HANSER

Leseprobe

zu

Grundlagen des modularen Softwareentwurfs

von Herbert Dowalil

ISBN (Buch): 978-3-446-45509-2
ISBN (E-Book): 978-3-446-45600-6

Weitere Informationen und Bestellungen unter
http://www.hanser-fachbuch.de/
sowie im Buchhandel

Inhalt

Vorwort

Motivation

„Entwickler an die Macht!“ Selbstbewusst läutete das Java Magazin im September 2016 [Zör16] eine neue Ära der Softwarearchitektur ein. Gerade und vor allem durch den Trend hin zu Microservices, so der Tenor, würde die Softwarearchitektur immer mehr zum Basisskill werden und eine eigene Architektenrolle überflüssig machen. Was ist von dieser Ansage zu halten? Ich nehme hier gleich vorweg, dass ich diese Ansicht weitgehend teile. Softwarearchitektur ist nämlich gar nicht so schwierig zu verstehen und umzusetzen. Bei der Demokratisierung des Themas gilt es aber vorsichtig zu sein. Ich sehe dabei auf der einen Seite die große Gefahr, dass es im Zuge dessen falsch verstanden oder im Extremfall zunächst bagatellisiert und danach weggelassen wird. Andererseits, und in diese Falle würde unsere Branche nicht zum ersten Mal tappen, könnte es wieder passieren, dass einige wenige Muster so in den Vordergrund rücken, dass diese eine gewisse Eigendynamik entwickeln und im Endeffekt die Architekturarbeit ersetzen. So geschehen zu Beginn des Jahrtausends im Zuge des Booms der SOA 1.0, die heute noch von einigen als Ziel mit Selbstzweck betrieben wird. Ähnliches konnte man auch in der Welt der Softwaredesigns beobachten, wie den ehemals gängigen Core JEE Pattern und deren Glaube daran, dass ganz besonders viele Schichten ganz besonders viel Nutzen bringen würden.

Microservices, als der zurzeit aktuelle Hype, machen hier leider keine Ausnahme. Ins Vokabular vieler sind sie bereits als Ersatz für Softwarearchitektur eingegangen. Und natürlich als neues Muster, welches angeblich absolut und immer perfekt passen soll. So sinnvoll sie in einigen Anwendungsfällen sein mögen, ein Allheilmittel stellen sie bestimmt nicht dar, dazu mehr in Kapitel 7. Inzwischen ist es ja sogar schon so, dass die diversen großen Beratungsfirmen die Microservices als neue Einnahmequelle für sich entdeckt haben und nun dieses zweite Muster parallel zur SOA 1.0 um teure Beratertagsätze als absolute Wahrheit verkaufen. Ich sehe hier leider durchaus die Gefahr, dass sich auch um das Thema Microservices, obwohl den Mustern der SOA 1.0 natürlich meilenweit voraus, ein neuer Cargo Cult entwickeln könnte, und das wäre wirklich das Letzte, was unsere Branche noch braucht.

Eine andere Tendenz, welche ich beobachte, ist die der Verwässerung des Themas Softwarearchitektur. In diesem Buch beschränke ich mich auf den Hauptaspekt, nämlich den Bau effizienter und langfristig wartbarer Strukturen beim Softwareentwurf, auch als Komponenten oder Module bekannt. Überraschend oft wird dies aber nach wie vor übersehen und man beschränkt sich bei der Architekturarbeit auf Dinge, die bestenfalls Randthemen

der Architektur sind, wie Lead Development, Requirements Engineering, Kommunikation oder Technologie Coaching. All diese immer noch weit verbreiteten Missverständnisse zum Thema Softwarearchitektur, welche einem das Arbeitsleben als Architekt manchmal ganz schön schwermachen, waren es unter dem Strich dann auch, welche mich motivierten, dieses Buch zu schreiben.

Können wir diesmal etwas besser machen? Auf jeden Fall! Zunächst einmal ist es wichtig, nicht in Mustern (Pattern) zu denken und diese als reines Mittel zum Zweck zu sehen. Bei jedem Muster, welches man einzusetzen gedenkt, muss man sich immer die Frage stellen, was man damit eigentlich erreichen möchte, und damit, ob es für den jeweiligen Anwendungsfall überhaupt geeignet ist. Bringt es auch den Mehrwert, den man sich von seinem Einsatz erwartet? Die Frage lautet demnach: Was kann man denn vom Thema Softwarearchitektur erwarten? Worum geht es dabei überhaupt? Ich möchte Sie hiermit einladen, diese Fragen durch die Lektüre dieses Buchs gemeinsam mit mir zu beantworten. Ich wünsche mir nämlich für die Zukunft, dass der Weg der Demokratisierung des Themas Softwarearchitektur anhält. Dabei aber auch, dass es im Zuge dessen nicht bagatellisiert oder falsch verstanden wird.

Für die Demokratisierung der Softwarearchitektur nur bedingt geeignet, sehe ich die Aspekte der Makro-Architektur, worauf ich in Kapitel 6 näher eingehen werde. Während ich keinen Sinn darin sehe, die Aufgaben der Mikro-Architektur in einem zentralen Team zu kapseln, wird einem bei den Themen der Makro-Architektur manchmal einfach nichts anderes übrigbleiben. Je mehr das Thema allerdings an der Basis angekommen ist, desto weniger zentrale Arbeit wird dafür nötig sein. Jedenfalls wird es uns Softwarearchitekten hoffentlich gelingen, das Thema Makro-Architektur in Zukunft nicht mehr so stiefmütterlich zu behandeln, wie dies momentan der Fall ist. Denn wenn wir das tun, überlassen wir es genau jenen Vertretern, die nur darauf warten, ein Tool zu verkaufen, welches im Endeffekt nur die Symptome misslungener Makro-Architekturen etwas lindern wird, anstatt für echte Abhilfe zu sorgen.

Worauf sollte man achten, wenn man das Thema Softwarearchitektur im Team etablieren möchte? Es geht dabei um eine objektive Betrachtung des Themas und darum, dass nicht eine ebenso eingeschränkte wie subjektive Sicht darauf in das Selbstverständnis der Mitarbeiter übergeht. Beginnen sollte man dabei auf jeden Fall mit den Aspekten der Mikro-Architektur, da diese zunächst einfacher zu erlernen und zu etablieren sind, bevor man sich an das abstraktere Thema der Makro-Architektur heranwagt. Es ist schlichtweg nicht denkbar, jemandem die Verantwortung für eine ganze Systemlandschaft zu übergeben, der noch niemals eine größere Menge Code mit seinem Team strukturiert hat. Nur dabei kann man miterleben, welche Herausforderung die Wartung einer komplexen und stetig wachsenden Codebasis mit sich bringt. Vor allem kann man aber nur dabei auch noch relativ kurzfristig reagieren, sollte einmal etwas nicht zu den erwarteten Ergebnissen führen.

Zielgruppe

Für wen ist dieses Buch von Interesse? Natürlich zunächst einmal für alle Developer oder Architekten, welche etwas mehr über den modularen Softwareentwurf erfahren möchten. Sowie jede handelnde Person in einem typischen Softwareentwicklungs-Projekt, welche mit diesen Rollen zu tun hat, wie Projektleiter, Scrum-Master oder Requirements Engineer. Dazu natürlich auch alle Manager und Führungskräfte, die sich die Frage stellen, wie denn

mit der ständig weiterwachsenden Systemlandschaft am besten umzugehen wäre, oder wenn diese ihre bestehende SOA-1.0-Strategie auf ihre Sinnhaftigkeit hinterfragen möchten.

Über dieses Buch

Zum Inhalt sei vorweg noch so viel gesagt, dass ich den Aufbau und die Reihenfolge der Themen ganz bewusst gewählt habe. Ich glaube nämlich, dass es wichtig ist, zunächst zu verstehen, worum es bei Softwarearchitektur überhaupt geht und welche Ziele man damit verfolgt. Dies bietet einem dabei das Rüstzeug, die Bedeutung der einzelnen Muster und ihre sinnvolle Anwendbarkeit einzuschätzen. Pattern, welche als Mittel zum Zweck betrieben werden, sind nämlich bereits viel zu oft anzutreffen und es handelt sich dabei um etwas, wozu ich auf keinen Fall noch weiter beitragen möchte.

Bei den Grafiken greife ich meist auf einen recht plakativen Stil der „Bubbles and Arrows" zurück. Die Spitze eines Pfeils zeigt dabei von einer Komponente aus in die Richtung zu der Komponente, zu der diese eine Abhängigkeit besitzt. Mir sind die Vorteile eines Standards wie UML natürlich voll und ganz bewusst. Ich möchte allerdings mit diesem Buch keine exakte Dokumentation oder Implementierungsvorgabe liefern, sondern eben gewisse Inhalte möglichst gut rüberbringen, und dafür ist diese Art und Weise der Darstellung einfach besser geeignet. Codebeispiele führe ich fast ausschließlich in Java an, einerseits, weil es im Moment die am weitesten verbreitete Programmiersprache ist, und andererseits, weil ich selbst diese Sprache am besten beherrsche. Ich bin mir sicher, dass jemand, der andere Programmiersprachen wie C# oder JavaScript beherrscht, sich in diesen Beispielen auch sehr gut zurechtfinden wird.

Eine der Schwächen unserer Branche und eine der Hauptquellen für viele Missverständnisse in unserem Berufsleben ist bestimmt die Tatsache, dass viele der Begriffe, welche wir tagtäglich verwenden, nicht genauer definiert sind. Aus diesem Grund gibt es am Ende dieses Buchs ein eigenes Glossar, welches für viele dieser Begriffe, welche auch in diesem Buch verwendet werden, eine Definition bietet, was genau darunter zu verstehen ist. Im Zweifelsfall bitte ich Sie, einfach dort nachzuschlagen.

Auf die „Randthemen" der Architektur werde ich dabei wenig bis gar nicht eingehen. Es ist schlichtweg eine Definitionsfrage, ob ein Thema wie Technologieauswahl noch zur Disziplin der Softwarearchitektur zu zählen ist oder nicht. Primär geht es mir darum, softwareintensive Systeme zu strukturieren, und genau darauf wird auch der Fokus dieses Buchs liegen. Eine detaillierte Darstellung, welche Messaging-Infrastrukturen (ob Bus, Broker oder ESB) es am Markt gibt und welche Vor- und Nachteile diese haben, würde alleine schon ein ganzes Buch füllen. Dies überlasse ich gerne anderen, was das betrifft berufeneren Autoren. Für mich besitzen Themen wie eben Technologie zwar eine starke Wechselwirkung mit dem Thema Architektur, sie gehören aber nicht zum Thema Architektur selbst. Zur Abgrenzung noch die Aspekte, die man üblicherweise im Umfeld des Softwareengineerings antrifft, und wie ich die Abgrenzung bzw. Wechselwirkung mit dem Thema Architektur bzw. Modularisierung sehe:

Architektur/Modularisierung

Beim Thema Architektur geht es in erster Linie um die Einteilung einer Software in seine einzelnen Komponenten bzw. Module. Dieses Buch wird fast ausschließlich davon handeln. Und zwar sowohl auf Mikro- als auch auf Makro-Architekturebene.

Requirements Engineering

Um zu wissen, welche Aufgaben eine Software zu erfüllen hat, ist es wichtig, die Wünsche des Auftraggebers zu verstehen. Da dies nicht so einfach ist, wie ein Laie meinen könnte, gibt es eine eigene Disziplin dafür. Die so erhobenen Anforderungen sind natürlich wichtig für den Entwurf einer angemessenen Modulstruktur, wobei die Funktionalen Anforderungen darauf mehr Einfluss haben als die Nichtfunktionalen Anforderungen! Als Architekt werden Sie also nicht darum herumkommen, die funktionalen/fachlichen Requirements in ihre Einzelteile zu zerlegen und deren Abhängigkeiten untereinander zu verstehen oder zumindest bei diesem Prozess zu unterstützen.

Technologie/Betrieb

Die Auswahl der einzelnen Technologien wie Frameworks ist so eng mit dem späteren Betrieb der Software verzahnt, dass ich dieses Thema gerne als ein gemeinsames betrachte. Es wird dabei sehr von den Nichtfunktionalen Anforderungen beeinflusst. Es besteht außerdem eine hohe Wechselwirkung mit dem Thema Architektur, da der Entwurf mancher Strukturen, Abläufe und Kopplungen nicht so ohne Weiteres mit jeder Technologie möglich ist. So könnte man aufgrund der Nichtfunktionalen Anforderung in Bezug auf hohe Skalierbarkeit eine Plattform wie vert.x im Zusammenspiel mit RxJava wählen, was dann die Möglichkeit zum Bau sehr feingranularer Strukturen mit sich bringt bzw. sogar nötig macht.

Management, Leitung, Kommunikation

Bei einem so hohen Vernetzungsgrad der einzelnen Themen ist es natürlich unerlässlich, im Zuge der Erstellung einer Software viel zu kommunizieren. Nicht selten wird diese Ehre automatisch dem Architekten zuteil. Es ist aber genauso denkbar, dass diese Aufgabe von einem versierten Projektleiter erledigt wird.

Development

Die Entwicklung der Software selbst ist eine von manchen Unternehmen leider noch immer unterschätzte Disziplin. Dabei hat dann oft das bürokratische Rundherum den Blick auf die eigentliche Aufgabe so verschleiert, dass man zu wenig Augenmerk darauf legt. Die Aufgabe des Developments ist es jedenfalls, die Anforderungen umzusetzen, und dies in Abstimmung mit der Architektur, wobei die Mikro-Architekturaspekte dabei idealerweise vom Development-Team selbst erledigt werden.

Wie auch immer, sollten Sie bei der Umsetzung eines der in diesem Buch behandelten Themen Unterstützung benötigen, so gibt es Leute wie mich, die Ihnen dabei gerne beratend oder mit einem gezielten Training zur Verfügung stehen.

Danksagung

Danken möchte ich zunächst einmal meiner Familie, beginnend bei meinen Eltern, welche mir als kleinem Kind meinen großen Wunsch nach einem Computer erfüllt hatten, und das, obwohl sie nie viel Geld hatten und Computer damals wirklich noch sehr, sehr teuer waren. Dann meiner lieben Frau, welche mich jederzeit bei diesem Buchprojekt unterstützt hat, auch wenn das Leben für berufstätige Eltern einer kleinen Tochter nicht selten stressig ist und am Anfang natürlich noch keineswegs klar war, ob es überhaupt jemals zu einer Veröffentlichung kommen würde. Dann natürlich meiner entzückenden kleinen Tochter, die es immer wieder akzeptierte, dass Papa, anstatt mit ihr zu spielen, sich lieber das Notebook schnappte, um an seinem Buch weiterzuarbeiten.

Dann gilt mein besonderer Dank noch meinen Freunden Gernot Starke und Alexander von Zitzewitz, da sie mich mit ihrem Feedback zu meinem Content ermutigt haben, mich als Autor zu versuchen. Meinem ehemaligen Boss Mohammad Kabiri, weil er es mir immer ermöglicht hat, mich Schritt für Schritt weiterzuentwickeln, auch wenn das Umfeld dafür nicht immer ideal war. Und zu guter Letzt noch vielen meiner vormaligen Kollegen, allen voran Michael Koitz, Miroslaw Szelag und Sebastian Bicchi (sec-research.com), da sie auch bei hoher Arbeitsbelastung immer die Zeit für mich fanden, um als Freunde für mich da zu sein, und auch immer ein offenes Ohr für Fachsimpeleien hatten, die mir halfen, die Inhalte für dieses Buch zu entwickeln. Philipp Krenn von Elastic gilt mein Dank für die Prüfung des Abschnitts über Skalierung und Sharding von Datenbanken. Last but not least möchte ich noch Brigitte Bauer-Schiewek, Irene Weilhart und Petra Kienle vom Carl Hanser Verlag danken für die prima Zusammenarbeit im Zuge des Lektorats.

Der Autor

Herbert Dowalil, Jahrgang 1976, ist glücklich verheiratet und lebt mit seiner Frau und der gemeinsamen Tochter im Großraum Wien. Er begann bereits zu seiner Grundschulzeit in den 80er-Jahren des letzten Jahrhunderts damit, sich selbst das Programmieren beizubringen. Dabei begann er mit Acorn Electron Basic, um später Turbo Pascal, C und C++ und zuletzt im 21. Jahrhundert Java zu verwenden. Dabei legte er sein Hauptaugenmerk bald auf die Frage, was gut wartbare und robuste Systeme von den Problemfällen, welche man in der Branche ja leider zu oft antrifft, unterscheidet. Heute ist er als selbstständiger Berater, Trainer und Autor zu Themen wie Vermessung von Software in Kennzahlen, Mikro- bzw. Makro-Architekturen, Design Pattern und Prinzipien des modularen Softwareentwurfs tätig. Spezialisiert ist er u. a. auf Refactorings und Evolution von Legacy-Architekturen.

- Kontakt: hdowalil@gmail.com
- Twitter: @hdowalil

Im World Wide Web:

- *http://improve-it.solutions*
- *http://software-architektur.training*

widersprechen. Das „Closed“ dieses Prinzips entspricht nämlich weitgehend dem Information Hiding Principle. Wie soll es also möglich sein, gezielt so viel wie möglich zu verbergen und gleichzeitig erweiterbar zu sein? In den folgenden Kapiteln werden wir dafür einige Möglichkeiten kennenlernen.

2.5 Lose Kopplung

Keine Frage, egal wie gut man einen Architekturplan hinbekommt, es wird zwangsläufig zu Abhängigkeiten zwischen den einzelnen Bausteinen kommen. Einfach weil die einzelnen Features und Bausteine eines gesamten Systems nicht völlig isoliert voneinander existieren können. Bei der Festlegung einer solchen Verbindung zweier Bausteine, Kopplung genannt, ist es immer empfehlenswert, dies auf möglichst leichtgewichtige Art und Weise zu tun. Dabei kann dies einerseits bedeuten, tendenziell eine eher geringe Anzahl solcher Verbindungen zu haben, andererseits aber auch, dass im Zuge dieser Abhängigkeiten eine Komponente möglichst wenige Annahmen von der anderen Komponente treffen sollte. Üblicherweise geht es bei diesen Annahmen um folgende Punkte:

Laufzeitumgebung, Ausführungsort

Die andere Komponente muss auf derselben Maschine laufen, damit die Integration möglich ist. In so einem Fall wäre es schwierig, Fehler, welche sich auf den stabilen Betrieb der Laufzeitumgebung auswirken (Verbrauch von zu viel Speicher oder CPU), von den anderen Komponenten zu isolieren. Es bestünde dann die Gefahr, dass das gesamte System in Mitleidenschaft gezogen wird. Im nicht so extremen Fall gibt es gewisse andere Formen der Einschränkung des operativen Systems, die im Falle einer Wiederverwendung für die andere Komponente gelten würden.

Technologie

Einschränkungen bei der Technologiewahl der angebundenen Komponente. Dabei kann es sich um Einschränkungen handeln, die noch einen gewissen Spielraum zulassen (wie alle Systeme, welche eine REST API konsumieren können). Im Extremfall würde die andere Komponente genau dieselbe Technologie verwenden müssen.

Zeit

Die andere Komponente kann beispielsweise zu gewissen Zeitpunkten nicht angesprochen werden, bietet aber keine asynchrone Variante ihrer Schnittstelle an. Wenn die aufrufende Komponente die fremde Schnittstelle benötigt, um mit einer Verarbeitung fortfahren zu können, wäre sie somit für diese Zeit blockiert.

Daten und Formate

Dabei kann es bei der Kommunikation gewisse allgemeine Einschränkungen geben in Bezug auf die möglichen Datenformate, die geparst und verstanden werden, beispielsweise Datumsformate oder Header, die gesetzt werden müssen. Hauptsächlich bestehen diese Kopplungen aber einfach aufgrund des konkreten Datenformats der Schnittstelle, welche verwendet wird. Wenn das fremde Schnittstellenformat überall in der eigenen Komponente verwendet wird, wird so eine Kopplung sogar noch stärker. Durch Integration mit dem Composite-UI-Pattern oder Verwendung des Tolerant-Reader-Musters kann man diese Form der Kopplung teilweise abschwächen.

Wenn nun ein Service einen anderen über eine SOAP-Schnittstelle aufruft, so ist dies eine losere Kopplung, als es ein lokaler Aufruf mit den üblichen Mitteln der jeweiligen Programmiersprache wäre. Der direkte Aufruf würde nämlich eine Kopplung bedeuten, was den Ausführungsort der anderen Komponente angeht, als auch was die Technologie betrifft. Wohingegen die Kommunikation über das Netzwerk, wie beim Aufruf des Webservices, keinerlei Annahmen in Bezug auf den Ausführungsort trifft und technologisch nur insoweit einschränkend ist, als die Zieltechnologie in der Lage sein muss, Webservices als Schnittstelle konsumieren zu können. Trotzdem ist es nicht wahr, was diesbezüglich nach wie vor oft behauptet wird: Dass man damit prinzipiell bereits eine lose Kopplung erreicht hätte. Nach wie vor gibt es dabei die Einschränkung in Bezug auf die zeitliche Komponente, weil diese Form der Integration eine synchrone ist. Außerdem gibt es Abhängigkeiten bezüglich des Schnittstellenformats. Am besten werfen wir gleich mal einen Blick auf unterschiedliche Formen der Integration, und zwar absteigend, was die Intensität der Kopplung angeht.

2.5.1 Code Reuse

Grad der Kopplung: sehr groß (Laufzeitumgebung, Technologie, Zeit, Datenformate)

Davon sprechen wir, wenn z. B. in Java direkt eine Methode einer anderen Klasse aufgerufen und die andere Komponente als .jar-File wiederverwendet wird. Das ist die stärkste Form der Kopplung und sollte nur in der Mikro-Architekturebene Anwendung finden bzw. beim Bau von Monolithen.

2.5.2 Datenbankintegration – gemeinsames Datenmodell

Grad der Kopplung: groß (Technologie, Datenformate)

Dabei greifen verschiedene Systeme direkt auf dieselben Tabellen einer Datenbank zu (Bild 2.3). Wir sind dabei zwar zeitlich voneinander entkoppelt, treffen aber sonst alle möglichen Annahmen. So müssen alle beteiligten Systeme mit dieser Datenbanktechnologie kommunizieren können und haben auch Abhängigkeiten zum konkreten Datenmodell.

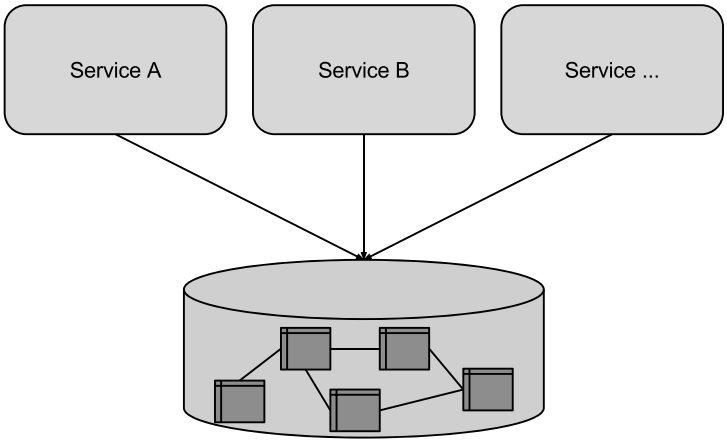

Bild 2.3 Unterschiedliche Services teilen sich dasselbe Datenmodell in derselben Datenbank

2.5.3 Datenbankintegration – selbe Datenbank, unterschiedliche Datenmodelle

Grad der Kopplung: mittel (Technologie)

Eine alternative Form der Datenbankintegration, bei der sich die einzelnen Services nicht mehr dasselbe Datenmodell teilen, liegt darin, wenn zwar noch dieselbe Datenbank genützt wird, aber jeder dort sein eigenes Datenmodell besitzt (Bild 2.4). Dadurch ist es dann einfach, Konsistenz zwischen den Services durch atomare Transaktionen zu gewährleisten. Allerdings wird man mit dieser Form der Integration kaum auskommen, weil auf diese Art und Weise alleine ein Service keine Aktionen in einem anderen Service auslösen kann, wodurch es bei der Datenbankintegration meist noch zu anderen Formen der Interaktion kommt (wie RPC).

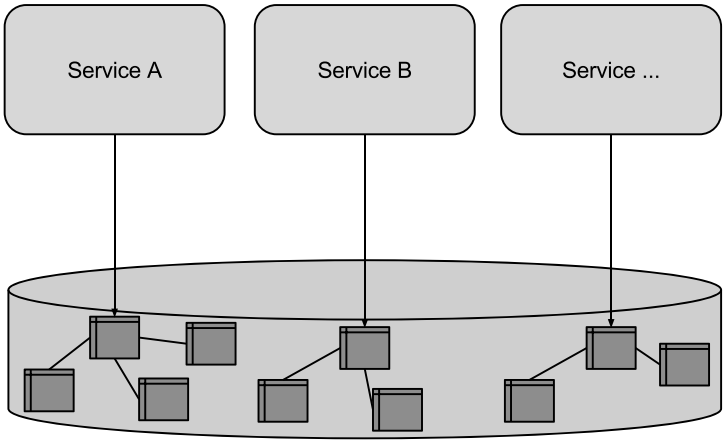

Bild 2.4 Unterschiedliche Services benutzen verschiedene Datenmodelle in derselben Datenbank

2.5.4 Synchroner Remote Procedure Call

Grad der Kopplung: mittel (Zeit, Datenformate)

Bei Remote Procedure Calls (RPC) kommuniziert eine Komponente mit einer anderen durch eine mehr (SOAP) oder weniger (REST) standardisierte Schnittstelle über das Netzwerk. Hier muss die andere Komponente nach wie vor zur selben Zeit verfügbar sein (Annahme: Zeit), dafür ist es uns egal, mit welcher Technologie die Gegenstelle gebaut wurde.

Völlig los werden Sie übrigens die technologische Kopplung mit einem synchronen RPC aber ebenfalls nicht. So gibt es in Java die sogenannte Remote Method Invocation, kurz RMI, welche zwar einen synchronen Call über das Netzwerk ermöglicht, aber immer noch sehr einschränkend auf die Gegenstelle wirkt. Sie können sich dabei zwar weitestgehend die Hardwareplattform des Services aussuchen, sind aber eingeschränkt auf die Java-Technologie, welche darauf lauffähig sein muss. Bei anderen Formen des synchronen RPC ist diese Einschränkung in irgendeiner Form eigentlich fast immer ebenfalls vorhanden, aber nicht gleich so offensichtlich. Wenn Sie eine SOAP oder REST API zur Verfügung stellen, unterliegen die potenziellen Consumer nach wie vor der technologischen Einschränkung, dass sie diese Art von Kommunikation beherrschen müssen. Bei einem Cobol-Entwickler, welcher Code für einen Mainframe schreibt, wird man mit einer WSDL wenig Euphorie auslösen. Üblicherweise behilft man sich dann mit einem Message-Bus, welcher auch die zeitliche Kopplung entfernen kann.

2.5.5 Datenreplikation

Grad der Kopplung: gering bis mittel (Datenformate)

Daten werden nicht wie beim RPC per Remote-Schnittstelle geholt, sondern asynchron repliziert. Damit wäre die zeitliche Kopplung entfernt. Der Aufwand lohnt sich aber meiner Meinung nach nur, wenn es einen triftigen Grund für die Aufgabe der zeitlichen Kopplung gibt. So kann z. B. der Anspruch an ein System, was die Verfügbarkeit angeht, so hoch sein, dass man sich entschließt, die Daten redundant zu halten. Außerdem kann man damit nur Daten einer anderen Komponente wiederverwenden, aber keinerlei Logik, wie eine Berechnung von was auch immer. Zur Umsetzung gibt es verschiedene Möglichkeiten, welche im Abschnitt 11.1 kurz beschrieben werden.

2.5.6 Messaging

Grad der Kopplung: gering (Datenformate)

Während Remote Procedure Calls darüber definiert sind, dass sie synchron erfolgen, geht es beim Messaging um asynchrone Kommunikation. Man reduziert die Abhängigkeit zwischen Sender und Empfänger um die zeitliche Komponente. Das bedeutet, dass Sender und Empfänger nicht gleichzeitig online sein müssen, es wird also keine zeitliche Annahme getroffen. Prinzipiell unterscheidet man dabei zwischen Message-Bus und Message-Broker. Während ein Bus agnostisch darüber ist, wer Nachrichten publiziert und wer welche konsu-

miert, ist ein Broker darüber hinaus für das Routing der Nachrichten zu den konkreten Empfängern zuständig. Ein Broker implementiert dabei wenigstens die folgenden beiden Routing-Muster:

Command

Hinter Commands steht der Wunsch des Absenders, eine bestimmte Aktion auszuführen. Ein Message-Broker ist dabei als dezidierte Indirektion dafür zuständig, einen solchen Command an den dafür zuständigen Empfänger weiterzuleiten. Dabei gibt es üblicherweise genau einen Empfänger der Nachricht, aber durchaus verschiedene Endpoints, die einen solchen Command erstellen dürfen. In manchen Fällen kann der konkrete Empfänger auch mittels eines gewissen Regelwerks von der Infrastruktur ermittelt werden. So gibt es in RabbitMQ die Möglichkeit, den Empfänger abhängig von einem sogenannten Topic der Nachricht auszuwählen (Topic Based Routing).

Events

Bei Events informiert ein Absender darüber, dass ein bestimmtes Ereignis stattgefunden hat. Interessierte Empfänger können solche Nachrichten abonnieren und werden dann bei Auftreten eines der abonnierten Ereignisse informiert. Der Absender ist dabei agnostisch darüber, wer und wie viele Empfänger seine Nachrichten abonniert haben. Es gibt dabei üblicherweise immer genau einen Absender einer solchen Nachricht, während sie von mehreren Endpoints empfangen werden kann.

2.5.7 Composite-UI

Grad der Kopplung: Sehr gering (Zeit)

Die einfachste Art und Weise, um Systeme zu integrieren, bietet Ihnen das User-Interface selbst. Wenn es für den jeweiligen Anwendungsfall sinnvoll ist, sollten Sie immer bestrebt sein, die Kopplung in der UI-Ebene herzustellen. Wobei das prinzipiell auf dem Server oder am Client und somit meist im Browser erfolgen kann.

Natürlich kann man es aber mit dieser Form der Optimierung auch übertreiben. Es kann sein, dass eine UI-Integration komplizierter ist als ein synchroner RPC, einfach weil beispielsweise die Aufwände zur Angleichung der User-Interfaces (wie der DOM-Struktur und der CSS-Klassen bei einer Webapplikation) größer sind, als sie es für einen synchronen RPC wären. Wie so vieles ist also auch das nur eine Abwägung des Pros gegen das Contra. Nähere Informationen und Beispiele zur Umsetzung einer solchen UI-Integration gibt es in Abschnitt 11.2.

Wenn Sie zwei Komponenten integrieren, so tun Sie das immer auf die Art und Weise, welche die geringstmögliche Kopplung bedeutet! Bei hohem Abstraktionsgrad wie der Makro-Architektur ist dies besonders wichtig! In der Mikro-Architektur oder Designebene kann dagegen auch ein hoher Grad an Kopplung (wie durch Code Reuse) noch akzeptabel sein!

2.6 Hohe Kohäsion

Innerhalb einer Komponente ist es natürlich nichts Schlechtes, wenn es zu Dependencies der einzelnen Subbausteine kommt. Im Gegenteil, dies ist ein Indiz dafür, dass die Abgrenzung einer Komponente gegen den Rest des Systems an den richtigen Stellen gemacht wurde. Wenn Sie dies gut und ausgewogen hinbekommen, so spricht man von einer hohen Kohäsion oder Zusammengehörigkeit der Subbestandteile eines Bausteins. Bei Kohäsion geht es also um Abhängigkeiten bzw. Kopplung innerhalb gewisser Bausteingrenzen. Hier ist es dann natürlich auch nicht unüblich, wenn es zu enger Kopplung kommt.

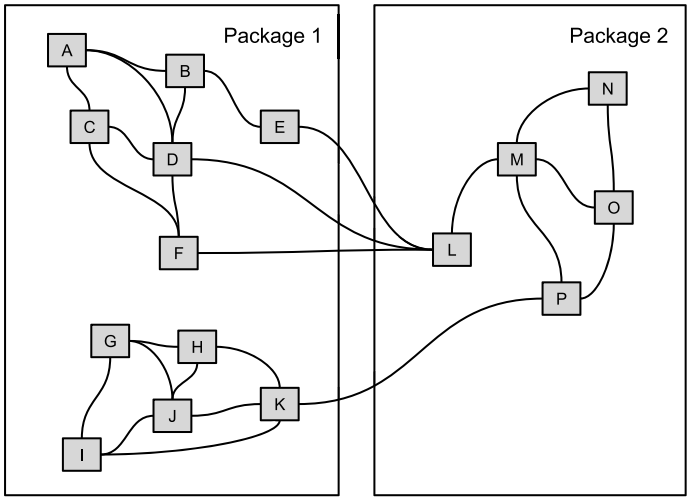

Bild 2.5 Schlechte Kohäsion

Wenn Sie einen Blick auf Bild 2.5 werfen, dann sehen Sie vermutlich auf den ersten Blick, dass diese Struktur wenig Sinn ergibt. Es existieren drei Abhängigkeiten zu Klasse L in Package 2 von Package 1 aus, während es nur eine Abhängigkeit zu L innerhalb des Package 2 gibt. Dann scheint es in Package 1 zwei Teilbereiche zu geben, die dort wiederum nichts miteinander zu tun haben. Hier empfiehlt sich dann ein Refactoring, hin zu einer Struktur, wie sie in Bild 2.6 Gute Kohäsion dargestellt ist, wo es dann drei Packages gibt, welche jeweils eine hohe innere Kohäsion aufweisen und nur an wenigen Stellen Abhängigkeiten zu den anderen Packages haben. Zunächst verschieben wir dafür die Klasse L in das Package 1. Danach trennen wir noch den in der Luft hängenden Teil von Package 1 als neues Package 3 ab.

Wir sehen also, dass eine Erhöhung der Kohäsion innerhalb der einzelnen Packages automatisch zu einer Verringerung der Kopplung zwischen diesen führt. Es gibt also eine Wechselwirkung zwischen den beiden Prinzipien der losen Kopplung und der hohen Kohäsion, wodurch diese manchmal auch als ein Prinzip zusammengefasst werden, welches dann als „High Cohesion And Low Coupling“ bezeichnet wird.

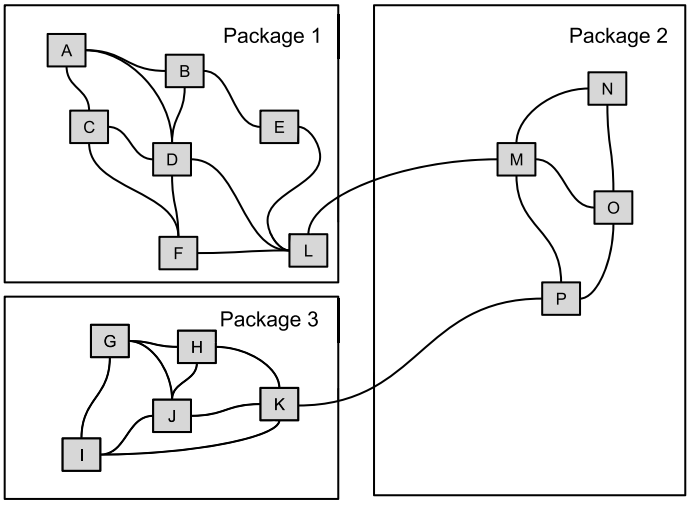

Bild 2.6 Gute Kohäsion

2.7 Separation Of Concerns

Auch bekannt als bzw. eng verwandt mit:

- Single Responsibility Principle: Jeder Komponente wird nur genau eine Aufgabe zu Teil, womit es für jede Komponente nur genau einen Grund geben sollte, diese zu ändern [Mar02].
- Prinzip der Modularität

Separation of Concerns, zu Deutsch etwa Trennung der Angelegenheiten, ist ein Prinzip, welches versucht, eine Vorgabe für jede Art der Modularisierung zu definieren. Es besagt, dass jede Angelegenheit von einem eigenen, dafür zuständigen Baustein abgebildet werden soll. Die einzelnen Aufgaben sollten also nicht x-beliebig auf die Systemstruktur verteilt sein. Verwirrend mag sein, dass diese Definition dabei die folgenden beiden Interpretationsmöglichkeiten zulässt:

- Ein Concern entspricht einem Teilbereich der fachlichen Anforderungen. Die Anforderungen werden runtergebrochen, zerlegt und schlussendlich jeweils in einer eigenen Komponente abgebildet.
- Ein Concern entspricht einer der Aufgaben, welche eine Software zu erledigen hat, um die gewünschten Anforderungen zu erfüllen. Es gäbe demnach beispielsweise eine Komponente für das User-Interface, eine für den Datenbankzugriff und so weiter.

Am besten gehen wir dem auf den Grund, indem wir einen Blick auf die mit Abstand häufigste Form der technischen Strukturierung werfen, nämlich auf einen Monolithen mit Schichtenarchitektur (Bild 2.7). Nehmen wir jetzt an, Sie fügen bei einer der Entitäten von Feature A ein Feld hinzu. Ich bin mir sicher, dass Sie in der Datenbank beginnen werden

und diese Änderung eine Kaskade an Änderungen in den Schichten darüber nach sich ziehen wird, während die anderen vertikalen Schnitte (Feature B, C etc.) davon unberührt bleiben. Es besteht also eine enge Kopplung über die gesamten Datenformate hinweg. Daran kann man erkennen, dass es zwischen den Schichten eine hohe Kopplung (oder eben Kohäsion) gibt, während diese zwischen den fachlichen Schnitten eher gering ist. Daraus schließe ich, dass man zuerst nach fachlichen Kriterien strukturieren sollte, um erst innerhalb dieser Fachlichkeit die technischen Strukturen zu bilden. Ein Concern im Sinne von Separation Of Concerns sollte demnach, so wie in Bild 2.8 dargestellt, in erster Linie fachlicher und nicht technischer Natur sein.

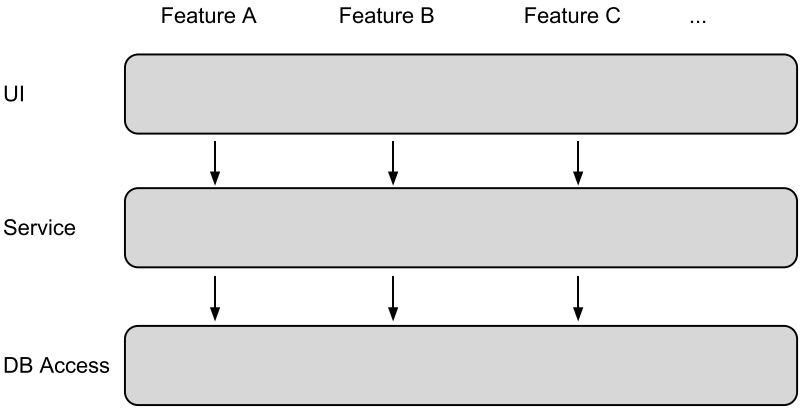

Bild 2.7 Ein System, nach technischen Kriterien in Schichten strukturiert

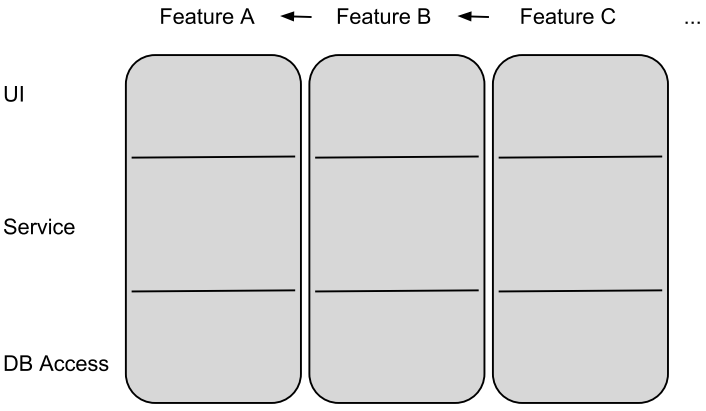

Bild 2.8 Dasselbe System, diesmal in fachlichen Schnitten strukturiert

Wenn Sie ein System in seine Einzelteile zerlegen, tun Sie dies in den oberen Hierarchieebenen am besten nach fachlichen Kriterien. Technische Aspekte sollten dann eher erst in den unteren Ebenen benützt werden, um Strukturen zu bilden.

Eines ist jedenfalls einfach: zu erkennen, ob dieses Prinzip verletzt wurde. Nämlich immer dann, wenn der Auftraggeber nach einer eindeutig fachlich gut abgrenzbaren Änderung der Software verlangt und für diese in Folge mehr als ein Baustein geändert werden muss. Eine solche Änderung könnte z. B. sein, dass keine Lieferkosten mehr ab einem Bestellwert von 100 Euro berechnet werden.

Einen Blick möchte ich noch auf das fast schon klassische Antipattern werfen, mit dem dieses Prinzip immer wieder verletzt wird. Es handelt sich dabei um eine sogenannte „Gottklasse“, wie beispielsweise der Controller, welcher in Bild 2.9 dargestellt ist. Das Problem bei dieser Umsetzung des Controllerablaufs ist, dass Kompetenzen der einzelnen Bausteine in eine zentrale Klasse verschoben werden, wo sie nichts verloren haben. Diesem Antipattern werden wir übrigens später noch einmal im Bereich der Makro-Architektur begegnen, wo ein zentraler Enterprise Service Bus als allumfassender Mediator fungiert.

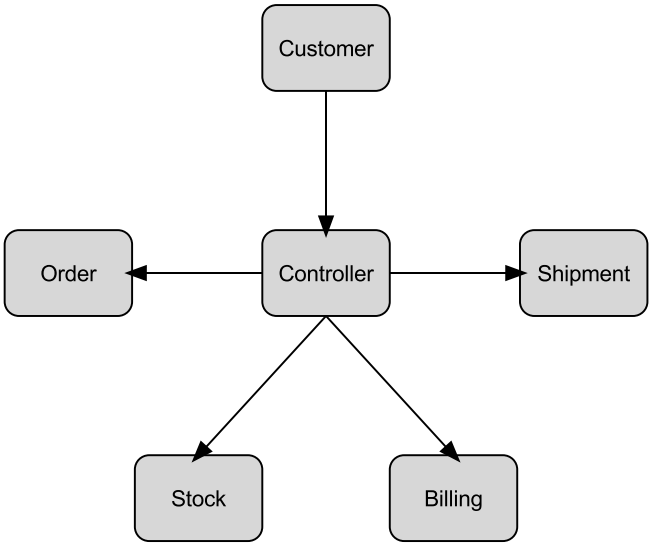

Bild 2.9 Verletzung des Separation-of-Concerns-Prinzips durch eine Gott-Klasse

Die Lösung für dieses Problem ist ebenso einfach wie offensichtlich. Genauso wie beim Prinzip Dumb Pipes and Smart Endpoints, welches wir später noch kennenlernen werden, gehört auch hier die Logik in die jeweils zuständigen Klassen verschoben. Die Umsetzung sollte also vielmehr so aussehen wie in Bild 2.10. Hierbei spricht der Kunde nur mit der Order-Klasse, um eine Bestellung abzugeben, welche gegenüber der Stock-Klasse den Lagerstand prüft. Wenn die Bestellung möglich ist, so wird die weitere Abarbeitung an die Shipment-Klasse delegiert, welche mithilfe der Billing-Klasse eine Rechnung erzeugt und im Endeffekt alles an den Kunden zustellt.

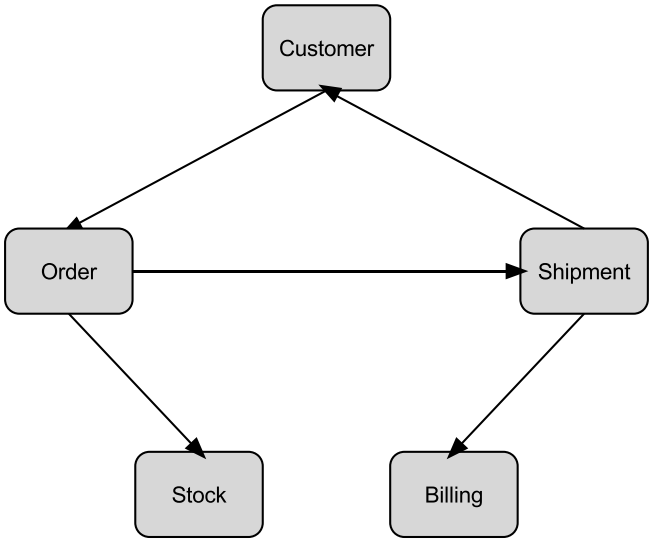

Bild 2.10 Sauber strukturiert und modular

2.8 Hierarchischer Aufbau

Das Prinzip des hierarchischen Aufbaus von Komponenten haben wir bereits in Abschnitt 1.1.1 kennengelernt. Dabei bilden Komponenten in ihrem Zusammenspiel wiederum Komponenten, die ihrerseits wiederum mit anderen Bausteinen auf ihrer Ebene zusammen Komponenten bilden, und so weiter [Sta14]. Im Grunde hat also eine Softwarearchitektur ab einer gewissen Größe, wenn man so möchte, eine fraktale Natur. Beispielhaft ist das in Bild 2.11 dargestellt. Falls Ihnen das noch nicht komplex genug sein sollte, so kann ich Ihnen noch empfehlen, einen Blick auf Bild 2.12 zu werfen. Diese Abbildung stimmt übrigens weitgehend mit der beispielhaften Zerlegung der Versicherungsdomäne in Abschnitt 4.7 überein. Die Aufteilung dort ist natürlich noch nicht vollständig. Komponenten wie CRM stellen Services oder Systeme dar, die sich jeweils noch weiter in einzelne Module zerlegen werden, um selbst ebenfalls den Rahmenbedingungen der Modularität zu folgen.

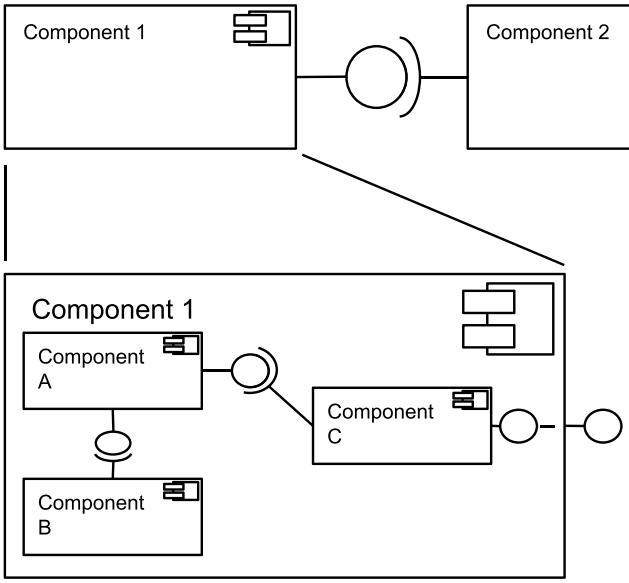

Bild 2.11 Eine Komponente kann selbst wiederum aus anderen Komponenten aufgebaut sein

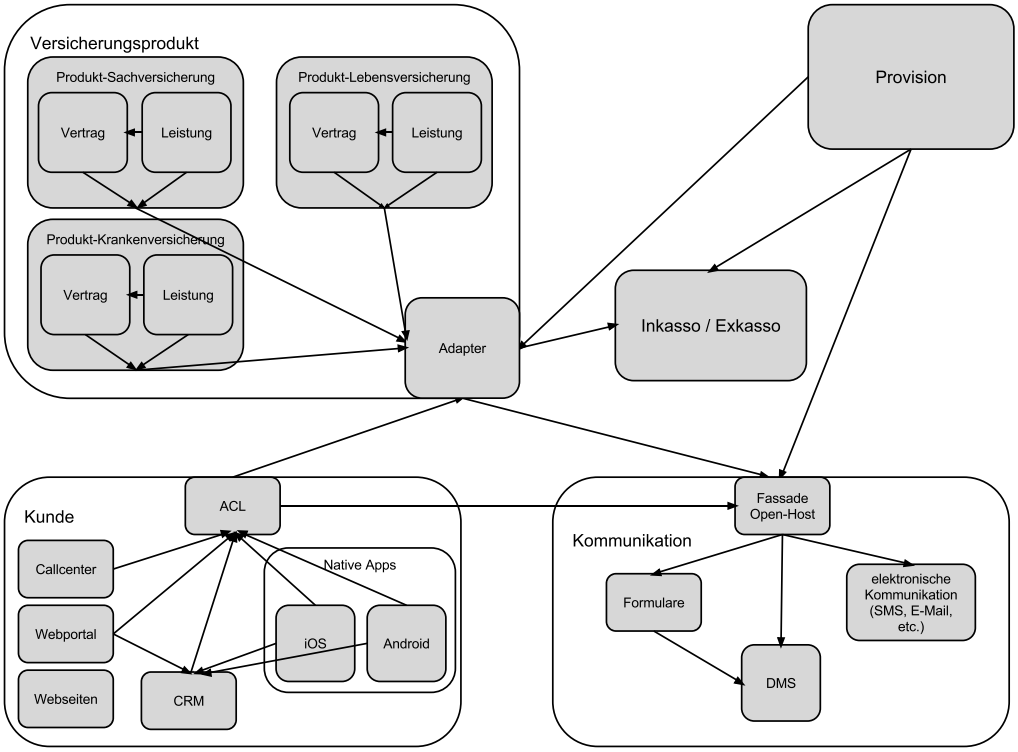

Bild 2.12 Beispielhafte teilweise Aufteilung der Versicherungsdomäne in den oberen drei Abstraktionsebenen

Durch die Anwendung dieses Prinzips werden Architekturen großer Systemlandschaften sehr gut beherrschbar, da unser menschliches Gehirn auf genau diese Art und Weise einen komplexen Sachverhalt begreift. Es zerlegt diesen nämlich einfach immer weiter in seine Teilbereiche [Lil17]. Bedenken Sie nur mal, wie ein Computer Zeit erfasst. Oft wird dafür die Unixzeit verwendet, welche die Millisekunden misst, welche seit dem 1.1.1970 vergangen sind. Wir Menschen könnten mit einer solchen Abstraktionsebene nichts anfangen. Stellen Sie sich vor, Sie würden jemanden nach seinem Geburtstag fragen und er würde diesen in Millisekunden vor bzw. nach einem allgemein anerkannten Referenzzeitpunkt angeben. Man könnte mit so einer Angabe recht wenig anfangen. Daher wechseln wir, wenn es um Zeitabläufe geht, gerne zwischen Größenordnungen hin und her. Wenn Sie den Kellner fragen, wie lange es denn noch dauern wird, bis das Essen serviert wird, dann wird er hoffentlich für seine Antwort eine eher kleine Zeitskala wie Minuten wählen. Wenn Sie sich fragen, wie lange es her ist, dass die Dinosaurier ausstarben, dann sind Sie wiederum gut damit beraten, auf die Abstraktionsebene der Erdzeitalter zu wechseln.

Bild 2.13 Wenn man näher rangeht, offenbart die Natur ihre fraktalen Eigenschaften. Sie wird dabei nicht weniger komplex, sondert zeigt immer mehr Strukturen und Details, aus denen sie aufgebaut ist. (Quelle: *http://panoramas.pictures*)

2.9 Zusammenfassung

Für mich ist es immer wieder erstaunlich, wie gut sich die in diesem Kapitel vorgestellten Prinzipien modularer Architekturen zusammenfassen und auf den Punkt bringen lassen. Falls Sie irgendwann jemandem eine Management Summary zum Thema Softwarearchitektur geben wollen, so möchte ich Ihnen den folgenden Abschnitt empfehlen:

Vereinfacht gesagt geht es bei modularer Softwarearchitektur darum, möglichst die einzelnen fachlichen Themen in eigenen Komponenten zu kapseln, mit klar definierten ein- und ausgehenden Schnittstellen, welche ihre jeweiligen inneren Details der Umsetzung vor der Außenwelt so gut es geht verbergen. Durch diese fachliche Aufteilung kommt es automatisch zu hoher innerer Kohäsion, wodurch die Kopplung dazwischen möglichst lose werden kann. Ab einer gewissen Größe macht es Sinn, diese Strukturen auf mehreren ineinander verschachtelten Hierarchieebenen aufzubauen. Dadurch ist das System auch jederzeit erweiterbar und somit ist es auch gar nicht erst notwendig oder sinnvoll, kommende Anforderungen vorwegzunehmen.

6 Makro-Architektur

„Wer nichts weiß, muss alles glauben."
Marie von Ebner-Eschenbach

In diesem Kapitel möchte ich einen Überblick darüber geben, welche Muster und Vorgehensweisen abseits des DDD (Kapitel 4) und der EAIP (Kapitel 5) zum Thema Makro-Architektur empfehlenswert sind und was inzwischen als überholt gilt. Dabei werde ich zunächst Muster vorstellen, welche aus heutiger Sicht keine Gültigkeit besitzen, und danach vernünftigere Alternativen dazu vorstellen. Dabei werden wir uns auch des Öfteren die Frage stellen, welche Prinzipien mit den jeweiligen Mustern verletzt bzw. umgesetzt werden.

Prinzipiell ist das oberste Ziel einer jeden guten Makro-Architektur dabei, dass ein Software-Entwicklungsprozess, welcher mehrere Teams umfasst, auf Dauer effizient bleibt. So unspektakulär dies auch klingen mag, so sind unkontrolliert gewachsene Makro-Architekturen bestimmt das größte Problem, mit dem viele IT-Landschaften früher oder später konfrontiert werden. Dabei schleicht sich diese Problematik langsam über die Jahre ein, sodass es kaum jemand bemerkt, und oft ist dann niemand in der Lage, die Ursache für die dadurch notwendige, aber auch lähmende Bürokratie zu identifizieren. Das geht nicht selten so weit, dass es zu einer existenziellen Bedrohung für das Unternehmen werden kann. Schließlich ist es im ökonomischen Darwinismus einer Marktwirtschaft nicht egal, wie effizient man ist, sondern eben eine Überlebensfrage!

Die Aufgaben einer Makro-Architektur sind, ganz ähnlich wie es die Mikro-Architektur im Kleinen tut, die Strukturen der obersten Abstraktionsebene zu planen und zu entwickeln. Dabei ist lose Kopplung sogar noch wichtiger als in einer Mikro-Architektur. Dies ist aber in einer Makro-Architektur naturgemäß ungleich schwieriger, da plötzlich mehrere Teams und Projekte betroffen sind. Auch können Fehlentscheidungen nicht so einfach behoben werden, sondern ziehen immer erhebliche Kosten und lange Durchlaufzeiten zur Behebung nach sich. Ein Aspekt, welchen es wiederum nur in der Makro-Architektur gibt, ist der Bedarf zur Abstimmung der einzelnen Teams untereinander. Dies darf allerdings nur passieren, um unnötige Abweichungen zu vermeiden, und niemals, um die einzelnen Teams zu kontrollieren, zu sehr einzuschränken oder um ihnen vorzuschreiben, wie sie ihren Job zu machen hätten.

Probleme durch schlechte Makro-Architekturen schleichen sich langsam in wachsende IT-Landschaften der Unternehmen ein. Oft kann dabei den Problemen nicht mehr die Ursache zugewiesen werden, was dadurch auf lange Sicht zu einer existenziellen Bedrohung für jedes zunächst erfolgreiche Unternehmen führen kann.

Makro-Architektur vs. Kontextsicht

Vielleicht kommt Ihnen das bereits bekannt vor. Schließlich ist es ja nicht unüblich, ja sogar unerlässlich, als Architekt im Zuge eines Projekts die strukturelle Kontextabgrenzung abzuklären, näher zu definieren und zu dokumentieren. Man kümmert sich dann um die Außenbeziehungen, die alle neu zu erstellenden Komponenten haben, aber niemals konkret um das größere Bild des Zusammenspiels auf der oberen strukturellen Abstraktionsebene. In so einem Fall läuft man Gefahr, dieses Thema zu übersehen. Es wäre die klassische „Mikroskopfalle“, in die man getappt ist, wenn man niemals gezielt einen Plan für die Makro-Architektur erstellt und davon ausgeht, dass sich die einzelnen Systeme schon darum kümmern werden. Ich habe jedenfalls noch niemals erlebt, dass sich ein so enorm komplexes Unterfangen von selbst erledigt hätte.

Makro-Architektur und Verteilte Systeme

Im nächsten Kapitel 7 werden Verteilte Systeme bzw. SOA besprochen. Dabei kommt es in einem gewissen Rahmen zu Überschneidungen mit diesem Kapitel, da es meist empfehlenswert ist, sehr große Architekturen als Verteilte Systeme zu konzipieren, um eine technologische Kopplung in großem Stil zu vermeiden. Trotzdem sollte man nicht den Fehler begehen, die Anwendungsmöglichkeiten von Dingen wie dem Saga Pattern auch in monolithischen Architekturen zu übersehen. Dies kann z. B. nötig sein, um große Transaktionen bewusst auf kleinere aufzuteilen, um so die Daten sharden zu können, oder um mit externen APIs zusammenzuarbeiten. Daher gibt es für Makro-Architekturen ein gesondertes Kapitel.

6.1 Antipattern

6.1.1 Maximierung des Reuse

Zugegeben, es klingt verlockend, viel Wiederverwendung zu nutzen. Schließlich bedeutet das, dass man sich viel redundante Entwicklung erspart, oder [Mur10]? Wie kann es also sein, dass ich mir erlaube, dies hier als Antipattern anzuführen? Vielleicht ist dies auch etwas übertrieben. Hauptsächlich möchte ich aufzeigen, dass Reuse nicht das ultimative Ziel einer Architektur sein kann, auch wenn diese Denkweise doch recht verbreitet ist. Der Wunsch nach möglichst viel Wiederverwendung gleicht nämlich dem Versuch, möglichst viele externe Abhängigkeiten zu haben. Und externe Abhängigkeiten sind dann doch wieder eher die Herausforderung der Softwarearchitektur und auf keinen Fall ihr ultimatives

Ziel, da man dadurch das Prinzip der möglichst losen Kopplung verletzen würde. Anstelle auf Reuse, sollten Sie eher auf die folgenden beiden Handlungsmaximen setzen:

Modularisierung

Versuchen Sie die einzelnen Themen im Sinne einer hohen Kohäsion möglichst gut zu kapseln und Implementierungsdetails so gut wie möglich von der Außenwelt zu verbergen (Information-Hiding-Prinzip). Dadurch werden externe Abhängigkeiten minimiert.

Ersetzbarkeit

Überlegen Sie sich bei jedem Baustein, welchen Sie erzeugen, auch immer, wie Sie diesen gegebenenfalls wieder loswerden können. Dazu ist das genaue Gegenteil eines hohen Grads an Wiederverwendung hilfreich, nämlich eine möglichst geringe Anzahl eingehender Abhängigkeiten, oder mit anderen Worten eben eine lose Kopplung. Wie wichtig das ist, darf man nicht unterschätzen. Ich habe jedenfalls noch nie ein Unternehmen erlebt, welches in Schwierigkeiten kam, weil manche Funktionalitäten redundant entwickelt wurden. Meistens kommen gewachsene IT-Landschaften in Probleme, wenn Komponenten oder Systeme, welche Schwierigkeiten welcher Art auch immer machen oder deren Betrieb einfach nur teuer wird, nicht mehr einfach ausgetauscht werden können.

Dasselbe gilt natürlich auch für Technologien, welche sie einsetzen. Jede Technologie kommt früher oder später an das Ende ihrer Lebensdauer und muss dann abgelöst werden. Sei es, weil neuere Technologien effizienter sind, oder weil es schlichtweg keinerlei Know-how mehr dazu am Arbeitsmarkt gibt. Dazu ist es wichtig, möglichst auf die Technologie betreffende Annahmen bei der Integration zu verzichten. Benutzen Sie also, wenn es geht, niemals eine konkrete Technologie zur Integration von einzelnen Bausteinen. Sobald nämlich eine konkrete Technologie für die Schnittstellen zuständig ist, verteilt sich die Annahme dieser Technologie auf die gesamte Systemlandschaft und wird zu einem großen Problem, sobald sie das Ende ihrer Lebensdauer erreicht hat. Integration sollte demnach offenen Standards folgen, wie denen des W3C. Wenn Sie beispielsweise auf Integration über REST-APIs setzen, so können Sie die Technologien ihrer Bausteine jederzeit durch solche ersetzen, mit denen es ebenfalls möglich ist, REST-APIs zu erstellen oder zu konsumieren. Und das sind heutzutage mehr oder weniger alle.

Ein hoher Grad an „Wiederverwendung“ kann ein Indiz für falsch verstandene Separation of Concerns sein und anzeigen, dass die Schnitte in der Systemlandschaft an den falschen Stellen getätigt wurden. An erster Stelle und auf oberster Abstraktionsebene sollten Sie immer nach fachlichen Kriterien entscheiden.

6.1.2 Kanonisches Modell

Bereits oft gescheitert und trotzdem immer wieder probiert, ist der Versuch, die Formate, mit denen Services Daten austauschen, über die gesamte Systemlandschaft hinweg zu vereinheitlichen. Ein Unterfangen, welches in seiner Komplexität dabei unterschätzt wird, weil

es durch die Annahme dieser global definierten Datenformate Kopplung zwischen allen Services und Bausteinen einer Systemlandschaft herstellt. Damit wäre dann das Prinzip der losen Kopplung auf drastische Art und Weise verletzt. Oder um Eric Evans, seines Zeichens Erfinder des Domain Driven Design, zu zitieren [Eva03]:

Total unification of the domain model for a large system will not be feasible or cost-effective.

Auch wenn man den initialen Wurf eines solchen Modells hinbekommen sollte, so bedeutet es auch in weiterer Folge bei jeder Änderung an einem der dabei angebundenen Services enorme Aufwände zur Abstimmung. Nach unzähligen Diskussionen, welche ich in meiner Karriere schon zu dem Thema zu führen gezwungen war, scheint es mir hauptsächlich zwei Gründe zu geben, warum dies manchmal nach wie vor irrtümlicherweise für sinnvoll erachtet wird. Diese sind:

- Es wird als notwendiges Mittel zum Zweck gesehen, welches es den Services ermöglichen soll, miteinander zu kommunizieren.
- Durch eine prinzipielle Adapterschicht zu einem kanonischen Modell werden die einzelnen konkreten Service-Implementierungen angeblich komplett austauschbar.

Zunächst möchte ich auf das erste Argument eingehen. Diesem liegt die irrtümliche Annahme zugrunde, dass einzelne Business-Themen auf mehrere Services verteilt zu implementieren seien. Man sollte stattdessen aber versuchen, jedes der fachlichen Themen (also Subdomänen im Sinne des DDD) in möglichst einem Service zu kapseln. Wenn Sie das hinbekommen, so modelliert jeder Service den jeweiligen Teil der Problemdomäne, für die er zuständig ist. Es bleiben allerdings immer manche Aspekte, die von mehr als einem Service bearbeitet werden. Hier ist es aber meist immer noch besser, wenn jeder Service diese für sich selbst modelliert, und zwar aus zwei Gründen: Einerseits, weil die einzelnen Subdomänen oft unterschiedliche Sichtweisen auf diese geteilten Aspekte haben, und andererseits, um Abstimmungsaufwände zwischen den einzelnen Services zu vermeiden.

Argument Nr. 2 für kanonische Modelle setzt ein Szenario voraus, wie es in Bild 6.1 dargestellt ist. Die Idee ist, dass die einzelnen Service Consumer auf den konkreten Provider immer über die globale Adapterschicht zugreifen, welche das konkrete Schnittstellenmodell A des Service Providers immer auf das kanonische Modell X konvertiert. Wenn man nun den Service Provider durch einen anderen austauschen möchte, welcher stattdessen ein anderes Schnittstellenmodell B liefern würde, so ist nichts weiter zu tun, als den Adapter für diesen Service Provider in der globalen Schicht auszuwechseln, welcher dann ein Mapping von B auf X macht, anstatt wie zuvor A auf X. Die einzelnen Service Consumer wären davon demnach nicht betroffen. Soweit die Theorie. Valide ist dieses Argument allerdings keineswegs. Einerseits ist nicht gesagt, dass der neue Service Provider, was die Schnittstelle (wie Modell und Abläufe) angeht, überhaupt prinzipiell kompatibel ist. Und auch, wenn dem so ist, macht es keinen Sinn, den Adapter vorwegzunehmen. Wenn es wirklich zu einem solchen gefürchteten Austausch des Providers kommt, so kann man später immer noch einen Adapter bauen, welcher das Mapping von B zu A übernimmt. Und diese Modelle müssen zueinander konvertibel sein, denn wenn der Umweg B->X->A möglich ist, so ist natürlich im Endeffekt die Übersetzung von B zu A ebenfalls möglich (und sei es mittels eines Umwegs über X).

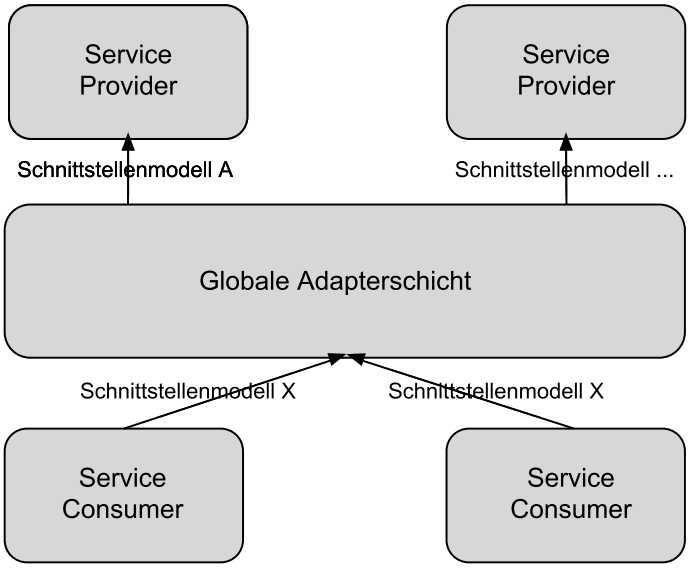

Bild 6.1 Das theoretische Einsatzszenario für ein kanonisches Modell

Noch eine Problematik eines dermaßen umfassenden Modells darf man nicht unterschätzen, und zwar die Gefahr, dass die einzelnen Teams gewisse Aspekte des Modells unterschiedlich interpretieren. Man denke nur an den Mars-Climate-Orbiter, welcher verloren ging, weil der Wert eines Felds von zwei Teams unterschiedlich interpretiert wurde. Während die NASA den Impuls im metrischen System berechnete, wurde die Navigationssoftware von Lockheed-Martin für das imperiale System ausgelegt.

6.1.3 Service Versioning

Verteilte Systeme bringen früher oder später immer die gleiche Herausforderung mit sich: Schnittstellen müssen geändert werden, weil der Service, welcher diese anbietet, weiterentwickelt wird. Dabei stellt sich die Frage, ob und wie man auf die jeweiligen Consumer der eigenen API Rücksicht nimmt. Schließlich wurden die Service Consumer mit der alten Version des zu ändernden Service Providers getestet. Früher war es daher üblich, die Änderungen am Service Provider eine Zeitlang parallel zur alten Version anzubieten, bis alle Consumer von der alten auf die neue Version umgestellt (und getestet) wurden. Meist lief es aber so ab, wie in dem hier folgenden Beispiel:

- Service A, aktuell in Version 1.1.0 deployed, stellt Erweiterungen online. Eine „Impact Analysis“ ergibt, dass davon potenziell die beiden Service Consumer B und C betroffen wären. Daher wird die Änderung als Version 1.2.0 parallel zur alten Version deployed. Die Teams von B und C unterliegen der Unternehmensrichtlinie, nach der sie innerhalb eines Quartals die Umstellung auf die neue Version machen müssen.
- Während Service C bald auf die neue Version 1.2.0 von Service A umstellt, sind die Benutzer von Service B mit dessen Stabilität unzufrieden und somit wird hier alles daran gesetzt, diese Probleme in den Griff zu bekommen. Irgendwelche Richtlinien der IT-Abtei-

lung sind dem Kunden egal und daher ist das Team von Service B die nächsten Monate mit dem Refactoring beschäftigt.

- Inzwischen müssen aber neue Features von Service A raus und somit erstellt man eine weitere Version 1.3.0.
- Team B bemerkt inzwischen, dass ihre Probleme teilweise von einem Bug in Version 1.1.0 des Service A verursacht werden, und um den Betrieb nicht zu gefährden, wird kurzerhand eine neue Version von Service A deployed, nämlich 1.1.1.

Nun laufen inzwischen drei Varianten von Service A, nämlich 1.1.1, 1.2.0 und 1.3.0. Stellen Sie sich das Spiel mit einer Systemlandschaft aus 100 Services oder mehr über einen Zeitraum von zehn Jahren vor. Sie sehen also, dass es keine gute Idee ist, Services gezielt in verschiedenen Versionen anzubieten. Viel besser ist es stattdessen, wenn Sie eine neue Version eines Services einfach als rückwärtskompatiblen Evolutionsschritt jederzeit deployen können. Um das ohne schlaflose Nächte hinzubekommen, bieten sich die Pattern an, welche ich Ihnen demnächst noch vorstellen werde, beispielsweise das Consumer Driven Contract Testing (Abschnitt 6.2.1) oder auch der Robustheitsgrundsatz (Abschnitt 6.2.2). Eine weitere Alternative wäre, jedesmal aufwändige System-Integrationstests durchzuführen.

6.1.4 Zentraler Mediator – Enterprise Service Bus (ESB)

Der Begriff Enterprise Service Bus, oder ESB, wurde ja bereits näher spezifiziert. Dabei haben wir auch den Funktionsumfang dieser Tools kennengelernt. Die Frage, welche wir uns jetzt stellen werden, lautet, ob und welche der Features, die über reines leichtgewichtiges Messaging hinausgehen, man überhaupt verwenden sollte. Konkret werden wir sehen, dass das Muster einer dezidierten Infrastruktur zur Orchestrierung von Services ein klares Antipattern darstellt. Im Grunde handelt es sich nämlich bei einem ESB um eine zentrale Komponente, die als Mediator zwischen den Services fungiert. Im Kapitel zum Thema Separation of Concerns haben wir dieses Muster bereits als Antipattern definiert. Der Grund dafür ist, dass es dadurch zu einer Verwässerung der Logik der einzelnen Bausteine in der Infrastrukturschicht kommt. Ein Refactoring eines solchen Prozesses im Sinne des Lightweight-Messaging-Prinzips ist dabei in den meisten Fällen überraschend einfach. Nehmen wir als Beispiel eine Online-Bestellung in einem Web-Shop, wobei wir zunächst einmal die ESB-Lösung betrachten (Bild 6.2).

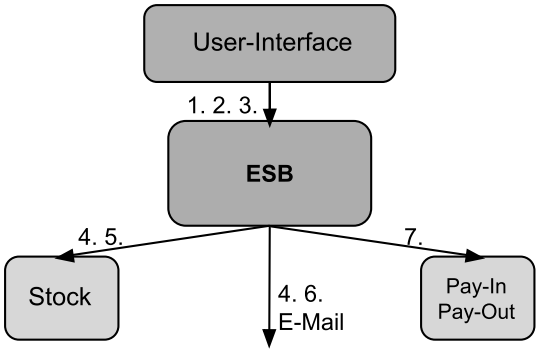

Bild 6.2 Online-Bestellung, als ESB-Prozess abgebildet

- Der Kunde legt im Web-GUI unseres Online-Shops diverse Artikel in seinen Warenkorb. (1.)
- Der Kunde möchte die Produkte haben, gibt sowohl seine Postadresse für die Lieferung, seine E-Mail-Adresse als Kontaktmöglichkeit als auch seine Bankverbindung zur Bezahlung an und klickt auf den Button zur Bestellung im Web-GUI. (2.)
- Es wird ein Prozess auf einem Enterprise Service Bus gestartet, welcher sämtliche Daten erhält und daraufhin den weiteren Ablauf steuern wird. (3.)
- Zunächst wird die Verfügbarkeit der Produkte im Lager durch den Service „Stock“ geprüft. Wenn einzelne Produkte der Bestellung nicht auf Lager sind, wird eine Order an den jeweiligen Lieferanten geschickt. Der Kunde bekommt eine E-Mail, dass sich die Lieferung verzögern wird. In diesem Fall wird täglich überprüft, ob die Produkte inzwischen lagernd sind, und sobald sie das sind, geht es weiter bei 5. (4.)
- Es werden die bestellten Produkte mittels Aufruf des Lagerservices vom Lagerbestand abgezogen. (5.)
- Es geht eine E-Mail an das Versandteam mit dem Auftrag, die gewünschten Produkte an den Kunden zu versenden. Sobald der Versand erfolgt ist, schicken die Mitarbeiter eine E-Mail an den Kunden, welche über den voraussichtlichen Liefertermin informiert. (6.)
- Es wird der Service für Inkasso „Pay-In Pay-Out“ aufgerufen, mit dem Auftrag, den fälligen Betrag vom Konto des Kunden abzubuchen. (7.)

Dies ist, wie Sie sehen, alles andere als gut modularisiert, da ein Großteil der Logik einfach in den zentralen ESB wandert. Dabei wird auch offensichtlich, dass hier das vom Domain Driven Design klar empfohlene Prinzip der Abgrenzung der Modelle in ihren jeweiligen Kontexten verletzt wird. Damit diese zentrale Ablaufsteuerung funktioniert, muss das Team, welches diesen ESB-Prozess entwickelt, mit sämtlichen Modellen und Schnittstellen der Systemlandschaft umgehen können. Nicht zufällig wird gerade in Unternehmen, welche auf dieses Antipattern setzen, oft der Ruf nach einem kanonischen Schnittstellenmodell laut, wobei es sich aber im Endeffekt nur um den Versuch handelt, das Feuer mit Benzin zu löschen. Schließlich haben wir auch dieses „Kanonische Modell“ bereits als Antipattern kennengelernt (Abschnitt 6.1.2). Nicht selten entwickeln die ESBs durch all diese negativen Aspekte gerne ein Eigenleben und beginnen zu wachsen, wodurch sie noch problematischer werden und man sie später kaum noch loswerden kann.

Setzen Sie niemals einen zentralen ESB zur Verbindung der Systeme ein. Achten Sie penibel auf die Einhaltung des Lightweight-Messaging-Prinzips, sowie auf die Gültigkeitsgrenzen der Modelle (Bounded Contexts).

Kommen wir also nun zu der Frage, wie dieser Ablauf aussehen würde, wenn wir auf einfaches Lightweight Messaging setzen und alle Logik in den jeweiligen Endpoints kapseln (Bild 6.3):

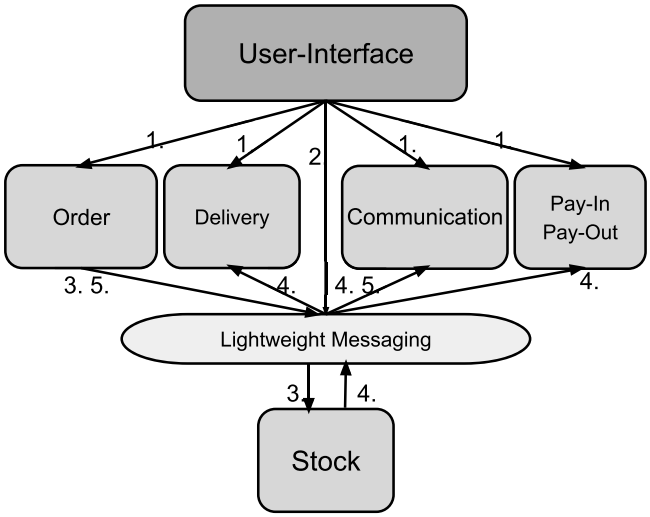

Bild 6.3 Derselbe Bestellvorgang mit lose durch Messaging gekoppelten Endpoints

- Der Kunde gibt im Zuge des Bestellprozesses seine Kontaktdaten an (Service „Communication“), seine Zahlungsweise (Service „Pay-In Pay-Out“) und seine Postadresse (Service „Delivery“). Die gewünschten Artikel werden im Warenkorb des Services „Order“ gespeichert und enthalten nur die ID des jeweiligen Produkts und die gewünschte Anzahl. (1.)
- Sobald der Kunde bestellen möchte, wird eine Message an einen eigenen Service „Order“ gesendet, mit dem Auftrag, den jeweiligen Warenkorb für den beauftragenden Kunden zu bestellen. (2.)
- Der Service „Order“ sendet eine Message an den Service „Stock“ mit dem Auftrag, die gewünschten Produkte zur Verfügung zu stellen. (3.)
- Sobald die Waren vorrätig sind, sendet der Service „Stock“ wiederum eine Message, welche unter anderem vom Service „Order“ abonniert wird. Dies kann entweder mehr oder weniger sofort erfolgen oder mit einer entsprechenden Zeitverzögerung, falls die gewünschten Produkte erst bei Lieferanten geordert werden müssen. Die Services „Communication“, „Delivery“ sowie „Pay-In Pay-Out“ haben ebenfalls diese Nachricht abonniert und kümmern sich jeweils um ihren Teil, der im Zuge der Auslieferung einer Bestellung nötig ist. (4.)
- Sollte der Service „Order“ nicht innerhalb von einem Tag die Bestätigung über die Verfügbarkeit der Produkte erhalten haben, sendet er eine Nachricht an den Service „Communication“ mit der Bitte, den Kunden über die Verzögerung zu informieren. (5.)

6.2 Empfohlene Pattern

6.2.1 Consumer Driven Contract Tests

Kommen wir nochmal auf das Dilemma aus Abschnitt 6.1.3 zu sprechen. Ein Service möchte seine Schnittstelle ändern und wir möchten mit möglichst viel Selbstvertrauen und ohne den Betrieb zu gefährden mit der neuen Version rausgehen. Das möglichst auch noch ohne große Aufwände zur Abstimmung dieser Änderung. Eine Möglichkeit ist es dabei immer, auf System-Integrationstests zu setzen, die aber teuer, aufwändig und teilweise schwierig durchzuführen sind. Eine einfachere Alternative dafür stellen die sogenannten Consumer Driven Contract Tests dar [Nyg07] (Bild 6.4). Die Idee ist, dass jeder Service Consumer dem Service, welchen er benutzt, auch einen Testfall zur Verfügung stellt. Dieser Testfall deckt die Erwartungen des Consumers an diesen Service Provider ab. Man dreht damit quasi die Richtung der Verantwortung um und überlässt den Service Consumern die Beweisführung, dass eine neue Version des Service Providers problemlos in Produktion gehen darf. Mit so einem Testfall …

- … sagt der Consumer: Wenn dieser Testfall durchläuft, so sind alle Erwartungen, welche ich an dich habe, erfüllt. Du kannst dann beruhigt jederzeit eine neue Release rausbringen.
- … sagt der Provider: Wenn du als Consumer meine Schnittstelle verwenden möchtest, so darfst du das gerne tun. Ich verlange dafür allerdings von dir einen solchen Consumer Driven Contract Test, sodass meine Weiterentwicklung davon nicht gebremst wird.

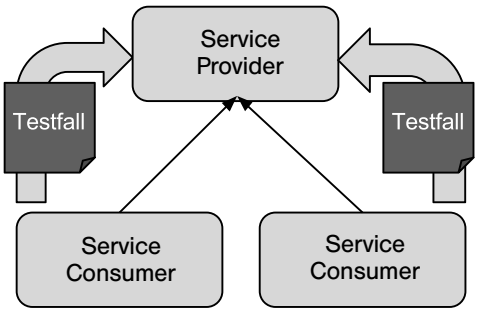

Bild 6.4 Consumer Driven Contract Testing zur organisatorischen Entkopplung der Entwicklung einzelner Services

6.2.2 Robustness Principle (Tolerant Reader)

Auch bekannt als:

Postel's Law

Betrachten wir nochmal dieselbe klassische Herausforderung jedes verteilten Systems aus dem vorigen Abschnitt über Consumer Driven Contract Testing. Ein Service Provider möchte mit einer Änderung seiner Schnittstelle produktiv gehen. Wenn die betroffenen Service

Index