Langlebige Software-Architekturen

Technische Schulden analysieren, begrenzen und abbauen

DIE LESEPROBE

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.

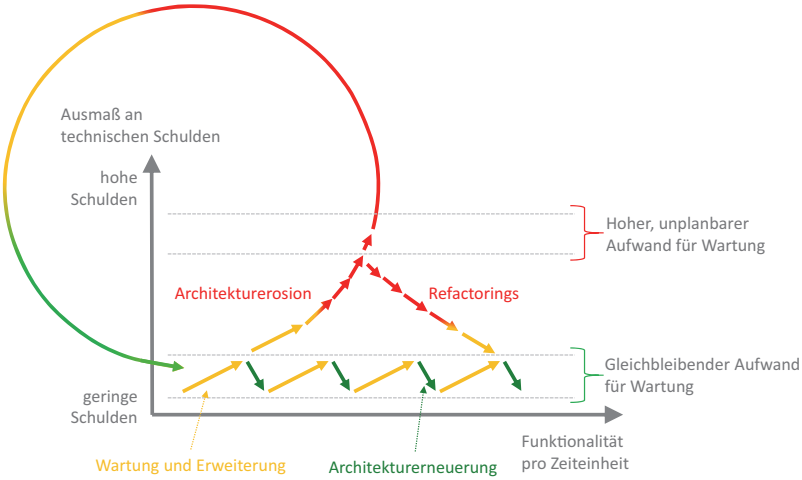


Abb. 1–1
Technische Schulden und Architekturerosion

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 ■ 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**
  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.

Unbeweglichkeit

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

Zähigkeit

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

1.3.4 Für Qualität bezahlen wir nicht extra!

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.

Individualsoftwareentwicklung = 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.

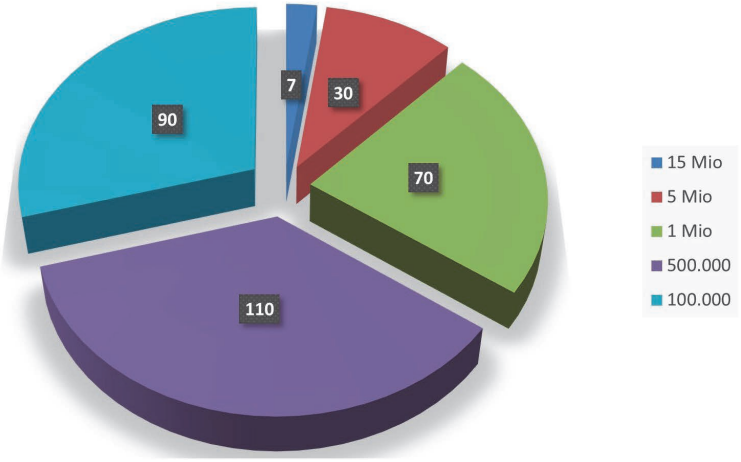


Abb. 1–2 *Größenordnung der untersuchten Systeme in Lines of Code (LOC)*

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.

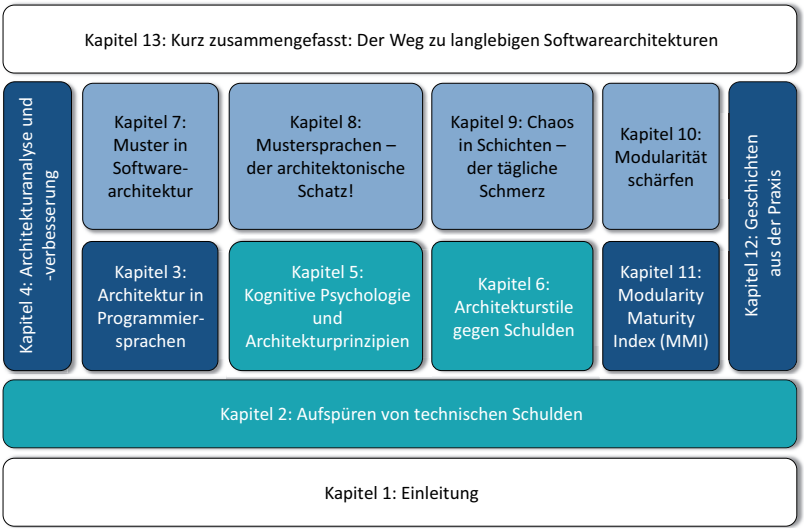


Abb. 1–3
Aufbau des Buches

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.

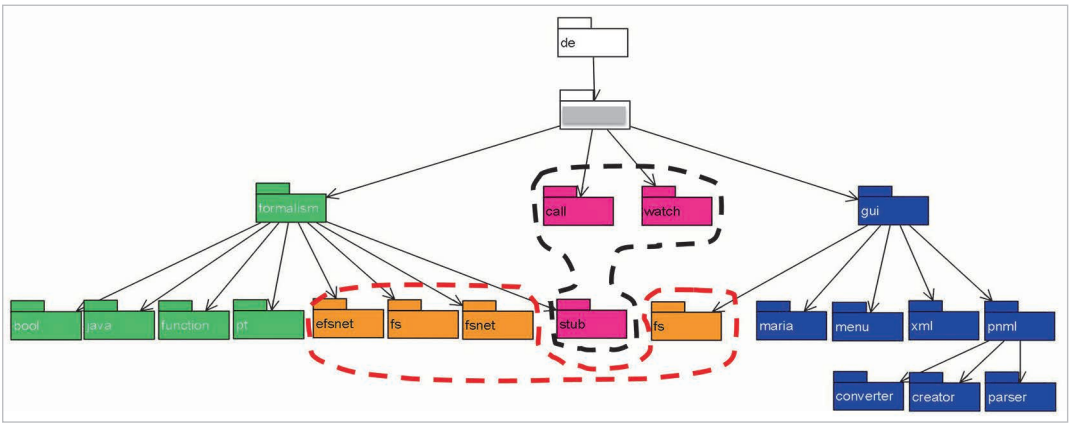
Architektur im Sourcecode

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:

Keine Zeit + kein Geld

- *Fehlendes Budget*
  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).

Großes Refactoring = hohe Testaufwände

- *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.

Veraltete Technologie

- *Widerspenstige Technik*
  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.

```
de.name.ui.risiko
           .angebot
           .produkt

de.name.logic.risiko
              .angebot
              .produkt
```

versus

```
de.name.risiko.ui
         .risiko.logic

de.name.angebot.ui
         .angebot.logic

de.name.produkt.ui
         .produkt.logic
```

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.

Deployment-Struktur

Nimmt man nun noch das Deployment der Software auf verschiedene Prozesse oder Devices als ein Kriterium für die Architektur mit auf, dann braucht man oberhalb der fachlichen Struktur ggf. noch eine weitere Ebene. Diese Ebene zerlegt den Sourcecode in mehrere Build-Artefakte, die auf verschiedene Prozesse oder Rechner deployed werden können. Getrennt werden so z.B. der Clientcode von Servercode oder die verschiedenen Microservice-Deployments. In Detail kann die Strukturierung Ihres Systems dann beispielsweise folgendermaßen aussehen:

- Bei einer *Rich-Client-Anwendung* oder *Native App* mit *Serverkomponente*
  Als oberste Ebene wird eine grobe Strukturierung in Client, Server und Common/Shared gewählt, um das Deployment abzubilden (s. Abb. 7–3). Gemeinsame Anteile, die von Client und Server verwendet werden sollen, werden üblicherweise in einem gemeinsamen Bereich mit dem Namen Common oder Shared zusammengefasst. Unterhalb dieser drei Bereiche existieren fachliche Build-Artefakte. Häufig sind die fachlichen Build-Artefakte im Client anders oder gröber geschnitten als auf dem Server, weil der Client verschiedene Serverkomponenten integriert (s. Abschnitt 6.3.4).
- Bei einer *Webanwendung*
  Die grobe Strukturierung besteht aus Integrationsschicht (s. Abschnitt 6.3.4) und fachlichen Build-Artefakten.
- Bei einer *reinen Serveranwendung*
  Die fachlichen Build-Artefakte bilden die erste Strukturierung.

Strukturierung in fachlichen Build-Artefakten

Unterhalb der ersten Ebene von fachlichen Build-Artefakten wird in allen Architekturvarianten auf die gleiche Weise strukturiert: Die fachlichen Build-Artefakte werden so lange weiter fachlich unterteilt, bis man bei fachlich nicht weiter zerlegbaren Features angekommen ist. Diese Pakete mit einzelnen fachlichen Features enthalten dann die entsprechenden Klassen aus allen technischen Schichten für die Umsetzung. Verwendet man eine Mustersprache, so findet man hier jeweils einen Satz von Musterelementen (s. Abschnitt 6.6).

In Abbildung 7–3 sieht man die Aufteilung einer Client-Server-Anwendung in die drei Bereiche Client, Server und Common. Unterhalb dieser drei Deployment-Bereiche finden sich die fachlichen Build-Artefakte. Die weiteren Ebenen darunter sind in Abbildung 7–3 nicht zu sehen.

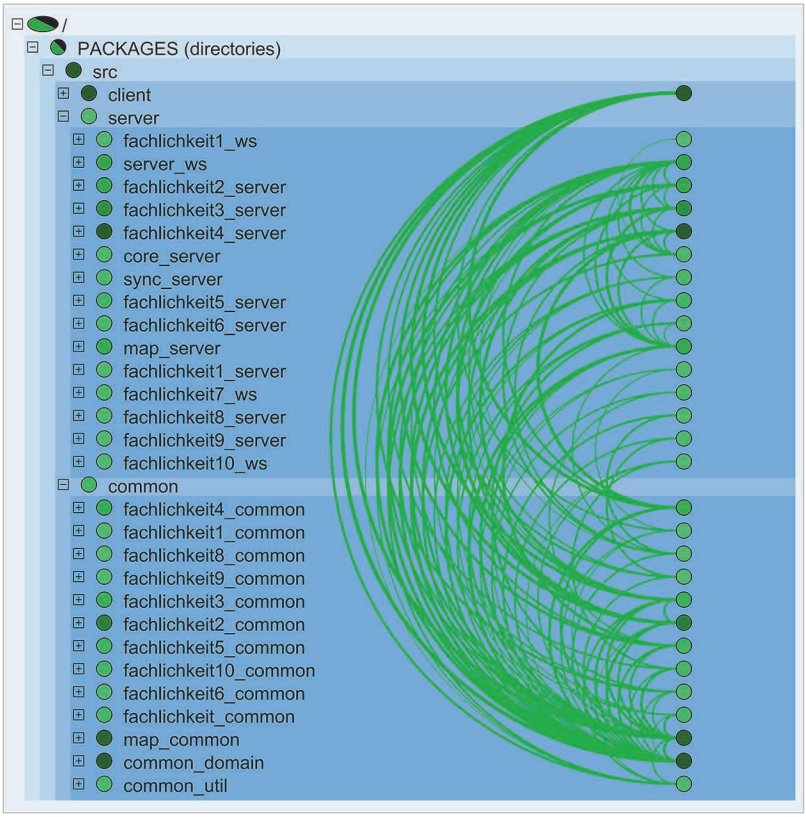


Abb. 7–3
Deployment und fachliche Build-Artefakte einer Rich-Client-Anwendung

Mischung im Umbruch

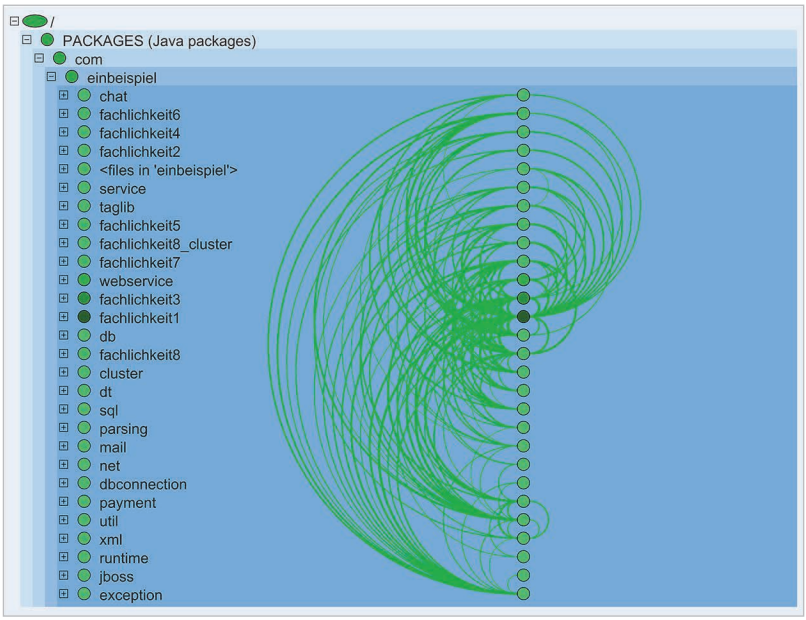
Schwierig sind Systeme, bei denen die Umstellung von der technischen Schichtung auf eine fachliche Aufteilung halb umgesetzt ist oder wo technische und fachliche Strukturen nebeneinander existieren. In solchen Systemen findet man häufig fachliche und technische Paketknoten auf einer Ebene. Bei diesen Systemen haben die älteren Entwickler in der Regel keine wohlstrukturierte Architekturvorstellung, sondern eine Reihe von parallelen Schemata, mit denen sie ihren Weg finden. Neue Entwickler haben Schwierigkeiten, sich in solchen Strukturen zurechtzufinden.

Untersuchung einer Baustelle

Bei der Architekturanalyse solcher Systeme ist viel Modellierungsaufwand zu leisten, weil die Abbildung des Sourcecodes auf die Soll-Architektur viel Wissen über die verschiedenen Varianten der Architekturmuster benötigt. Als Ursachen für diesen baustellenartigen Zustand bringen die Architekten und Entwickler meist Budgetmangel und widerspenstige Technik an (s. Abschnitt 7.1). Oder aber sie haben gar keine Vorstellung davon, dass es eine Architektur geben sollte, die man in der Struktur des Systems ausdrücken könnte.

In Abbildung 7–4 sehen Sie so ein uneinheitliches System. Dabei handelt es sich um ein Java-System mit 700.000 LOC. Das System wird in vier voneinander abhängigen Eclipse-Projekten entwickelt. Die Packages sind über alle Eclipse-Projekte verteilt und zum Teil gibt es dieselben Packages in verschiedenen Eclipse-Projekten. Mehr zu diesem Code-Smell der Split-Packages finden Sie in Abschnitt 3.1.

Abb. 7–4
Fachliche und technische Package-Knoten gemischt



Chaos in Komponenten

Eine andere Problematik bilden Softwaresysteme, bei denen auf der obersten Ebene zwar eine einheitliche Strukturierung, z.B. nach Fachlichkeit, gewählt wurde, unterhalb dieser ersten Ebene aber ganz unterschiedliche Strukturen zu finden sind. Das Ergebnis ist, dass man sich jede Komponente im System einzeln erklären lassen muss. Für diese uneinheitliche Struktur bringen die Architekten oft die folgenden Argumente an:

Mein Code, dein Code

- *No-Collective-Code-Ownership*
  Die Uneinheitlichkeit der Paketierung hat ihre Ursache darin, dass die einzelnen Entwickler jeweils für einen Teilbereich des Softwaresystems zuständig sind. Jeder Entwickler hat seine persönliche Vorstellung über die Strukturierung seines Systemteils. Eine Einheitlichkeit ist nicht vorgesehen und entsteht folglich auch nicht. Microservices gehen mit ihrer Idee von unabhängigen Technologie-Entscheidungen pro Microservice sogar noch einen Schritt weiter: Verschiedene Microservices werden je nach Bedarf mit unterschiedlichen Programmiersprachen und Technologien umgesetzt.

- *Künstler am Werk*
  Ein anderer Architekt sagte mir, dass er eine Vereinheitlichung der Strukturierung nicht durchsetzen könne. Seine Entwickler müssten sich als Softwarekünstler ausleben können und würden so das System jedes Jahr um neue Strukturen und Muster erweitern. Er sagte mir, dass sein System deshalb *Jahresringe* aus Mustern habe.

Selbstverwirklichung

Den Ausführungen in den letzten Abschnitten können Sie schon entnehmen, dass ich mich eher für Einheitlichkeit als für Selbstverwirklichung aussprechen werde. Selbstverständlich lasse ich mich durch gute Argumente davon überzeugen, dass sich bestimmte Technologien und Programmiersprachen für bestimmte Probleme besser eignen und die Architektur deshalb heterogen sein muss. Trotzdem können die Strukturen in den verschiedenen Systemteilen einheitlich aufgebaut werden.

Einheitlichkeit vor Selbstverwirklichung

Einheitliche Muster

Verwenden Sie für die Strukturierung Ihrer Architektur (fachlich, technisch) einheitliche Muster, die immer auf die gleiche Art im Paket-Baum bzw. den Build-Artefakten wiederzufinden sind.

Hat man sich über die grundsätzliche Strukturierung des Systems in technische und fachliche Einheiten geeinigt, kommt bei größeren Systemen als Nächstes die Frage nach den Schnittstellen auf.

7.3 Schnittstellen von Bausteinen

Bei den ersten Java-Systemen, die ich analysiert habe, hatte ich den Eindruck, dass Schnittstellen in Java kein wirklich wichtiges Konzept darstellen.

Braucht man Schnittstellen im System?

Das wurde mir besonders bei der Analyse eines Java-Systems deutlich, das von einem Team gebaut worden war, das früher in PL1 programmiert hatte. Diese Gruppe von Entwicklern legte sehr viel Wert auf ihre internen Schnittstellen – so waren sie es von früher gewohnt! Gleich zu Beginn der Entwicklung des Systems hatten die Entwickler die Schnittstellen der Bausteine festgelegt. Damit einher ging natürlich auch eine Entwicklung mit Code-Ownership: Jeder Entwickler hatte seinen Baustein, den er entwarf und implementierte. Wie die Bausteine zusammenarbeiten, wird über die Schnittstelle abgesprochen. Ohne Schnittstellen ging in diesem Fall gar nichts.

Dokumentierte Schnittstellen

Die Schnittstellen waren für diese Teams das entscheidende Mittel, um die Architektur ihres Softwaresystems stabil zu halten. Diese Archi-

tekten waren ganz besonders daran interessiert, zu überprüfen, ob die Schnittstellen ihrer Bausteine eingehalten werden. Meistens findet man bei solchen Teams eine umfassende Dokumentation aller Schnittstellen. Diese Dokumentation bildet das Kernstück der Architekturbeschreibung und wird an alle Veränderungen angepasst. Bausteine dürfen nur über die dort dokumentierten Klassen und Methoden benutzt werden. Die Architekten solcher Systeme waren in den Analysen überzeugt davon, dass ihr Softwaresystem nur deshalb erweiterbar und wartbar geblieben ist, weil sie sich dieser Restriktion unterworfen haben.

Schnittstellen kommen später.

Im Gegensatz dazu konzipieren die meisten mit objektorientierten Sprachen groß gewordenen Entwicklern zu Beginn des Projekts keine Schnittstellen für die Bausteine ihres Systems oberhalb von Klassen und Paketen. Auf Nachfrage zeigte sich, dass den Architekten Schnittstellen durchaus wichtig sind. Die Definition von Schnittstellen hielten die Architekten aber am Anfang des Projekts nicht für notwendig oder sinnvoll. Die Architekten sagten meistens sinngemäß: »Die Bausteine sind noch zu klein, da ist der gesamte Inhalt in der Schnittstelle.«

Schnittstellen bilden sich üblicherweise ab der Größe von 400.000 LOC zwischen den Bausteinen eines Systems. Die ersten Schnittstellen findet man dabei häufig zwischen Client- und Server-Sourcecode. Das ist ein sehr natürlicher Prozess, weil die Entwickler und Architekten den Client vom Server entkoppeln wollen. Eine solche Schnittstelle besteht aus den Klassen und Interfaces des Servers, die vom Client-Sourcecode verwendet werden.

Fachliche Schnittstellen

Genau wie bei der Schichtung (s. Abschnitt 7.1) beginnen die Architekten und Entwickler also auch hier eher mit einer Schnittstelle in der technischen Schichtung und nicht mit einer Schnittstelle zwischen fachlichen Modulen. Sinnvoll ist aber ähnlich wie bei der Schichtung, dass rechtzeitig die Schnittstellen zwischen den fachlichen Modulen entworfen werden:

Schnittstellen zwischen fachlichen Modulen

Beginnen Sie mit dem Design der fachlichen Schnittstellen, sobald sich die erste Aufteilung in fachliche Module abzeichnet.

Schnittstellen kamen zu spät.

Beachten Sie diesen Merksatz nicht, kommen Sie möglicherweise in die Situation eines Architekten, bei dem ich vor einiger Zeit eine Architekturanalyse machen durfte: Sein System hat eine beachtliche Größe von 850.000 LOC in Java. Schnittstellen für Bausteine waren in der Architektur aber noch nicht vorhanden. Als dieser Architekt gesehen hatte, dass mit dem Sotographen eine Überprüfung der Schnittstellen möglich

ist, fragte er uns, ob wir vielleicht auch die inverse Menge prüfen könnten. Er sagte: »Ich hätte gerne eine Abfrage, die feststellt, welche Klassen in den einzelnen Bausteinen bisher nicht von außen verwendet werden. Dann würde ich gerne täglich überprüfen, dass niemand anfängt auch noch diese Klassen zu benutzen.«

Muster für Schnittstellen

Die spannende Frage ist nun: Wo findet man die Schnittstellen in den Systemen? Es gibt einige Varianten, wie Entwicklungsteams ihre Schnittstellen ablegen:

1. *Verteilte Schnittstelle*
   Die Schnittstelle wird von allen Interfaces gebildet, die in irgendwelchen Paketen in der Paketstruktur des Bausteins zu finden sind.
2. *Wurzelpaketschnittstelle*
   Im obersten Paket, dem Wurzelknoten des Bausteins, werden die Elemente der Schnittstelle abgelegt.
3. *Schnittstellenpaket*
   In jedem Baustein gibt es ein spezielles Paket für die Schnittstelle, das sich direkt unterhalb des Wurzelknotens des Bausteins befindet.
4. *Schnittstellenprojekt*
   Die Schnittstelle wird in ein eigenes Projekt ausgelagert.

Verteilte Schnittstellen

Verteilte Schnittstellen finden sich oft in kleineren und jüngeren Systemen, bei denen sich alle Entwickler im gesamten Sourcecode auskennen. In Abbildung 7–5 sieht man einen Ausschnitt eines Systems, bei dem in der Service-Schicht zwei Namespaces existieren: Berater und Niederlassungsleiter. Jeder dieser Namespaces enthält jeweils Interface und Implementierung der jeweiligen Services. Allerdings sollen nur die Interfaces von außerhalb der Service-Schicht verwendet werden, die Implementierungen nicht.

Verteilte Schnittstellen reduzieren die Übersicht.

Das Verteilen der Schnittstellen über die gesamte Paketstruktur ist keine gute Lösung. Will sich ein Nutzer dieses Bausteins die Schnittstelle anschauen, so muss er durch die gesamte Paketstruktur wandern, um einen Einblick zu bekommen. Die Architekten dieser Lösungen argumentieren oft damit, dass man so direkt neben der konkreten Klasse das Interface findet und schneller bei Fragen der Implementierung zu der konkreten Klasse navigieren kann. Da die meisten Entwicklungsumgebungen aber die Vererbungs- oder Implementierungsbeziehungen auswerten können und dem Entwickler die entsprechenden Unterklassen anzeigen können, ist mir dieses Argument nicht ausreichend. Darüber hinaus gehören oft auch Exception-Klassen zur Schnittstelle, für die es kein Interface gibt.

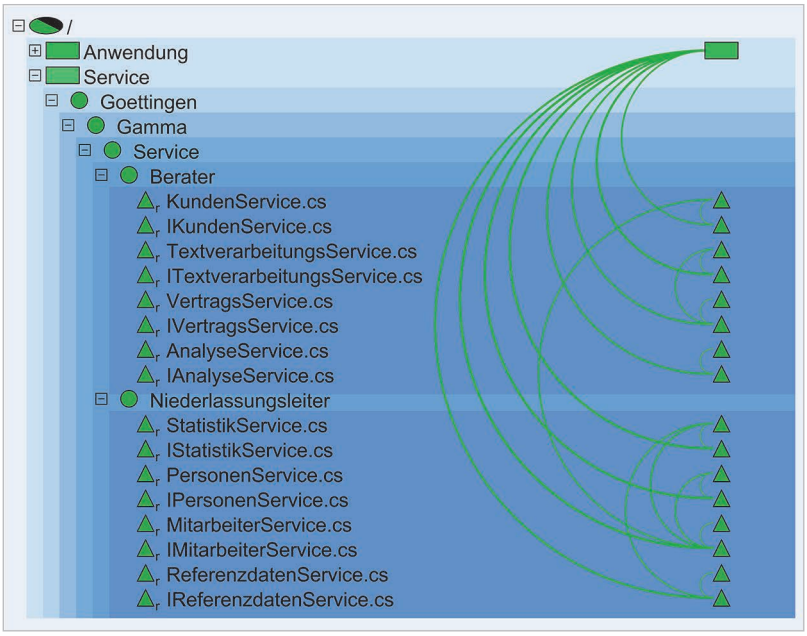


Abb. 7–5
Verteilte Schnittstellen

Wurzelpaketschnittstelle

Die Variante Wurzelpaketschnittstelle geht hier in die richtige Richtung: Alle Klassen und Interfaces der Schnittstelle sind an einer Stelle versammelt (s. Abb. 7–6). Sie können gemeinsam untersucht werden – Chunking wird zumindest durch die räumliche Nähe möglich. Ob inhaltlich auch Nähe zwischen den Klassen und Interfaces der Schnittstelle besteht, ist natürlich nicht gesagt.

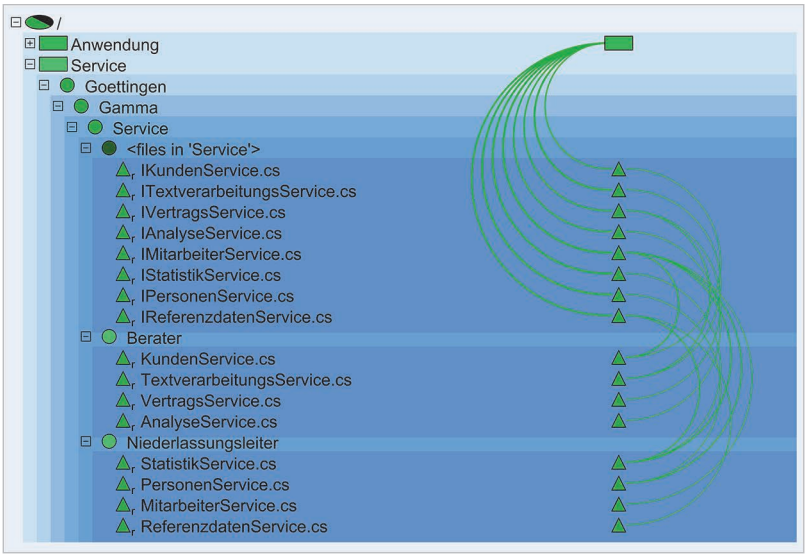


Abb. 7–6
Wurzelpaketschnittstelle

Wurzelknoten für Schnittstelle und Hilfsklassen

Das Problem dieser Variante ist, dass in das Wurzelpaket oft nicht nur die Klassen und Interfaces der Schnittstelle gelegt werden. Entwicklungsteams suchen oft einen Ort für Hilfsklassen, die im gesamten Baustein verwendet werden. Da es keinen natürlichen Ort für diese Hilfsklassen gibt, werden sie häufig auch im Wurzelpaket untergebracht. So entsteht im Wurzelpaket eine Mischung aus Schnittstelle und Hilfsklassen, was dem Verwender dieses Bausteins keinen klaren Eindruck von der Schnittstelle gibt.

Schnittstellenpaket

Diese Unklarheit löst Variante 3, Schnittstellenpaket, auf (s. Abb. 7–7): Für die Klassen und Interfaces der Schnittstelle gibt es ein Paket mit einem eigenen Namen. Manche Teams, wie das Eclipse-Team, nennen dieses Paket `api`, manche nennen es `interface` oder `public`. Gibt es parallel dazu dann noch ein Paket namens `impl`, in dem sich die gesamte Implementierung befindet, so ist der Entwickler sehr gut unterstützt. Er wird von einem für alle Bausteine einheitlichen Muster angeleitet und findet die Schnittstelle an einem Ort versammelt.

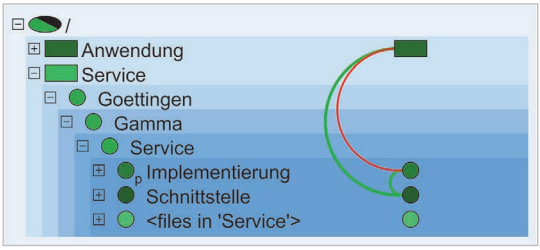


Abb. 7–7 *Schnittstellenpaket in Java*

In Abbildung 7–7 heißen die beiden Namespaces `Schnittstelle` und `Implementierung`. An dem kleinen p am Namespace `Implementierung` sieht man, dass dieser Knoten privat sein soll. Der Bogen von der Anwendungsschicht zum Namespace `Implementierung` ist rot eingefärbt, denn diese Beziehungen sind von der Architektur verboten.

Schnittstellenprojekt

Hat das Entwicklungsteam seinen Sourcecode in mehrere Projekte aufgeteilt, entsteht oft auch die Idee, eigene Schnittstellen- und Implementierungsprojekte zu bilden, also Variante 4. Dadurch wird die Schnittstelle noch deutlicher von der Implementierung getrennt. Diese Aufteilung hat den Vorteil, dass ein Projekt, das auf einem anderen aufbaut, allein mit dem Interface-Projekt zusammen übersetzt werden kann. Das Implementierungsprojekt wird nicht benötigt. Welche Gefahren sie bergen und warum man sie deshalb nur mit äußerster Vorsicht einsetzen sollte, zeigt der nächste Abschnitt.

7.4 Interfaces – das architektonische Allheilmittel?

In der Softwareentwicklung haben sich verschiedene Prinzipien durchgesetzt, die mit Interfaces und Interface-Projekten zusammenhängen. Software wird flexibler und änderbarer durch:

- Kapselung und lose Kopplung
- Abhängigkeit von Abstraktionen
- Programmierung gegen Abstraktionen

Problem von Interface-Projekten

Problematisch wird der Einsatz von Interfaces und Schnittstellenprojekten dann, wenn sie eingesetzt werden, um Zyklen aufzulösen. Keine Frage: Mit Interfaces kann man Zyklen auflösen und viele Bücher empfehlen dieses Vorgehen. Mir gefällt das nicht! Interfaces sind erfunden worden, damit zu einer abstrakten Beschreibung verschiedene Implementierungen angeboten und eingesetzt werden können. Interfaces sind nicht entwickelt worden, um Zyklen aufzulösen.

Abb. 7–8
Zweierzyklus auf Klassenebene

Wir haben einen Zyklus.

Schauen wir uns das einmal genauer an. Stellen wir uns vor, wir haben zwei Klassen (A und B) programmiert (s. Abb. 7–8) und diese beiden Klassen brauchen sich gegenseitig. Wir haben also einen Zyklus auf Klassenebene.

Um diesen Zyklus aufzulösen, haben wir zwei Möglichkeiten:

1. Das Design der Klassen überprüfen und es überarbeiten, sodass der Zyklus nicht mehr nötig ist.
2. Den Zyklus mithilfe von Interfaces auflösen.

7.4.1 Die Basistherapie

Ich wäre natürlich dafür, Strategie 1 zu verfolgen und den Zyklus durch besseres Design loszuwerden. Aber stellen wir uns einmal vor, wir würden Strategie 2 anwenden. Wir erhalten dann eine Struktur aus zwei Klassen und einem Interface, in der es keinen Zyklus mehr gibt (s. Abb. 7–9).

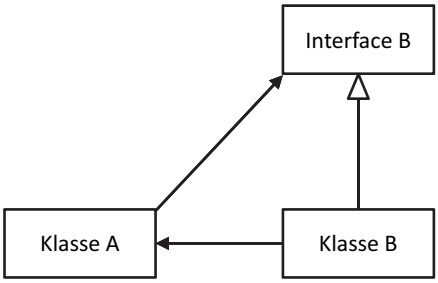


Abb. 7–9
Auflösung eines Zweierzyklus mit Interface

Klasse B implementiert Interface B. Die Beziehung geht von Klasse B zu Interface B, sodass man von Klasse B aus keinen Zyklus mehr hat.

Interfaces verschleiern Zyklen.

Liegt das Hauptaugenmerk des Projektleiters oder der Qualitätsabteilung darauf, bei Metriken besonders gut dazustehen, so ist das Einführen von Interfaces bei zyklisch verkoppelten Klassen die Lösung! Alle Tools zur Qualitätsmessung, die im Projekt eingesetzt werden, schlagen bei den Zyklen nun nicht mehr an. Yes!

Aber ist das Design wirklich besser geworden, dadurch dass wir ein fachliches Konzept – das Klasse B hoffentlich darstellt – in zwei Bestandteile zerlegt haben? Sicherlich nicht! Die vorher statisch sichtbare zyklische Abhängigkeit zwischen Klasse A und B haben wir jetzt in die Laufzeitumgebung verschoben.

Bausteinzyklen

Richtig deutlich wird die Problematik, wenn wir uns vorstellen, dass Klasse A und Klasse B in zwei verschiedenen Bausteinen als Services wichtige Rollen spielen. Wir haben hier also nicht nur einen Zyklus auf Klassenebene, sondern auch auf Bausteinebene (s. Abb. 7–10).

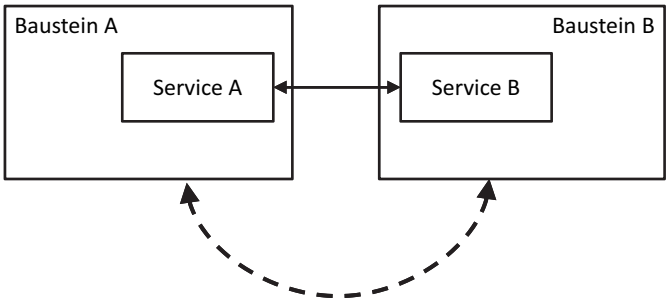


Abb. 7–10
Zweierzyklus auf Bausteinebene

Wenden wir nun unseren Zaubertrick mit dem Interface an, verbessert das erst einmal nicht unsere Situation.

Bausteinzyklen mit Interfaces

Das Interface B wird zusammen mit der Klasse B im Baustein B untergebracht und der Zyklus zwischen den beiden Bausteinen ist weiterhin vorhanden. Hier hilft uns das Zerlegen von Service B in Interface und Implementierung nicht weiter (s. Abb. 7–11).

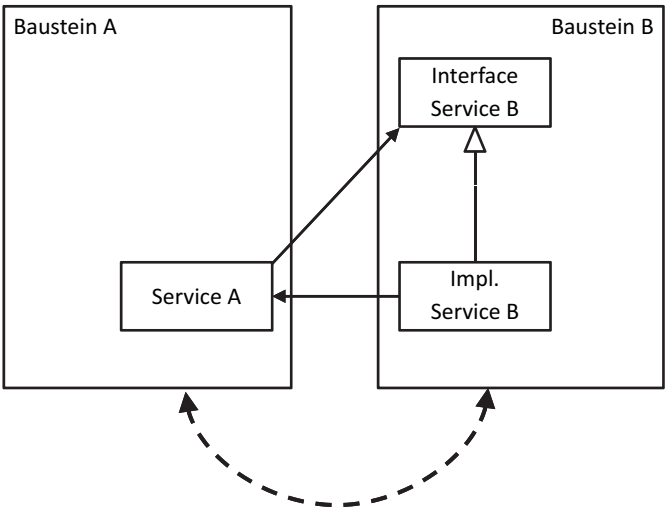


Abb. 7–11
Interface im Zweierzyklus

Um den Zyklus zwischen den Bausteinen doch noch loszuwerden, sind in der Praxis zwei Varianten zu finden: Das Interface von Service B wird Baustein A zugeordnet (s. Abb. 7–12) oder das Interface von Service B wird in einen eigenen Baustein abgelegt (s. Abb. 7–13).

7.4.2 Die Nebenwirkungen

Interface in Plug-in-Architekturen

Die Variante in Abbildung 7–12 hat sich bei Plug-in-Architekturen sehr bewährt. Hier wird von Baustein A ein Interface für ein Plug-in definiert, das von anderen Bausteinen geliefert werden muss, damit Baustein A funktionieren kann. Es ist bei dieser Lösung zu erwarten, dass nicht nur Baustein B eine Implementierung für das Interface Service B liefern wird, sondern eine Reihe anderer Bausteine ihre eigene Implementierung mitbringen. Alle diese Implementierungen erben von Interface Service B. Bei dieser Variante wird das Konzept »Interface« genau im Sinne seiner ursprünglich von den Erfindern geplanten Lösung verwendet. Wir erinnern uns: Interfaces sind eine abstrakte Beschreibung, für die verschiedene Implementierungen angeboten und eingesetzt werden können.

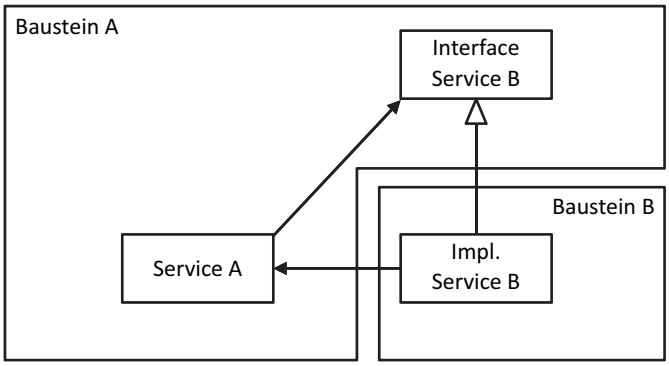


Abb. 7–12
Schneiden von Bausteinen, um Zyklen unsichtbar zu machen

Dependency Injection

Damit die Plug-in-Lösung funktionieren kann, braucht Baustein A natürlich zur Laufzeit die passende Implementierung zu seinem Interface – entweder aus Baustein B oder aus einem anderen Baustein, der eine Implementierung des Plug-in-Interface anbietet. Heute steht eine Reihe von Dependency-Injection-Frameworks zur Verfügung, die diese Aufgabe übernehmen. Alternativ kann auch eine Factory eingesetzt werden, die außerhalb von Baustein A liegen muss, sonst holt man sich den Zyklus zwischen Baustein A und B direkt wieder ins System.

Doch keine Plug-in-Architektur!

Ob die Plug-in-Lösung trägt bzw. ob es eine gute Idee ist, das Interface von Service B dem Baustein A zuzuordnen, merkt man sehr schnell. Wenn andere Bausteine als Baustein A auch das Interface von Service B verwenden wollen. Dann kann das Interface Service B kein Plug-in-Interface von Baustein A mehr sein. In diesem Fall ist das Interface Service B leider nur das Interface von der Implementierung von Service B. Das Interface Service B hat also nichts im Baustein A verloren und wir stehen weiterhin mit unserem Zyklus zwischen Baustein A und B da (s. Abb. 7–11).

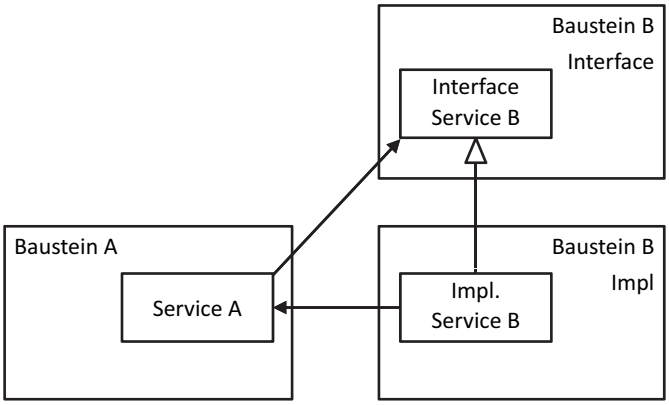
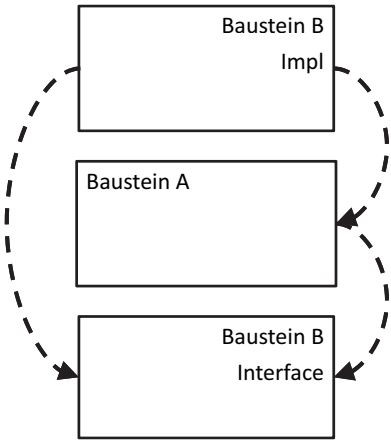


Abb. 7–13
Schneiden von Bausteinen, um Zyklen unsichtbar zu machen

Interface-Projekt?

Um diesem Problem zu entgehen, wird häufig die Variante in Abbildung 7–13 gewählt, bei der das Interface in einen eigenen Baustein verschoben wird. Normalerweise sind in diesem Interface Baustein B natürlich alle Interfaces aus Baustein B versammelt. Hier wird das Problem auf Bausteine mit nur jeweils einer einzelnen Klasse und einem Interface vereinfacht, damit das Prinzip deutlicher wird.

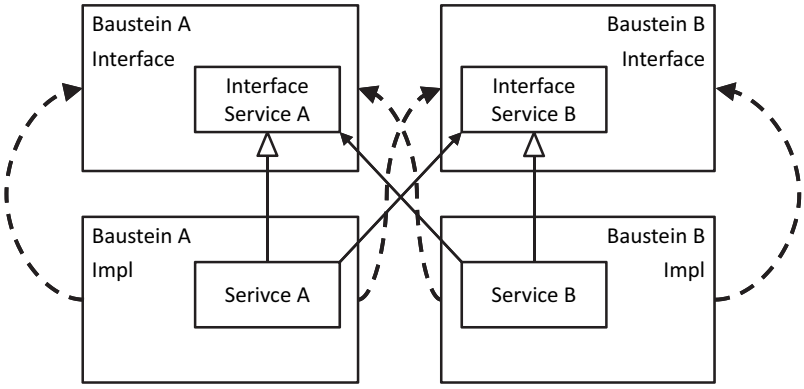
Abb. 7–14
Interface-Baustein



Getrennte Bausteine, die zusammengehören

Dadurch, dass man den Baustein B in zwei Teile aufspaltet, erhält man auf Bausteinebene eine Struktur wie in Abbildung 7–14. Baustein A und Baustein B Impl brauchen Baustein B Interface, aber die gegenseitige Beziehung existiert nicht mehr. Um diese Beziehungen hierarchisch zu ordnen, müssen die Bestandteile von Baustein B um Baustein A herum verteilt werden. Der fachliche Baustein B zerfällt also in zwei Bausteine, die im System nicht mehr nahe beieinander liegen.

Abb. 7–15
Jeder fachliche Baustein hat zwei technische Teile.



Wenn man das Spiel mit den Interfaces zur Vermeidung von Zyklen weiter treibt, wird irgendwann auch für Baustein A ein Interface-Projekt notwendig (s. Abb. 7–15).

Interface-Projekte als Muster

Wenn Baustein A und Baustein B bereits ein getrenntes Interface-Projekt haben, dann wird dieses Muster irgendwann zur Regel für die gesamte Entwicklung. Jedes Projekt muss ein Interface-Projekt haben, über das auf seine Dienste zugegriffen wird. Oder man legt ein großes Interface-Projekt für das ganze Softwaresystem an, wo alle Interfaces versammelt sind (s. Abb. 7–16). Beide Ansätze klingen doch erst einmal nach einer sehr guten Regel bzw. Leitplanke für die Architektur, nicht wahr?

Interface-Projekte verschleiern Architekturprobleme.

In Wirklichkeit werden aber durch diese Aufteilung in Projekte nur die Abhängigkeiten zwischen Bausteinen verschleiert. Eine echte Modularisierung z.B. im Sinne von fachlichen aufeinander aufbauenden Schnitten oder Microservices wird so nicht erreicht. Vielmehr werden fachlich zusammengehörige Einheiten zerlegt, um technisch Zyklenfreiheit zu erreichen (s. Abb. 7–16).

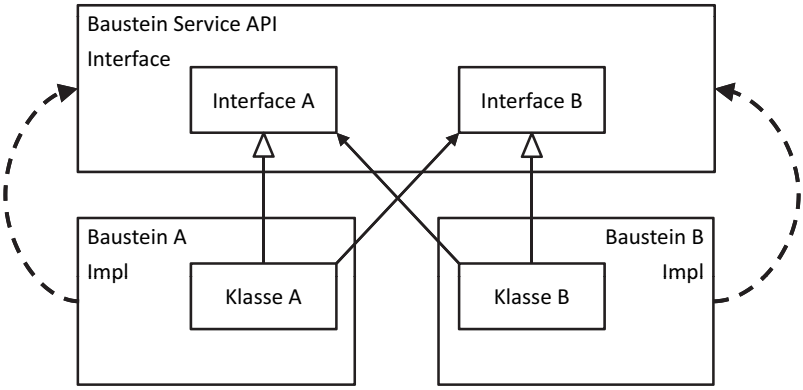


Abb. 7–16
Ein Interface-Projekt für alle

7.4.3 Feldstudien am lebenden Patienten

Für alles ein Interface- und ein Implementierungsprojekt

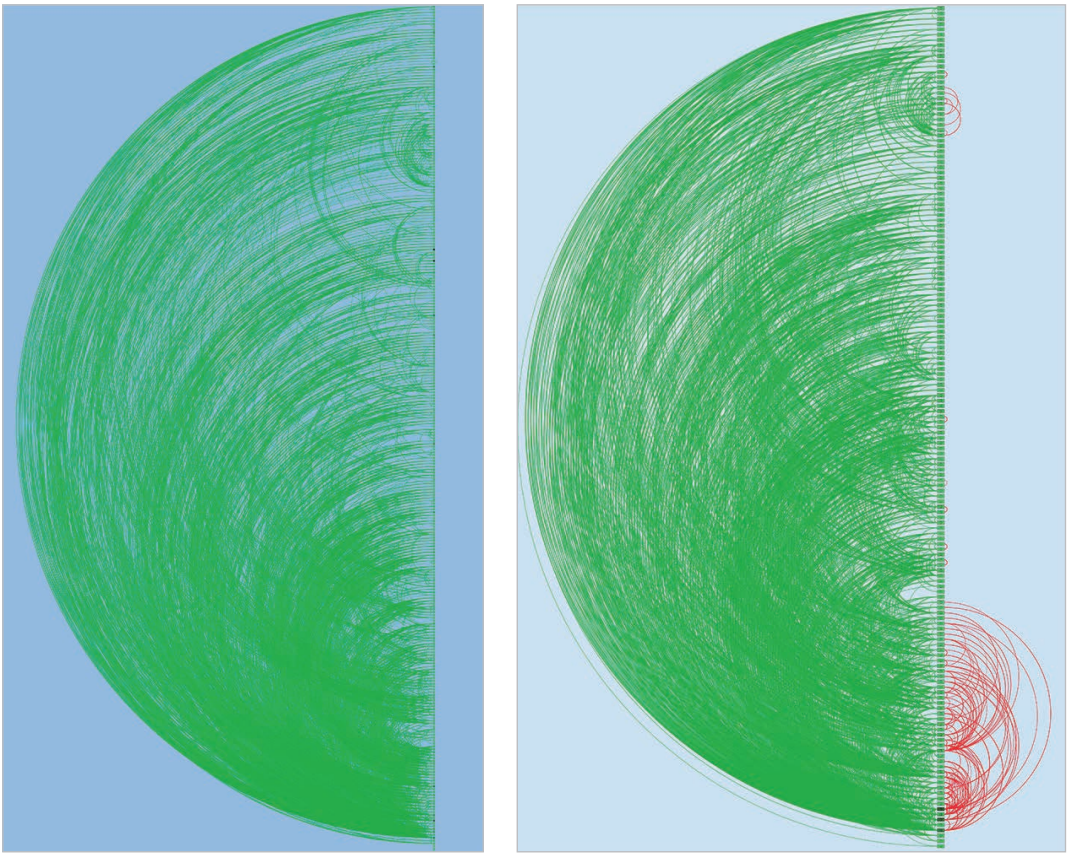
Zwei Beispiele folgen hier, damit Sie sich das Problem an echten Systemen vergegenwärtigen können. Das erste System hat 715.000 LOC in 7.400 C#-Klassen, die in 590 Projekte aufgeteilt sind. Da Visual Studio keine Zyklen zwischen C#-Projekten und den daraus gebildeten DLLs zulässt, mussten die Projekte, die potenzielle Zyklen verursachen, jeweils in Interface- und Implementationsprojekte aufgeteilt werden. Für 552 Projekte gibt es in diesem System ein Interface- und ein Implementationsprojekt. Die einzige andere Möglichkeit wäre gewesen, einige sehr große Projekte zu bilden.

In der folgenden Abb. 7–17 sehen Sie auf der linken Seite das wunderbar geschichtete System aus 590 Projekten. Auf der rechten Seite

sehen Sie, was passiert, wenn man die Interface- und Implementierungsprojekte zusammenlegt. Es entsteht eine Reihe von roten Bögen, die für Beziehungen gegen die Richtung der Schichtung stehen.

Besonders im unteren Bereich des Systems, wo die Basisfunktionalität angesiedelt ist, existiert ein Knäuel aus Projekten, die sich gegenseitig brauchen. Die fachliche Diskussion der Verletzungen während der Analyse hat diesem System sehr in der Reifung seines Designs weitergeholfen.

Abb. 7–17
Dasselbe System mit Trennung und ohne Trennung von Interface- und Implementierungsprojekten



Ein Projekt für alle Interfaces

Beim zweiten System haben die Architekten und Entwickler alle Interfaces in ein gemeinsames Projekt gelegt, sodass alle anderen Projekte genau einen Einstiegspunkt haben, um die Funktionalität der anderen Projekte aufzurufen.

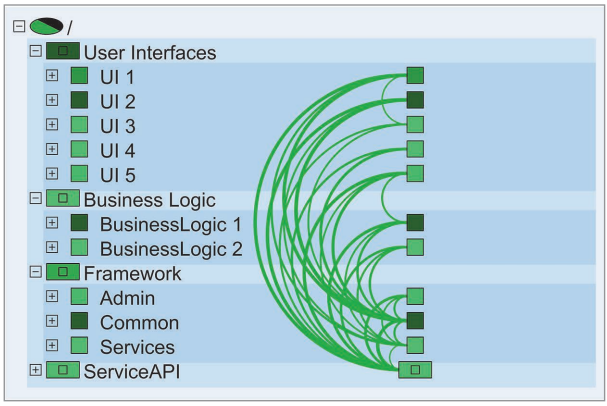


Abb. 7–18
Alle Projekte benutzen die ServiceAPI, um sich gegenseitig aufzurufen.

Die ServiceAPI

Dieses System hat 190.000 LOC in 2.000 Java-Klassen, die in 42 Projekten aufgeteilt sind. In Abbildung 7–18 sieht man, dass alle Projekte Beziehungen zu ServiceAPI haben. Dadurch, dass ServiceAPI alle Interfaces enthält, gehen alle Beziehungen von den Projekten nach unten. Die einzelnen Projekte rufen sich gegenseitig fast gar nicht auf. Die Stellen, an denen es direkte Beziehungen zwischen den einzelnen Projekten gibt (UI 1 → UI 3, UI 5 → BL 1, Zugriffe auf Common und Services), wurden von den Entwicklern und Architekten als Verletzungen betrachtet und von uns bei der Analyse untersucht.

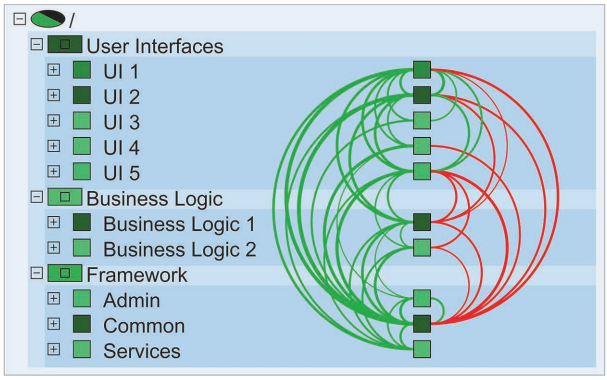


Abb. 7–19
Interfaces den Implementationen zugeordnet

Verwobene Fachlichkeit

Integriert man die Interfaces in die jeweiligen fachlichen Projekte, so entsteht ein ganz anderes Bild (s. Abb. 7–19). Die verschiedenen Projekte sind stark miteinander gekoppelt. Besonders das Projekt Common wird stark verwendet und benutzt die über ihm liegenden Projekte umfassend.

7.4.4 Der Kampf mit dem Monolithen

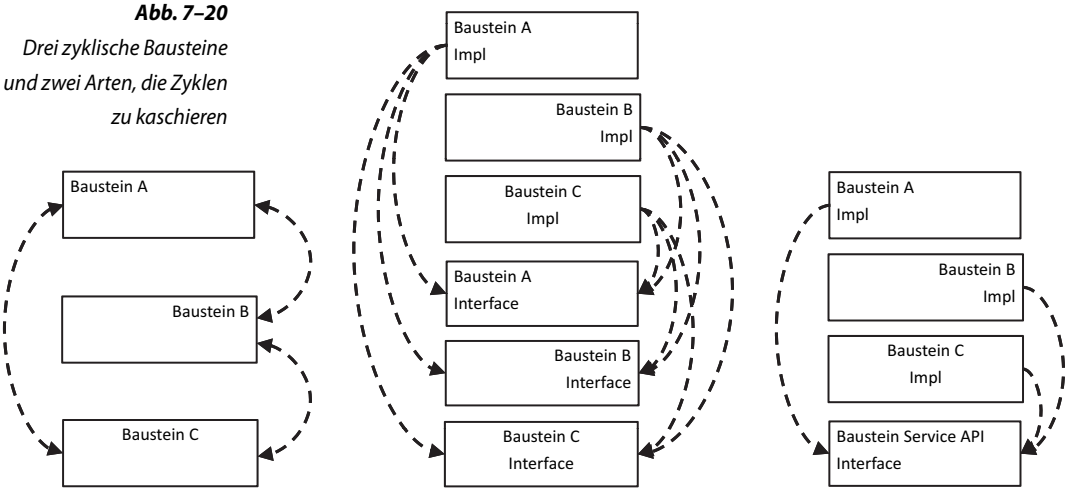
Fachliche Hierarchien im Monolithen?

Viele Entwicklungsteams und Architekten schlagen diesen Weg ein und zerlegen ihre Bausteine in ein Interface- und ein Implementierungsprojekt. Meistens liegt es daran, dass die Teams ihr System mit wachsender Größe gerne entlang fachlicher Gesichtspunkte modularisieren wollen. Dadurch wird es möglich, dass die Teams an kleineren Projekten arbeiten, ohne das gesamte Projekt auschecken zu müssen. Verwenden die Teams eine Entwicklungsumgebung, wie Visual Studio, oder ein Build-System, wie Maven, kommt jetzt der Moment der Wahrheit: Hat das System eine fachlich hierarchische Struktur?

Keine technischen Notlösungen, bitte!

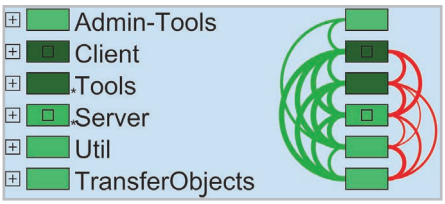
Denn: Projekte in Visual Studio oder mit Maven gebaute Projekte dürfen nicht zyklisch miteinander verknüpft sein. Die einfachste Methode, mit dieser Beschränkung umzugehen, besteht darin, Interface-Projekte zu bilden. Abbildung 7–20 fasst die üblichen Aufteilungen zusammen.

Abb. 7–20
Drei zyklische Bausteine und zwei Arten, die Zyklen zu kaschieren



Diese Aufteilung ist eine Notlösung, die Sie für eine langlebige Architektur nicht wählen sollten!

Abb. 7–21
System mit vielen Zyklen



Zerschlagung des Monolithen

Wenn Sie ein monolithisches System wie in Abbildung 7–21 haben, das Sie gerne in mehrere Projekte aufteilen wollen, dann empfiehlt sich das folgende Vorgehen:

Client-Server-Trennung

- Trennen Sie Ihr System als Erstes dort in zwei Projekte auf, wo sich automatisch ein zyklenfreier Schnitt ergibt bzw. ergeben sollte: zwischen Client- und Servercode. Wenn das schwierig ist, haben Sie viel Refactoring-Aufwand vor sich, weil das System wahrscheinlich gar keine Struktur hat (s. Abschnitt 12.1)!

Trennung von Oberflächen- und Businesslogik

- Bei einer Webarchitektur nehmen Sie die Stelle, wo sich im Sourcecode der beste Schnitt zwischen der Oberflächenlogik und der Businesslogik durchführen lässt.

Server-Interface-Projekt zum Client

- Falls Sie den Clientcode unabhängig vom Servercode übersetzen wollen, dann legen Sie die Interfaces für den Servercode in ein eigenes Projekt.
- Das Server-Interface-Projekt sollte nur von dem oder den Clients verwendet werden dürfen. Andere Serverprojekte sollten weiterhin direkt auf den Serverklassen arbeiten.

Fachliche Module im Server

- Beginnen Sie nun den Sourcecode Ihres Serverprojekts zu untersuchen. Wenn Sie die Konstruktion mit einem Fokus auf Modularität, Muster und Zyklenfreiheit angegangen sind, dann sollten die fachlichen Schnitte im Serverprojekt möglich sein und nur eine Richtung der Benutzung zwischen den fachlichen Schnitten existieren.
- Für die einzelnen fachlichen Serverprojekte stellen Sie nun jeweils Interfaces zur Verfügung. Diese Interfaces liegen beim jeweiligen Serverprojekt und dürfen vom jeweils übergeordneten Schnitt gerufen werden.

Fachliche Schichtung und Microservices

- Auf diesem Wege bekommen Sie die erste Version einer fachlichen Schichtung Ihres Systems (s. Abb. 7–22). Falls Ihr Ziel Microservices sind, dann müssen Sie sich nun noch Gedanken über die Kommunikation zwischen den fachlichen Modulen machen. Anschließend können Sie sie in verschiedenen Prozessen laufen lassen.

Schnittstellen beim Modul ablegen

Legen Sie die Schnittstellen für die in einem Modul implementierte Funktionalität im gleichen Projekt ab wie die Klassen, die die Funktionalität implementieren.

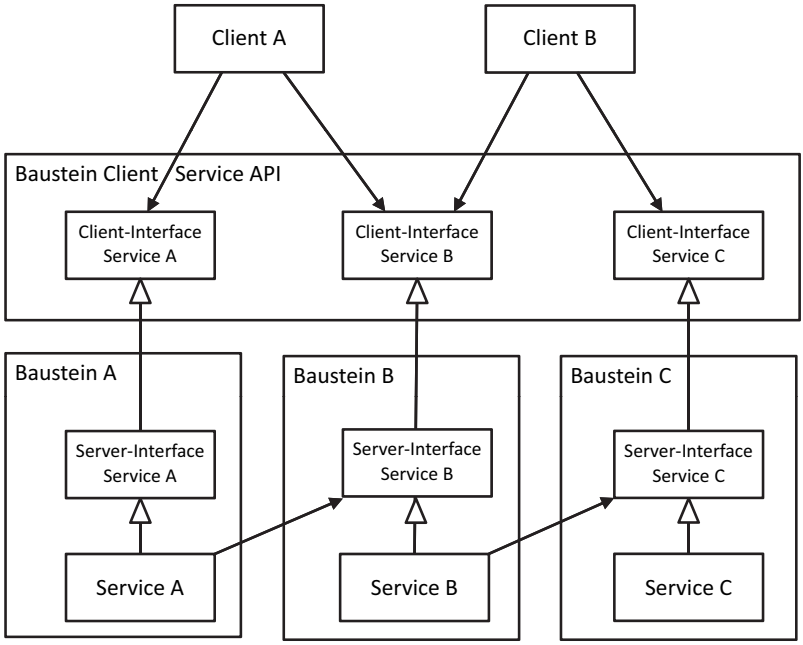


Abb. 7–22
Aufteilung der Bausteine in einer langlebigen Architektur

7.5 Der Wunsch nach Microservices

Microservices müssen sein.

Eine Reihe von 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. Aus diesem Grund gibt es von meinem Kollegen Henning Schwentner und mir inzwischen ein weiteres Buch mit dem Titel »Domain-Driven Transformation – Monolithen und Microservices zukunftsfähig machen«, in dem wir das Zerlegen von Softwaresystemen mit Domain-Driven Design detailliert beschreiben (s. [Lilienthal & Schwentner 2023]).

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 inzwischen macht. Wenn dann noch Themen wie DevOps, Cloud 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 ich 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 einem Großteil der 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 bezüglich 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.

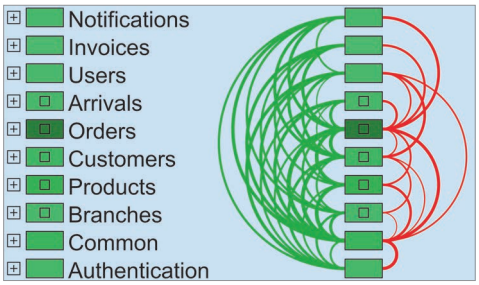


Abb. 7–23
Erste fachliche Aufteilung

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).

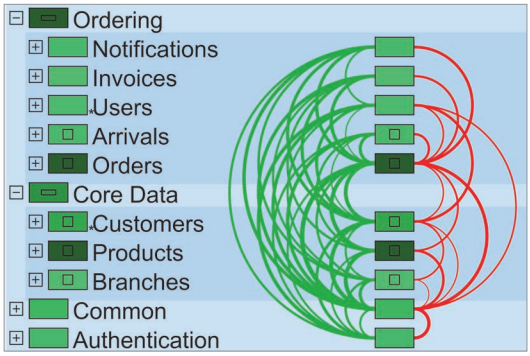


Abb. 7–24
Aufteilung in zwei Microservices

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.

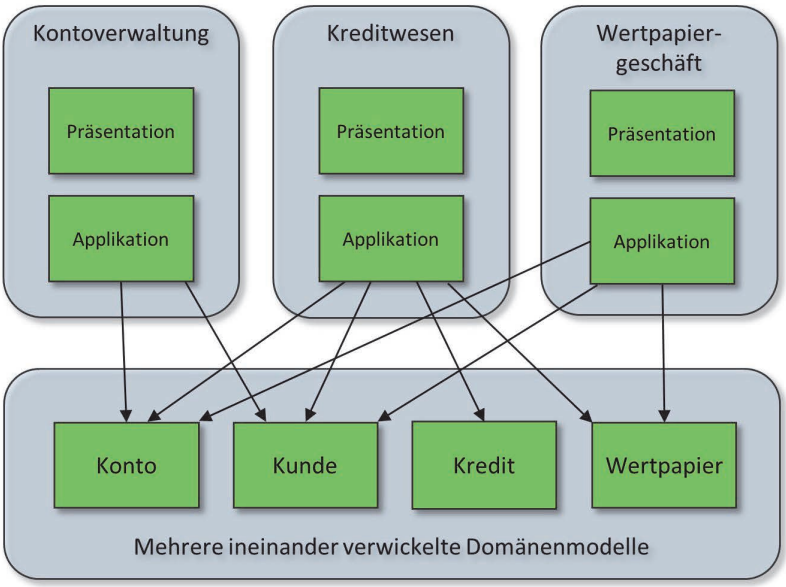
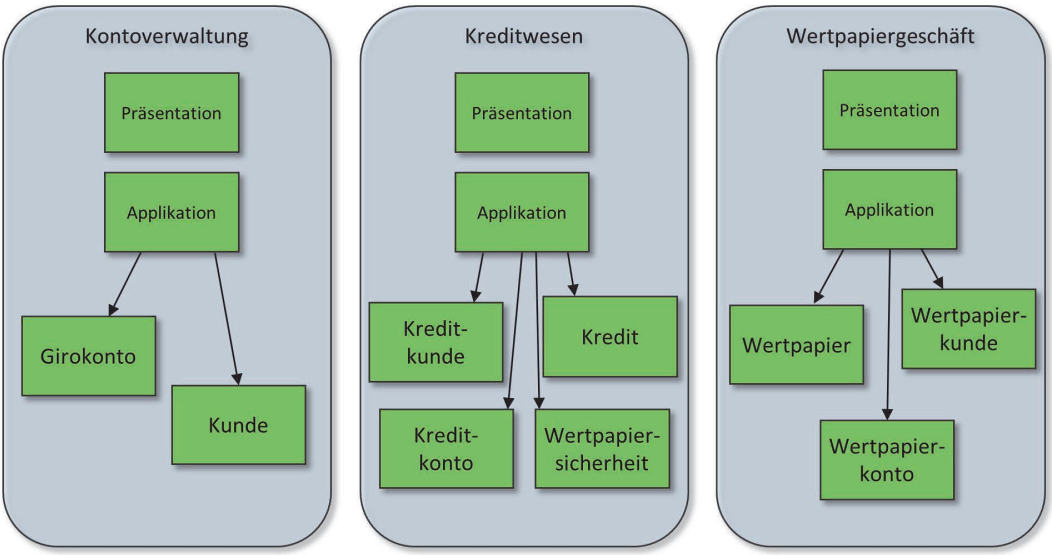


Abb. 7–25
Monolith mit einem großen Domänenmodell

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.