Carola Lilienthal

Langlebige Softwarearchitekturen

Technische Schulden analysieren, begrenzen und abbauen

3., 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, Frank Heidt
Herstellung: Stefanie Weidner
Umschlaggestaltung: Helmut Kraus, *www.exclam.de*
Druck und Bindung: mediaprint solutions GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.d-nb.de* abrufbar.

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

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

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

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

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

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

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

5 4 3 2 1 0

Geleitwort

Eigentlich ...

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

Dennoch ...

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

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

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

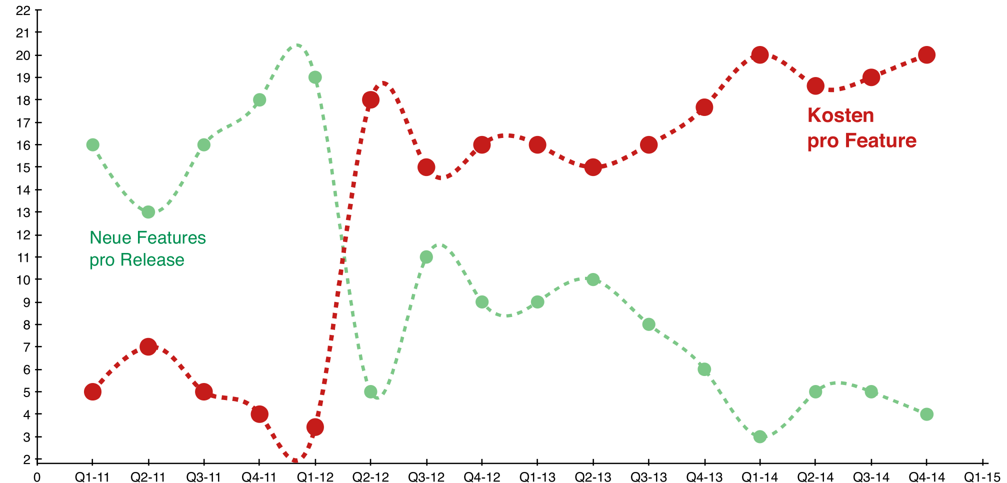


Abb. 1
Steigende Pflegekosten

Endlich ...

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

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

Prima ...

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

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

Die Dritte ...

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

Gernot Starke
Köln, September 2019
www.arc42.de

Vorwort zur 3. Auflage

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

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

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

Carola Lilienthal
Hamburg, August 2019

@cairolali
www.langlebige-softwarearchitektur.de

Inhaltsverzeichnis

1 Einleitung 1

1.1 Softwarearchitektur 1

1.2 Langlebigkeit 3

1.3 Technische Schulden 4

1.3.1 »Programmieren kann jeder!« 8
1.3.2 Komplexität und Größe 9
1.3.3 Die Architekturerosion steigt unbemerkt 11
1.3.4 Für Qualität bezahlen wir nicht extra! 13
1.3.5 Arten von technischen Schulden 14

1.4 Was ich mir alles anschauen durfte 15

1.5 Wer sollte dieses Buch lesen? 16

1.6 Wegweiser durch das Buch 16

2 Aufspüren von technischen Schulden 19

2.1 Begriffsbildung für Bausteine 19

2.2 Soll- und Ist-Architektur 21

2.3 Verbesserung am lebenden System 25

2.4 False Positives und generierter Code 43

2.5 Spickzettel zum Sotographen 45

3 Architektur in Programmiersprachen 47

3.1 Java-Systeme 47

3.2 C#-Systeme 52

3.3 C++-Systeme 54

3.4 ABAP-Systeme 56

3.5 PHP-Systeme 57

3.6 TypeScript -Systeme 59

4 Architekturanalyse und -verbesserung 61

4.1 Entwickler und Architektur . . . 61
4.2 Architekturarbeit ist eine Holschuld . . . 62
4.3 Live-Workshop zur Architekturverbesserung . . . 63
4.4 Der Umgang mit den Vätern und Müttern . . . 65
4.5 Modularity Maturity Index (MMI) . . . 66
4.6 Technische Schulden im Lebenszyklus . . . 68

5 Kognitive Psychologie und Architekturprinzipien 71

5.1 Modularität . . . 72
5.1.1 Chunking . . . 72
5.1.2 Übertragung auf Entwurfsprinzipien . . . 74
5.1.2.1 Einheiten . . . 75
5.1.2.2 Schnittstellen . . . 77
5.1.2.3 Kopplung . . . 78
5.2 Musterkonsistenz . . . 79
5.2.1 Aufbau von Schemata . . . 80
5.2.2 Übertragung auf Entwurfsprinzipien . . . 82
5.3 Hierarchisierung . . . 86
5.3.1 Bildung von Hierarchien . . . 86
5.3.2 Übertragung auf Entwurfsprinzipien . . . 88
5.4 Zyklen = misslungene Modularität + Muster . . . 90
5.5 Konsequenzen für die Architekturanalyse . . . 91

6 Architekturstile gegen technische Schulden 93

6.1 Regeln von Architekturstilen . . . 93
6.2 Trennung von fachlichen und technischen Bausteinen . . . 94
6.3 Schichtenarchitekturen . . . 96
6.3.1 Technische Schichtung . . . 97
6.3.2 Fachliche Schichtung . . . 98
6.3.3 Infrastrukturschicht . . . 100
6.3.4 Integration von fachlichen Schichten . . . 102
6.4 Hexagonal, Onion und Clean Architecture . . . 103
6.5 Microservices und Domain-Driven Design . . . 105
6.6 Mustersprachen . . . 108
6.6.1 WAM-Mustersprache . . . 110
6.6.2 DDD-Mustersprache . . . 112
6.6.3 Typische Framework-Muster . . . 114
6.7 Langlebigkeit und Architekturstile . . . 116

7 Muster in Softwarearchitekturen 117

7.1 Abbildung der Soll-Architektur auf die Ist-Architektur 117
7.2 Die ideale Struktur: fachlich oder technisch? 120
7.3 Schnittstellen von Bausteinen 125
7.4 Interfaces – das architektonische Allheilmittel? 130
7.4.1 Die Basistherapie 130
7.4.2 Die Nebenwirkungen 132
7.4.3 Feldstudien am lebenden Patienten 135
7.4.4 Der Kampf mit dem Monolithen 138
7.5 Der Wunsch nach Microservices 140
7.5.1 Früh übt sich 141
7.5.2 Der Knackpunkt: das Domänenmodell 143

8 Mustersprachen – der architektonische Schatz! 147

8.1 Die Schatzsuche 147
8.2 Die Ausgrabungsarbeiten 149
8.3 Aus der Schatztruhe 151
8.4 Den Goldanteil bestimmen 155
8.5 Jahresringe 156
8.6 Unklare Muster führen zu Zyklen 157

9 Chaos in Schichten – der tägliche Schmerz 161

9.1 Bewertung des Durcheinanders 163
9.1.1 Ausmaß der Unordnung 164
9.1.1.1 Architekturstile und Zyklen 166
9.1.1.2 Programmzeilen in Zyklen 167
9.1.1.3 Dependency Injection und Zyklen 169
9.1.2 Umfang und Verflochtenheit 169
9.1.3 Reichweite in der Architektur 172
9.2 Das große Wirrwarr 176
9.2.1 Der Schwarze-Loch-Effekt 178
9.2.2 Der Befreiungsschlag 180
9.2.3 Technische Schichtung als Waffe 182
9.2.4 Mustersprache als Leuchtturm 184
9.3 Uneven Modules 187

10 Modularität schärfen 191

10.1 Kohäsion von Bausteinen 192
10.2 Größen von Bausteinen 196
10.3 Größen von Klassen 196

10.4 Größe und Komplexität von Methoden 202
10.5 Lose Kopplung 205
10.6 Kopplung und Größe von Klassen 211
10.7 Wie modular sind Sie? 213

11 Geschichten aus der Praxis 215
11.1 Das Java-System Alpha 216
11.2 Das C#-System Gamma 224
11.3 Das C++-System Beta 232
11.4 Das Java-System Delta 241
11.5 Das Java-System Epsilon mit C#-Satelliten 248
11.5.1 Java-Epsilon 248
11.5.2 C#-Epsilon 1 256
11.5.3 C#-Epsilon 2 259
11.6 Das ABAP-System Lambda 264

12 Fazit: der Weg zu langlebigen Architekturen 271

Anhang

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

Literatur 287

Index 295

oder die Namespaces? Je nachdem, welchen Weg das Entwicklungsteam eingeschlagen hat, muss die eine oder die andere Struktur als Grundlage gewählt werden. In Abbildung 3–11 sieht man die Struktur von Magento auf Basis des Directory-Baums. Magento ist eine Open-Source-E-Commerce-Plattform in PHP. Namespaces werden in Magento nicht eingesetzt, sodass als Basis für die Architekturuntersuchung nur der Directory-Baum infrage kommt.

Build-Artefakte aus Frameworks

In manchen PHP-Umgebungen findet man oberhalb der Namespaces und Directories weitere Artefakte zur Strukturierung: Symfony, ein MVC-Framework für PHP, hat ein Konzept von Bundles für Module. Manche Teams verwenden Composer, ein Tool zum Management von Abhängigkeiten für PHP, das ähnliche Aufgaben übernimmt wie Maven. Hier werden die Abhängigkeiten in einem JSON-File beschrieben.

3.6 TypeScript -Systeme

Ein Aufsatz auf JavaScript

TypeScript als Aufsatz von JavaScript macht aus einer ungetypten eine statisch getypte Sprache. Dadurch lassen sich Typen und statische Referenzen zuverlässig aus dem Quelltext extrahieren. Selbstverständlich kann man den Mechanismus der statischen Typisierung in TypeScript aushebeln, indem man `any` als Typ verwendet. Dennoch lässt sich der TypeScript-Anteil von Systemen zuverlässig statisch analysieren.

TypeScript-Frameworks

In der Regel werden die Strukturen in TypeScript-Systemen maßgeblich von den verwendeten Frameworks (z.B. Angular) beeinflusst. Viele Abhängigkeiten werden nicht direkt in TypeScript programmiert, sondern ergeben sich aus den Beziehungen im Framework. In Angular ist es beispielsweise so, dass ein wesentlicher Teil der Beziehungen über die Templates vorgegeben wird. Um einen Gesamtblick auf die Beziehungen und Abhängigkeiten eines TypeScript-Systems zu bekommen, muss man die im TypeScript vorhandenen Beziehungen um solche ergänzen, die mit Framework-Mitteln definiert werden.

Sind Teile des analysierten Systems in reinem JavaScript programmiert, versucht der TypeScript-Transpiler auch in diesen Systemteilen Typinformationen abzuleiten, die für die Analyse zur Verfügung gestellt werden können.

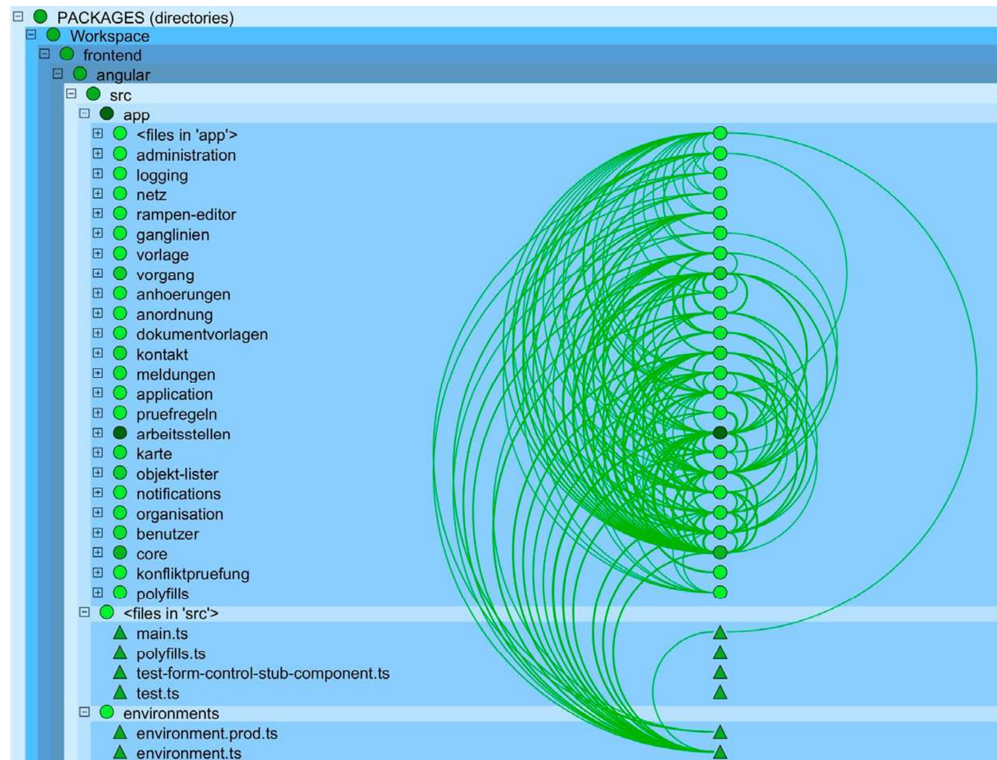


Abb. 3–12
Ein TypeScript-System mit ca. 60.000 LOC

Um ihre Architekturidee in einem TypeScript-System auszudrücken, können sich TypeScript-Programmierer der Directories und der Dateien (.ts) bedienen. Anders als in C# oder Java bildet jede Datei im Prinzip ihren eigenen Namespace bzw. ihr eigenes Package. In TypeScript ist es guter Stil, in eine Datei Klassen und Funktionen gemeinsam abzulegen, die zusammen einen Baustein, z.B. einen fachlichen Service (s. Abschnitt 6.6.2), implementieren.

Schmale Integrationsschicht

Eine etwas abgemilderte Form könnte sein, dass auch die Oberfläche in einzelne fachliche Anteile zerschnitten wird, die wieder den fachlichen Modulen zugeordnet werden. Übrig bleibt am Schluss eine möglichst dünne Schicht der Integration (s. Abb. 6–7).

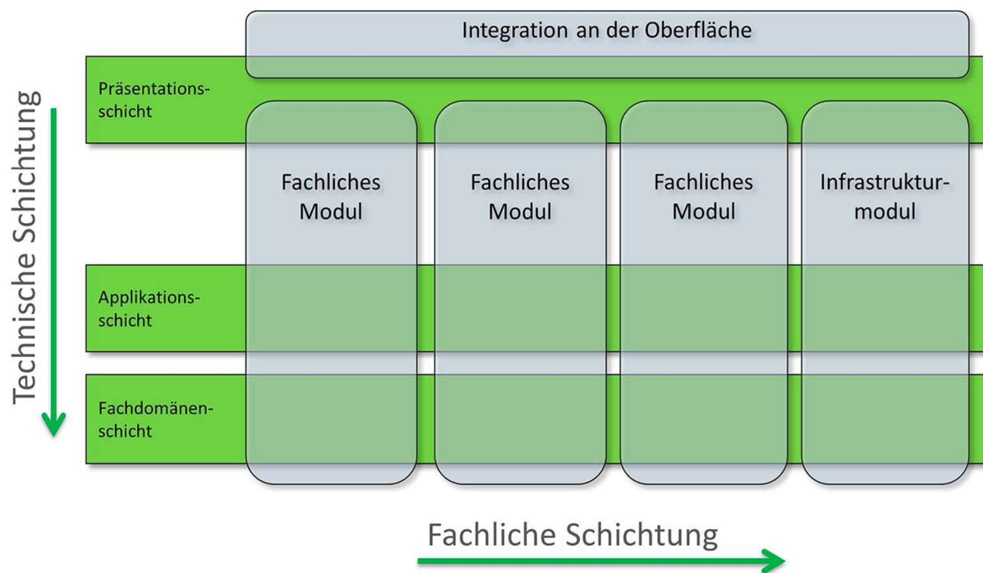


Abb. 6–7
Schmale Integration von fachlichen Modulen

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

6.4 Hexagonal, Onion und Clean Architecture

In den letzten Jahren sind verschiedene weitere Architekturstile aufgetaucht, die Aspekte von Schichtenarchitekturen und Quasar aufnehmen und verbessern:

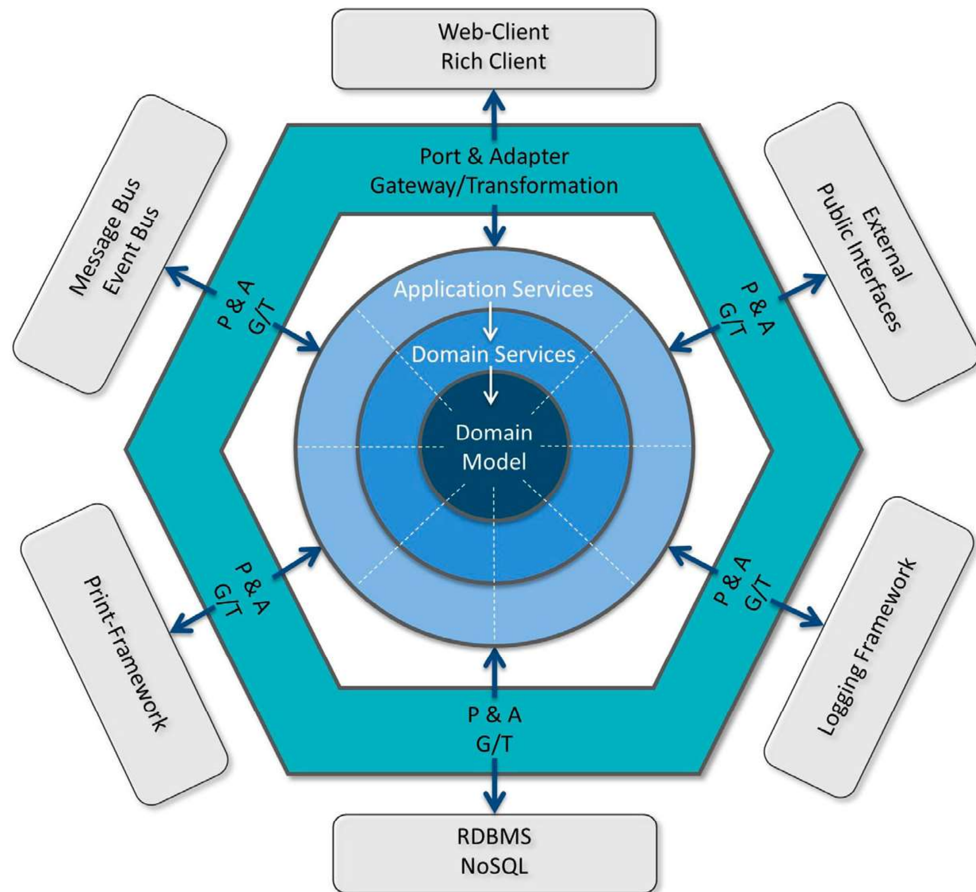
- Hexagonal Architecture (Ports & Adapters) von Alistair Cockburn im Jahre 2005[7]
- Onion Architecture von Jeffrey Palermo in 2008[8]
- Clean Architecture von Robert C. Martin in 2012[9]

Obwohl sich diese Architekturen in ihren Details etwas unterscheiden, haben sie alle das gleiche Ziel: Modularität, wie sie in diesem Buch verfolgt wird. Dieses Ziel erreichen alle diese Stile, indem sie die Trennung der Fachlichkeit von der Technik und den hierarchischen Aufbau der fachlichen Module in den Vordergrund stellen.

7. *http://wiki.c2.com/?PortsAndAdaptersArchitecture*
8. *https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/*
9. *https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html*

In Abbildung 6–8 habe ich versucht, diese verschiedenen Architekturen zusammenzufassen. In der Mitte sieht man die fachlichen Bausteine als Zwiebel aus drei Schichten, deren Beziehungen von außen nach innen gehen. Das Zerlegen der Zwiebel in fachliche Module wird mit gestrichelten Linien angedeutet. Um die Zwiebel herum ist das Hexagon dargestellt, das über Port & Adapter oder auch über Gateway oder Transformation die Zusammenarbeit mit den technischen Bausteinen ermöglicht. Wie genau Port und Adapter funktionieren, lässt sich auf verschiedenen Webseiten nachlesen.[10]

Abb. 6–8
Hexagonal, Onion und Clean Architecture



Mit diesen Architekturstilen werden die fachlichen Bausteine tatsächlich unabhängig von den technischen. Die fachlichen Bausteine können ohne die Benutzeroberfläche, Datenbank, Webserver oder andere externe Schnittstellen getestet werden. Benutzeroberfläche, Datenbank und Schnittstellen können geändert oder ausgetauscht werden, ohne den Rest des Systems zu beeinflussen, weil die fachlichen Bausteine tat-

10. *https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/* oder *https://www.thinktocode.com/2018/07/19/ports-and-adapters-architecture/* oder *https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html*

sächlich nichts vom Rest der Welt wissen. Das ist eine klare Verbesserung gegenüber den Schichtenarchitekturen mit der Infrastrukturschicht. Diese als Zwiebeln in Schichten angelegten Architekturstile lassen sich mit den Mitteln des Sotographen ebenso analysieren wie Schichtenarchitekturen und Mustersprachen. Bei zwei Teams haben wir tatsächlich eine entsprechende Analyse durchgeführt.

Hat man die Zwiebel in fachliche Module aufgeteilt und will man unabhängige deploybare Einheiten haben, so eignen sich Microservices und Domain-Driven Design als ein nächster Schritt.

6.5 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

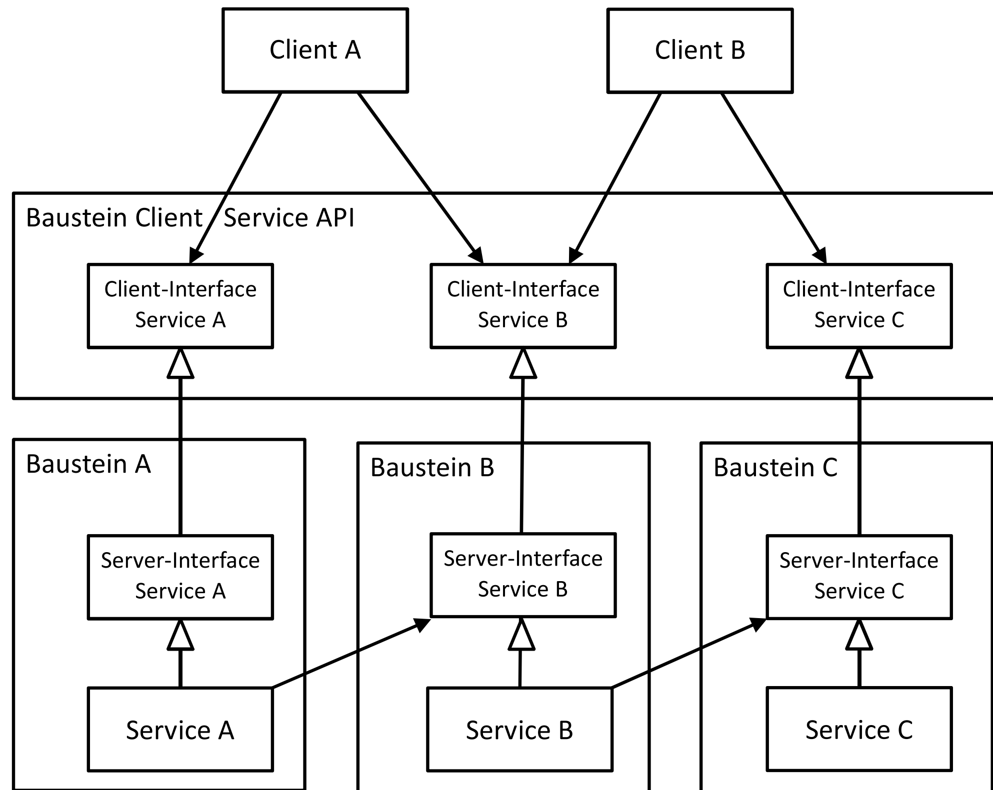
Durch die Kombination von Microservices mit dem strategischen Design aus Domain-Driven Design (DDD) von Eric Evans[11] wird der Fokus beim Zerteilen eines Softwaresystems in Microservices aber auf die Fachlichkeit gelegt. Das ist eine deutliche und aus meiner Sicht sehr wichtige Entwicklung hin zu einer 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 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 fachli-

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

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 6.5), 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 (dt.: große Matschkugel, großes verworrenes Knäuel) konfrontiert.

7.5.1 Früh übt sich

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.

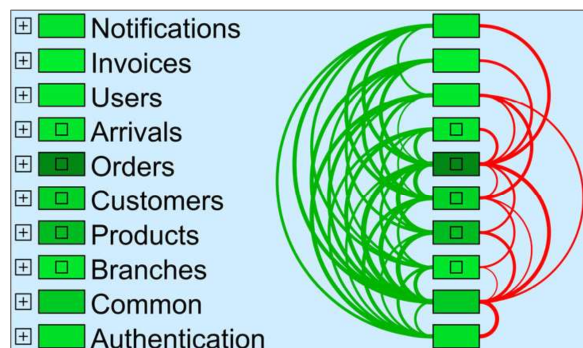
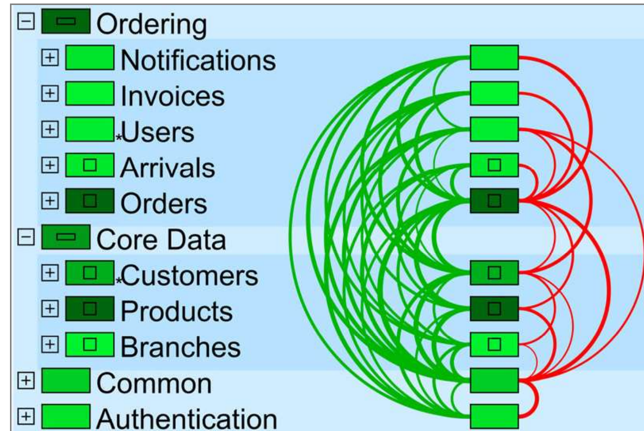


Abb. 7–23
Erste fachliche Aufteilung

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 6.5)?
- 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 6.5), 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

Spätestens wenn 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.

7.5.2 Der Knackpunkt: das Domänenmodell

Wiederverwendung

Die meisten Entwickler haben gelernt, dass sie vorhandene Klassen und Komponenten in ihrer Software wiederverwenden sollen. Folglich bauen sie die ganze Software auf einem Domänenmodell auf, das alle Fachbegriffe der gesamten Domäne enthält. Alle Aspekte jedes Fachbegriffs werden in einer Klasse integriert und es entstehen große Domänenklassen. Dieses Domänenmodell wird aus allen Teilen der Software verwendet und die Klassen im Domänenmodell haben im Vergleich zum Rest des Systems sehr viele Methoden und viele Attribute. Die zentralen Klassen des Domänenmodells, wie z.B. Produkt, Vertrag, Kunde, sind dann meistens auch die größten Klassen im System. Was ist passiert?

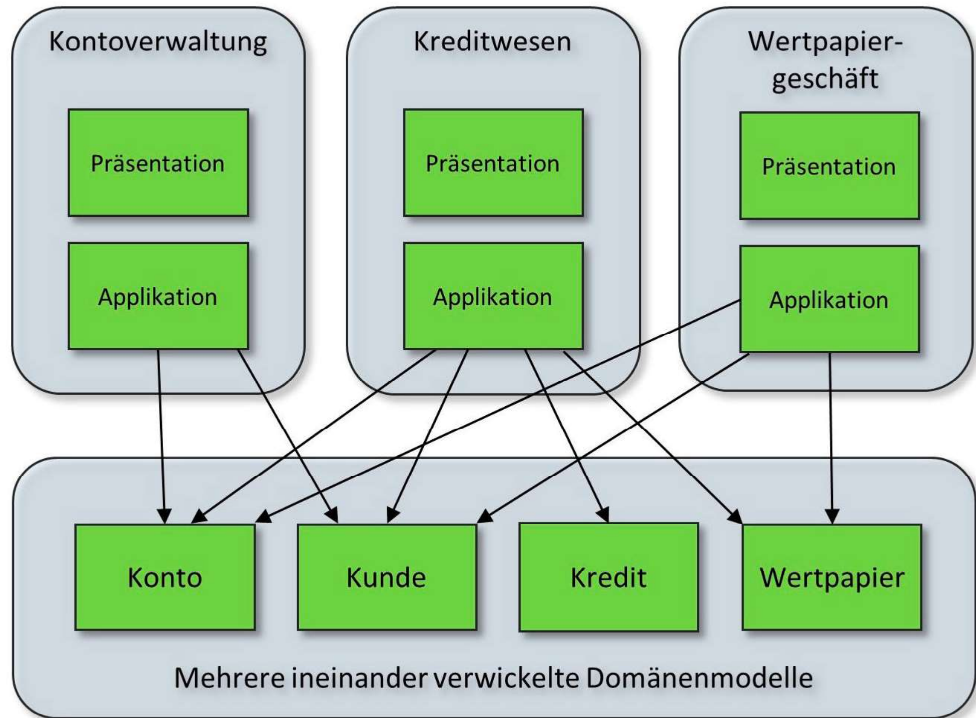
Erweiterung der zentralen Domänenklassen

Jeder Entwickler, der neue Funktionalität in das System eingebaut hat, hat dafür die zentralen Klassen des Domänenmodells benutzt. Allerdings musste er diese Klassen auch ein bisschen erweitern, damit seine neue Funktionalität umgesetzt werden konnte. So bekamen die zentralen Klassen mit jeder neuen Funktionalität ein bis zwei neue Methoden und Attribute hinzu. Genau! So macht man das! Wenn es schon eine Klasse Produkt im System gibt und Funktionalität entwickelt wird, die das Produkt braucht, dann wird die eine Klasse Produkt im System verwendet und so erweitert, dass es passt. Man will nämlich die vorhandene Klasse wiederverwenden und nur an einer Stelle suchen müssen, wenn beim Produkt ein Fehler auftritt. Schade ist nur, dass diese neuen Methoden im Rest des Systems gar nicht benötigt werden, sondern nur für die neue Funktionalität eingebaut wurden.

Großes übergreifendes Domänenmodell

Hier hat Wiederverwendung zu einem großen verwirrenden Domänenmodell geführt. Diesen Fehler machen leider viele Teams. Sie entwickeln in der Hoffnung auf Wiederverwendung ein Softwareprodukt, das als Monolith oder Big Ball of Mud daherkommt. Ein Monolith, der mehrere ineinander verwickelte Domänenmodelle ohne klare Grenzen enthält. Verschiedenartige Konzepte, die eigentlich nicht miteinander in Beziehung stehen, werden so über viele Module verteilt und durch sich widersprechende Elemente verbunden. Zusätzlich arbeiten mehrere Teams an einem großen Domänenmodell, was für die Geschwindigkeit der Entwicklung und die Unabhängigkeit der Teams sehr problematisch ist. In Abbildung 7–25 sieht man ein zentrales Domänenmodell, das von den darauf aufbauenden fachlichen Modulen gemeinsam verwendet wird.

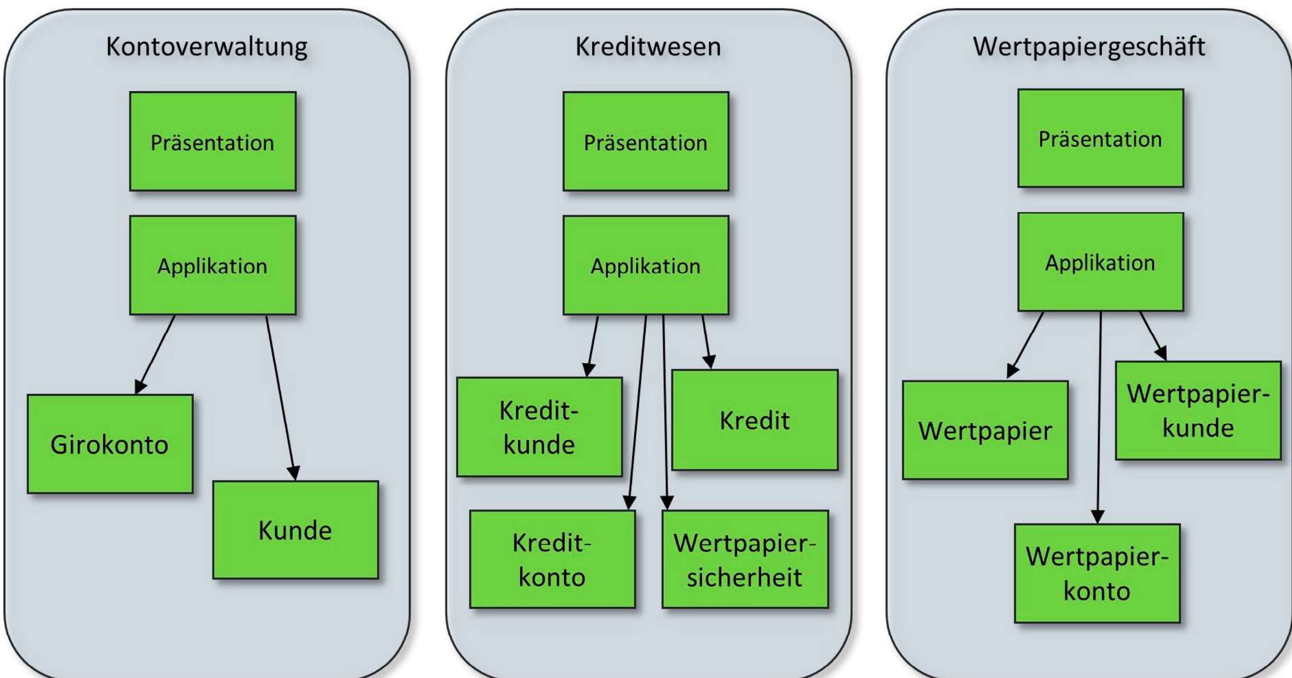
Abb. 7–25
Monolith mit einem großen Domänenmodell



Kontextspezifische Domänenmodelle

Domain-Driven Design empfiehlt uns eine andere Lösung (s. Abb. 7–26). Das zentrale Domänenmodell aus Abbildung 7–25 ist auf die verschiedenen fachlichen Bounded Contexts oder Microservices aufgeteilt. Jeder einzelne Bounded Context hat sein eigenes kleines Domänenmodell. Die jeweils kontextspezifischen Konzepte der lokalen Domänenmodelle werden je nach den Bedürfnissen des jeweiligen Bounded Context strukturell und sprachlich spezialisiert.

Abb. 7–26
Mehrere Bounded Contexts mit spezifischem Domänenmodell



270 Microservices mit einem model.jar

Vor einiger Zeit war ich bei einer Firma mit einem großen Websystem für den Verkauf von Gebrauchtwagen. Die Architekten sagten mir voller Stolz, dass sie 270 getrennt deploybare Microservices hätten. Ich war beeindruckt und fragte, wo ich denn helfen könnte. Die Antwort war: »Immer wenn ein Team in seinem Microservice ein neues Feld braucht, müssen alle Teams zusammenkommen und die Änderung besprechen! Das ist sehr aufwendig!« Nun war ich überrascht und ließ mir den Sourcecode und den Build-Prozess mehrerer zentraler Microservices zeigen. Dabei war das Problem nicht zu übersehen: Alle Microservices verwendeten ein »model.jar«. Dieses Jar enthielt das gesamte Domänenmodell für alle 270 Microservices und bildete gleichzeitig die Schnittstelle zu der einen und einzigen Datenbank. Diese 270 getrennt deploybaren Bausteine kann man sicherlich nicht als Microservices im Sinne des Domain-Driven Design bezeichnen. Die Skalierung ist möglicherweise gut, aber getrennte Teams, die unabhängig voneinander entwickeln, erreicht man auf diese Weise selbstverständlich nicht.

Nicht zu klein anfangen!

Bevor Sie jetzt aber dazu übergehen, Ihr System in viele kleine Bounded Contexts zu zerlegen, möchte ich Sie auf ein Missverständnis hinweisen: Bedenken Sie bei der Aufteilung in Bounded Contexts, dass diese nicht zu klein werden sollten. Jeder Bounded Context sollte einen weitgehend eigenständigen Systemteil umfassen, für das ein Team verantwortlich ist. Folgen Sie diesem Rat, so kann es bei einer Neuentwicklung passieren, dass die ersten Versionen des neuen Systems, die bestimmte Teilprozesse umsetzen, als ein Bounded Context von einem Team begonnen werden. Nach einiger Entwicklungszeit wird der erste Bounded Context zu groß für ein Team, sodass er aufgeteilt werden kann.

Code duplizieren und zurückbauen

Bei einem Altsystem ist der Prozess auf gewisse Weise entgegengesetzt. Ein umfassendes Domänenmodell in einem meistens großen Monolithen zu zerschlagen, ist eine Herkulesaufgabe. Zu verwoben sind die auf dem Domänenmodell aufsetzenden Teile des Systems mit den Klassen des Domänenmodells. Um hier weiterzukommen, zerlegen wir den Big Ball of Mud nicht sofort in alle Bounded Contexts, die wir in der Analyse mit den Fachanwendern entdeckt haben. Ein solches Refactoring wäre viel zu groß. Vielmehr wird der erste Schnitt für eine Aufteilung in zwei immer noch sehr große Bounded Contexts gesetzt (s. Abschnitt 7.5.1). Dabei kopieren wir die Domänenklassen in jeden der beiden Bounded Contexts, wir duplizieren also Code! Anschließend bauen wir die Domänenklassen jeweils für ihren Bounded Context zurück. So bekommen wir kontextspezifische Domänenklassen, die von ihrem jeweiligen Team unabhängig vom Rest des Systems

erweitert und angepasst werden können. Wenn dieses Refactoring gelungen ist, wird die weitere Aufteilung Schritt für Schritt vorangetrieben.

Ob Sie Microservices einsetzen wollen oder nicht, für die Langlebigkeit Ihres Systems ist eine Aufteilung in der fachlichen Dimension mit kleinen kontextspezifischen Domänenmodellen ein guter Weg.

Fachliche kontextspezifische Strukturen

Mit jeder Funktionalität, die Sie in Ihr System einbauen (lassen), diskutieren Sie in Ihrem Team, ob eine Aufteilung in getrennte Bounded Contexts mit eigenen kontextspezifischen Domänenmodellen angebracht ist. Je später Sie damit anfangen, desto aufwendiger wird es.