4 Was sind Microservices?

Der Abschnitt 1.1 hat schon eine erste Definition des Begriffs Microservice gegeben. Es gibt aber noch andere Möglichkeiten, Microservices zu definieren. Die unterschiedlichen Definitionen zeigen die Eigenschaften von Microservices und geben damit an, aus welchen Gründen Microservices vorteilhaft sind. Am Ende des Kapitels sollte eine eigene persönliche Definition des Begriffs Microservice stehen – abhängig vom eigenen Szenario.

Den Begriff »Microservice« betrachtet das Kapitel aus unterschiedlichen Perspektiven:

- In Abschnitt 4.1 steht die Größe des Microservice im Mittelpunkt.
- Abschnitt 4.2 stellt eine Beziehung zwischen Microservices, Architektur und Organisation mithilfe des Gesetzes von Conway her.
- Am Ende zeigt Abschnitt 4.3 eine fachliche Aufteilung von Microservices anhand von Domain-Driven Design (DDD) und BOUNDED CONTEXT.
- Der Abschnitt 4.4 erläutert, warum Microservices eine UI enthalten sollten.

4.1 Größe eines Microservice

Der Name »Microservices« verrät schon, dass es um die Servicegröße geht – offensichtlich sollen die Services klein sein.

Ein Möglichkeit, die Größe eines Microservice zu definieren, sind Lines of Code (LoC, Codezeilen) [1]. Ein solcher Ansatz hat jedoch einige Probleme:

- Er hängt von der verwendeten Programmiersprache ab. Einige Sprachen benötigen mehr Code, um dasselbe auszudrücken – und Microservices sollen gerade nicht den Technologie-Stack fest definieren. Dementsprechend ist eine Definition anhand dieser Metrik kaum sinnvoll.

- Schließlich geht es bei Microservices um einen Architekturansatz. Architektur sollte sich aber nach den Gegebenheiten in der fachlichen Domäne richten und nicht so sehr nach technischen Größen wie LoC. Auch aus diesem Grund ist eine Bestimmung der Größe anhand der Codezeilen kritisch zu sehen.

LoC können trotz aller Kritik ein Indikator für die Größe eines Microservice sein. Es stellt aber dennoch die Frage nach der idealen Größe eines Microservice. Wie viele LoC darf ein Microservice haben? Auch wenn es keine absoluten Richtwerte gibt, gibt es dennoch Einflussfaktoren, die für größere oder kleinere Microservices sprechen.

Modularisierung

Ein Faktor ist die Modularisierung. Teams entwickeln Software in Modulen, um die Komplexität handhabbar zu machen: Statt die gesamte Software zu verstehen, muss ein Entwickler nur jeweils das Modul verstehen, das er gerade ändert, und das Zusammenspiel der Module. Nur so kann ein Team trotz der Komplexität eines typischen Software-Systems überhaupt produktiv arbeiten. Es gibt in der Praxis oft Probleme, weil Module größer werden als ursprünglich geplant. Dadurch werden die Module schwer zu verstehen und schwer zu warten, weil Änderungen ein Verständnis der Software voraussetzen. Also ist es sinnvoll, Microservices möglichst klein zu halten.

Dagegen spricht, dass Microservices im Gegensatz zu vielen anderen Modularisierungsansätzen einen Overhead haben:

Verteilte Kommunikation

Microservices laufen in eigenständigen Prozessen. Daher ist die Kommunikation zwischen Microservices verteilte Kommunikation über das Netzwerk. Für diese Art von Systemen gibt es die »erste Regel für verteilte Systeme«. Sie besagt, dass Systeme möglichst nicht verteilt werden sollen [2]. Der Grund dafür ist, dass der Aufruf eines anderen Systems über das Netzwerk einige Größenordnungen langsamer ist als der direkte Aufruf im selben Prozess. Zu der reinen Latenzzeit kommen noch der Aufwand für die Serialisierung und Deserialisierung der Parameter und Ergebnisse hinzu. Diese Operationen dauern nicht nur lange, sondern kosten auch CPU-Kapazität.

Hinzu kommt, dass verteilte Aufrufe fehlschlagen können, weil das Netzwerk gerade nicht verfügbar oder der aufgerufene Server nicht erreichbar ist – weil er beispielsweise abgestürzt ist. Das macht die Implementierung verteilter Systeme noch komplexer, denn der Aufrufer muss mit diesen Fehlern sinnvoll umgehen.

Die Erfahrung zeigt, dass Microservices-Architekturen trotz dieser Probleme funktionieren [3]. Wenn Microservices besonders klein gewählt werden, gibt es mehr verteilte Kommunikation und das System wird insgesamt langsamer. Das spricht für größere Microservices. Wenn ein Microservice eine UI enthält und einen guten funktionalen

Schnitt hat, kann er in den meisten Fällen ohne den Aufruf anderer Microservices auskommen, weil alle Bestandteile der Fachlichkeit in einem Microservice umgesetzt sind. Das Vermeiden verteilter Kommunikation ist ein weiterer Grund, Systeme fachlich sauber aufzubauen.

Nachhaltige Architektur

Microservices nutzen Verteilung auch dazu, um die Architektur durch Aufteilung in einzelne Microservices nachhaltig zu gestalten: Es ist viel schwieriger, einen Microservice zu nutzen als eine Klasse. Der Entwickler muss sich dazu mit der Verteilungstechnologie beschäftigen und die Schnittstelle des Microservice nutzen. Gegebenenfalls muss er für Tests Vorkehrungen treffen, um den aufgerufenen Microservice in Tests einzubeziehen oder durch einen Stub zu ersetzen. Schließlich muss der Entwickler mit dem Team Kontakt aufnehmen, das für den Microservice zuständig ist.

Eine Klasse in einem Deployment-Monolith zu nutzen, ist hingegen sehr einfach – auch wenn sie aus einem ganz anderen Bereich des Monolithen kommt und von einem anderen Team verantwortet wird. Weil es so einfach ist, eine Abhängigkeit zwischen zwei Klassen umzusetzen, schleichen sich in Deployment-Monolithen oft unbeabsichtigte Abhängigkeiten ein. Bei Microservices ist eine Abhängigkeit komplizierter umzusetzen und kann sich daher nicht einfach so einschleichen.

Refactoring

Allerdings führen die Grenzen zwischen den Microservices auch zu Herausforderungen beispielsweise beim Refactoring. Wenn sich herausstellt, dass eine bestimmte Funktionalität in einem Microservice nicht sinnvoll untergebracht ist, muss sie in einen anderen Microservice verschoben werden. Wenn der Ziel-Microservice in einer anderen Programmiersprache geschrieben ist, entspricht das letztendlich einer Neuimplementierung. Solche Probleme entfallen, wenn die Funktionalitäten innerhalb eines Microservice verschoben werden sollen. Auch dieser Faktor spricht eher für größere Microservices. Dieser Themenkomplex steht im Mittelpunkt von Abschnitt 8.3.

Teamgröße

Das unabhängige Deployment der Microservices und die Aufteilung in Teams ergeben eine obere Grenze für die Größe eines Microservice. Ein Team soll in einem Microservice Features unabhängig von anderen Teams implementieren und in Produktion bringen können. Dadurch erlaubt die Architektur, die Entwicklung zu skalieren, ohne dass dabei die Teams zu viel koordinieren müssen.

Ein Team muss Features unabhängig von den anderen Teams umsetzen können. Es scheint daher zunächst so, dass der Microservice so groß sein muss, dass Features sinnvoll in dem Microservice implementiert werden können. Wenn die Microservices kleiner sind, kann ein Team aber einfach für mehrere Microservices zuständig sein, die zusammen die Implementierung von fachlichen Features erlauben.

Eine untere Grenze für die Größe ergibt sich durch das unabhängige Deployment und die Aufteilung in Teams nicht.

Wohl aber eine obere Grenze: Wenn der Microservice so groß ist, dass er von einem Team nicht mehr weiterentwickelt werden kann, ist er zu groß. Ein Team sollte dabei eine Größe haben, wie sie für agile Prozesse besonders gut funktioniert. Das sind typischerweise drei bis neun Personen. Also darf ein Microservice auf keinen Fall so groß werden, dass ein Team ihn nicht mehr alleine weiterentwickeln kann. Neben der Größe spielt auch die Menge an Features eine Rolle, die in einem Microservices implementiert werden müssen. Wenn gerade sehr viele Änderungen notwendig sind, kann ein Team schnell überlastet sein. Abschnitt 13.2 zeigt Alternativen, um mehreren Teams die Arbeit an einem Microservice zu erlauben. Dennoch sollte ein Microservice nicht so groß sein, dass mehrere Teams an ihm arbeiten müssen.

Infrastruktur

Ein weiterer Einflussfaktor auf die Größe eines Microservice ist die Infrastruktur. Jeder Microservice muss unabhängig deployt werden können. Er muss eine Continuous-Delivery-Pipeline haben und eine Infrastruktur zum Ausführen des Microservice, die nicht nur in der Produktion, sondern auch in den verschiedenen Test-Stages vorhanden sein muss. Zu der Infrastruktur können auch Datenbanken oder Application Server gehören. Und es muss auch ein Build-System für den Microservice geben. Der Code des Microservice muss unabhängig von den anderen Microservices versioniert werden. Also muss es ein Projekt in der Versionskontrolle für den Microservice geben.

Abhängig davon, wie aufwendig es ist, für einen Microservice eine Infrastruktur bereitzustellen, kann die sinnvolle Größe eines Microservice unterschiedlich sein. Wenn die Größe besonders klein gewählt wird, wird das System in viele Microservices aufgeteilt und es müssen entsprechend mehr Infrastrukturen bereitgestellt werden. Bei größeren Microservices gibt es im System weniger von ihnen und es müssen auch nicht so viele Infrastrukturen bereitgestellt werden.

Build und Deployment der Microservices sollten sowieso automatisiert sein. Es kann aber dennoch aufwendig sein, alle benötigten Bestandteile der Infrastruktur für einen Microservice bereitzustellen. Wenn das Aufsetzen einer Infrastruktur für einen neuen Microservice automatisiert ist, sinkt der Aufwand für die Bereitstellung von Infrastruktur für neue Microservices. So kann die minimal mögliche Größe eines Microservice weiter reduziert werden. Unternehmen, die schon länger mit Microservices arbeiten, vereinfachen meistens das Erstellen neuer Microservices durch die automatisierte Bereitstellung von Infrastruktur.

Außerdem gibt es Technologien, mit denen der Overhead der Infrastruktur so weit sinkt, dass wesentlich kleinere Microservices möglich sind – dann allerdings mit einigen Einschränkungen. Solche Nanoservices behandelt Kapitel 15.

Ersetzbarkeit

Ein Microservice sollte möglichst einfach zu ersetzen sein. Das kann sinnvoll sein, wenn die Technologie des Microservice veraltet ist oder der Code des Microservice von so schlechter Qualität ist, dass er nicht mehr weiterentwickelt werden kann. Die Ersetzbarkeit von Microservices ist ein Vorteil gegenüber monolithischen Anwendungen, die kaum zu ersetzen sind. Wenn ein Monolith nicht mehr wartbar ist, muss er entweder mit erheblichen Kosten weiterentwickelt werden oder es muss doch eine Migration stattfinden, die auch sehr kostspielig ist. Je kleiner ein Microservice ist, desto einfacher ist es, ihn durch eine Neuimplementierung zu ersetzen. Über einer bestimmten Grenze ist der Microservice kaum noch zu ersetzen, weil es dieselben Herausforderungen wie bei einem Monolithen gibt. Ersetzbarkeit begrenzt die Größe eines Microservice also nach oben.

Transaktionen und Konsistenz

Transaktionen haben die ACID-Eigenschaften:

- Atomizität bedeutet, dass die Transaktion entweder ganz oder gar nicht ausgeführt wird. Bei einem Fehler werden alle Änderungen wieder rückgängig gemacht.
- Consistency (Konsistenz) bedeutet, dass vor und nach der Ausführung der Transaktion die Daten konsistent sind – also beispielsweise keine Validierungen verletzt sind.
- Isolation bedeutet, dass die Operationen der Transaktionen voneinander getrennt sind.
- Durability steht für Dauerhaftigkeit: Die Änderungen in der Transaktion werden gespeichert und stehen auch nach einem Absturz noch zur Verfügung.

Innerhalb eines Microservice können Änderungen in einer Transaktion stattfinden. Ebenso kann die Konsistenz der Daten in einem Microservice sehr einfach garantiert werden. Über einen Microservice hinaus wird das schwierig. Dann ist eine übergreifende Koordination notwendig. Bei einem Zurückrollen einer Transaktion müssten alle Änderungen in allen Microservices rückgängig gemacht werden. Das ist aufwendig und schwer umzusetzen, weil die Entscheidung, ob Änderungen rückgängig gemacht werden sollen oder nicht, garantiert zugestellt werden muss. Kommunikation in Netzwerken ist aber unzuverlässig. Bis zur Entscheidung, ob die Änderung erfolgen darf, sind weitere Änderungen an den Daten ausgeschlossen. Gegebenenfalls ist es nach weiteren Änderungen sonst schon nicht mehr möglich, eine bestimmte

Änderung noch zurückzunehmen. Wenn Microservices aber länger keine Änderungen an Daten durchführen können, reduziert das den Durchsatz des Systems.

Allerdings sind Transaktionen bei der Kommunikation über Messaging-Systeme möglich (siehe Abschnitt 9.4). Mit diesem Ansatz sind Transaktionen auch ohne enge Bindung der Microservices möglich.

Konsistenz

Neben Transaktionen ist auch die Konsistenz der Daten wichtig. Eine Bestellung muss beispielsweise auch irgendwann als Umsatz verbucht werden. Nur dann sind Umsatz und Bestelldaten konsistent. Die Konsistenz der Daten kann nur mit einer engen Abstimmung erreicht werden. Konsistenz von Daten kann über Microservices hinweg kaum zugesichert werden. Das bedeutet nicht, dass der Umsatz für die Bestellung nicht irgendwann gebucht wird – aber nicht genau zum selben Zeitpunkt und vielleicht auch nicht innerhalb einer Minute nach Bearbeitung der Bestellung. Schließlich erfolgt die Kommunikation über das Netzwerk – und ist damit langsam und unzuverlässig.

Änderungen von Daten innerhalb einer Transaktion und Konsistenz der Daten sind nur möglich, wenn alle betroffenen Daten in einem Microservice sind. Damit stellen sie eine untere Grenze für einen Microservice dar: Wenn Transaktionen mehrere Microservices umfassen sollen und Konsistenz der Daten über mehrere Microservices notwendig ist, sind die Microservices zu klein gewählt.

Kompensationstransaktionen über Microservices hinweg

Zumindest bei Transaktionen gibt es noch eine Alternative: Wenn eine Änderung an den Daten später zurückgerollt werden muss, dann können dafür Kompensationstransaktionen genutzt werden.

Das klassische Beispiel für eine verteilte Transaktion ist eine Reisebuchung, die aus einem Hotel, einem Mietwagen und einem Flug besteht. Nur alles zusammen soll gebucht werden oder nichts von allem. In realen Systemen und auch bei Microservices wird die Funktionalität in drei Microservices aufgeteilt, weil es drei fachlich sehr unterschiedliche Aufgaben sind. Dann wird bei den Systemen nachgefragt, ob das gewünschte Hotelzimmer, der gewünschte Mietwagen und der gewünschte Flug verfügbar sind. Danach wird dann alles reserviert. Wenn plötzlich beispielsweise das Hotel nicht mehr verfügbar ist, muss auch die Reservierung für den Flug und den Mietwagen rückgängig gemacht werden. In der realen Welt werden die Firmen für die Stornierung der Buchung jedoch wahrscheinlich eine Gebühr berechnen. Damit ist die Stornierung nicht einfach ein technisches Ereignis hinter den Kulissen wie das Zurückrollen einer Transaktion, sondern ein Geschäftsprozess. Das ist viel besser mit einer Kompensa-

tionstransaktion abbildbar. Mit diesem Ansatz sind auch Transaktionen über mehrere Elemente in Microservice-Umgebungen umsetzbar, ohne dass es eine enge technische Bindung gibt. Eine Kompensationstransaktion ist einfach ein normaler Service-Aufruf. Sowohl technische als auch geschäftliche Gründe können für eine Nutzung von Mechanismen wie Kompensationstransaktionen über Microservices sprechen.

Zusammenfassung

Es ergeben sich die folgenden Einflussfaktoren auf die Größe eines Microservice (siehe Abb. 4–1):

- Die Teamgröße stellt eine obere Grenze dar: Ein Microservice darf auf keinen Fall so groß sein, dass mehrere Teams an ihm arbeiten müssen. Schließlich sollen die Teams unabhängig voneinander entwickeln und Software in Produktion bringen. Das ist nur möglich, wenn jedes Team an einer eigenen Deployment-Einheit arbeitet – also einem eigenen Microservice. Ein Team kann aber auch an mehreren Microservices arbeiten.
- Die Modularisierung schränkt die Größe eines Microservice weiter ein: Er sollte möglichst so groß sein, dass ein Entwickler ihn verstehen und weiterentwickeln kann. Noch kleiner ist natürlich besser. Diese Grenze liegt unterhalb der Teamgröße: Was ein Entwickler noch verstehen kann, sollte ein Team auch noch weiterentwickeln können.
- Die Ersetzbarkeit nimmt mit der Größe des Microservice ab. Daher kann sie die obere Grenze für die Größe eines Microservice beeinflussen. Diese Grenze liegt unterhalb der Modularisierung: Wenn jemand den Microservice ersetzen kann, muss er ihn auch verstehen können.
- Eine untere Grenze ist die Infrastruktur: Wenn es zu aufwendig ist, für einen Microservice Infrastruktur bereitzustellen, sollte die Anzahl der Microservices eher kleiner sein – und damit ergibt sich, dass die Microservices eher groß sind.
- Ebenso nimmt die verteilte Kommunikation mit der Anzahl der Microservices zu. Auch aus diesem Grund sollte die Größe des Microservice nicht zu klein gewählt werden.
- Die Konsistenz der Daten und Transaktionen kann nur in einem Microservice sichergestellt werden. Daher dürfen die Microservices nicht so klein sein, dass Konsistenz und Transaktionen mehrere Microservices umfassen.

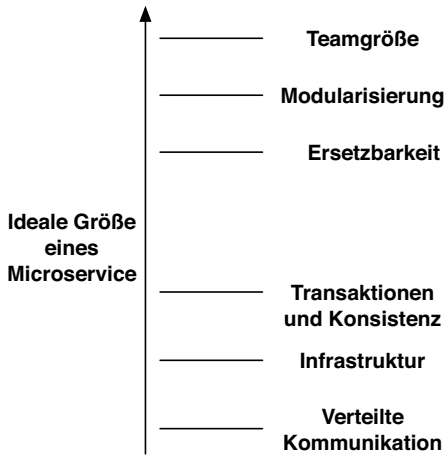

Abb. 4–1
Einflussfaktoren für die Größe eines Microservice

Diese Faktoren beeinflussen nicht nur die Größe der Microservices, sondern sie entsprechen auch einem bestimmten Verständnis von Microservices. Hauptvorteile der Microservices sind demnach unabhängige Deployments und das unabhängige Arbeiten der verschiedenen Teams. Dazu kommt die Ersetzbarkeit der Microservices. Daraus lassen sich Grenzen für die Größe eines Microservice ableiten.

Aber es gibt auch andere Gründe für Microservices. Wenn Microservices zum Beispiel wegen der unabhängigen Skalierung eingeführt werden, muss die Größe so gewählt werden, dass jeder Microservice eine Einheit ist, die unabhängig skalieren muss.

Wie klein oder groß ein Microservice sein kann, ist aus dieser Aufstellung alleine nicht zu ermitteln. Es hängt auch von der Technologie ab. Insbesondere der Aufwand für die Infrastruktur eines Microservice und für die verteilte Kommunikation hängt von der verwendeten Technologie ab. Kapitel 15 betrachtet Technologien, mit denen sehr kleine Services möglich werden – sogenannte Nanoservices. Sie haben andere Vor- und Nachteile als Microservices, die beispielsweise mit den Technologien aus Kapitel 14 umgesetzt werden.

Es gibt also keine ideale Größe, sondern die Größe hängt von der Technologie und dem Einsatzkontext der Microservices ab.

Selber ausprobieren und experimentieren

- Wie ist der Aufwand für das Deployment eines Microservice in Deiner Sprache, Plattform und Infrastruktur?
  - Ist es nur ein einfacher Prozess? Oder eine komplexe Infrastruktur mit Application Server oder anderen Infrastrukturelementen?
  - Wie kann der Aufwand für das Deployment gesenkt werden, sodass kleinere Microservices möglich werden?

Auf Basis dieser Informationen kannst du eine untere Grenze für die Größe eines Microservice definieren. Obere Grenzen sind durch die Teamgröße und die Modularisierung gegeben – auch hier solltest du dir entsprechende Grenzen überlegen.

4.2 Das Gesetz von Conway

Das Gesetz von Conway stammt von dem amerikanischen Informatiker Melvin Edward Conway [6][7] und besagt:

> *Organisationen, die Systeme designen, können nur solche Designs entwerfen, welche die Kommunikationsstrukturen dieser Organisationen abbilden.*

Wichtig ist, dass es nicht nur um Software geht, sondern um jegliche Art von Design. Die Kommunikationsstrukturen, von denen Conway spricht, müssen nicht mit dem Organigramm übereinstimmen. Oft gibt es informelle Kommunikationsstrukturen, die ebenfalls in diesem Kontext betrachtet werden können. Ebenso kann die geografische Verteilung der Teams die Kommunikation beeinflussen. Schließlich ist es einfacher, mit einem Kollegen im selben Raum oder zumindest am selben Standort zu sprechen, als wenn der Kollege in einer anderen Stadt oder gar einer anderen Zeitzone arbeitet.

Gründe für das Gesetz

Der Grund für das Gesetz von Conway liegt darin, dass jede organisatorische Einheit einen bestimmten Teil der Architektur entwirft. Sollen zwei Teile der Architektur eine Schnittstelle haben, ist eine Abstimmung über diese Schnittstelle notwendig – und damit eine Kommunikationsbeziehung zwischen den organisatorischen Einheiten, die für die jeweiligen Teile zuständig sind.

Aus dem Gesetz lässt sich auch ableiten, dass eine Modularisierung des Designs sinnvoll ist. Durch ein solches Design wird erreicht, dass sich nicht jeder im gesamten Team mit jedem anderen absprechen muss. Stattdessen können die Personen, die am selben Modul arbeiten,

sich eng abstimmen, während Personen, die an unterschiedlichen Modulen arbeiten, sich nur abstimmen müssen, wenn es eine Schnittstelle gibt – und dann auch nur über die Gestaltung der Schnittstelle.

Aber die Kommunikationsbeziehungen gehen weiter. Es ist einfacher, mit einem Team im selben Gebäude zu kollaborieren als einem Team in einer anderen Stadt, einem anderen Land oder gar einer anderen Zeitzone. Daher können Teile der Architektur, die viele Kommunikationsbeziehungen haben, besser von Teams in räumlicher Nähe umgesetzt werden, weil sie leichter miteinander kommunizieren können. Es geht eben beim Gesetz von Conway nicht um das Organigramm, sondern um die realen Kommunikationsbeziehungen.

Conway stellt übrigens auch die These auf, dass eine große Organisation viele Kommunikationsbeziehungen hat. Dadurch wird die Kommunikation schwieriger oder sogar ganz unmöglich. In der Folge kann auch die Architektur immer mehr in Mitleidenschaft gezogen werden und schließlich zusammenbrechen. Zu viele Kommunikationsbeziehungen sind also letztendlich für ein Projekt ein echtes Risiko.

Das Gesetz als Einschränkung

Normalerweise wird das Gesetz von Conway gerade in der Software-Entwicklung als eine Einschränkung angesehen. Nehmen wir an, ein Projekt wird nach technischen Aspekten aufgeteilt (Abb. 4–2). Alle Entwickler mit einer UI-Ausrichtung kommen in ein Team, die Entwickler mit Backend-Ausrichtung in ein zweites und das dritte Team sind die Datenbankexperten. Die Aufteilung hat den Vorteil, dass die drei Teams jeweils aus den Experten für die entsprechende Technologie bestehen. Dadurch ist die Organisationsform sehr einfach und klar umsetzbar. Außerdem erscheint diese Aufteilung auch logisch. Teammitglieder können sich sehr einfach unterstützen und der technische Austausch ist auch einfacher.

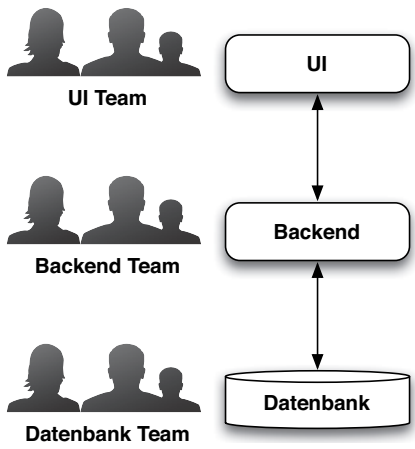

Abb. 4–2
Technische Aufteilung des Projekts

Aus dieser Aufteilung folgt nach dem Gesetz von Conway, dass die Teams drei technische Schichten implementieren werden: eine UI, ein Backend und eine Datenbank. Diese Aufteilung entspricht der Organisation, die auch durchaus sinnvoll aufgebaut ist. Aber sie hat einen entscheidenden Nachteil: Ein typisches Feature benötigt Änderungen an der UI, am Backend und an der Datenbank. Die UI muss die neuen Features für Nutzer verwendbar machen, das Backend muss die Logik umsetzen und die Datenbank muss Strukturen für das Speichern der entsprechenden Daten schaffen. Das hat folgende Nachteile:

- Derjenige, der das Feature umgesetzt haben will, muss mit drei Teams sprechen.
- Die Teams müssen sich untereinander koordinieren und neue Schnittstellen schaffen.
- Die Arbeiten der Teams müssen so koordiniert sein, dass sie zeitlich zueinander passen. Das Backend kann beispielsweise ohne Zulieferung der Datenbank kaum sinnvoll arbeiten – und die UI auch nicht ohne Zulieferungen des Backends.
- Wenn die Teams in Sprints arbeiten, führen die Abhängigkeiten zu zeitlichen Verzögerungen: Das Datenbankteam erstellt im ersten Sprint die nötigen Änderungen, im zweiten Sprint implementiert das Backend-Team die Logik und im dritten Sprint kommt die UI. So dauert die Implementierung eines Features drei Sprints.

Letztendlich führt dieser Ansatz zu einer großen Menge von Abhängigkeiten, Kommunikation und Koordination. Daher ist diese Organisation nicht sehr sinnvoll, wenn es darum geht, möglichst schnell neue Features umzusetzen.

Vielen Teams, die einen solchen Ansatz nutzen, ist die Auswirkung auf die Architektur nicht klar und sie beachten diesen Aspekt nicht weiter. Im Mittelpunkt einer solchen Organisation steht eher der Aspekt, dass Mitarbeiter mit ähnlichen Skills in der Organisation an einer ähnlichen Stelle positioniert sein sollen. So wird die Organisation zu einem Hemmnis für fachlich geschnittene Module wie Microservices, dem die Aufteilung der Teams in technische Schichten entgegensteht.

Das Gesetz als Enabler

Das Gesetz von Conway kann aber auch zur Unterstützung von Ansätzen wie Microservices genutzt werden. Wenn das Ziel ist, einzelne Komponenten möglichst unabhängig voneinander zu entwickeln, kann das System in fachliche Komponenten aufgeteilt werden. Anhand dieser fachlichen Komponenten können jeweils Teams gebildet werden. Abbildung 4–3 zeigt das: Es gibt jeweils ein Team für Produktsuche, Benutzer und Bestellprozess. Sie arbeiten an den jeweiligen Komponenten, die technisch in UI, Backend und Datenbank aufgeteilt sein kön-

nen. Übrigens sind die fachlichen Komponenten in der Abbildung gar nicht extra benannt, weil sie identisch mit der Bezeichnung der Teams sind. Komponenten und Teams sind synonym. Dieser Ansatz entspricht der Idee von sogenannten cross functional Teams, wie Methoden wie Scrum sie fordern. Diese Teams sollen verschiedene Rollen umfassen, sodass sie ein möglichst breites Aufgabenspektrum abdecken. Nur ein solches Team kann eine Komponente verantworten – von den Anforderungen über die Implementierung bis hin zum Betrieb.

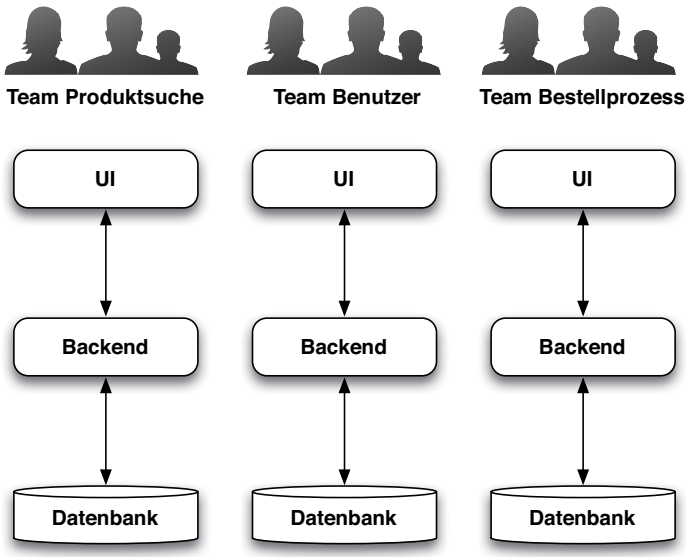

Abb. 4–3
Aufteilung nach Fachlichkeiten

Die Aufteilung in die technischen Artefakte und die Schnittstelle zwischen den Artefakten können nun innerhalb des Teams geklärt werden. Im einfachsten Fall muss dazu nur ein Entwickler mit dem Entwickler sprechen, der neben ihm am Tisch sitzt. Zwischen den Teams ist die Abstimmung komplizierter. Die ist aber auch nicht so oft notwendig, weil Fachlichkeiten idealerweise nur von einem einzigen Team umgesetzt werden müssen. Außerdem entstehen durch diesen Ansatz dünne Schnittstellen zwischen den Fachlichkeiten, weil für die Definition der Schnittstelle eine aufwendige Koordination über Teamgrenzen hinweg notwendig ist.

Letztendlich lautet die zentrale These aus dem Gesetz von Conway, dass Architektur und Organisation nur zwei Seiten derselben Medaille sind. Wenn das geschickt genutzt wird, hat das System eine recht saubere und für das Projekt nützliche Architektur. Gemeinsames Ziel von Architektur und Organisation ist die reibungslose Arbeit der Teams mit möglichst wenig Koordination.

Die saubere Aufteilung der Fachlichkeiten in die Komponenten erleichtert auch die Wartung. Und da für jede Fachlichkeit und jede Komponente nur ein Team zuständig ist, wird diese Aufteilung auch langfristig stabil und das System so auch langfristig wartbar bleiben.

Die Teams benötigen Anforderungen, an denen sie arbeiten können. Das bedeutet, dass die Teams fachliche Ansprechpartner haben müssen, die solche Anforderungen stellen können. Das hat Auswirkungen auf die Organisation über das Projekt hinaus, denn die Anforderungen kommen aus den Fachbereichen und auch diese müssen entsprechend dem Gesetz von Conway den Teamstrukturen im Projekt und der fachlichen Architektur entsprechen. Das Gesetz kann über die Software-Entwicklung hinaus auf die Kommunikationsstrukturen der gesamten Organisation einschließlich der Anwender ausgeweitet werden. Oder umgekehrt können sich die Teamstruktur eines Projekts und damit die Architektur eines Microservice-Systems aus der Organisation der Fachbereiche ergeben.

Das Gesetz und Microservices

Die bisherige Betrachtung hat nur allgemein eine Beziehung zwischen der Architektur und der Organisation des Projekts gezeigt. Es wäre ohne Weiteres denkbar, die Architektur an Fachlichkeiten auszurichten und jeweils einem Team die Verantwortung für eine Fachlichkeit zu übergeben, ohne dabei Microservices zu nutzen. Dann würde das Projekt einen Deployment-Monolithen entwickeln, in dem alle Funktionalitäten umgesetzt werden. Microservices unterstützen den Ansatz aber. Abschnitt 3.1 hat schon gezeigt, dass Microservices eine technische Unabhängigkeit bieten. Zusammen mit der fachlichen Aufteilung werden die Teams noch unabhängiger voneinander und müssen sich noch weniger koordinieren. Sowohl die technische als auch die fachliche Koordination der Teams wird auf das Notwendigste reduziert. Dadurch wird es deutlich einfacher, an vielen Features parallel zu arbeiten und die Features auch in Produktion zu bringen.

Microservices sind als technische Umsetzung einer Architektur besonders gut geeignet, den Ansatz einer fachlichen Aufteilung mithilfe des Gesetzes von Conway zu unterstützen. Tatsächlich ist genau dieser Aspekt eine wesentliche Eigenschaft einer Microservices-Architektur.

Allerdings bedeutet die Ausrichtung der Architektur an den Kommunikationsstrukturen, dass eine Änderung des einen auch eine Änderung des anderen bedingt. Dadurch werden Änderungen der Architektur zwischen den Microservices aufwendiger und der Prozess insgesamt weniger flexibel. Wenn eine Funktionalität von einem Microservice in einen anderen verschoben wird, kann das auch zur Folge haben, dass die Funktionalität von einem anderen Team weitergepflegt wird. Sol-

che organisatorischen Änderungen machen Änderungen an der Software komplizierter.

Nun muss geklärt werden, wie diese fachliche Aufteilung sinnvoll umgesetzt werden kann. Dazu hilft Domain-Driven Design (DDD).

Selber ausprobieren und experimentieren

Betrachte ein dir bekanntes Projekt:

- Wie sieht die Teamstruktur aus?
  - Ist sie technisch oder fachlich getrieben?
  - Muss die Struktur für einen Microservices-Ansatz geändert werden?
  - Wie muss sie geändert werden?
- Ist die Architektur auf Teams sinnvoll verteilbar? Schließlich soll jedes Team unabhängige fachliche Komponenten verantworten und dort Features umsetzen können.
  - Welche Änderungen an der Architektur wären notwendig?
  - Wie aufwendig wären die Änderungen?

4.3 Domain-Driven Design und Bounded Context

In seinem gleichnamigen Buch hat Eric Evans [5] Domain-Driven Design (DDD) als eine Pattern-Sprache formuliert. Es ist eine Sammlung von zusammenhängenden Entwurfsmustern und soll die Entwicklung von Software vor allem in komplexen Domänen unterstützen. Die Namen der Entwurfsmuster sind in KAPITÄLCHEN gesetzt.

Domain-Driven Design ist für ein Verständnis von Microservices wichtig, weil es bei der Strukturierung von größeren Systemen nach Fachlichkeiten hilft. Genau so ein Modell benötigen wir für den Schnitt eines Systems in Microservices. Jeder Microservice soll eine eigene fachliche Einheit bilden, die so geschnitten ist, dass für Änderungen oder neue Features nur ein Microservice geändert werden muss. Dann ziehen wir aus der unabhängigen Entwicklung in den Teams den maximalen Nutzen, weil mehrere Features parallel ohne größere Koordinierung umgesetzt werden können.

Ubiquituous Language

Als Basis definiert DDD, wie ein Modell für eine Domäne entworfen werden kann. Eine wesentliche Grundlage von DDD ist UBIQUITUOUS LANGUAGE (allgemeingültige Sprache). Die Idee ist, dass die Software genau dieselben Begriffe nutzen soll wie die Domänenexperten. Das gilt auf allen Ebenen: sowohl für den Code und die Variablennamen wie auch für die Datenbankschemata. Dadurch wird sichergestellt, dass die Software tatsächlich die wesentlichen Elemente der