1.3 Serviceorientierte Architektur

Services, not packaged software. Tim O'Reilly [wha]

TL;DR

- Ein *Service* ist eine Laufzeiteinheit, die unabhängig installiert werden kann.
- Ein Service kann durch seinen Quelltext (9.5), seine Anforderungen (5.7), seine Schnittstellen (2.3) sowie seine Qualitätsmerkmale (3) beschrieben werden.

- *Geschäftssysteme* sind soziotechnische Feedbacksysteme zur Unterstützung der Arbeit von Menschen und folgen einem ähnlichen Qualitätsmodell (3.1).
- Eine *Serviceorientierte Architektur (SOA)* ist ein Architekturstil, der auf die Erhöhung von Effizienz, Agilität und Produktivität einer Organisation zielt. Dies wird erreicht durch die Positionierung von Services als primäre Quelle von Geschäftslogik [Erl08].

■

Das Geschäftssystem

Zu Beginn dieses Abschnitts steht die Definition des Geschäftssystems, da die meisten Geschäftssysteme Teil einer serviceorientierten Architektur sind.

Ein Geschäftssystem ist verteiltes Softwaresystem, das heute de facto über den Browser, eine mobile App oder eine Web API mit seinen Benutzern kommuniziert. Geschäftssysteme sind *soziotechnische Feedbacksysteme*, also Systeme, die von Menschen genutzt werden. Ihnen ist zu eigen, dass sie sich durch Feedback ihrer Stakeholder sowie durch Änderungen in der Umwelt stetig verändern [Leh80].

Ein Geschäftssystem ist genau dann erfolgreich, wenn es die *Bedürfnisse* seiner Stakeholder erfüllt. Man sagt auch, dass die Stakeholder den *Anforderungsraum* aufspannen. Die möglichen Einflüsse auf ein Geschäftssystem durch seine Stakeholder und die Umwelt werden im Detail im Abschnitt über die Wartbarkeit (Teil II) besprochen.

Serviceorientierte Architektur

In einer serviceorientierten Architektur (SOA) stellt ein Service die primäre Quelle von Geschäftslogik dar. Diese Architektur besteht aus Services (2.3), die nach den Regeln eines Vertrags (2.4) miteinander kommunizieren. In welcher Technologie ein Service gefertigt ist, spielt für die Kollaboration keine Rolle. Die SOA ist ein Architekturstil (1.2).

SOA kam im Jahr 2000 auf und wurde durch das Platzen der Dotcom-Blase in 2001 beflügelt, als man feststellte, dass die Serviceorientierung Marktvorteile bietet ... wenn man sie richtig einsetzt.

Die serviceorientierte Architektur erwuchs aus zwei Strömungen: Die objektorientierte Analyse und Design hatte die Architekten geschult, auf Erweiterbarkeit, Wiederverwendbarkeit, Flexibilität, Robustheit und die Erfüllung von Geschäftszielen zu achten.

Die andere Strömung waren Webservices. Die Interoperabilität und Unabhängigkeit von einer konkreten Technologie bei der Servicekommunikation war ein fehlender Faktor bei der Verbindung von bisher getrennten Systemen. Plötzlich war es möglich, dass Systeme miteinander kommunizieren konnten, obwohl sie nicht in derselben Programmiersprache oder vom selben Hersteller stammten. Über APIs konnten die Systeme nun auch ferngesteuert werden. Webservices machten vor allem auch die Kommunikation zwischen Geschäftspartnern leichter und in vielen Fällen dadurch erst möglich. Schon bald sprach man nur noch von *Services, not packaged software* [wha].

Entwurfsprinzipien der SOA

Da Services bzw. deren Anbindung für ein Geschäftssystem so bedeutend sind, lohnt es, sich die Eigenschaften einer SOA im Detail anzuschauen. Hier kann man etwas lernen und auf eine Microservice-Architektur übertragen. Nach Thomas Erl gibt es acht Entwurfsprinzipien, die für eine SOA gelten sollten [Erl08]:

- *Servicevertrag:* Verträge formen die Basis der Kommunikation zwischen Services und bilden damit das Fundament der Architektur. Ein Servicevertrag besteht aus einer Sammlung von Dokumenten, die den Service und seine Nutzung beschreiben. Welche Dokumente das sind, hängt vom Service ab. Bei einem Microservice kann dies beispielsweise eine Swagger-Definition[swa] sowie eine Kontextkarte mit Domänenmodell sein. Der in diesem Buch beschriebene Entwurfsstandard sieht einen technischen Vertrag (2.3) in Kombination mit Nutzungsbedingungen (2.4) vor.
- *Loose-Kopplung:* Je enger zwei Services aneinander gekoppelt sind, desto abhängiger sind sie voneinander. In einer SOA sollten Services möglichst loose gekoppelt sein, damit sie austauschbarer und unabhängiger werden. Allerdings ist ein gewisser Grad an Kopplung unvermeidbar. Ein Team sollte die Abhängigkeiten seines Service von anderen Services kennen. In einem späteren Kapitel werden die verschiedenen Beziehungsformen (1.6) zwischen Entwicklungsteams besprochen.
- *Serviceabstraktion:* Dieses Entwurfsprinzip sucht die Funktion eines Services zu kapseln, sodass nur die für die Schnittstelle wesentlichen Konzepte nach außen sichtbar sind. Beim domänengetriebenen Entwurf wird über die Kontextgrenze das Äußere vom Inneren eines Service getrennt.
- *Servicewiederverwendbarkeit:* Aus der objektorientierten Entwicklung stammt das Prinzip der Wiederverwendbarkeit, das sich auch auf eine SOA übertragen lässt. Die Idee ist einfach: Ein und derselbe Service kann von verschiedenen Akteuren genutzt werden. Mittel zum Zweck ist ein Servicekatalog (2.2), in dem die in einer Organisation verfügbaren Services verzeichnet sind. Services sollten kooperativ sein, damit sie gut von Dritten genutzt werden können.

  Adam Jacob fügt dem hinzu, dass ein Service hierfür so entwickelt werden sollte, dass er keine Annahmen über seine Umwelt oder seine Nutzung machen sollte [JA10]. Damit meint Jacob, dass ein Service beispielsweise keine Dateien nach *c:*
  temp schreiben sollte. Natürlich darf ein Service zum Beispiel Annahmen über seine Persistenzschicht treffen.
- *Serviceautonomie:* Die Autonomie meint die Vorgaben und Kontrolle in der Entwicklung eines Services sowie die Kontrolle des Service über seine Laufzeitumgebung. Je mehr Autonomie ein Service in der Entwicklung genießt, desto höher kann die Kontrolle über die Laufzeitumgebung sein, so die Theorie.
- *Servicezustandslosigkeit:* Ist ein Service zustandslos, so muss ein Akteur nichts über seine Geschichte wissen, um eine Anfrage platzieren zu können. Das bedeutet, dass ein Service gut skalierbar ist, denn wir können mehrere Instanzen des Service unabhängig voneinander anfragen. Aus diesem Grund sind die Zustandslosigkeit und die Idempotenz wichtige Prinzipien im Serviceentwurf.
- *Service Discoverability:* Dieses Prinzip besagt, dass Services (automatisch) entdeckt werden sollten. So sollte ein Service etwa einen Storage-Dienst automatisch entdecken kön-

nen, der dann von Umgebung zu Umgebung verschieden konfiguriert wird. Zudem können bestehende Dienste manuell über einen Servicekatalog im Unternehmen kommuniziert werden. Die automatische Entdeckung von Services skaliert besser als eine manuelle Konfiguration.

- *Service Composability:* Ebenfalls ein altes Prinzip der Softwareentwicklung ist die Komposition von Programmen aus verschiedenen Modulen. Dieses Prinzip lässt sich auch auf eine SOA übertragen, bei der eine Anwendung aus verschiedenen Diensten besteht. Aus einzelnen Geschäftsprimitiven kann eine anspruchsvolle Geschäftsanwendung geschaffen werden. Wenn die Infrastrukturdienste wie Lastverteilung, Bootstrapping, Konfiguration, Artefakt Repository und Automation eine saubere API haben, kann ich darüber einen Continuous Deployment (9.4)-Prozess bauen.

Emergente Eigenschaften

Irgendwann hat die Organisation den Punkt erreicht, an dem die Services als kleine, modulare Einheiten vorliegen, die wunderbar funktionieren. Nun zahlt sich die Architektur aus, denn neue Anwendungen können auf Basis der bestehenden Dienste komponiert werden, ohne dass Services dafür umgeschrieben werden müssen.

An diesem Punkt kann man dann feststellen, dass das Ganze mehr ist als die Summe seiner Teile. Diese Eigenschaft von Informationssystemen ist bekannt als *Emerging Properties* oder *emergente Eigenschaften*. Bei der Suche nach einer guten Definition für Emerging Properties bin ich auf folgendes Zitat gestoßen, das von einem Chemiker stammt, aber dennoch einen interessanten Bezug zur Dekomposition von Services aufweist:

> *An emergent property is a property which a collection or complex system has, but which the individual members do not have. A failure to realize that a property is emergent [...] leads to the fallacy of division. - Issam Sinjab*

In der Physik gibt es einfache Beispiele für solche Eigenschaften: Mischt man rot und gelb, so erhält man grün! Auf eine Servicearchitektur übertragen bedeutet dies, dass sobald Geschäftsfunktionen modularisiert sind und neu „gemischt" werden können, kann das Fach neue und innovative Möglichkeiten der Komposition zur Verbesserung des Geschäfts entdecken. Beispielsweise kann ein gut dokumentierter *Lagerdienst*, der für den Online-Shop entwickelt wurde, auch leicht in die App für die Filialmitarbeiter integriert werden, damit diese dem Kunden sofort Auskunft geben können. Dies steigert den Geschäftsnutzen und bedeutet einen Wettbewerbsvorteil.

Damit diese emergenten Eigenschaften genutzt werden können, ist die Herstellung der Kommunikation zwischen den Stakeholdern im Rahmen der Service Governance von größter Bedeutung.

Qualitäten der SOA

Nach Thomas Erl führt die rigorose Anwendung der genannten acht Entwurfsprinzipien zu den folgenden Eigenschaften des Gesamtsystems:

- Es entsteht eine erhöhte Konsistenz, wie Funktionalität und Daten in der Organisation repräsentiert werden.
- Es gibt weniger Abhängigkeiten zwischen den Services.
- Anwendungen benötigen weniger Wissen über die Funktionsweise von Services, die sie konsumieren.
- Es gibt mehr Möglichkeiten zur Wiederverwendung von Services für unterschiedliche Einsatzmöglichkeiten.
- Es entstehen mehr Möglichkeiten bei der Aggregation und Komposition von Services in verschiedensten Konfigurationen.
- Die Vorhersagbarkeit von Verhalten erhöht sich.
- Die Verfügbarkeit und Skalierbarkeit der Services und Anwendungen erhöht sich.
- Die Wahrnehmung von bereits existenten Dienste steigt, was wiederum die Wiederverwendung begünstigt.

Typisierung von Services

Hat man eine Vielzahl von verschiedenen Diensten und möchte diese kategorisieren, beispielsweise um Entwurfsstandards anzuwenden, so stellt sich die Frage, ob sich verschiedene Typen von Services definieren lassen. Wiederum war es Thomas Erl, der eine solche Typisierung nach Entity Service, Task Service oder Utility Service vorschlug:

- *Entity Service*: Ein Entity Service fokussiert sich auf Daten und bildet Teile des Domänenmodells auf die Persistenzschicht ab. Wenn das Domänenmodell stimmt, dann ist so ein Service gut wiederverwendbar, da ihn viele Geschäftsprozesse nutzen können.

  Allerdings bietet solch ein Service keine Funktionalität. Reine Datendienste gelten aber als anämisch [New15]. In einer Microservice-Architektur werden auch Entitäten manipuliert, allerdings sowohl ihr Verhalten als auch ihre Daten, wie wir im nächsten Kapitel sehen werden.
- *Task Service*: Ein Task Service erfüllt eine spezifische Aufgabe in einem konkreten Geschäftsprozess und ist aus diesem Grund wenig wiederverwendbar [Erl08]. Interessanterweise werden Microservices nur wenige Jahre später nicht mehr nach diesem Prinzip entworfen. Hier achtet man darauf, dass der Service seine Domäne besonders gut beherrscht. Er trifft so wenig Annahmen wie möglich über seine Verwendung und ist dadurch wiederverwendbar. Task Services sind also ein veraltetes Konzept.
- *Utility Service*: Der Utility Service ist ein Dienst, der von vielen anderen Services benötigt wird. Gute Beispiele sind Logging, Notifications oder die Autorisierung. Man nennt solche Dienste auch *vertikale Services* oder *Infrastrukturdienste*.

Die Unterteilung in Infrastrukturdienste und Geschäftsdienste, die sowohl Task Service als auch Entity Service sind, macht für mich Sinn. Was die beiden unterscheidet, ist die rein technische Aufgabe in der Infrastruktur (zum Beispiel ein HTTP Cache), im Gegensatz zur geschäftlichen Aufgabe eines Geschäftsdiensts.

Kritik an der SOA

Die Einführung serviceorientierter Architekturen zu Beginn des Jahrtausends hat die Unternehmen viel Geld gekostet, und in vielen Fällen sind die Einführungen gescheitert. Als 2009 die IT-Budgets aufgrund der Finanzkrise weltweit zusammengestrichen wurden, waren Programme rund um SOA ganz oben auf den roten Streichlisten. Und das trotz der vielen Vorteile, die eine SOA einer Organisation bringen kann. Wie konnte es also sein, dass das Akronym SOA für viele Entscheider in der IT heute immer noch ein rotes Tuch ist? Hierfür gibt es viele verschiedene Gründe:

- In einer serviceorientierten Architektur steigt die Komplexität der Unternehmensarchitektur. Services, die früher nur von einer Abteilung verwendet wurden, werden plötzlich von allen konsumiert. Dies führt beispielsweise zu höheren Anforderungen an die Performance und die Skalierbarkeit und im täglichen Betrieb zu langsamen Diensten.

  Die reibungslose Kommunikation der Dienste war noch nicht erlernt und Ausfälle und Unterbrechungen die Folge. Es war schlicht ein neues Paradigma, das erst erlernt werden wollte. Fortschrittliche Mechanismen der Resilienz wie beispielsweise der Circuit Breaker kamen erst Jahre später.
- Es wurde damals viel über technische Standards gesprochen, aber wenig über das Geschäft. Wie auch? Es waren ja zwei getrennte Abteilungen: das Business und die IT. So war es wichtiger, schwergewichtige Standards wie SOAP zu etablieren, beispielsweise um Services vermeintlich sicherer zu machen, als sich über den eigentlichen Geschäftsnutzen der Dienste zu unterhalten. Auch hier fehlten die Methoden und das Wissen zur fachlichen Dekomposition der Unternehmensdienste. Eric Evans schrieb sein Buch über domänengetriebenes Design erst 2003. Es dauerte viele Jahre, bis dieses Denken in der Praxis angenommen wurde. Bis vor kurzem waren der Monolith und seine inhärente Komplexität immer noch ein Standard in der Architektur.
- Eine SOA wurde damals in vielen Fällen zum Anlass genommen, die Datenarchitektur einer ganzen Organisation zu harmonisieren. Heute weiß man, dass dieses Unterfangen zu komplex ist, um beherrscht werden zu können. Möchte man erreichen, dass verschiedene Systeme ein- und dasselbe Datenmodell verwenden, so koppelt man diese aneinander und behindert ihre individuelle Entwicklung.
- Damals war Hardware noch ein Investitionsgut. Neue Maschinen mussten erst bewilligt und bestellt werden, bevor auf ihnen ein neuer Dienst laufen konnte. Keinesfalls konnten Maschinen ad hoc provisioniert werden, so wie wir es heute in virtualisierten Umgebungen gewöhnt sind. Viele der Vorteile einer SOA konnten also gar nicht genutzt werden.
- Die Reife in der Softwareentwicklung ist seit Anfang des Jahrtausends enorm gestiegen. Seit der allmählichen Einführung von Continuous Integration hat sich seit 2006 die Anzahl funktionaler Fehler drastisch reduziert. Funktionale Fehler in Services sind aufgrund automatischer Testfälle selten geworden. Als es noch keine automatischen Testfälle gab, waren diese Fehlerraten höher und die Systeme entsprechend unzuverlässiger. Auch dies trug zum Misserfolg der Servicekompositionen bei.

Fazit

Die Wiederverwendbarkeit von Geschäftsdiensten gewährt Unternehmen Vorteile, weil sich die Architektur besser anpassen lässt. Das Versprechen möglicher emergenter Eigenschaften ist darüber hinaus ein verlockender Punkt. Der Zusammenschluss der Services eröffnet Organisationen in jedem Fall neue Möglichkeiten.

Jedoch fehlte es bei der serviceorientierten Architektur heute noch oft an einer geschäftlichen Vision zum Wohle des Unternehmens. Viel zu oft wird Technologie heute noch mit unklaren Geschäftszielen und keiner übergeordneten, inhaltlichen Architektur versehen. Wie ich im nächsten Kapitel zeigen werde, kann der domänengetriebene Entwurf diese Lücke zwischen Geschäft und Technik schließen.

Quelle: Architektur für Websysteme. Takai, S. 11ff.

1.4 Microservice-Architektur

SOA is dead; Long Live Services. - Anne Thomas Manes

By focusing each service in a narrow band, the services become easier to manage, develop, and test. - Adam Jacob

TL;DR

- Ein *Microservice* ist isolierter, kooperativer und autonomer Service (2.3), der nur eine Aufgabe hat.
- Eine *Microservice-Architektur* ist ein Architekturstil (1.2), bei dem Services zur Laufzeit komponiert werden.
- Eine *Anwendung* ist in diesem Buch ein Microservice mit einer Benutzerschnittstelle. Anwendungen sind *Kompositionen* von *Services*.
- Ein *Monolith* ist ein einschichtiges, untrennbares und technologisch homogenes System, das verschiedene Services in sich vereint.
- Ein Microservice ist leichter testbar, analysierbar, änderbar, schätzbar, skalierbar und prüfbar als ein Monolith.
- Ein wenig diplomatisches Synonym für Monolith ist *Big Ball of Mud* [mud].
- Die Komposition über das Netzwerk erzeugt Komplikationen, die bei einer monolithischen Architektur nicht gegeben sind.
- Eine Microservice-Architektur hat eine höhere Latenz (11) als ein Monolith.

Was ist ein Microservice?

Ein Microservice ist ein isolierter Dienst mit eigener Laufzeitumgebung und Non-Shared Storage State (13.3). Er hat nur eine einzige Geschäftsaufgabe, aber erledigt diese besonders

gut. Zusammen mit anderen Diensten lässt sich ein Microservice zu einer Microservice-Architektur *komponieren*. In diesem Kapitel werden die Vor- und Nachteile einer solchen Architektur besprochen und mit der serviceorientierten Architektur verglichen.

Eine *Anwendung* ist eine Komposition von Microservices mit einer Benutzerschnittstelle. Bild 1.2 zeigt die Komposition eines Systems aus Anwendungen und Microservices.

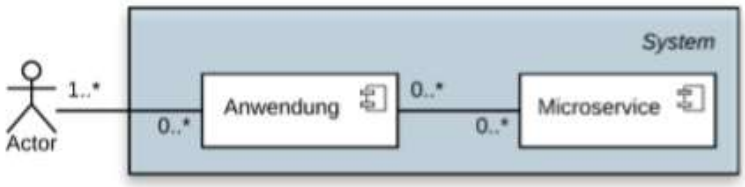

Bild 1.2 Ein Geschäftssystem als Komposition von Anwendung und Microservices

Single-Responsibility-Prinzip

Das *Single-Responsibility-Prinzip* ist eines der SOLID-Prinzipien der objektorientierten Entwicklung. Das Prinzip besagt, dass jede Klasse nur eine einzige Aufgabe haben sollte und sich auch nur aus diesem Grund verändern darf. Diese Aufgabe soll die Klasse kapseln und damit gleichzeitig eine hohe Kohäsion erzeugen. Dieses Prinzip lässt sich auch auf einen Service anwenden, um diesen besser zu kapseln und kohärenter zu machen. Tatsächlich ist das Single-Responsibility-Prinzip das wichtigste Prinzip eines Microservice und gibt ihm auch seinen Namen: Ein Microservice darf nur eine einzige Aufgabe haben, aber diese soll er besonders gut beherrschen. Deswegen hat der Microservice eine hohe Kohärenz und ändert sich nur, wenn sich auch seine geschäftliche Funktion verändert.

Wie diese Geschäftsfunktionalität identifiziert werden kann, ist nicht Teil des Microservice-Architekturstils, der nur die technische Dekomposition eines Systems umfasst. Wie man das Geschäft zerlegt, ist Aufgabe des domänengetriebenen Entwurfs, den ich im nächsten Kapitel bespreche.

Komplexität bewusst machen

Die konsequente Anwendung des Single-Responsibility-Prinzips führt zu einer optimierten *Komplexität* des Microservice. Das bedeutet aber nicht, dass die Geschäftslogik trivial sein muss, denn die Geschäftswelt ist komplex.

Diese inhärente Komplexität, die auch von Brooks besprochen wird [Bro75], heißt *unverzichtbare Komplexität*. Man nennt sie unverzichtbar, weil man sie nicht reduzieren kann, ohne gleichzeitig das Geschäft zu reduzieren und sich dadurch selber Marktvorteile zu nehmen. Das Ziel der Softwareentwicklung ist nicht, dem Geschäft seine Eigenarten zu rauben, sondern dieses zu unterstützen.

In einem Monolithen müssen viele verschiedene Domänen Platz finden, was zu einer größeren Komplexität des Service führt. Der Service kann dann zu einem verworrenen Haufen Spaghetti-Code werden, der sich nur noch schwer ändern lässt. Man kann auch sagen: Die

Software ist hässlich, weil das Problem hässlich ist oder zumindest nicht gut verstanden wurde [mud].

Häufig werden monolithische Entwicklungsprojekte auch ohne Veränderung an der Organisationsstruktur vorgenommen, sodass in ein und derselben Software unterschiedliche Interpretationen des Geschäfts durch verschiedene Abteilungen Platz finden müssen. Nach Melvin Conway ist das eine schlechte Idee, denn die Organisation sollte das System reflektieren und umgekehrt [Con68]. Im domänengetriebenen Entwurf, der im kommenden Kapitel besprochen wird, werden die Domänen häufig nach Abteilungsgrenzen geschnitten.

Das Streben nach optimaler Komplexität ist eine wesentliche Aufgabe des Architekten und *Keep it simple* eine zentrale Designphilosophie. Tatsächlich sollte ein Architekt stets danach streben, unnötige und vor allem unkontrollierte Komplexität zu vermeiden. Die Ethikrichtlinien der Schweizer Informatik Gesellschaft weisen dies explizit aus [sig].

Sie können die Komplexität dadurch reduzieren, dass Sie entscheiden, welche Teile der unverzichtbaren Komplexität abgebildet werden sollen.

Mit optimaler Komplexität steigt die Änderbarkeit und Flexibilität, weil es leichter ist, einen solchen Service zu formen. Wenn wir den Service sogar zur Laufzeit verstehen können, dann können wir ihn auch gut diagnostizieren und kommen Fehlern schneller auf die Spur.

Isolation

Die zweite Charakteristik eines Microservice ist die Isolation, wodurch sich der Dienst ohne Einfluss auf andere Dienste ändern lässt. Dies beginnt bei der Entwicklung, denn ein Microservice kann in einem eigenen Repository versioniert werden, verfügt über eigene Build-Pläne und kann unabhängig von anderen Services installiert werden. Mehr Informationen dazu finden Sie im Kapitel über die Änderbarkeit (9). Die Verwaltung der Quelltexte in einem eigenen Repository ist meiner Meinung nach wichtig, und zwar aus Gründen des Lifecycle Managements. Viele Teams speichern jedoch alle Quelltexte in einem einzigen Repository, und das Thema ist umstritten [sin].

Eine Microservice-Architektur wird zuverlässig (Teil IV) ausgelegt, sodass Störungen lokal begrenzt bleiben und keine anderen Dienste stören. Dies ist in einer monolithischen Architektur nicht machbar, denn die klassische Endlosschleife führt hier zum Versagen aller Dienste. Auf der anderen Seite kann ein Microservice-System durch Schneeballeffekte beim Versagen einzelner Instanzen abstürzen.

Des Weiteren ist ein Microservice auch beim Storage auf sich alleine gestellt. Jeder Service hat sein eigenes Schema bzw. unter Umständen sogar sein eigenes Storage-System, auf das kein anderer Service Zugriff hat. Integrationen auf der Datenbankebene (ein verbreitetes Anti-Pattern, auch *Distributed Monolith* genannt) können so effektiv verhindert werden. Zudem ist die Größe der Daten für den Service optimiert und ihre Pflege deswegen besonders einfach.

Ein Downstream-Service kann den Zustand eines Microservice nur über seine API abfragen. Durch diese Kapselung kann die Entwicklung durch Refactorings im Rahmen der Kontextgrenze besser gepflegt werden.

Unterschiede zur SOA

Eine serviceorientierte und eine Microservice-Architektur unterscheiden sich in einigen wenigen, aber bedeutsamen Punkten. In beiden Fällen dienen die Services als primäre Quelle der Geschäftslogik.

Der kleine große Unterschied zwischen den beiden ist, dass ein Microservice nur eine einzige Aufgabe wahrnimmt, in einer SOA darf ein Dienst jedoch beliebig viele Aufgaben haben. De facto verbindet man mit einer Microservice-Architektur außerdem die Verwendung von REST (2.3) zur Kommunikation zwischen den Diensten.

In einer SOA ist SOAP prädominant, aber es gibt daneben in der Regel noch viele andere Protokolle, sodass man es mit einem babylonischen Wirrwarr von Dialekten zu tun hat. Ein Mittel zur Integration dieser unterschiedlichen Sprachen für ältere Systeme ist der Einsatz eines Service Bus (2.5).

SOAP steht auch in der Kritik, weil es ein schwergewichtiges Protokoll ist, das von Menschen schlecht lesbar ist. Eine WSDL ist ein komplexes Artefakt mit erstaunlich wenig Aussagekraft. Entsprechend sind auch die SOAP-Implementierungen nicht einfach zu handhaben.

Komposition der Microservice-Architektur

In der klassischen, monolithischen Entwicklung unterscheidet man zwischen *Modul* und *Komponente*. Noch im Jahr 2012 definierte Len Bass den Unterschied folgendermaßen [BKC13]:

- Ein *Modul* ist eine gekapselte Implementierungseinheit. Wenn wir über ein Modul sprechen, so haben wir immer einen konkreten Bezug zur Implementierung.
- Eines oder mehrere Module können bei einer *Komponente* zusammengestellt werden. Eine Komponente ist eine Laufzeiteinheit, die unabhängig deployed werden kann. Eine Komponente besteht aus verschiedenen Modulen, wobei die Abbildung surjektiv, aber nicht injektiv ist.

Interessanterweise ist ein Microservice sowohl Modul als auch Komponente, denn ein Microservice wird isoliert entwickelt und ist gleichzeitig eine Laufzeiteinheit, die unabhängig deployed werden kann. Der wesentliche Unterschied zur klassischen Definition ist, dass Microservices zur *Laufzeit* komponiert werden können. D.h. ich muss meine Software nicht neu bauen, wenn sich an einer Funktion etwas ändert, sondern kann zur Laufzeit beispielsweise einen neuen Service hinzufügen. Dies eröffnet dem Team im Umgang mit einem System fundamental neue Möglichkeiten. Die gewonnene Flexibilität sorgt vor allem für mehr Geschwindigkeit in der Entwicklung.

Die Isolation auf den Ebenen Entwicklung, Fehlerpropagierung und Datenbank ist im Zusammenspiel mit der Komponierbarkeit zur Laufzeit förderlich für Continuous Deployment (9.4).

Vorteile einer Microservice-Architektur

Zusammenfassend kann man die folgenden Vorteile einer Microservice-Architektur anrechnen:

- *Programmierung*: Die Microservice-Architektur begünstigt Software Craftsmanship, denn ein Microservice zeichnet die Grenzen des Geschäfts nach, das er abbilden soll [New15].
- *Technische Heterogenität und Flexibilität*: Wenn unser System aus diskreten Diensten besteht, dann dürfen diese aus verschiedenen Technologien bestehen, ohne dass sich die Qualität des Systems ändert. Generell rate ich aus Gründen der Konsistenz (6) von zu viel Verschiedenheit ab und empfehle, immer dieselben Technologien einzusetzen. Aber in einer Microservice-Architektur ist die Möglichkeit der Inkonsistenz ein Vorteil, wenn bereits bestehende Systeme oder externe Systeme, die nicht kontrolliert werden können, integriert werden sollen. Dann ist die Architektur flexibel genug, diesen Rahmenbedingungen genüge zu tun.
- *Innovation*: Eine Microservice-Architektur vereinfacht die Einführung neuer Technologien, denn neue Services können in anderen Programmiersprachen geschrieben werden oder andere Frameworks (6.3) verwenden, ohne dass dies die Qualität der Architektur beeinflusst. Wenn eine neue Technologie vielversprechend, aber riskant erscheint, kann bei einer Microservice-Architektur ein wenig kritischer Dienst zur Probe gewählt werden.
- *Zuverlässigkeit*: In einer Microservice-Architektur sind viele verschiedene Dienste an einem Anwendungsfall beschäftigt. Der Ausfall eines Service kann durch einen guten Entwurf kompensiert werden, beispielsweise durch einen Wechsel von synchronem zu asynchronem Messaging. Dies steigert die Widerstandsfähigkeit gegen Fehler und erhöht die Zuverlässigkeit (IV) [New15].
- *Skalierbarkeit*: In einem Monolithen müssen alle Services zusammen skaliert werden, aber bei einer Microservice-Architektur können nur die Services skaliert werden, bei denen das auch nötig ist [New15]. Dies erhöht die Ressourceneffizienz und dient auch der Kostenoptimierung.
- *Leichte Deployments*: Eines der größten Probleme monolithischer Architekturen sind die Deployments, da das System immer als Ganzes veröffentlicht werden muss. Hierfür haben sich in der Vergangenheit elaborate Prozesse in der Versionskontrolle (9.5) etabliert, die sicherstellen sollen, dass nur funktionierende Commits ihren Weg in den Release Branch finden. Bei Microservices stellt sich dieses Problem nicht mehr, da jeder Service unabhängig von anderen Diensten deployed werden kann. Das schafft viel organisatorische Flexibilität und ist ein echter Vorteil, weil neuer Geschäftswert schneller in Produktion gebracht werden kann.
- *Gesetz von Conway*: Bei einer Microservice-Architektur kann das Team leichter an die Architektur des Systems und der Organisation angepasst werden, da die Wartung einer spezifischen Codebase weniger Personal benötigt. Dies schafft vor allem Flexibilität und eine verbesserte Abstimmung durch optimierte Kommunikationswege, wenn man es richtig macht. Mehr zu Conway's Law finden Sie in Abschnitt 5.11.

- Der Service ist einfach zu benutzen, da seine API so primitiv ist, dass sie jeder versteht. Tatsächlich lädt dieses Vorgehen andere ein, die API zu benutzen, anstatt ähnliche Funktionalität zu entwickeln, was auch die die Wiederverwendung fördert [JA10].
- Der Service ist einfach zu entwickeln, da nur wenige Anforderungen erfüllt werden müssen. Dies zieht kurze Turnaround-Zeiten nach sich, d.h. der Geschäftswert kann früher erzeugt werden. [JA10]
- Der Service ist einfach zu testen, und somit lassen sich automatische Tests günstig entwickeln. Eine gute Testabdeckung führt wiederum zu höherer Qualität und steigert das Vertrauen der Benutzer, was auch die Wiederverwendung steigert. [JA10]
- Der Service ist einfach zu betreiben, und in Kombination mit einer funktionierenden Virtualisierung (oder Containerisierung) erlaubt dies eine schlanke und vorhersagbare Budgetierbarkeit. [JA10]
- Der Service hat eine hohe Konzeptionelle Integrität (5).

Damit dies funktionieren kann, benötigt es einige Anforderungen an den Service, die erfüllt sein müssen, damit sich der Service nahtlos in die Dienstlandschaft einer Organisation integrieren kann. Der hierfür benötigte Entwurfsstandard des Service (2.3) ist in einem kommenden Kapitel beschrieben.

Nachteile einer Microservice-Architektur

Wo Licht ist, ist auch Schatten, und so haben Microservice-Architekturen gegenüber Monolithen auch einige Nachteile:

- *Latenz*: Durch die Komposition verschiedener Dienste in eigenen Laufzeitumgebungen entsteht mehr Verkehr im Netzwerk, der langsamer ist als Aufrufe innerhalb derselben Maschine. Die Latenz (11) der Aufrufe steigt, und das System ist langsamer. Insbesondere bei Systemen mit viel Traffic, kann dies deutliche Auswirkungen auf das Benutzererlebnis haben.
- *Netzwerkkomplikationen*: Ein verteiltes System ohne Fehler ist nicht möglich, und Störungen und Ausfälle können gravierende Konsequenzen für die Funktion eines Systems haben. Ein Monolith ist nicht auf ein funktionierendes Netzwerk angewiesen, weswegen die Fehlerbehandlung hier einfacher ist. Die möglichen Fehlersituationen sind in Abschnitt 15.2 diskutiert.
- *Referenzielle Integrität*: Dadurch, dass die Geschäftslogik auf verschiedene Dienste mit jeweils eigener Persistenzschicht verteilt ist, können keine Datenbankmechanismen zur Wahrung der referenziellen Integrität genutzt werden. Dies kann bei komplexen Domänen Auswirkungen auf die Komplexität im Code und die notwendigen Transaktionen sowie die Performance haben.
- *Neues Paradigma*: In einer Microservice-Architektur kapseln die Dienste ihre Geschäftslogik, und dadurch werden diese in sich weniger komplex. Die Komplexität des Geschäfts verschwindet dadurch aber nicht, sondern verlagert sich hin zur Komposition der Dienste zu einem funktionierenden Ganzen. Dies ist für viele Teams heute Neuland und benötigt auch Kompetenzen im Bereich System und Reliability Engineering, vor allem aber in der Domänenanalyse.

Fazit

Viele Organisationen sind heute auf Monolithen zur Durchführung ihres Geschäfts angewiesen. Diese *Systems of Record* bilden in vielen Branchen das Rückgrat der Geschäftsfähigkeit und sind das Ergebnis hoher Investitionen, die geschützt werden müssen. Die Idee, dass kleine, agile und flexible Microservices zur Erbringung von Geschäftsdiensten eingesetzt werden können, ist neu. Ebenfalls neu ist aber auch, dass Unternehmen Vorteile haben, wenn ihre Informationssysteme klein, agil und flexibel sind, weil sie so schneller neue Dienstleistungen am Markt anbieten können. Der Trend hin zu diesen Architekturen darf also nicht ignoriert werden. Da Microservice-Architekturen auch Mischformen von Monolithen und Microservices erlauben, können Unternehmen Mischformen adaptieren und so iterativ wettbewerbsfähig bleiben.

Quelle: Architektur für Websysteme. Takai, S. 11ff.

1.5 Domänengetriebener Entwurf

Cells can exist because their membranes define what is in and out and determine what can pass. - Eric Evans

TL;DR

- Die *Domäne* ist die Arbeit einer Organisation in ihrer Umwelt.
- Ein *Domänenmodell* repräsentiert den Teil der Domäne, der durch Software unterstützt werden soll, und ist in der Allgemeinsprache (5.3) beschrieben.
- Ein Domänenmodell kann in *Subdomänen* zerlegt werden.
- In einer Domäne repräsentieren *Aggregate* die Daten und das Verhalten von Objekten.
- Das *Aggregat* ist eine Transaktionsgrenze, die kohärent zur Arbeit der Organisation ist.
- Ein Domänenmodell hat eine *Kontextgrenze*, die den Gültigkeitsbereich seiner Konzepte bestimmt.
- Der *domänengetriebene Entwurf* zerlegt die Arbeit der Organisation in Domänenmodelle und komponiert diese im *strategischen Entwurf* zu einem System.
- Ein Microservice (1.4) implementiert entweder ein Domänenmodell, ein Aggregat oder einen Domänendienst.
- Eine domänengetriebene Microservice-Architektur bietet Wettbewerbsvorteile.

Was ist der domänengetriebene Entwurf?

Im Kapitel über Microservices haben wir gesehen, dass die technische Trennung von Services eine gute Idee ist, um die *unverzichtbare Komplexität* [Bro75] eines Systems in der

Entwicklung beherrschen zu können. Die Gretchenfrage lautet nun, wie diese Teilung vorzunehmen ist, um das Geschäft optimal zu unterstützen. Hier kommt der domänengetriebene Entwurf ins Spiel, der uns eine *fachliche Dekompositionstechnik* an die Hand gibt, die sich gut mit der technischen und physischen Dekomposition einer Microservice-Architektur verbinden lässt. In diesem Abschnitt gehe ich auf die wichtigsten Punkte des domänengetriebenen Entwurfs ein.

Die Allgemeinsprache

Die Allgemeinsprache (5.3) ist ein Werkzeug des domänengetriebenen Entwurfs. Die Begriffe dieser Sprache bilden die konzeptionelle Grundlage für die Arbeit des Teams. Durch den rigorosen Einsatz einer gemeinsamen Sprache kann das Team seine natürlichen linguistischen Fähigkeiten bei der Entwicklung des Systems nutzen.

Event Storming

Grundlage des domänengetriebenen Entwurfs ist die Identifikation der wirkenden Geschäftsereignisse. Im Anforderungsmanagement wird schon seit vielen Jahren die Erstellung eines Systemkontexts (5.4) gelehrt [PR15]. Der Systemkontext umfasst solche Geschäftsereignisse, auf die das System eine Antwort haben sollte. Die Geschäftsereignisse sind ein Synonym für Domänenereignisse und Ausgangspunkt für die Analyse der Arbeit der Organisation.

Domänenereignisse sind nicht technisch, obwohl sie später auch auf die technische Architektur abgebildet werden müssen. Stattdessen repräsentieren sie fachliche Ereignisse und werden deswegen von den Stakeholdern gut verstanden. Die Formulierung der Ereignisse durch Nomen und Verben der Allgemeinsprache fördert eine informative Kommunikation über die Domäne. Dies vertieft und schärft das Verständnis des Systems im Team.

Im Anforderungsmanagement (5.7) ist für die Beantwortung eines Geschäftsereignisses ein Anwendungsfall (5.8) vorgesehen, anhand dessen sich Verhalten und Daten der Domäne ableiten lassen. Die Erkundung des Ereignisraums einer Domäne geschieht im *Event Storming*. Das ist ein Workshop, zu dem die fachlichen Stakeholder, das Management, Benutzer, aber auch Entwickler und Architekten eingeladen werden – kurzerhand alle Personen mit einem potenziellen Einfluss auf das System. In einem gemeinsamen Workshop leitet ein Moderator die Gruppe an, gemeinsam die Domänenereignisse zu identifizieren. Dabei entstehen auch die ersten Domänenmodelle und Aggregate, die die Ereignisse verarbeiten und die im Folgenden beschrieben werden. Weitere Informationen über die Durchführung von Workshops finden Sie in [Unt15].

Domänenmodelle schneiden

Um das System zu zerlegen, verwenden wir Domänenmodelle. Ein *Domänenmodell* ist ein konzeptionelles Modell der Domäne, das sowohl Daten als auch Verhalten enthält. Das heißt, die Domäne modelliert die Arbeit einer Organisation in ihrer Umwelt. Eine Domäne

kann in *Subdomänen* zerlegt werden, die durch eine *Kontextgrenze* (engl. *bounded context*) voneinander getrennt sind. Hierbei wird der also fachliche Kontext in disjunkte Modelle zerlegt, die jeweils einen Teil der Arbeit des Geschäfts repräsentieren.

Bild 1.3 zeigt ein vereinfachtes Beispiel für einen Online-Shop, bei dem Preisberechnung, Artikelkatalog, Lager und der Warenkorb jeweils als eigene Subdomänen modelliert sind. Alle Konzepte der Domänenmodelle sind Teil der Allgemeinsprache (5.3) unseres Systems.

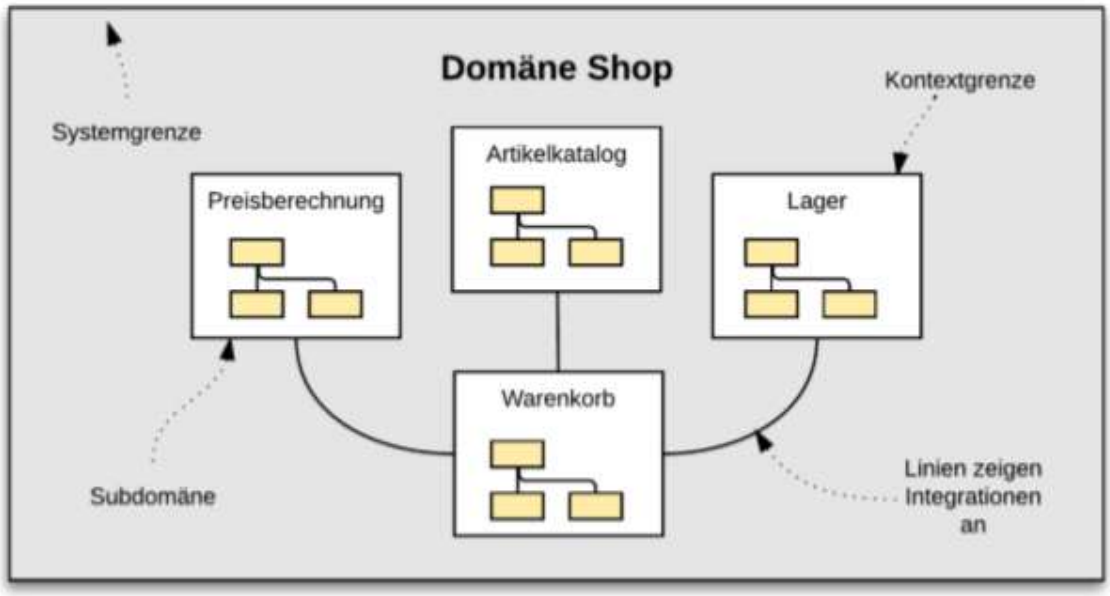

Bild 1.3 Beispiel für eine Dekomposition der Domäne am Beispiel Shop

Domänenmodelle bilden zusammen mit ihren Entitäten, Wertobjekten und anderen Basiskonzepten die *Bausteine* des domänengetriebenen Entwurfs. Die aus den Bausteinen konstruierten Modelle können dann im *strategischen Entwurf* zu einem System komponiert werden.

Die Etablierung der Kontextgrenze ist beim domänenorientierten Entwurf eine Schlüsseltechnik, denn die Grenze steckt den Bereich ab, in dem ein Konzept, seien es Daten oder Verhalten, seine Bedeutung hat. Dieser Punkt ist zentral: Ein Konzept muss außerhalb seines Kontexts keine Bedeutung haben. Dies macht das Modell unabhängig und verschafft dem Team Freiraum bei der Modellierung. Und nur, wenn es Freiraum hat, ist es handlungsfähig und kann Geschwindigkeit entwickeln.

Möchte man mehrere Modelle zusammenlegen, so wird das Gesamtmodell unweigerlich komplexer. Tatsächlich ist solch ein Abgleich verschiedener Modelle weder praktikabel noch kosteneffektiv [Eva03]. Ein Beispiel hierfür ist das sogenannte *Master Data Management*, das alle möglichen Daten querschnittlich durch eine Organisation modellieren möchte. Hier hat man es automatisch mit vielen verschiedenen Teams zu tun, die auf die Daten angewiesen sind, weil es die IT-Strategie nun so verlangt. Ein Abgleich der Modelle wird dadurch kompliziert, schwerfällig, langsam und teuer. Darüber hinaus müssen die Teams Kompromisse eingehen und können ihre eigene Domäne also nicht vollständig abbilden.

Trotzdem müssen Domänenmodelle aber in vielen Fällen zusammenarbeiten. In Bild 1.3 zeigen die Linien zwischen den Subdomänen Integrationspfade der Domänen an. So ist

im Beispiel der Warenkorb abhängig vom Artikelkatalog, um Artikel im Warenkorb anzeigen zu können, vom Lager, um die Verfügbarkeit von Artikeln darzustellen, sowie von der Preisberechnung. Wie genau das funktioniert, werden wir später sehen. Wir halten an dieser Stelle fest, dass trotz der Integration die Modelle nicht zwingend etwas miteinander zu tun haben müssen.

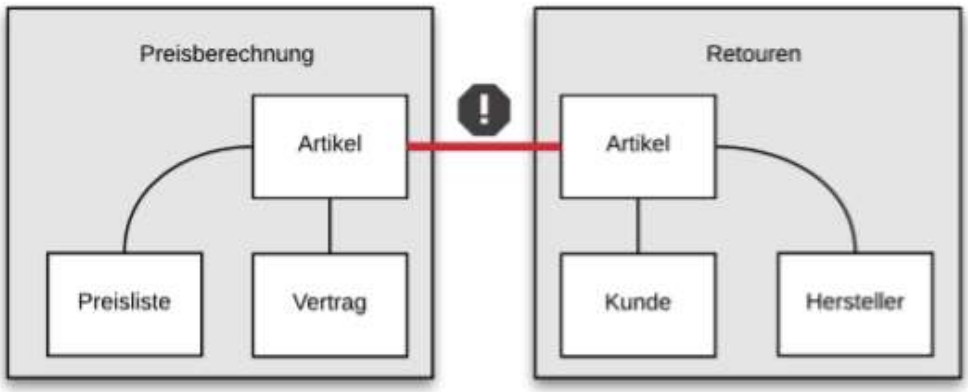

Bild 1.4 Beispiel für zwei Domänenmodelle, die nicht gemischt werden müssen

Jedes Konzept des Modells sollte unmissverständlich sein. Ein Konzept wie der *Artikel* taucht aber in vielen Subdomänen auf, bedeutet aber jeweils etwas anderes. Bild 1.4 zeigt ein vereinfachtes Beispiel für zwei Domänen, die beide das Konzept Artikel enthalten. In einem Monolithen sehe ich häufig, dass für beide Modelle ein und dieselbe Klasse genutzt wird. Tatsächlich hat der Artikel bei der Preisberechnung und bei den Retouren jeweils andere Bedeutungen und deswegen auch andere Eigenschaften. Die beiden Konzepte zu vermischen, wäre ein Fehler, weil damit das *Single-Responsibility-Prinzip* verletzt wird: Eine solche Artikelklasse muss sich dann sowohl um Retouren als auch die Preisberechnung kümmern.

Die Modelle können wir mithilfe der UML (8.4) oder auch freihändig modellieren, um so die in der Sprache definierten Konzepte und ihre Relationen untereinander aufzuzeigen. Die Diskussion der Modelle mit dem Fach schafft in der Regel interessante Erkenntnisse für das gesamte Team. Die Fachexperten sollten sich dabei gegen Begriffe wehren, die das Wissen über die Domäne nicht gut transportieren. Entwickler sollten auf mögliche Fallstricke in der Implementierung achten.

Konzepte des domänengetriebenen Entwurfs

Zum domänengetriebenen Entwurf gehören die folgenden Konzepte:

- Zum Domänenmodell gehören Entitäten, Wertobjekte, Aggregate, Repositories, Factories sowie Domänendienste.
- Ein Konzept wird als *Entität* (engl. *entity*) modelliert, wenn seine Einzigartigkeit, also die Unterscheidbarkeit von allen anderen Konzepten, wichtig ist.
- Ein *Wertobjekt* (engl. *value object*) beschreibt oder misst ein Konzept der Allgemeinsprache, das keine eigene Identität besitzt.

- Ein *Aggregat* fasst Wertobjekte und Entitäten innerhalb einer Kontextgrenze zusammen. Ein Aggregat hat eine als *Aggregatwurzel* definierte Entität über die der Zugriff auf alle anderen Entitäten und Wertobjekte des Aggregats erfolgt.

Entitäten und Wertobjekte

Innerhalb eines Domänenmodells können wir die Konzepte auf verschiedene Weise modellieren, je nachdem um was es sich handelt. Im domänengetriebenen Design gibt es hierfür zwei wesentliche Möglichkeiten: die Entität und das Wertobjekt. Ein Konzept wird als *Entität* modelliert, wenn seine Einzigartigkeit, also die Unterscheidbarkeit von allen anderen Konzepten, wichtig ist [Ver13]. Eine Entität ist einzigartig und hat einen Lebenszyklus. Oft möchte man Änderungen an einer Entität während seines Lebenszyklus beobachten können. Änderungen an Entitäten können Nebenwirkungen haben. Da Entitäten langlebig sind und eine Identität haben, müssen sie gespeichert werden und sind deswegen also teure Konzepte.

Die andere Möglichkeit ist die Modellierung als *Wertobjekt* (engl. *value object*), das keine eigene Identität besitzt. Wertobjekte beziehen ihre Identität durch ihren Wert und sind leichter zu erzeugen, testen, benutzen, optimieren oder zu warten als Entitäten. Änderungen an einem Wertobjekt haben keine Nebenwirkungen [Eva03]. Wertobjekte sind aus diesen Gründen Entitäten vorzuziehen, wenn dies möglich ist.

Aggregate bilden

Häufig machen Objekte nur in Kombination mit anderen Objekten Sinn und werden auch zusammen gespeichert. Eine Bestellung enthält mehrere Artikel, ein Menü mehrere Gerichte und ein Buch viele Kapitel (ich spreche aus Erfahrung). Im domänengetriebenen Entwurf heißen diese Objektgraphen *Aggregate* (engl. *aggregates*). In einem Graphen darf jeweils nur eine Entität nach außen sichtbar sein, und diese Entität heißt *Wurzelentität* (engl. *root entity*). Bild 1.5 zeigt ein Beispiel für das Domänenmodell eines Artikelkatalogs. Der Artikel ist eine Wurzelentität mit verschiedenen Eigenschaften (engl. *traits*), die über eine Suchfunktion gefunden werden können.

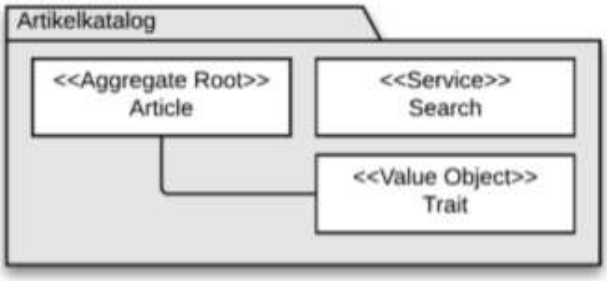

Bild 1.5 Beispiel für das Domänenmodell eines Produktkatalogs

Die Aggregate und die Wurzelentität bilden die *Transaktionsgrenze* unseres Modells. Da unsere Entitäten und Wertobjekte das Geschäft modellieren, ist diese Transaktionsgrenze konsistent mit der Geschäftspraxis der Organisationseinheit. Die Erzeugung und Löschung eines Aggregats und seiner Attribute sind miteinander verbunden, während die Erzeugung und Löschung ihrer Bestandteile unabhängig voneinander ist [Eva03].

Diese Art der Modellierung erzeugt eine konzeptionelle Integrität zwischen System und Geschäft des Unternehmens und macht es möglich, die unverzichtbare Komplexität beherrschbar zu machen, weil andere Modelle keinen Einfluss auf die Konzepte der Domäne haben.

Factories, Repositories und Services

Im domänengetriebenen Entwurf gibt es noch ein paar weitere Konzepte, auf die ich an dieser Stelle kurz eingehen möchte:

- Wenn die Erzeugung von Aggregaten zu kompliziert wird, sollte eine *Factory* eingesetzt werden, um das Softwaredesign zu vereinfachen und den Prozess der Erzeugung zu vereinfachen.
- Alle Operationen zum Speichern von Daten sollten an ein *Repository* delegiert werden. Achten Sie darauf, dass Ihr Service dabei crash-konsistent (17.4) bleibt.
- Manchmal macht es keinen Sinn, eine Methode an ein Aggregat, eine Entität oder ein Wertobjekt zu hängen. In diesem Fall gibt es das Konzept des *Service*, der die Geschäftslogik implementiert, die auf mehreren Objekten basiert.

Modelle im Kontext sichtbar machen: Die Kontextkarte

Wie wir schon gesehen haben, stehen unsere Domänenmodelle in Beziehung zueinander. Beispielsweise müssen für den Warenkorb die Preise berechnet und Lieferzeiten angezeigt werden. Wir möchten diese Abhängigkeiten zu anderen Modellen gerne explizit machen, aber wir möchten unsere Freiheit behalten. Gleichzeitig stellt sich nun das erste Mal die Frage nach der physischen Verteilung von Modellen auf Laufzeiteinheiten. Ein Diagramm, das Kontextgrenzen sichtbar macht, heißt *Context Map* oder *Kontextkarte*. Diese Kontextkarte ist ein Werkzeug des *strategischen Entwurfs* und wird aus der Perspektive des Teams gezeichnet, das für ein bestimmtes Modell verantwortlich ist.

Das Hauptziel der Kontextkarte ist es aufzuzeigen, welche Abhängigkeiten zwischen Domänenmodellen bestehen, und ob es Konzepte gibt, die nicht zueinander passen und deswegen übersetzt werden müssen. Die Kontextkarte nimmt deswegen die tatsächliche Situation auf und identifiziert die Kontaktpunkte zwischen den Modellen.

In einer serviceorientierten Architektur ist nicht jeder Service ein Microservice. Ist eine integrierte Domäne durch ein *System of Record* oder *Legacy System* implementiert, so ist die Wahrscheinlichkeit groß, dass die Konzepte nicht gut passen und übersetzt werden müssen. Mittel zum Zweck ist hierfür der Anti-Corruption Layer, der im folgenden Abschnitt beschrieben wird. Häufig werden ältere Systeme gekapselt mit dem Ziel, sie besser integrieren zu können. Dieses Thema beschreibe ich im Kapitel über den Service Bus (2.5).

Domänengetriebene Microservices

In einer Microservice-Architektur bilden wir jeweils ein Aggregat oder ein Domänenmodell auf einen Service ab. Ich nutze für die Notation UML-Komponenten, um anzuzeigen, dass es sich tatsächlich um eine physische Laufzeiteinheit handelt. Bild 1.6 zeigt eine sogenannte *Kontextkarte* für einen Checkout-Service. Je nachdem, wie die Abhängigkeiten gerichtet sind, stehen Microservices in einer Upstream- oder Downstream-Beziehung zueinander: Derjenige Service, der eine API aufruft, ist der Upstream-Service. Im Diagramm werden die Assoziationen zwischen den Services mit U für Upstream und D für Downstream beschriftet. Über diese Kontextkarte kann das Team eindeutig kommunizieren, von welchen anderen Diensten es abhängig ist. Weitere Informationen zur Dokumentation mit Kontextkarten finden Sie in einem Artikel von Alberto Brandolini [bra].

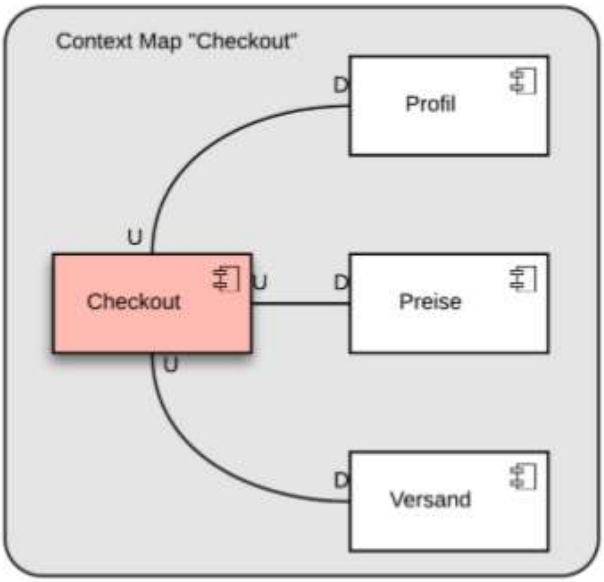

Bild 1.6 Beispiel für Upstream- und Downstream-Beziehungen von Services

Microservices und domänengetriebener Entwurf ergänzen sich aus den folgenden Gründen gut zur Unterstützung des Geschäfts:

- Es bietet sich an, einzelne Domänenmodelle oder Aggregate auf einen Microservice abzubilden. Das muss nicht so sein, aber da unser System am Ende des Tages aus Microservices besteht, die auch von verschiedenen Teams entwickelt und gewartet werden, macht diese Dekomposition Sinn, weil die Dienste dann die geschäftliche Dekomposition nachbilden. Mehr zur Organisation von Teams im kommenden Kapitel.
- Werden Domänenmodelle auf Microservices abgebildet, so lassen sich diese auch über erhobene Metriken (12.2) im Rahmen der Service Governance (2.1) zur Laufzeit aufzeichnen und beobachten. Die Auswertung der Metriken zugunsten strategischer Entscheidungen ist ein Wettbewerbsvorteil [PF13]. Da bei einer domänengetriebenen Architektur die aufgezeichneten Geschäftsereignisse besonders informativ sind, können Vorteile allenfalls besser genutzt werden.
- Die durch Geschäftsereignisse ausgelöste Modellierung passt gut zum asynchronen Entwurf (14.4), der die Skalierbarkeit begünstigt.

Fazit

Das Domänenmodell ist eine Vereinfachung. Es handelt sich um eine Interpretation der Wirklichkeit, die die für das System relevanten Aspekte abstrahiert und überflüssige Details ignoriert. Zusammenfassend hat das Domänenmodell drei Verwendungszwecke, die alle die konzeptionelle Integrität beeinflussen:

1. Das Modell und der Entwurf beeinflussen einander, was dazu führt, dass das Modell für die Architektur und die Softwareentwicklung relevant wird. Das gut recherchierte Modell spiegelt sich im fertigen Produkt. Im Laufe der Entwicklung können Quelltexte durch das Modell leichter interpretiert werden.
2. Das Modell bildet die Grundlage für das gemeinsame Verständnis im Team. Die Entwickler können mit dem Fach über die definierte Allgemeinsprache (5.3) kommunizieren. Unsere natürlichen linguistischen Fähigkeiten können dazu genutzt werden, das Modell zu verbessern.
3. Das Modell ist das destillierte Wissen des Teams über das System und die Domäne gleichermaßen.

Weitere Informationen zum Thema des domänengetriebenen Entwurfs finden Sie in den Büchern von Eric Evans [Eva03] und Vaughn Vernon [Ver13].

Quelle: Architektur für Websysteme. Takai, S. 11ff.