2.3 Technische Schichtung als Waffe 174
9.2.4 Mustersprache als Leuchtturm 176
9.3 Uneven Modules 179

10 Modularität schärfen 183
10.1 Kohäsion von Bausteinen 184
10.2 Größen von Bausteinen 188
10.3 Größen von Klassen 188
10.4 Größe und Komplexität von Methoden 194

10.5 Lose Kopplung 197
10.6 Kopplung und Größe von Klassen 203
10.7 Wie modular sind Sie? 205

11 Geschichten aus der Praxis 207
11.1 Das Java-System Alpha 208
11.2 Das C#-System Gamma 216
11.3 Das C++-System Beta 224
11.4 Das Java-System Delta 233
11.5 Das Java-System Epsilon mit C#-Satelliten 240
11.5.1 Java-Epsilon 240
11.5.2 C#-Epsilon 1 248
11.5.3 C#-Epsilon 2 251
11.6 Das ABAP-System Lambda 256

12 Fazit: der Weg zu langlebigen Architekturen 263

Anhang

A Analysewerkzeuge 269
A.1 Lattix 271
A.2 Sonargraph Architect 272
A.3 Sotograph und SotoArc 274
A.4 Structure101 275

Literatur 279

Index 287

kommt auch noch eine Gutachterin und haut in die Kerbe, die dem Entwicklungsteam die ganze Zeit schon bewusst ist!

Selbstreflexiv

An dieser Stelle der Diskussion erzähle ich oft eine Geschichte, die mir der Chefentwickler des Sotographen vor Jahren mitgegeben hat: Als der Sotograph einen ersten lauffähigen Stand erreicht hatte, hat der Chefentwickler den Source- und Bytecode des Sotographen mit dem Sotographen analysiert. Seine Erwartung war, dass die Architektur sehr gut aussieht. Er ging davon aus, dass keine roten Bögen existieren würden und alles wunderbar gut strukturiert wäre. Schließlich baut sein Entwicklungsteam eine Software zur Analyse von Strukturen im Sourcecode. Leider musste der Chefentwickler feststellen, dass es eine Reihe roter Bögen und schlechter Strukturen gab. Das lag nicht daran, weil er ein schlechtes Entwicklungsteam hatte. Die Ursache war vielmehr, dass seine Entwickler ihr System noch nie von dieser Flughöhe aus betrachtet hatten. Sie konnten nicht sehen, wo sie die Architektur verletzen. Und selbst wenn, gab es Stellen, wo einfach Flüchtigkeitsfehler passiert waren. Diese Geschichte entspannt die Situation bei Analysen oft ein wenig.

Fingerspitzengefühl

Der Rest ist ein Abwägen und Austarieren, sodass ich kritische Punkte ansprechen kann und trotzdem die Offenheit bei der Analyse nicht verloren geht. Je nach Setting gelingt das mal besser und mal schlechter. Wenn mich ein IT-Chef einlädt, damit sein Team eine Gelegenheit bekommt, an seiner Architektur zu arbeiten und es weniger um eine externe Bewertung geht, ist die Lage meistens entspannter. Als Gutachter wandert man dagegen immer auf einem schmalen Grat zwischen zu harter Beurteilung und zu freundlicher Beschönigung. Das ist kein einfaches Unterfangen. Deshalb möchte ich an dieser Stelle alle Systemeltern um Entschuldigung bitten, bei denen ich das notwendige Fingerspitzengefühl habe vermissen lassen.

4.5 Modularity Maturity Index (MMI)

Standardisiertes Verfahren

In den letzten zwei Jahren haben mich immer wieder Kunden gefragt, wie ihr System denn im Verhältnis zu anderen Systemen abschneiden würde. Dadurch dass das Vorgehen bei den Analysen seit 2008 standardisiert ist (s. Kap. 4), ist eine Reihe vergleichbarer Werte und Bewertungen von ca. 150 Systemen vorhanden. Auf dieser Basis war es möglich, ein einheitliches Bewertungsschema zu erstellen, um den Zustand von Systemen in Relation zu setzen: den Modularity Maturity Index (MMI). Den Namen Modularity Maturity Index haben wir gewählt, weil für Langlebigkeit der Aspekt der Modularität auf allen Ebenen die entscheidende Rolle spielt (s. Abschnitt 5.4).

MMI von 1 bis 10

Der Zustand der Systeme wird durch den MMI auf eine Skala von 0 bis 10 abgebildet (Y-Achse). In Abbildung 4–2 ist eine Auswahl von 22 Softwaresystemen dargestellt, die in einem Zeitraum von fünf Jahren analysiert wurden. Die Größe der Systeme in Lines of Code (LOC) ist jeweils unterhalb der Punkte angegeben. Bei System 2 handelt es sich um ein 14,7 Millionen LOC großes Java-System. System 16 ist ein knapp 10 Millionen LOC großes C#-System. Insgesamt finden sich in Abbildung 4–2 Java-, C#-, C++- und PHP-Systeme.

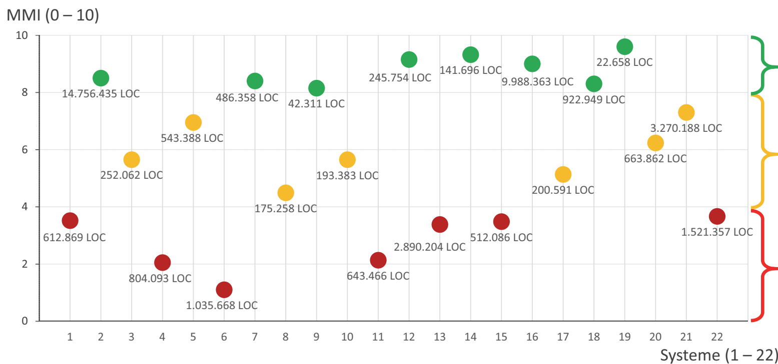


Abb. 4–2
MMI für 22 verschiedene Systeme

Grüner und gelber Bereich

Liegt der MMI eines Systems zwischen 8 und 10, also im grünen Bereich, so ist der Anteil an technischen Schulden gering. Das System befindet sich im Korridor des »Gleichbleibenden Aufwands für Wartung« aus Abbildung 1–2 in Abschnitt 1.3. Systeme mit einem MMI zwischen 4 und 8 haben bereits einiges an technischen Schulden angesammelt. Architekturerosion verlangsamt die Wartungs- und Entwicklungsgeschwindigkeit immer mehr und es sind Refactorings notwendig, um in den grünen Bereich zurück zu gelangen. In diesem Fall sollte das Team zusammen mit dem Reviewer Refactorings definieren und priorisieren, die die technischen Schulden reduzieren. Schritt für Schritt müssen diese Refactorings in die Wartung oder auch Erweiterung des Systems eingeplant und die Ergebnisse regelmäßig überprüft werden. So kann ein System schrittweise in den Bereich »Gleichbleibender Aufwand für Wartung« überführt werden.

Roter Bereich

Systeme mit einem MMI unterhalb der Marke 4 sind nur mit sehr viel Aufwand zu warten und zu erweitern. Sie sind im Korridor »Hoher, unplanbarer Aufwand für Wartung« aus Abbildung 1–2 in Abschnitt 1.3 angelangt. Bei diesen Systemen muss sehr genau abgewogen werden, ob sich ein Erneuern durch Refactorings lohnt oder ob das System ersetzt werden sollte. Vor einiger Zeit durften wir beispielsweise das System mit der Nummer 13 (2,8 Millionen LOC) analysie-

ren. Dieses System war sowohl in der Wartung als auch in der Erweiterung sehr teuer. Die Analyse konnte den schlechten Eindruck, den dieses System in den letzten Jahren hinterlassen hatte, mit eindeutigen Fakten aus dem Inneren des Systems belegen. Mit einem MMI von 3,5 war das System im roten Bereich angekommen. Das die Software einsetzende Unternehmen hatte sich bereits vor einiger Zeit nach Ersatz umgesehen und ein Standardprodukt ausfindig gemacht. Um hier nicht in dieselbe Falle der ständig steigenden Wartungskosten zu tappen, wurde dieses Standardprodukt ebenfalls untersucht. Dabei handelt es sich um System 18 aus Abbildung 4–2. Mit einem MMI von 8,6 ist dieses System in einem guten Zustand und das in der Analyse identifizierte Verbesserungspotenzial lässt sich mit vertretbaren Kosten umsetzen.

MMI misst Langlebigkeit.

In den Kapiteln 7–10 werden die Aspekte beschrieben, die in den MMI einfließen. Einerseits sind es tatsächlich messbare Werte, wie Größenverhältnisse und Kopplung zwischen Bausteinen. Auf der anderen Seite fließen qualitative Bewertungen durch die Reviewer z. B. zu dem Schnitt der Bausteine, ihren Schnittstellen und ihren Namen ein. Bisher hat sich der Eindruck der untersuchten Systeme während des Live-Workshops immer im MMI widergespiegelt. Wir sind also zuversichtlich, mit diesem Index ein Mittel der Vergleichbarkeit für Langlebigkeit gefunden zu haben.

4.6 Technische Schulden im Lebenszyklus

Dauerhafter Schutz

Will man seine Architektur dauerhaft gegen technische Schulden schützen, so sollte man von Anfang an die folgenden Maßnahmen einsetzen:

Architekturdiskussion

- *Architekturdiskussion im Team*
  Während der laufenden Iteration sammeln die Teammitglieder Architekturthemen, die sie gerne diskutieren möchten. An einem regelmäßigen Termin zur Architekturretrospektive werden diese Themen und weitere Punkte, die den erfahreneren Entwicklern und Architekten wichtig sind, diskutiert.

Monitoring

- *Monitoring*
  Verschiedene Kennzahlen werden automatisiert mit passenden Metrikwerkzeugen erhoben (s. Anhang A), sodass das Team Abweichungen und Anomalien frühzeitig erkennen und dagegensteuern kann, bevor die anfänglich noch kleinen Probleme zu großen Geschwüren werden.

- *Pair Programming*
  Programmiert wird grundsätzlich nur in Zweierteams, die regelmäßig gewechselt werden. Auf diese Weise wird das Wissen über die Architektur im Team verteilt und gleichzeitig diskutiert.

Pair Programming

- *Mob Programming*
  Wichtige Refactorings werden vom gesamten Team an einem großen Bildschirm mit einer Person an Maus und Tastatur gemeinsam durchgeführt. So wird das Verständnis für diese Refactorings an alle Mitglieder des Teams weitergegeben.

Mob Programming

- *Mob Architecting*
  In Analogie zum Mob Programming bezeichne ich die Live-Workshops zur Architekturverbesserung (s. Abschnitt 4.3) oft als Mob Architecting. Das ganze Team sitzt in regelmäßigen Abständen zusammen und überarbeitet zuerst mit externer Hilfe und später auch alleine mithilfe eines passenden Tools seine Architektur. So stellt sich das Team der sich ansonsten unbemerkt einschleichenden Architekturerosion entgegen (s. Abschnitt 1.3.3).

Mob Architecting

In einigen Firmen sind diese Prozesse bereits etabliert. Für das tägliche Monitoring werden Metrikwerkzeuge eingesetzt und es gibt regelmäßige Architekturdiskussionen im Team. Andere Firmen haben bereits eine Codebasis, für die sie sich in der Zukunft eine längerfristige Begleitung bei der Architekturverbesserung wünschen.

Regelmäßige Analyse

Eine solche längerfristige Begleitung besteht aus regelmäßigen Architekturanalysen und für gewöhnlich aus begleitender Architekturberatung. Bei einer solchen Begleitung durchlaufe ich mit den Kunden in der Regel die folgenden drei Phasen der Architekturanalyse (s. Abb. 4–3): Aufräumen (Phase 1), Verbessern (Phase 2), Erhalten (Phase 3).

Abb. 4–3
Die drei Phasen bei der Architekturanalyse

Aufräumen

In der ersten Phase des Aufräumens finden wir Abweichungen zwischen Soll- und Ist-Architektur, die für das Entwicklungsteam manchmal überraschend sind. Diese Abweichungen müssen repariert werden, damit der Sourcecode überhaupt zur Soll-Architektur passt. Teil dieser Phase ist aber auch, die Soll-Architektur zu präzisieren und sie um feh-

lende Architekturkonzepte zu ergänzen. Hier diskutieren wir viel über verschiedene Architekturstile und Designprinzipien, die für das jeweilige System sinnvoll sein könnten (s. Kap. 6). Für diese Phase des Aufräumens benötigt man üblicherweise einen 2-Tage-Workshop. Für Systeme bis 500.000 LOC ist das eine gute Größenordnung. Hat man es mit einem größeren System zu tun, sollte man einen längeren Workshop von drei bis fünf Tagen einplanen. Systeme unter 100.000 LOC lassen sich auch an einem Tag aufräumen. Die Tiefe der Analyse hängt dabei sehr von der vorhandenen Architektur und dem Architekturverständnis der Teilnehmer ab. Außerdem hat die Anwesenheit der Systemeltern Einfluss auf die Geschwindigkeit, mit der wir bei der Analyse vorankommen.

Verbessern

Die zweite Phase des Verbesserns ist die Phase, in der Architekten und Entwicklungsteam den meisten Spaß haben. In dieser Phase ist die Architektur bereits aufgeräumt, man hat die ersten Diskussionen zur Architektur geführt und alle zusammen haben die Verbesserung als Ziel. Diese Phase steht und fällt mit den zeitlichen Freiräumen der Architekten und Entwickler. Sitzen sie unter Zeitdruck in solchen Workshops ist nur wenig zu erreichen. Sehr beeindruckt haben mich hier die Eclipse-Entwicklungsteams. Für diese Teams war es Usus, dass nach einem Release vier Wochen lang keine neuen Features gebaut wurden. Aufgabe der Teams war es in dieser Zeit, den Sourcecode aufzuräumen und die Architektur zu verbessern.

Erhalten

Schließlich folgt die dritte Phase, in der die Architektur erhalten werden muss, damit das System so lange wartbar und weiterentwickelbar bleibt, bis eine Neuentwicklung sinnvoll wird. Diese Phase können Entwicklungsteams in der Regel alleine bewältigen, wenn sie die entsprechenden Analysewerkzeuge und die zeitlichen Ressourcen zur Verfügung haben. Als Architekt oder Projektleiter sollte man in dieser Phase aufpassen, wenn eine größere Änderung des Softwaresystems ansteht. Größere Änderungen haben oft auch Effekte auf die Architektur des Systems. Ein Zurückgehen in Phase 2 (Verbessern) ist an dieser Stelle oft sinnvoll, damit die Architektur nach der Änderung immer noch tragfähig bleibt.

Verbündete finden

Am meisten freue ich mich bei den Live-Workshops, wenn – meistens am zweiten Tag – einer der Teilnehmer zum Rechner kommt und sagt: »Darf ich auch mal?« Voller Freude gebe ich ihm die Maus und lasse ihn seine Architektur anfassen und verändern. Wenn das passiert, weiß ich: Ich habe einen Verbündeten im Unternehmen gewonnen. Hier ist jemand, der sich für die Architekturanalyse und -verbesserung interessiert und gerne selbst Hand anlegen will. Jetzt kann langfristige Architekturverbesserung gelingen!

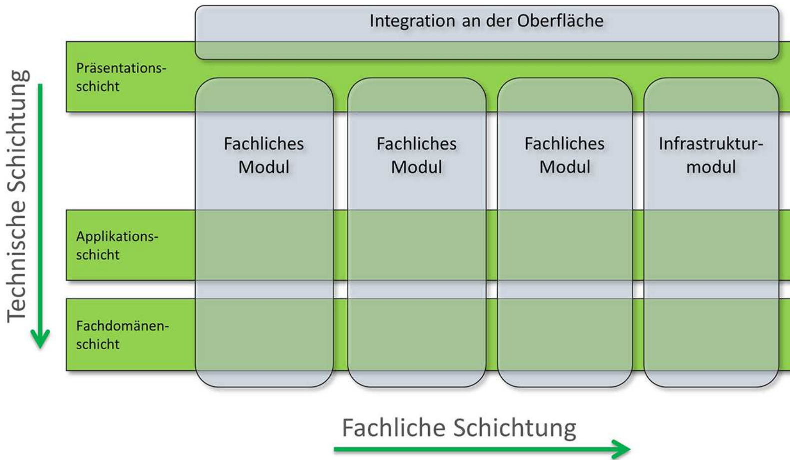


Abb. 6–7
Schmale Integration von fachlichen Modulen

Diese schmale Integration von fachlichen Modulen bildet bei Webanwendungen, die nach dem Prinzip der Microservices entwickelt sind, eine typische Lösung.

6.4 Microservices und Domain-Driven Design

Unabhängige Services

Microservices sind in den letzten Jahren als neuer Architekturstil aufgekommen. Viele Entwickler und Architekten dachten, es ginge bei diesem Architekturstil nur darum, Softwaresysteme in voneinander unabhängig deploybare Services aufzuteilen und die Entwicklung durch eine synchrone Teamstruktur zu beschleunigen. Dann wären sie für dieses Buch nicht weiter interessant gewesen.

Bounded Context → Microservice

Durch die Kombination von Microservices mit dem strategischen Design aus Domain-Driven Design (DDD) von Eric Evans[7] wird der Fokus beim Zerteilen eines Softwaresystems in Microservices aber auf die Fachlichkeit gelegt. Das ist eine deutliche und aus meiner Sicht sehr wichtige Entwicklung hin zu einer fachlichen Struktur in der Software (s. Abschnitt 6.3.2). Das strategische Design von Eric Evans gibt Entwicklungsteams und Business-Analysten eine Anleitung, wie sie die Domäne ihres Softwaresystems in Teilbereiche, die sogenannten Bounded Contexts, zerlegen können. Für jeden Bounded Context wird ein Microservice entwickelt und so die fachliche Grobstruktur einer Fachdomäne direkt in der Struktur des Softwaresystems abgebildet.

Ubiquitous Language im Microservice

Innerhalb eines Bounded Context sind alle Begriffe der Fachsprache – oder wie Eric Evans sagt: der Ubiquitous Language – eindeutig und klar definiert. Einerseits gibt es in jeder Domäne Begriffe, die man

7. s. [Evans 2004], [Vernon 2017].

eindeutig einem Bounded Context zuordnen kann. Andererseits kann derselbe Begriff aber mit einer jeweils etwas anderen Definition in zwei Bounded Contexts existieren. Das liegt einfach daran, dass zwei Abteilungen in einer Firma unterschiedlich über denselben Begriff und das Konzept, was er widerspiegelt, nachdenken. Genau diese semantische Verschiebung von Begriffen zwischen Kontexten wird bei der Implementierung der Fachsprache in den verschiedenen Microservices übernommen. Deshalb ist zu erwarten, dass eine Klasse, die einen fachlichen Begriff aus einem Bounded Context in Microservice A abbildet, unter gleichem Namen auch in Microservice B existiert, aber anders implementiert ist. Beispielsweise wird die Klasse `Konto` im Microservice *Kontoführung* anders implementiert sein als die Klasse `Konto` im Microservice *Kredit*. Beide Klassen haben sicherlich ähnliche Anteile, wie Kontonummer und das Ein- bzw. Auszahlen von Beträgen, während es andere Funktionalitäten, wie den Überziehungsrahmen beim Girokonto oder die Ratenzahlung beim Kredit, jeweils nur in einer der beiden Implementierungen gibt.

Microservices + Modularität

Schneiden Entwicklungsteams ihre Microservices anhand der Bounded Contexts in ihrer Fachdomäne, so entsteht eine Architektur mit einem starken Fokus auf Modularität (s. Abb. 6–8). Und zwar auf Modularität, die aus den fachlichen Anforderungen erwächst. Die Entwickler und Architekten versuchen Microservices zu entwerfen, die als fachlich zusammenhängende Einheiten fürs Chunking dienen können. Dabei wird ein sehr großer Wert auf eine minimale Schnittstelle und lose Kopplung gelegt. Fowler schreibt in seinem Artikel zu Microservices: »Ziel von Anwendungen im Microservice-Stil ist es, so entkoppelt und in sich kohäsiv wie möglich zu sein« (s. [Fowler & Lewis 2014/2015]). Für die Modularität und die fachliche Zerlegung ist dieser Architekturstil also ein absoluter Gewinn.

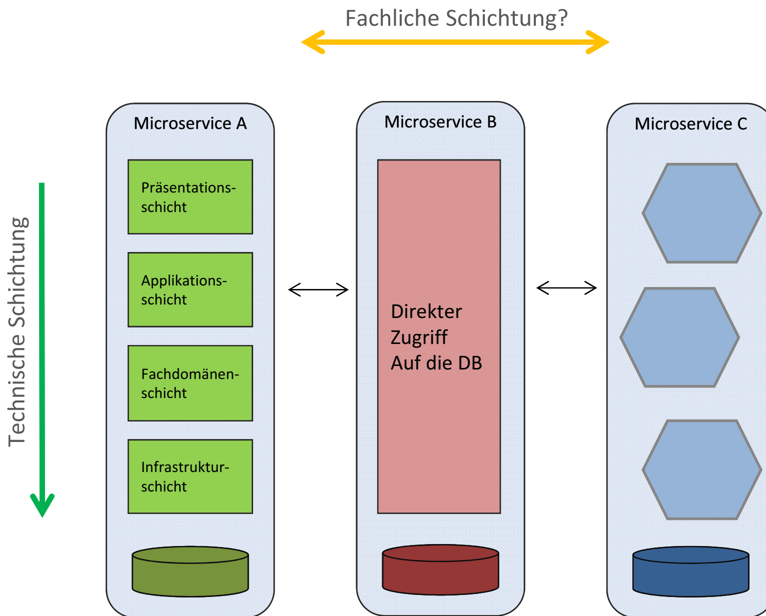


Abb. 6–8
Microservices mit unterschiedlichen Implementierungen

Kommunikation

Wie in Abbildung 6–8 durch die Doppelpfeile bereits angedeutet wird, ist bei der Konstruktion mit Microservices neben dem fachlichen Schneiden außerdem noch zu klären, wie die Kommunikation zwischen den Microservices umgesetzt wird. Fowler & Lewis sagen dazu, dass die Kommunikation zwischen den Services so schlank wie möglich erfolgen soll und die Schnittstellen der Microservices explizit beschrieben werden müssen. Die Logik darf nicht in der Verbindung zwischen den Microservices, sondern in den Microservices selbst implementiert sein (s. [Fowler & Lewis 2014/2015]).

Beachtet man diese Vorgabe nicht, so machen Fowler & Lewis folgende Beobachtung: »Man denkt schnell, dass alles in Ordnung ist, wenn man sich die Innenseite einer Komponente ansieht, übersieht dabei aber die chaotischen Verbindungen zwischen den Services.«

Schnittstellen + Context Mapping

Um dieses Chaos an den Verbindungen zwischen den Services, also den Schnittstellen, konzeptionell zu erfassen, schlägt das strategische Design von DDD sieben verschiedene Möglichkeiten vor. Eric Evans bezeichnet die Zusammenarbeit zwischen Bounded Contexts bzw. die Schnittstellen zwischen den Microservices als Context Mapping. Je enger die Zusammenarbeit, desto mehr müssen sich die Teams, die für die Microservices bzw. Bounded Contexts zuständig sind, absprechen. Die von Evans beschriebenen Arten des Context Mapping sind in der Reihenfolge von der engsten Zusammenarbeit bis hin zu gar keiner Zusammenarbeit die folgenden:

- *Shared Kernel*
  Microservices teilen sich einen Teil ihres Modells, d.h., es gibt einige Klassen, die in beiden Microservices gleich sind und nur gemeinsam geändert werden können. Die Teams, die für diese Bounded Contexts zuständig sind, müssen sich sehr viel absprechen.
- *Customer/Supplier*
  Der Supplier-Microservice stellt dem Customer-Microservice bestimmte Dienste zur Verfügung. Diese Dienste werden zwischen Customer-Team und Supplier-Team abgesprochen.
- *Published Language*
  Eine Published Language ist eine wohldokumentierte Sprache zum Informationsaustausch, die eine einfache Verwendung und Übersetzung von einer beliebigen Anzahl von Microservices ermöglicht. Alle, die die Published Language entwickeln und weiterentwickeln wollen, müssen sich absprechen.
- *Open-Host-Service*
  Ein Microservice bietet seine Dienste als Schnittstelle an, die von anderen Microservices verwendet werden kann. Oft bietet ein Open-Host-Service an seiner Schnittstelle eine Published Language an.
- *Anticorruption Layer*
  Ein Microservice schützt sich und sein Modell vor einem Altsystem durch einen Layer, sodass Änderungen am eigenen Modell möglich sind und bei Änderungen am Altsystem nur der Anticorruption Layer geändert werden muss.
- *Conformist*
  Ein Microservice übernimmt das Modell eines anderen Microservice eins zu eins, weil kein Geld vorhanden ist oder weil klar ist, dass der andere Microservice das perfekte Modell hat.
- *Separate Ways*
  Die Microservices haben keine Verbindung. Jeder hat sein eigenes Modell und es werden keine Daten zwischen den Microservices ausgetauscht.

Zyklenfreiheit + Microservices

Eine zyklenfreie Ordnung wird durch diese Arten des Context Mapping nicht hergestellt. Auch mit den in den neueren Versionen von DDD eingeführten Domain Events wird dieses Problem nicht gelöst. Domain Events entkoppeln sicherlich stärker als REST- oder RPC-Aufrufe. Aber auch Domain Events führen nicht automatisch zu einer geordneten Kommunikation. Jede Organisation, die Microservices in Teams entwickelt, muss sich deshalb intensiv Gedanken machen, wie die Kommunikation zwischen den Microservices geordnet werden soll.

den. Wie die Schnittstellen der einzelnen Musterelemente aussehen müssen, wird mithilfe von Elementregeln festgelegt[12].

6.5.2 DDD-Mustersprache

Eric Evans stellt in seinem Buch *Domain-Driven Design* (s. [Evans 2004]) einerseits das strategische Design vor, um Microservices anhand von Bounded Contexts zu schneiden und zusammenarbeiten zu lassen (s. Abschnitt 6.4). Auf der anderen Seite entwickelt er mit dem taktischen Design eine Mustersprache, um Vorgaben für die Konstruktion innerhalb von Bounded Contexts bzw. Microservices zu machen. Die Schichtenarchitektur von Domain Driven Design (DDD) entspricht den vier Schichten, die in Abschnitt 6.3.1 bei der technischen Schichtung vorgestellt wurden.

Elemente der DDD-Mustersprache

Die DDD-Mustersprache gibt Muster für die Applikations-, die Fachdomänen- und die Infrastrukturschicht vor (s. Abb. 6–10):

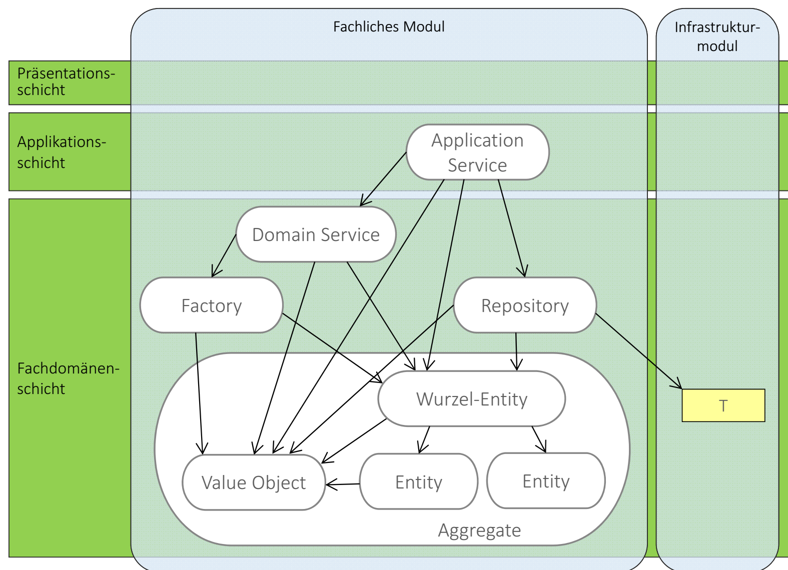
- **Value Objects** besitzen keine eigene Identität, beschreiben den Zustand anderer Objekte und können aus anderen Value Objects bestehen, niemals aber aus Entitäten. Dieses Muster entspricht den Fachwerten im WAM-Ansatz.
- **Entities** sind Kernobjekte einer Fachdomäne mit unveränderlicher Identität und einem klar definierten Lebenszyklus. Entitäten bilden ihren Zustand mit Value Objects ab und sind praktisch immer persistent. Hier finden wir die Materialien des WAM-Ansatzes wieder.
- **Aggegrates** kapseln vernetzte Entities und ihre Value Objects, besitzen grundsätzlich eine einzige Entität als Einstiegspunkt (Wurzel) und werden hinsichtlich Änderungen als Einheit betrachtet. Aggregates stellen über die Wurzel-Entity die fachliche Konsistenz und Integrität ihrer enthaltenen Entities sicher und sind komplexe Materialien im Sinne des WAM-Ansatzes.
- **Domain Services** stellen Abläufe oder Prozesse der Domäne dar, die nicht von Entitäten ausgeführt werden können. Services sind zustandslos und die Parameter und Ergebnisse ihrer Operationen sind Entities und Value Objects. Im WAM-Ansatz heißt dieses Musterelement fachlicher Service.
- **Application Services** bilden die Kapsel für ihr fachliches Modul und bieten nach außen eine Schnittstelle an, über die die fachliche Funktionalität aufgerufen werden kann, die in den Domain Ser-

12. s. [Becker-Pechau et al. 2006], [Knodel & Popescu 2007], [Lilienthal 2007/2008] und [Scharping 2006].

vices, Entities und Aggregates implementiert ist. Application Services enthalten keine fachliche Funktionalität. Im WAM-Ansatz existiert ein solches Muster nicht.

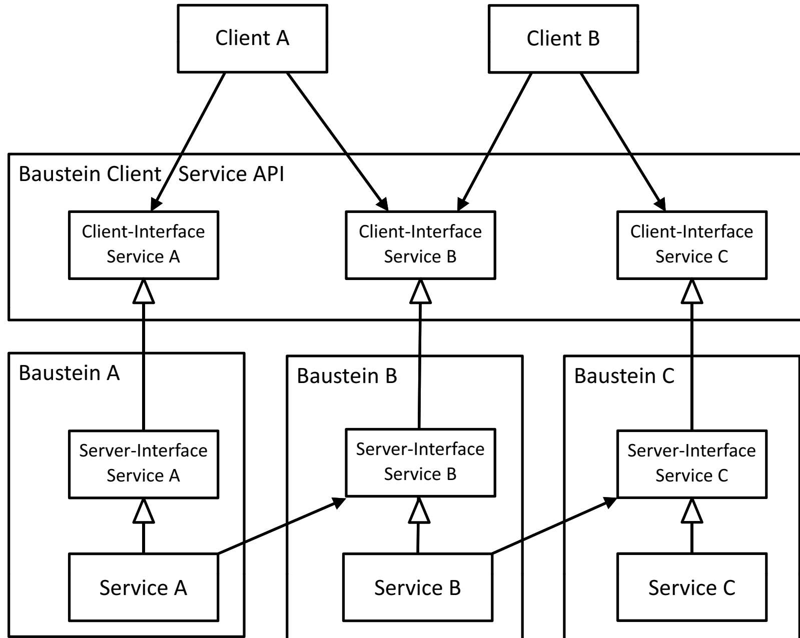
- **Factories** kapseln die Erzeugung von Aggregates, Entities und Value Objects. Factories arbeiten ausschließlich innerhalb der Domäne und haben keinen Zugriff auf die technischen Bausteine. Ein solches Muster gibt es im WAM-Ansatz nicht.
- **Repositories** kapseln die technischen Details (»T«) der Infrastrukturschicht (technisch) gegenüber der Domänenschicht (fachlich). Repositories beschaffen Objektreferenzen von Entities, die aus Datenbanken gelesen werden müssen. Die Entsprechung hierzu ist im WAM-Ansatz der technische Service.

Abb. 6–10
DDD-Architektur



Die Pfeile in Abbildung 6–10 stehen für die Benutzt-Beziehungen, die in der DDD-Architektur zwischen den Musterelementen erlaubt sind. Das Aggregate ist eine Hülle für ein oder mehrere Entities. Im Sourcecode findet man das Aggregate nicht als eigene Klasse, sondern die Wurzel-Entity wahrt die fachliche Konsistenz. Die Architekturelemente und die Regeln zu ihren Beziehungen müssen in jedem fachlichen Modul einer Architektur verwendet werden, um den Designraum einzuschränken. Dadurch entsteht eine einheitliche Struktur für alle fachlichen Module.

Abb. 7–22
Aufteilung der Bausteine in einer langlebigen Architektur



7.5 Der Wunsch nach Microservices

Microservices müssen sein.

Einige Unternehmen sind in den vergangenen zwei Jahren mit der Bitte an mich herangetreten, ihr System daraufhin zu untersuchen, wie man es in Microservices zerlegen kann. Solche Anfragen sind hochspannend, weil in vielen Unternehmen nur eine verschwommene Vorstellung davon existiert, was eine Aufteilung des Systems in Microservices bedeutet. Zu Beginn einer Analyse gilt es in diesem Fall also zu klären, was Microservices im Prinzip sind und welchen Nutzen das Unternehmen von der Einführung von Microservices haben kann und will.

Die aktuellen Hype-Themen

Ziele, die mir genannt wurden, sind häufig bessere Skalierbarkeit und eine diffuse Idee von einer besseren Architektur durch Microservices, weil das ja alle Welt gerade macht. Wenn dann noch Themen wie DevOps und automatisierte Deployment-Pipeline hinzukommen, dann haben wir den gesamten aktuellen Hype beisammen. Das sind alles gute und wichtige Themen! Keine Frage. Aber da es mir in meinen Analysen um die Verständlichkeit der Architektur geht, wende mich an dieser Stelle erst einmal dem Sourcecode und seiner Struktur zu.

Wo ist die fachliche Struktur?

Der Architekturstil »Microservices« führt zu einer fachlichen Aufteilung des Systems (s. Abschnitt 6.4), also zu einer Strukturierung des Systems nach fachlichen Kriterien, die sehr gut zu dem Merksatz »Fachlichkeit vor Technik« aus Abschnitt 7.2 passt. Insofern ist das

ein absolut zu begrüßender Wunsch. In allen fünf Analysen, in denen es bisher um Microservices ging, waren die Voraussetzungen für eine simple fachliche Aufteilung leider nicht gegeben. Die Systeme hatten meistens eine gute technische Schichtung mit wenigen Verletzungen. In der fachlichen Dimension wurde ich aber mit einem Big Ball of Mud konfrontiert.

Ein kleines, wachsendes System

In Abbildung 7–23 sieht man die fachliche Aufteilung eines Systems zu der Bestellung von Waren. Die Software ist in C# geschrieben und hat nach 9 Monaten Entwicklungszeit ca. 100.000 LOC. Das System ist also noch relativ klein, soll aber in absehbarer Zeit um die Bestellung weiterer Warenkategorien mit spezifischen Bestellprozessen wachsen. Damit das System die damit einhergehende größere Last an Anfragen performant beantworten kann, wäre eine Aufteilung in Microservices sinnvoll.

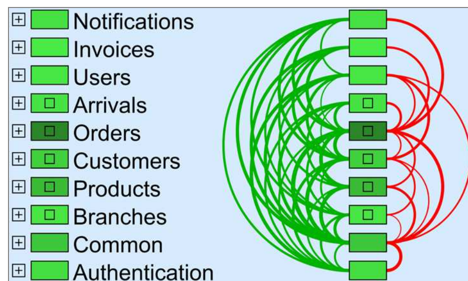
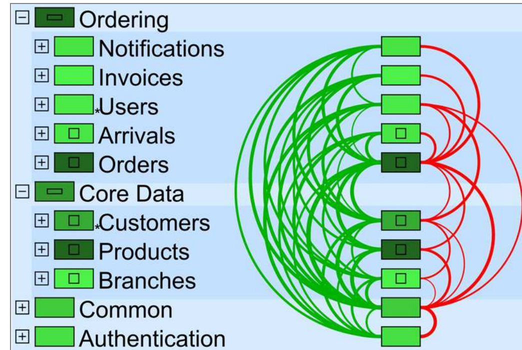


Abb. 7–23
Erste fachliche Aufteilung

Technische Schichtung gut gelungen, aber …

Die fachliche Struktur in Abbildung 7–23 war in den Namespaces und Directories des Systems nicht vorhanden. Dort gab es als erste Ordnung die technische Schichtung – ein übliches Phänomen bei Systemen dieser Größe (s. Abschnitt 7.2). Ein Großteil der Architekturarbeit bei der Analyse dieses Systems haben das Team und wir deshalb mit der Erarbeitung einer sinnvollen fachlichen Struktur verbracht. Das Ergebnis in Abbildung 7–23 ist ein erster Anfang mit 25.804 Beziehungen auf der linken (grün) und 587 Beziehungen auf der rechten Seite (rot). Ein möglicher Schnitt wäre, *Orders* und alles, was darüber liegt, zu einem Microservice »Ordering« zusammenzufassen. Als zweiten fachlichen Microservice »Core Data« bieten sich Customers, Products und Branches an (s. Abb. 7–24).

Abb. 7–24 Aufteilung in zwei Microservices



Die notwendigen Schritte

Bevor diese beiden Microservices allerdings als solche bezeichnet und dann auch getrennt deployed werden könnten, müsste sich das Team den folgenden Aufgaben widmen:

- Schnittstelle zwischen den beiden Microservices *Ordering* und *Core Data* bestimmen. Welche Art von Context Mapping soll hier gelten (s. Abschnitt 6.4)?
- Schnittstelle zwischen *Ordering* und *Core Data* verschlanken, sodass eine Kommunikation über Prozessgrenzen performant erfolgen kann.
- Klären, wie die beiden Komponenten *Common* und *Authentication* auseinandergenommen und jeweils *Ordering* oder *Core Data* hinzugefügt werden können. Möglicherweise sind Teile davon eher ein Shared Kernel (s. Abschnitt 6.4), sodass sie in beide Microservices kopiert werden müssen.

Aufwendige Refactorings

All diese Aufgaben werden zu einer Reihe von Refactorings führen und sind aller Voraussicht nach zeitaufwendig. Selbst bei einem so jungen und noch relativ kleinen System wäre es daher besser gewesen, rechtzeitig über die fachlichen Grobstrukturen nachzudenken.

Microservices vorausplanen

Sobald Sie voraussehen können, dass Ihr System in Microservices aufgeteilt werden muss, beginnen Sie damit, das System fachlich zu strukturieren und die Schnittstellen für die einzelnen Microservices festzulegen. Je später Sie damit anfangen, desto aufwendiger wird es.

Hat man viele Jahresringe aus verschiedenen Generationen von Mustersprachen, so baut das System ebenfalls technische Schulden auf. Ein guter Indikator dafür sind Muster, die ähnliche Dinge tun und die von unterschiedlichen Entwicklern erklärt werden müssen. Oder Aussagen wie: »In dem Teil kennen sich X und Y aus. In dem Teil nur Z.« Das Problem mit Jahresringen ist, dass man bei Erweiterungen oder Wartung jeweils die Muster aus dem Jahresring nachvollziehen muss, an dem man gerade Anpassungen macht.

Jahresringe aus Mustern vermeiden

Gehen Sie sehr vorsichtig mit der Erweiterung Ihrer Mustersprache um. Achten Sie darauf, dass die Muster einheitlich und durchgängig eingesetzt werden.

Neue Muster dürfen nur nach einer ausführlichen Diskussion im Entwicklungsteam in die Mustersprache aufgenommen werden.

8.6 Unklare Muster führen zu Zyklen

Mühe mit Mustern

Gibt sich ein Team keine Mühe mit seinen Mustern oder ist es sich nicht bewusst, dass Muster der Struktur des Systems guttun, dann gehen die Muster mit der Zeit kaputt und es werden unklare Muster eingeführt. In Abbildung 8–5 sieht man 22 Klassen, die in einem Klassenzyklus zusammenhängen. Keine der Klassen kann ohne Refactoring aus dem Zyklus entfernt werden, weil sie von einer anderen Klasse im Zyklus verwendet wird.

Sternzyklen

Auf der linken Seite von Abbildung 8–5 sieht man eine sehr typische zyklische Struktur in alten Java-Systemen: einen Sternzyklus. Um den `PluginManager` ist eine Reihe von `Command`-Klassen gruppiert, die den `PluginManager` alle selbst kennen und die vom `PluginManager` gekannt werden. Also eine direkte bidirektionale Beziehung. Dieser Teil des Zyklus wäre kein Problem, wenn er lokal und für sich alleine im System vorkommen würde. Leider hängen andere Klassen an diesem Zyklus dran und erzeugen so ein viel größeres undurchsichtigeres Gebilde als einen Sternzyklus. Wir schauen uns zunächst die Fälle an, in denen solche Sternzyklen üblicherweise vorkommen, und diskutieren anschließend das eigentliche Problem in dem Klassenzyklus von Abbildung 8–5.

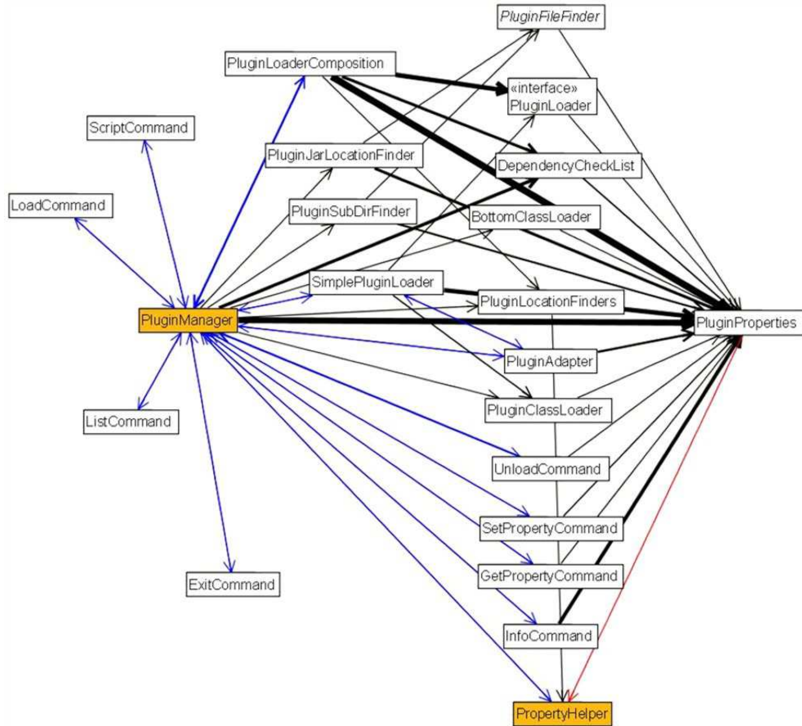


Abb. 8–5
Kaputte Muster

Sternzyklen, wie der in Abbildung 8–5 um den `PluginManager` herum, folgen in der Regel zwei häufig anzutreffenden Mustern:

Sternzyklus um Factories

- Alle Blattklassen im Sternzyklus werden von einer zentralen Klasse, einer Factory oder einem Manager, erzeugt und für andere Klassen zur Verfügung gestellt. Gleichzeitig benutzen die Blattklassen die Factory oder den Manager, um sich Zugang zu ihren Schwesterklassen zu verschaffen. Besser wäre es an dieser Stelle, wenn die Factory bzw. der Manager den Blattklassen bei ihrer Erzeugung die benötigten anderen Klassen per Dependency Injection zur Verfügung stellen würde. Dann bräuchten die Blattklassen die Factory oder den Manager nicht zu kennen und könnten ggf. auch auf anderem Wege erzeugt werden.

Sternzyklus um Oberklassen

- Alle Blattklassen im Sternzyklus haben eine zentrale Oberklasse, die ihre Unterklassen kennt. In diesem Fall wurde der Oberklasse neben der Rolle, gemeinsame Funktionalität für alle Klassen bereitzustellen, auch noch die Aufgabe einer Factory für die Objekte der Unterklassen übertragen. Hier wäre es besser, wenn die beiden Aufgaben auf zwei Klassen verteilt wären. Jedes Hinzufügen einer

neuen Unterklasse führt sonst dazu, dass die Oberklasse angepasst und alle anderen Unterklassen neu übersetzt werden müssen.

Properties sind dumm.

Das eigentliche Problem des Klassenzyklus in Abbildung 8–5 ist aber nicht der Sternzyklus um den `PluginManager`, sondern die Beziehungen, die sich um die Klasse `PluginProperties` herum ergeben. Eine Klasse namens `PluginProperties` sollte nur Properties zur Verfügung stellen. Das bedeutet, dass sie von anderen Klassen benutzt wird, um Properties abzufragen. Die Klasse selbst sollte aber keine anderen Klassen aufrufen. Sie sollte ganz dumm am Ende der Aufrufkette stehen. Leider geht in diesem Klassenzyklus aber doch eine Beziehung von `PluginProperties` aus, und zwar zur Klasse `PropertyHelper` (s. roter Pfeil).

Helfen kann man immer und überall.

Helper-Klassen finden sich in immer mehr Systemen und sind eine sehr problematische Entwicklung. Das Problem entsteht dadurch, dass das Muster »Helper« sehr generisch ist. Der Name zeigt an, dass es sich um eine Hilfsklasse handelt, er macht aber keine Einschränkungen für die Zusammenarbeit der Helper-Klasse mit anderen Klassen. Eine Helper-Klasse kann also von überall her benutzt werden und jede andere Klasse benutzen. Manchmal habe ich den Eindruck, dass die Helper-Klassen das neue Util sind. Irgendwie hat es sich herumgesprochen, dass Util-Klassen keine gute Idee sind. Helper klingt um eine Stufe besser. Leider sind Helper-Klassen aber nicht besser als Util-Klassen.

Unklare Muster vermeiden

Vermeiden Sie Helper- und Util-Klassen, wann immer es möglich ist, und suchen Sie die passende fachliche Klasse, in die die Funktionalität der Helper- bzw. Util-Klasse integriert werden muss.

11 Geschichten aus der Praxis

Den ganzen Charme einer Architekturanalyse kann man natürlich nur erleben, wenn man sein eigenes System untersuchen darf oder bei der Untersuchung des eigenen Systems dabei ist. Zu einem solchen Erlebnis kann Ihnen ein Buch nicht verhelfen. Aber vielleicht gelingt es, Ihnen mit den nun folgenden Fallstudien einen kleinen Eindruck zu vermitteln.

Anonyme Fallstudien

Für die Fallstudien auf den folgenden Seiten haben mir die Unternehmen ihr Einverständnis zur Veröffentlichung gegeben. Voraussetzung für die Veröffentlichung war, dass das jeweilige System nicht erkannt werden darf. Immer dort, wo es fachlich wird, werden Sie auf den nächsten Seiten also mit anonymen Bezeichnern zu tun haben. Eine Ausnahme bildet das System Epsilon, bei dem mir dankenswerterweise erlaubt wurde, die Fachlichkeit beizubehalten.

Einordnung in den MMI

In Abschnitt 4.5 habe ich den Modularity Maturity Index (MMI) vorgestellt, mit dessen Hilfe es möglich ist, die Modularität verschiedener Systeme zu vergleichen. In Abbildung 11–1 ist der MMI für die nun folgenden sechs Fallstudien dargestellt.

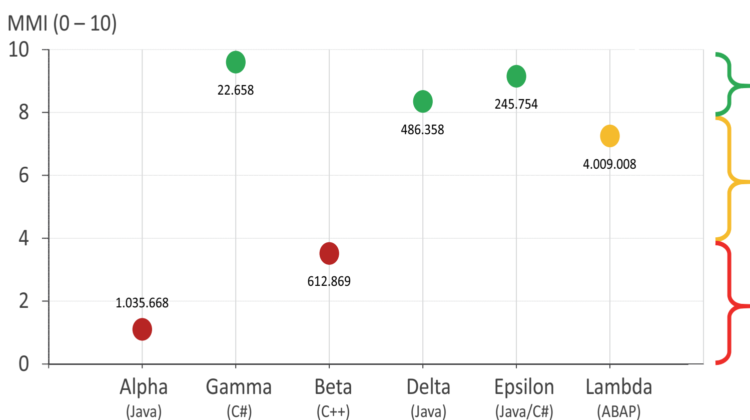


Abb. 11–1
MMI für die Geschichten aus der Praxis

Selbst bei dieser kleinen Menge an Systemen kann man erkennen, dass weder die Größe noch die Programmiersprache der entscheidende Faktor dafür ist, ob ein System beim MMI gut abschneidet oder nicht. Entscheidend ist vielmehr, ob das jeweilige Entwicklerteam Modularität, Musterkonsistenz und Hierarchisierung im Sourcecode und in den gröberen Strukturen des Systems berücksichtigt. Das kann selbstverständlich nur dann gelingen, wenn die Entwickler und Architekten, die ein System im Laufe seines Lebens betreuen, eine entsprechende Aus- und Weiterbildung erfahren durften.

Direkt nach dem Studium der (Wirtschafts-)Informatik oder einem ähnlichen Studium sind die meisten Entwickler in der Lage, kleine Systeme zu bauen, die wartbar bleiben, solange sie klein sind. Die Struktur dieser kleinen Systeme ist von jungen Entwicklern in der Regel nicht so angelegt, dass ihre Systeme beliebig erweitert und so zu großen Systemen werden können. Erst mit der Zeit haben sie die Erfahrungen, die sie brauchen, um große Systeme entwickeln zu können. Erfahrungen, die an der Universität ohne Praxiskenntnisse nicht vermittelbar sind. Entweder durchleben die jungen Entwickler und die Organisation, für die sie versuchen, ihre ersten großen Systeme zu bauen, diesen Lernprozess, der für beide Seiten schmerzhaft sein kann, oder den jungen Entwicklern wird ein erfahrener Entwickler/Architekt zur Seite gestellt. Dann können die jungen Entwickler von den erfahreneren lernen. In den nun folgenden Geschichten aus der Praxis lässt sich dieses Phänomen wiederfinden.

11.1 Das Java-System Alpha

Das System Alpha der Firma Aachen ist in den 90er-Jahren entstanden und wurde im Laufe der Jahre von verschiedenen Entwicklern weiterentwickelt. Anfänglich wurde Alpha in einer funktionalen Programmiersprache gebaut, wurde 1996 aber dann auf Java portiert. Seit 1993 ist es im produktiven Einsatz und wird regelmäßig um neue Funktionalität erweitert. 2005 entstand beispielsweise eine SOAP-Schnittstelle und 2007 wurden umfassende Anstrengungen zur Optimierung der Performance unternommen. Des Weiteren wurden die Oberflächen auf modernere Technologien umgestellt und ältere Oberflächen abgelöst.

1 Mio. LOC Java

Zum Zeitpunkt der Analyse von Alpha im Jahr 2014 war es für das Wartungsteam anstrengend, sich in dem Projekt zurechtzufinden. Alpha umfasst inzwischen 1 Million Lines of Code. Es ist damit ein großes Java-System, was durchaus einiges an Komplexität bergen kann. Die Aussagen des Wartungsteams machten aber den Eindruck,