daniel TAKAI

Architektur für Websysteme

// Serviceorientierte Architektur
// Microservices
// Domänengetriebener Entwurf

HANSER

Takai

Architektur für Websysteme

Daniel Takai

Architektur für Websysteme

Serviceorientierte Architektur, Microservices, Domänengetriebener Entwurf

HANSER

Der Autor:
Daniel Takai, Thun (CH)
takai@silberruecken.ch

Bibliografische Information der Deutschen Nationalbibliothek:

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

Lektorat: Brigitte Bauer-Schiewek
Copy editing: Jürgen Dubau, Freiburg/Elbe
Herstellung: Irene Weilhart
Layout: le-tex publishing services GmbH
Umschlagdesign: Marc Müller-Bremer, www.rebranding.de, München
Umschlagrealisation: Stephan Rönigk
Datenbelichtung, Druck und Bindung: Kösel, Krugzell
Ausstattung patentrechtlich geschützt. Kösel FD 351, Patent-Nr. 0748702
Printed in Germany

Print-ISBN: 978-3-446-45056-1
E-Book-ISBN: 978-3-446-45248-0

Inhalt

TEIL I

Geschäftssysteme

1 Einleitung

1.1 Buchmodell

If you want to build a ship, don't drum up people together to collect wood and don't assign them tasks and work, but rather teach them to long for the endless immensity of the sea. – Antoine de Saint Exupéry

Die Sprache als Werkzeug

Softwaresysteme sind komplex und schwierig zu begreifen. Die Fehler unsichtbarer Laufzeitumgebungen geben uns bei der Analyse Rätsel auf. Wenn wir verstehen möchten, was ein System tut, müssen wir sein Umfeld und die Arbeit verstehen, die mit dem System verrichtet wird. Dieses Verständnis ist schwer aufzubauen, wenn uns die gemeinsame Sprache der Konzepte des Systems fehlt. Haben wir jedoch ein gemeinsames, fachliches Verständnis der Domäne und sind Technik und Geschäft kohärent, so können wir unsere natürlichen linguistischen Fähigkeiten für die Analyse und Synthese eines Systems nutzen.

Aus diesem Grund habe ich für dieses Buch ein eigenes Modell entwickelt, eine Art von Allgemeinsprache (5.3), in der jeder verwendete Fachbegriff genau beschrieben ist. Was im Entwurf eines Systems funktioniert, das sollte für ein Buch lange reichen.

Jeweils zu Beginn eines Abschnitts habe ich die im Text folgenden Beschreibungen und Definitionen der besseren Übersichtlichkeit halber als *TL;DR* aufgeführt. Der ganz eilige Leser kann nur diese Textboxen lesen, um die Inhalte dieses Buches zu erfassen.

Für wen ist dieses Buch geschrieben?

Dieses Buch enthält das Wissen um die Architektur von Web- bzw. Geschäftssystemen, also Systemen, die über das Internet funktionieren und für die Arbeit mit Menschen entworfen werden. Das Buch richtet sich an Softwarearchitekten oder jene, die Softwarearchitekt werden möchten. Ebenfalls interessant ist das Buch für Personen, die im Anforderungsmanagement arbeiten. Durch die durchgängige Illustration mit Qualitätsszenarien soll das Buch helfen, Anforderungen besser zu spezifizieren.

Nicht geeignet ist das Buch zum Erlernen spezifischer Technologien oder Programmiersprachen. Zwar sind immer mal wieder konkrete Beispiele für Technologien angegeben, aber dieses Buch konzentriert sich auf Entwurfsmuster und Methoden.

Aufbau des Buches

Im ersten Teil dieses Buches werden verschiedene Architekturstile und Entwurfsmuster für Geschäftssysteme beschrieben: Was bedeutet Architektur und wie wird sie angewendet? Die vorgestellten Möglichkeiten werden in den Teilen danach über die Qualitätsmerkmale beschrieben, die für ein Geschäftssystem heute relevant sind. Jedes Qualitätsmerkmal ist dabei in Submerkmale untergliedert, welche wiederum durch verschiedene Techniken und Methoden im Detail erläutert werden.

Somit eignet sich der Aufbau des Buches auch als Checkliste für Systembewertungen oder als Grundlage für die Planung des Entwurfs. Ich selber nutze in meiner Praxis genau diesen Aufbau, um effizient Expertisen zu schreiben und Beratungsleistungen zu erbringen.

Geschäftssysteme

In diesem Buch geht es um den Entwurf, die Bewertung, Implementierung und Wartung von Geschäftssystemen (1.3), also Systemen, die die Arbeit mit Menschen in Organisationen über das Internet unterstützen. Geschäftssysteme reagieren auf Geschäftsereignisse mit Geschäftsfällen, deren Geschäftstätigkeiten durch Geschäftsregeln bestimmt werden. Das Geschäft steht in diesem Buch für die Arbeit einer Organisation und kann jeden erdenklichen fachlichen Hintergrund haben. Allgegenwärtig ist dabei die *unverzichtbare Komplexität*, die ein Geschäftssystem abbilden muss und die nicht reduziert werden kann. Eine Hauptaufgabe des Architekten ist neben der Möglichkeit, diese Komplexität zu bewältigen, vor allem die Kontrolle und Eliminierung unnötiger Komplexität. Hierfür bietet dieses Buch eine Dokumentation verschiedener Architekturstile, Entwurfsstandards, Qualitätsmodelle sowie Qualitätsmerkmale, die für Geschäftssysteme relevant sind.

Architekturstile

In diesem Buch werden drei Architekturstile vorgestellt, die aufeinander aufbauen und komplementär zueinander stehen. Zum besseren Verständnis gehe ich auf die Geschichte und Hintergründe der Stile ein. Zudem gibt es hier und dort Vergleiche mit dem Architekturstil des Monolithen.

Sammlungen von Entwurfsprinzipien heißen *Architekturstile*. In diesem Buch sind drei verschiedene Stile beschrieben, die zueinander passen und gemischt werden dürfen.

- Serviceorientierte Architektur (1.3)
- Microservice-Architektur (1.4)
- Domänengetriebener Entwurf (1.5)

Entwurfsstandards

Für Microservices, sonstige Services, Monolithen, Service-Bus-Systeme und Öffentliche APIs haben sich in den letzten Jahren Entwurfsmuster gebildet, die ich in diesem Buch festhalten möchte.

Der Entwurfsstandard ist eine Vorschrift der Governance für den Systementwurf. In diesem Buch sind drei Standards zur Vorgabe enthalten.

- Entwurfsstandard des Service (2.3)
- Entwurfsstandard des Open Host (2.4)
- Entwurfsstandard des Service Bus (2.5)

Qualitätsmodelle für Geschäftssysteme

Ich habe es oft erlebt, dass die für den Entwurf eines Systems notwendigen Grundlagen in Form von Qualitätsanforderungen nicht vorhanden waren. Zwar gibt es immer den einen oder anderen Satz, dass das System superschnell und sehr sicher sein soll, aber diese Beschreibungen sind meistens unzulänglich. Ich denke, dass der Grund hierfür in mangelndem Wissen um die möglichen Qualitäten und ihre Beschreibung ist. Es ist mir also ein Anliegen, die für ein Geschäftssystem notwendigen Qualitäten zu beschreiben, damit diese als Bewertungsgrundlage dienen kann: Welche Eigenschaften muss ein Service haben, damit er im Kontext erfolgreich sein kann?

Ein Qualitätsmodell beschreibt, bewertet und sagt Qualität voraus. Das Modell bildet damit die Basis des Entwurfs, der bewertet werden soll. In diesem Buch sind drei Qualitätsmodelle für Geschäftssysteme enthalten.

- Cloud-Native Systeme (3.1)
- Reaktive Systeme (3.1)
- Geschäftssysteme (3.1)

Es gibt heute viele verschiedene Methoden und Techniken in der Softwareentwicklung wie Eskalationsmanagement, Open Auth Connect, Mehrfaktorauthentifizierung oder Sandboxing. Aber es ist nicht in jedem Fall klar, welche Qualitäten des Produkts diese Methoden beeinflussen, und dies erschwert die Investitionsplanung. Wird ein wartbarer Service benötigt, aber nicht in Methoden investiert, die Wartbarkeit schaffen, so ist das ein Problem. Ein System kann funktional fehlerfrei sein und doch maßlos enttäuschen, weil stillschweigende Erwartungen nicht erfüllt wurden. Dem Architekten kommt die schwierige Rolle zu, diese Erwartungen im Rahmen des vorhandenen Budgets über die Qualitätsmerkmale zu verhandeln.

Qualitätsmerkmale von Geschäftssystemen

Qualitätsmerkmale sind Eigenschaften eines Systems. In diesem Buch sind 19 Qualitätsmerkmale beschrieben. Zu jedem Qualitätsmerkmal gibt es ein Herleitung, Qualitätsszenarien, Entwurfsmuster und Methoden für den praktischen Einsatz.

Danksagung

Ohne die Hilfe meiner hochgeschätzten Kollegen wäre dieses Buch niemals entstanden. Besonderen Dank möchte ich jenen aussprechen, die mit mir gemeinsam die Diskussion gesucht haben. Viele der Inhalte und Aussagen dieses Buches sind aus dem gemeinsamen Schreiben von Artikeln für Fachmagazine entstanden. Mein ausdrücklicher und großer Dank gilt Christoph Huber, Olaf Otto, Nicolas Bär, Christian Wittwer, Thomas Jaggi, Gion Manetsch, Verena Linder, Urs Siegenthaler, Carlo Bonati, Christoph Camenisch, Marcel Wiedemeier, Karsten Petersen, Stefan Zörner und Thomas Walser. Besonderen Dank gebührt Daniel Rey, ohne dessen exakte und fundierte Kritik mir die abschließende Korrektur des Werks sehr viel schwerer gefallen wäre.

Ich möchte mich außerdem bei all denjenigen Architekten und Autoren bedanken, die das Wissen um die Serviceentwicklung im Web vorantreiben und aktiv teilen. Dazu gehören insbesondere auch Organisationen, die ihr Wissen durch das Führen von Engineering Blogs und die Veröffentlichung von Open Source teilen.

1.2 Architektur und Entwurf

TL;DR

- Architektur ist der Schlüssel zum Verständnis eines Systems [CKK02].
- *Softwarearchitektur* ist die systematische Beschreibung der Services (2.3) und Qualität (3) eines Systems in der Allgemeinsprache (5.3).
- Der *Entwurf* ist die schöpferische Leistung, die zur Softwarearchitektur führt.
- Die Eingaben des Entwurfs sind Rahmenbedingungen, Annahmen, Anforderungen (5.7) und Qualitätsmerkmale (3).
- Die Ausgaben des Entwurfs sind die Architektur, eine Planung, Risiken und Kosten.
- Ein *Entwurfsprinzip* ist eine verallgemeinerte, akzeptierte Industriepraxis.
- Eine Sammlung von Entwurfsprinzipien heißt *Architekturstil*. In diesem Buch sind drei Architekturstile beschrieben: Die serviceorientierte Architektur (1.3), die Microservice-Architektur (1.4) und der domänengetriebene Entwurf (1.5).
- Eine konkrete Entwurfsvorlage, die ein verbreitetes und wiederkehrendes Problem löst, heißt *Entwurfsmuster*. In diesem Buch werden verschiedenste Entwurfsmuster zur Unterstützung von Qualitätsmerkmalen angegeben.
- Vorgaben für den Entwurf eines Systems heißen *Entwurfsstandards*. Entwurfsstandards spiegeln die Rahmenbedingungen einer Organisation. In diesem Buch sind drei Entwurfsstandards beschrieben: Service (2.3), Open Host (2.4) und Service Bus (2.5).

Was ist Softwarearchitektur?

In fast jedem Buch über Softwarearchitektur steht zu Beginn die Frage, was Softwarearchitektur eigentlich ist. Die Frage ist in der Literatur so offen wie ungeklärt. Tatsächlich bietet das Software Engineering Institute der Carnegie Mellon Universität sogar eine Sammlung vieler verschiedener Definitionen von Softwarearchitektur an [sei]. Es folgen verschiedene Definitionen und Erläuterungen über Softwarearchitektur:

- *Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled.* - Eoin Woods [sei]

  In dieser Definition tritt weder das Wort Komponente, Microservice oder Schnittstelle auf, beschreibt einen wesentlichen Teil von Architektur aber sehr gut: In der Architektur geht es um *Entscheidungen*, die hohe Kosten verursachen können und sich später schwer rückgängig machen lassen.

- *The set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both.* - Paul Clements [CBB+10]

  In dieser Definition geht es um Strukturen von Software und deren Eigenschaften. Zweifellos ist es wichtig zu wissen, aus welchen Bausteinen ein System besteht und welche Eigenschaften, auch Qualitätsmerkmale (3.1) genannt, diese Bausteine haben sollen. Es

ist vor allem die Dokumentation des Systems, die in dieser Definition eine tragende Rolle spielt, auch wenn sie nur implizit genannt ist.

- *Architecture is first and foremost key to achieving system understanding. As a vehicle for communication among stakeholders, it enables high-bandwidth, informed communication among developers, managers, customers, users, and others who otherwise would not have a shared language.* Paul Clements [CKK02]

  Dies ist meine Lieblingsdefinition von Softwarearchitektur, weil sie die Bedeutung der gemeinsamen Sprache in der täglichen Kommunikation mit den Stakeholdern so schön hervorhebt. Für mich ist die Sprache ein subtiles, aber wertvolles Werkzeug der Entwicklung, dem ich in diesem Buch viel Platz gelassen habe. So bezieht sich dann auch die letzte Definition von Softwarearchitektur, die zu meinem Buchmodell passt, auf die Allgemeinsprache (5.3):

- *Softwarearchitektur ist die systematische Beschreibung der Services und Qualität eines Systems in der Allgemeinsprache.* Daniel Takai

  Dies ist die Definition von Architektur, die also für den Rest dieses Buches gelten soll.

Das gesamte Gebiet der Informatik unterliegt einer zunehmenden Spezialisierung von Wissen und Rollen, und so auch die Architektur. Die Unterschiede zwischen Softwarearchitektur, Unternehmensarchitektur, Geschäftsarchitektur oder Technologiearchitektur sind in diesem Buch jedoch nicht beschrieben, weil der Prozess der Standardisierung noch im Gange ist und es schlicht zu früh wäre, trennscharf zu unterscheiden.

Architekturstile mischen

Beginnt man mit der Analyse oder Synthese eines Systems, so können bekannte Entwurfsformen eine Hilfe sein. Eine verallgemeinerte, akzeptierte Industriepraxis ist ein *Entwurfsprinzip*, vergleichbar mit einer Best Practice. Im Rahmen der Softwarearchitektur führt die konsistente Anwendung von Entwurfsprinzipien zur Erreichung bestimmter Qualitätsmerkmale. Beispielsweise führt die Anwendung des Prinzips der Zustandslosigkeit einer Schnittstelle zu einem skalierbaren Service (14).

Eine Sammlung von Entwurfsprinzipien heißt *Architekturstil*. Da die Anwendung der Prinzipien gewisse Qualitätsmerkmale fördert, führen diese zu einer homogeneren IT-Landschaft, weil sich die Services ähnlich verhalten. In der Praxis findet man häufig Mischformen der Architekturstile: eine serviceorientierte Architektur, die aus Monolithen besteht, oder eine Microservice-Architektur, die domänengetrieben entworfen wurde. Architekturstile dürfen also gemischt werden.

Sammlungen von Entwurfsprinzipien heißen *Architekturstile*. In diesem Buch sind drei verschiedene Stile beschrieben, die zueinander passen und gemischt werden dürfen:

- Serviceorientierte Architektur (1.3)
- Microservice-Architektur (1.4)
- Domänengetriebener Entwurf (1.5)

Entwurfsstandard

Ein *Entwurfsmuster* ist eine konkrete Entwurfsvorlage, die ein verbreitetes Problem löst, beispielsweise der Entwurf eines Test Harness (7.5) für automatische Tests.

Ein *Entwurfsstandard* ist eine verbindliche Vorgabe für den Entwurf eines Systems, die die konkrete Umwelt einer Organisation als Standard spiegeln. Solch ein Standard ist ein Architekturstil in Kombination mit einer Sammlung von Entwurfsmustern, die auf die Organisation zugeschnitten sind. So kann beispielsweise im Rahmen einer IT-Strategie eine bestimmte Datenbanktechnologie vorgegeben werden. Die Kontrolle der Umsetzung von Entwurfsstandards heißt *Compliance*, die unter anderem über das Test Management (7.3) sichergestellt werden kann.

Der Entwurfsstandard ist eine Vorschrift der Governance für den Systementwurf. In diesem Buch sind drei Standards zur Vorgabe enthalten.

- Entwurfsstandard des Service (2.3)
- Entwurfsstandard des Open Host (2.4)
- Entwurfsstandard des Service Bus (2.5)

Architektur und Programmierung

Die Softwareentwicklung beschäftigt sich mit der *Programmierung* eines Service und einem konkreten Bezug zur Implementierung. Die Softwarearchitektur hingegen definiert die Kommunikationswege zwischen den Services und beschäftigt sich mit ihren Schnittstellen. Aus verschiedenen Services komponiert die Architektur ein System. Das bedeutet aber nicht, dass Architektur und Programmierung nicht von derselben Person ausgeführt werden können, es sind nur andere Aufgaben.

Die Programmierung ist die Aufgabe des Entwicklers. Durch *Software Craftsmanship*, *Clean Code* und den Dialog mit dem Team stellt er eine bestmögliche Performance des Service sicher. Da er für den Service die Verantwortung übernimmt, sollte er auch bestimmen, mit welchen Technologien gearbeitet wird. Der Entwickler sollte anhand der eigenen Kompetenzen und Erfahrungen die Entscheidungen in Bezug auf die Programmierung fällen dürfen. Für mich gehört dazu auch die Wahl der Programmiersprache und der verwendeten Frameworks (6.3). Es ist aber auch verständlich, dass in vielen Organisationen aus Compliance-Gründen Vorgaben für einsetzbare Technologien gemacht werden. Dies ist zum Beispiel dann der Fall, wenn damit zu rechnen ist, dass viele verschiedene Entwickler im Laufe der Zeit an dem Service arbeiten werden. In diesem Fall wählt man eine *Plattform*, die populär ist und für die es auch in zehn Jahren noch genügend Personal geben wird.

Interessanterweise wird die Qualität eines Systems nicht nur durch die Qualität eines Service bestimmt, sondern mehrheitlich durch die Architektur, also die Komposition der Dienste [KKB+98]. Somit teilen sich also die Architektur und die Arbeit des Entwicklers die Aufgabe der Qualitätserreichung und sollten deswegen Hand in Hand arbeiten.

Der Entwurfsprozess

Um zur Architektur zu kommen, durchläuft der Architekt einen *Entwurfsprozess*, bei dem viele *Entscheidungen* getroffen werden. Viele Entscheidungen haben eine lange Lebensdauer und lassen sich so gut wie nicht mehr revidieren. Denken Sie nur an die Wahl eines Frameworks oder die Auswahl eines Cloud Providers: Können solche Entscheidungen noch rückgängig gemacht werden, wenn die Anwendungen auf dieser Basis ein paar Wochen später produktiv laufen? Die wesentlichen Entscheidungen zu identifizieren und teilweise auch zu treffen, ist die Arbeit des Architekten. Im *Entwurfsprozess*, der in Bild 1.1 visualisiert ist, sammelt der Architekt die folgenden Informationen:

- Informationen, die noch nicht vorliegen, müssen durch *Annahmen* ersetzt werden. Die Annahmen sollte der Architekt dokumentieren, auch damit man sie verfolgen kann. Bei Werksverträgen können solche Annahmen wertvoll sein, um nachzuweisen, dass es eine andere Ausgangslage gab als *angenommen*. Durch die Dokumentation der Annahmen, beispielsweise in einem Angebot, werden diese zum Vertragsbestandteil.
- Rahmenbedingungen (5.7) sind Anforderungen, die nicht verändert werden können. Beispiele für Rahmenbedingungen sind das verfügbare Budget, die Ausbildung der Projektmitarbeiter oder regulatorische Vorgaben, denen das Geschäft unterliegt. Rahmenbedingungen sind nicht veränderbar, aber sie müssen nicht notwendigerweise statisch sein. So kann es sein, dass sich die Gesetzgebung im Laufe des Projekts verändert. Die Dokumentation der Rahmenbedingungen kann wichtig sein, um Entscheidungen später nachvollziehen zu können.
- Die geplante Qualität (3) eines Systems zu verhandeln und im Entwurf zu berücksichtigen, ist eine Kernkompetenz des Architekten. Hierfür werden Qualitätsszenarien (3.2) für die relevanten *Qualitätsmerkmale* erhoben, die diese Qualität möglichst vollständig beschreiben.
- Neben den Qualitätsmerkmalen und Rahmenbedingungen werden auch Informationen über die funktionalen Anforderungen (5.7) benötigt. Vom *Requirements Engineer* bekommen Sie eine Zusammenfassung der Geschäftsziele, Epics und Features des Systems,

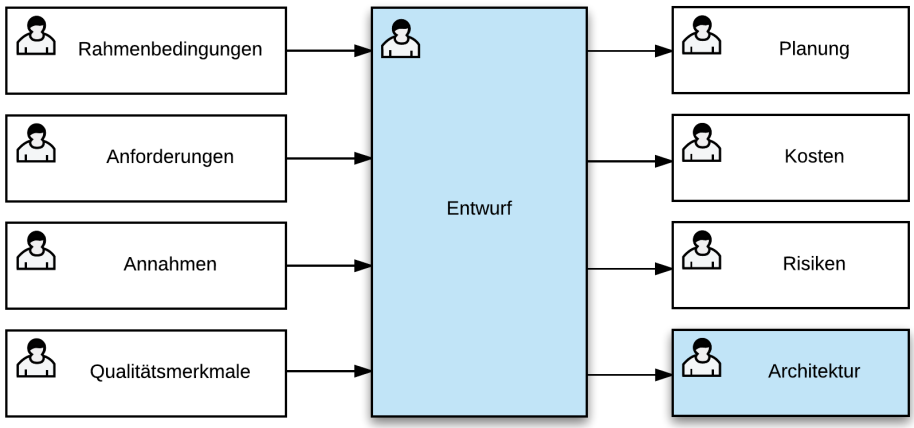

Bild 1.1 Entwurf der Architektur

die die funktionalen Anforderungen auf einer hohen Abstraktionsebene bündeln. So haben Sie eine Chance zu verstehen, was mit dem System erreicht werden soll, und kommen mit der Architektur weiter. Meiden Sie Detaildiskussionen zu funktionalen Anforderungen, denn diese sind für die Architektur selten relevant.

Die Analyse der Eingaben dient dem Architekten als Grundlage für den *Entwurf*. Der Architekt muss diesen Syntheseprozess nicht alleine durchlaufen, er kann den Entwurf auch gemeinsam mit dem Team oder anderen Stakeholdern durchführen. Da beim Entwurf sehr viele Dinge bedacht werden müssen, ist es sogar angeraten, Ergebnisse gemeinsam zu besprechen. Ich lasse mich immer wieder gerne von kreativen Ideen aus unerwarteten Richtungen überraschen. Gleichzeitig erhöht der Dialog über die Lösung den Buy-In der Stakeholder. Der Entwurfsprozess erzeugt vier Ausgaben:

- Die *Planung* des Systems wird anhand der definierten Services und Anwendungen sowie ihrer räumlichen Verteilung möglich. Die Planung wird durch das Engineering Management (9.1) begleitet. Ein Hilfsmittel der Kommunikation ist die Dokumentation (8.3) des geplanten Systems.
- Im Tandem mit der Planung lassen sich auch die *Kosten* für das System abschätzen. Kosten entstehen vor allem durch die Allokation von Personal, den Aufbau und Betrieb von Produktionstechnologien wie Continuous Deployment oder Versionskontrolle, die Definition von Produktionsprozessen, den Betrieb der verschiedenen benötigten Umgebungen (7.4) sowie möglicherweise Lizenzen, wenn Standardprodukte eingekauft werden. Hinzu kommen laufende Mittel für IaaS- oder PaaS-Produkte sowie die verdeckten Kosten, die entstehen, wenn der Architekt eine falsche Entscheidung trifft. Diese Kosten sind allerdings nicht finanzieller Natur, sondern nagen eher am Renommee: Architekten werden daran gemessen, wie gut ihre Entscheidungen sind.
- Der Architekt weist *Risiken* aus, die im Zusammenhang mit der Entwicklung und dem Betrieb des Systems stehen. Risiken entstehen immer dann, wenn gewünschte und geplante Qualität nicht übereinstimmen, weil dies bedeutet, dass Anforderungen der Stakeholder nicht erfüllt werden können. Stellen Sie sich vor, Ihr System muss von Hunderten Fachkräften eingesetzt werden, und es wird kein Budget für Usability-Tests reserviert. Es besteht dann das begründete Risiko, dass die Anwendung von den Benutzern nicht akzeptiert wird. Ähnliches gilt für die Wartbarkeit (II), Performance (III), Zuverlässigkeit (IV) oder Sicherheit (V).

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.

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.

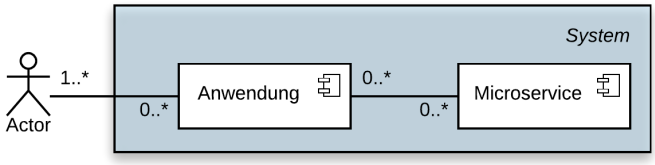

Bild 1.2 Ein Geschäftssystem als Komposition von Anwendung und Microservices

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.