Nachhaltige und flexible Softwareentwicklung

Wiederverwendbare Microservices durch Domain-Driven Design

Heutzutage sollten Anwendungssysteme so gestaltet sein, dass sie sich möglichst schnell an neue Anforderungen anpassen lassen. Da hohe Wirtschaftlichkeit und Flexibilität erstrebenswert sind, setzen Unternehmen zunehmend auf sogenannte Microservice-Architekturen, welche die Anforderungen durch feingranulare und autonome Microservices erfüllen. Der Entwurf der Microservices hat einen erheblichen Einfluss auf die letztliche Wiederverwendbarkeit und Flexibilität eines Anwendungssystems. Dabei sind allerdings zahlreiche Entwurfsentscheidungen zu treffen, die sich auf die Qualität des Anwendungssystems auswirken. Der Ansatz des Domain-Driven Design liefert interessante Konzepte für den Entwurf von Microservices und der daraus ableitbaren Architektur.

Für das Verständnis ist es notwendig, zunächst die grundlegenden Konzepte zu erläutern, auf die sich dieser Artikel stützt. Aus diesem Grund erläutert **Kasten 1** zunächst die Begrifflichkeiten Microservice und Microservice-Architektur, im Anschluss fassen wir die notwendigen Konzepte von Domain-Driven Design kurz zusammen.

Zentrale Konzepte des Domain-Driven Design (DDD)

Bei der Betrachtung komplexer Problemstellungen, die das zu entwickelnde Anwendungssystem lösen soll, bedarf es einem tief gehenden Verständnis der Fachdomäne des Kunden (im Folgenden nur noch *Domäne*). Ohne dieses ist eine optimale Unterstützung der Geschäftsprozesse des Unternehmens durch das Anwendungssystem nicht gewährleistet.

Ein Ansatz zur Auflösung dieser Problematik ist das *Domain-Driven Design (DDD)* von Eric Evans aus dem Jahr 2003 [Eva03, Ver13, Mil15]. Bei DDD handelt es sich in erster Linie um eine Ansammlung verschiedener Konzepte, welche uns bei dem Entwurf der Domäne unterstützen. Das zentrale Konzept von DDD ist das Formalisieren der Erkenntnisse über die Domäne, das *Domänenwissen*, in einem *Domänenmodell*. Semantisch und syntaktisch lässt DDD freien Spielraum, was die Gestaltung des Domänenmodells angeht; jede Form der Präsentation von Domänenwissen ist erwünscht, solange das Verständnis der Domäne gefördert wird. Prinzipiell sind jedoch in das Domänenwissen nur die für das Anwendungssystem relevanten Geschäftsobjekte aufzunehmen, die im Domänenmodell als *Domänenobjekte* bezeichnet werden. Es erfolgt also keine allgemeine Analyse der Domäne.

Das für ein Anwendungssystem erstellte Domänenwissen wird bei der Entwicklung eines nachfolgenden (in der gleichen Domäne angesiedelten) Anwendungssystems genutzt und entsprechend erweitert. Das Ergebnis ist ein Domänenmodell, das die Basis der zu der Domäne beitragenden Anwendungssysteme darstellt. Diese Basis besteht auf der Architekturebene aus den Microservices, die in unterschiedlichen Anwendungskontexten wiederverwendet werden. Zu jeder Entwicklungsphase des Anwendungssystems sieht DDD den Kontakt zu den Verantwortlichen aufseiten des Unternehmens vor, welche die Domä-

Microservice und Microservice-Architekturen

- *Microservice:* Autonomer Service, der einen minimalen Ausschnitt einer Fachlichkeit mit hoher Kohäsion abbildet und seine immaterielle Dienstleistung über wohldefinierte Schnittstelle veräußert. Weitere Eigenschaften sind die Berücksichtigung des Single Responsibility Principle (SRP) sowie eine lose Kopplung.
- *Microservice-Architektur:* Verteiltes Softwaresystem, bei dem die zugrunde liegenden Systembausteine aus Microservices bestehen.

Kasten 1

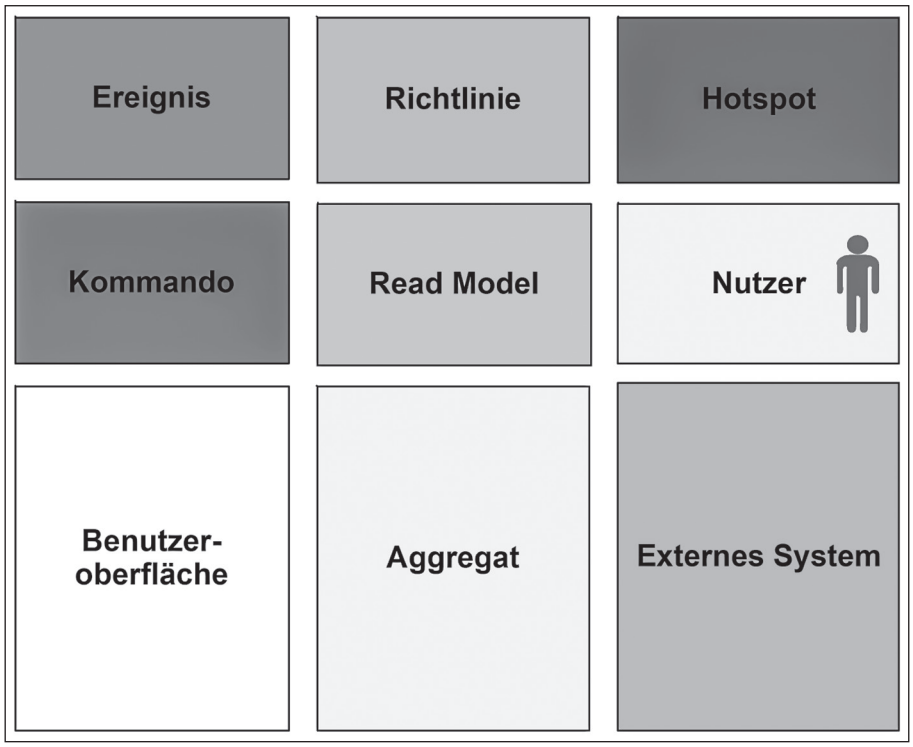

Abb. 1: Event-Storming-Elemente

nenexperten darstellen. Die *Domänenexperten* sind diejenigen, die ein tief greifendes Wissen über die Domäne besitzen, welches letztlich durch Anwendungssysteme abgebildet werden soll.

Um sich initial ein Bild von der Domäne zu machen und die zu lösende Problemstellung zu verstehen, liefert *Event Storming* einen entsprechenden Ansatz. Zusammen mit den Domänenexperten erarbeiten sich die IT-Experten im Rahmen eines Workshops ein gemeinsames Verständnis, das dann in Form von Haftnotizen auf eine Tafel oder Wand aufgebracht wird. Diese Haftnotizen repräsentieren dabei unterschiedliche Elemente, die in **Abbildung 1** dargestellt sind. Beispielsweise sind Ereignisse bedeutungsvolle Aktionen, die in der Domäne geschehen. Bei einem Kommando handelt es sich um Ereignisse, welche durch Nutzer ausgelöst werden. Die Bedeutung der weiteren Elemente ist in [Bra17] beschrieben.

Die wiederkehrende Diskussion mit den Domänenexperten wird auch als *Knowledge Crunching* bezeichnet. Um solche Diskussionen effizienter und effektiver zu gestalten, etabliert DDD schließlich eine sogenannte *ubiquitäre Sprache*, die den Wortschatz der Projektmitglieder (IT-Experten und Domänenexperten) in Form eines Vertrages festlegt. Üblicherweise wird die ubiquitäre Sprache in Form eines Glossars gepflegt.

Für die Implementierung der Artefakte von DDD bietet sich die objektorientierte Programmierung (OOP) an. Die Domänenobjekte und ihre Beziehungen zueinander ähneln stark einem UML-Klassendiagramm (Unified Modeling Language). Hier kommt ein zentrales Prinzip von DDD zum Tragen, denn die in dem Domänenmodell festgehaltenen Strukturen müssen sich unverändert in dem Quellcode des Anwendungssystems wiederfinden. Konkret bedeutet dies, dass zu jedem Domänenobjekt eine eigene Klasse existiert. Neben dieser grundsätzlichen Verwendung des Domänenmodells wird auch für die Implementierung auf die ubiquitäre Sprache zurückgegriffen. Durch diese Konsistenzhaltung wird eine bessere Auffindbarkeit des Domänenwissens im Quellcode erreicht und so die Wartbarkeit erleichtert.

Weiterhin stellt DDD für die Gestaltung der Makroarchitektur des Anwendungssystems eine *Schichtenarchitektur* (engl. layered architecture) vor. Die in **Abbildung 2** dargestellte Architektur besteht aus den vier Schichten:

- *Präsentationsschicht:* Stellt die Oberflächenelemente (Web-GUI) zur Verfügung.
- *Anwendungsschicht:* Beinhaltet die anwendungsspezifischen Funktionalitäten.
- *Domänenschicht:* Repräsentiert das Domänenwissen.
- *Infrastrukturschicht:* Realisiert unter anderen die Persistierung der Daten, Bereitstellung von Informationen oder die Anbindung externer Systeme.

Zwar führt DDD diese Schichten ein, zielt allerdings nur auf die Analyse und Konzeption der Domänenschicht ab. Zu den restlichen Schichten werden keine Konzepte vorgestellt. Die explizite Trennung der Domänenschicht von der restlichen Logik der Anwendung soll zu einer hohen Wiederverwendbarkeit und auch einer besseren Wartbarkeit der essenziellen Bestandteile des Anwendungssystems führen. Dies lässt sich darauf zurückführen, dass domänenspezifische Aspekte unabhängig von einer konkreten Anwendung oder Anwendungsgruppe gültig sind. In der Praxis wird dies allerdings oftmals nicht berücksichtigt, was die Wiederverwendbarkeit der entstehenden Microservices teilweise stark einschränkt.

Entwurf von Microservices und Microservice-Architektur

Beim Entwurf von Microservices sind einige Entwurfsentscheidungen, die sich positiv oder negativ auf das Anwendungssystem auswirken können, zu treffen. Aus diesem Grund sollte jede Entwurfs-

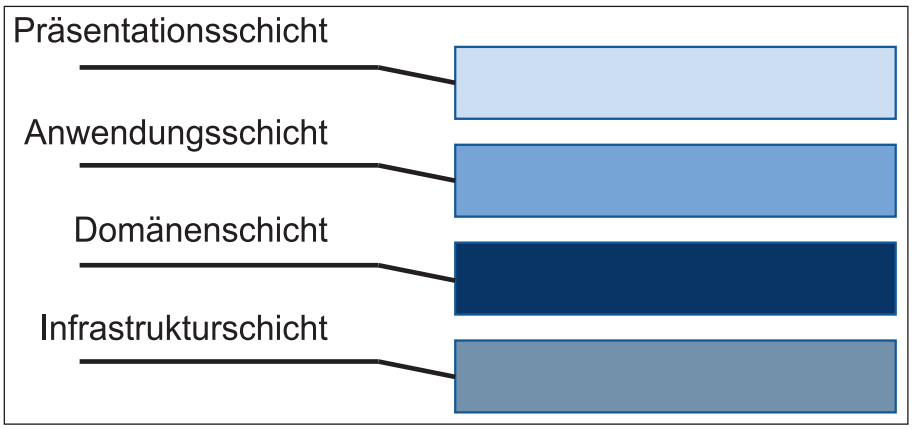

Abb. 2: Schichtenarchitektur eines Anwendungssystems basierend auf DDD

entscheidung mit Bedacht gewählt sein, was wiederum bedingt, dass notwendige Informationen vorliegen müssen, um fundierte Entscheidungen treffen zu können. Gleiches gilt auch für die Umsetzung der Microservices. Hier liefert beispielsweise Adam Wiggins zwölf „Regeln" [Wig17], die bei der Umsetzung von Microservices oder im Allgemeinen von Software-as-a-Service-Anwendungen zu beachten sind. Dieser Abschnitt soll einige Aspekte aufzeigen, die beim Entwurf zu berücksichtigen sind.

Anwendungssystem und Anforderungen als Ausgangspunkt

Ein Anwendungssystem soll stets für einen bestimmten Zweck (beispielsweise Unterstützung von Geschäftsprozessen in Unternehmen) eingeführt werden. In diesem Artikel betrachten wir die Unterstützung von Trainierenden im Rahmen eines Connected Training Systems (CTS).
Bei einem CTS handelt es sich um ein System, welches aus der Symbiose von Hardware und Software hervorgeht, um den Bedürfnissen eines Trainierenden gerecht zu werden und so einen optimalen Trainingsablauf sicherzustellen. In den Anforderungen des Anwendungssystems werden diese Zwecke detailliert festgehalten. In jeder Phase der Softwareentwicklung werden das Anwendungssystem und dessen Zweck fokussiert. Auch DDD hebt diese Fokussierung hervor. Gleiches gilt bei der Entwicklung eines microservice-basierten Anwendungssystems. Keineswegs wird ein Softwareentwicklungsprojekt mit der Absicht gestartet, einen einzelnen Microservice zu entwickeln. Beispielswiese werden für den Trainingsbetrieb verschiedene Microservices benötigt, welche verschiedene Teilbereiche der CTS-Domäne verwalten. Der Ausgangspunkt für die Entwicklung neuer Microservices – und auch für die Wiederverwendung dieser – ist die Entwicklung eines Anwendungssystems und dessen Anforderungen.
Durch die Verwendung von DDD wird die Domäne des Unternehmens für das Anwendungssystem beziehungsweise dessen Microservices genauer analysiert. Jedoch werden, wie auch die Schichtenarchitektur von DDD zeigt, weitere Aspekte des Anwendungssystems ausgelassen. Hierunter fallen auch die eigentlichen Anforderungen des Anwendungssystems, welche typischerweise in der Präsentations- und Anwendungsschicht verankert sind. Gerade die Anwendungsschicht ist für microservice-basierte Anwendungen Dreh- und Angelpunkt. In dieser wird die Orchestrierung der im Backend befindlichen Microservices vorgenommen. Neben DDD bedarf es somit eines weiteren Ansatzes der Softwareentwicklung.

```
1 Feature: Analyse des Trainings
2  Als Trainierender
3  Möchte ich mein Training analysieren
4  Sodass ich den Trainingsplan an meinen
   Fortschritten anpassen kann
5
6 Background: Meine Trainingsdaten werden aufgezeichnet
7
8 Scenario: Steigerung der Fitness
9  Angenommen ich habe mein Training absolviert
10 Wenn ich meine Trainingsdaten analysiere
11 Und sich eine Steigerung meiner Fitness ergibt
12 Dann wird mir diese Information mitgeteilt
13 Und eine Anpassung des Trainingsplans vorgenommen
```

Listing 1: Beispiel eines BDD-Features zur Trainingsanalyse

Ein möglicher Ansatz ist das *Behavior-Driven Development* (BDD, [Sma15]), welches auf die Erhebung der Anforderungen des Anwendungssystems abzielt. Die Idee von BDD basiert auf *Test-Driven Development* (TDD). Mit BDD werden textuelle *Features* erstellt, welche später der Implementierung als ausführbares Artefakt zur Verfügung stehen. Diese Ausführbarkeit wird durch vordefinierte Bausteine erreicht (in **Listing 1** fett dargestellt). Gerade für microservice-basierte Anwendungssysteme ist die Erstellung der Features wertvoll. Aus diesen können die Informationen der Anwendungsschicht und somit die Orchestrierung der Microservices gezogen werden.
Zwischen den Ansätzen BDD und DDD bestehen Synergien. Beide Ansätze bauen teilweise auf denselben Konzepten (Domänenobjekte, ubiquitäre Sprache) auf. Dies lässt sich mit dem in **Listing 1** dargestellten Feature verdeutlichen. Aus dem Text des Features lassen sich Domänenobjekte und Teile der ubiquitären Sprache identifizieren, welche für die Modellierung des Domänenmodells verwendet werden können. Beispielsweise lassen sich die Domänenobjekte Fitness, Training und Trainingsplan aus dem Feature entnehmen.

Betrachtende Domäne im Mittelpunkt

Die Domäne sollte im Mittelpunkt des Entwurfs eines Anwendungssystems stehen und kontinuierlich durch Domänenexperten und Geschäftsanalysten verfeinert und erweitert werden, weshalb auch hier ein iteratives und inkrementelles Vorgehensmodell zum Einsatz kommt.

Es wird dabei nicht das Ziel eines allumfassenden Modells fokussiert, sondern stattdessen lediglich der Ausschnitt, welcher für die Umsetzung der Anforderungen vonnöten ist. Andernfalls würde die Gefahr eines sogenannten *Big Ball of Mud* bestehen. Wie bereits erwähnt, haben anwendungsbezogene Anforderungen in der Domäne nichts zu suchen und sollten stattdessen auf Anwendungsseite umgesetzt werden. Falls die Notwendigkeit aufseiten der Microservices besteht (z. B. aufgrund etwaiger Leistungsanforderungen), so sollten diese in einem dedizierten Service gekapselt werden, wie beispielsweise in Form eines *Backend-for-Frontend*-Service (BFF) oder in einem dedizierten lokalen *API-Gateway*, welches als Zwischenglied zwischen der Anwendung und den benötigten Microservices dient. Dort findet sich dann auch die Orchestrierung der entsprechenden Microservices wieder.
Als Architekturstil für die Umsetzung von Microservices mit DDD hat sich die sogenannte *hexagonale Architektur* (oder auch *Onion-Architektur*, **siehe Abbildung 3**) zunehmender Beliebtheit erfreut, die sich wie folgt zusammensetzt:

- *Domäne:* Beinhaltet alle Domänenobjekte (Entitäten und Werteobjekte).
- *Domänenservices:* Services, die auf Domänenobjekten operieren und selbst keinen eigenen Zustand halten.
- *Applikationsservices:* Services, die als Bindeglied zwischen Ports- und Adapter sowie Domänenservice und Domäne dienen.
- *Ports/Adapter:* Anknüpfungspunkt von externen Abhängigkeiten.

An dieser Stelle muss zwischen dem Architekturstil des Anwendungssystems und des Microservice unterschieden werden. Während der Microservice sich auf die he-

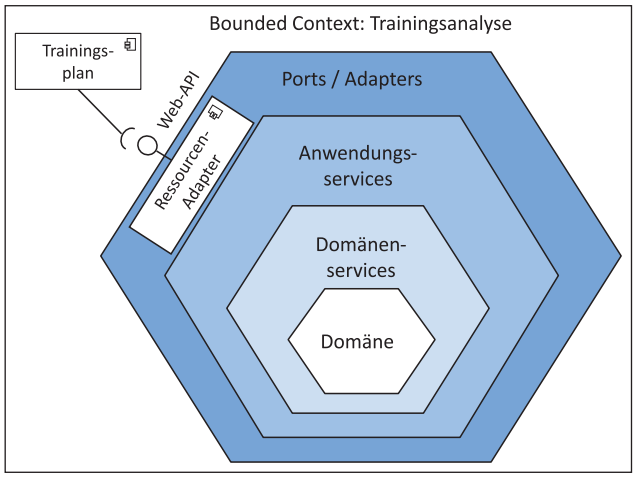

Abb. 3: Hexagonale Architektur für den Entwurf eines Microservice

xagonale Architektur beruft, wird für das Anwendungssystem die Schichtenarchitektur von DDD verwendet. Das Offerieren der Dienstleistung eines Microservice erfolgt über sogenannte *Web Application Programming Interfaces* (Web-APIs). Im Beispiel der CTS-Domäne stellt der Bounded Context „Trainingsanalyse" die Ressourcen über das Web-API bereit. Weitere Bounded Contexts (wie der Trainingsplan) können die bereitgestellten Ressourcen über das angebotene Web-API nutzen. Für den Entwurf von Web-APIs genießt hier insbesondere der ressourcenorientierte Architekturstil (zweite Ebene des Richardson *Maturity Model*, [Fow10]) zunehmender Beliebtheit, was sich beispielsweise anhand der aktuellen und neu veröffentlichten Web-APIs zeigt. Ein Web-API bildet durch seine vertragliche Eigenschaft einen zentralen und essenziellen Bestandteil eines Microservice, da nur über dieses mit dem Microservice kommuniziert werden kann. Hier haben sich insbesondere die Benutzbarkeit, Mächtigkeit und Interoperabilität als wichtige Qualitätsmerkmale eines Web-API in den letzten Jahren hervorgetan. Um letztlich auch die Auffindbarkeit eines Microservice zu begünstigen, sollte das resultierende Web-API derart beschrieben sein, dass es die gekapselte Domäne in adäquater Form abbildet. Idealerweise wird die zuvor eingeführte ubiquitäre Sprache zu einer *Published Language*.

Verknüpfung von Microservices mittels Context Mapping

Die Makroarchitektur eines microservicebasierten Anwendungssystems unterscheidet sich maßgeblich von einem monolithischen Anwendungssystem. Die für die als BDD-Feature erhobenen Anforderungen notwendige Domänenlogik findet sich verteilt in der Microservice-Architektur wieder. Über eine Orchestrierung der Microservices in der Anwendungsschicht des Anwendungssystems wird diese Domänenlogik vereint und die Anforderungen werden erfüllt. Die Orchestrierung der Microservices ist dabei eine Herausforderung und deren Auffindbarkeit ist eine Grundvoraussetzung.

DDD unterstützt die Entwicklung eines microservice-basierten Anwendungssystems mittels zweier Konzepte. Ein Konzept bezeichnet den *Bounded Context*, der zunächst nur eine explizite Grenze für die Gültigkeit eines Domänenmodells darstellt (**siehe Abbildung 4**). Genau bedeutet dies, dass das in dem Domänenmodell festgehaltene Wissen nur innerhalb dieser Grenze gültig ist; Selbiges gilt für die ubiquitäre Sprache.

Sam Newman ist der Auffassung, dass der Bounded Context ein ideales Mittel zum Entwurf von Microservices darstellt [New14]. Denn neben der Beschränkung der Gültigkeit führt diese Grenze auch zu einer Trennung zwischen geteiltem und verstecktem Domänenwissen. Das geteilte Domänenwissen kann als Schnittstelle zu der Domänenlogik des Microservice gesehen werden.

Nimmt man nun wieder Bezug zu der Orchestrierung der Microservices für ein Anwendungssystem, kann das Konzept der *Context Map* genutzt werden. Die Context Map ist ein Diagramm, welches Bounded Contexts und ihre Beziehungen zueinander darstellt. Wichtig anzumerken ist, dass Beziehungen zwischen den Bounded Contexts nicht nur technischer Natur sind. Eine Beziehung zwischen den Bounded Contexts kann ebenfalls als Kommunikationsweg zwischen zwei Entwicklungsteams gesehen werden. Die Context Map kann somit als Werkzeug für die Orchestrierung der Microservices in der Anwendungsschicht des Anwendungssystems verwendet werden.

Wie für DDD üblich, ist auch die Darstellungsform der Context Map frei wählbar. Ein sehr vereinfachtes Beispiel zur Illustration für eine Context Map findet sich in **Abbildung 4**, welche die Bounded Contexts – oder Microservices – der CTS-Domäne und deren Beziehungen zueinander abbildet. Für die CTS-Domäne wurden sieben Bounded Contexts identifiziert, welche jeweils einen bestimmten Bereich der Domäne beschreiben. Diese sind jeweils durch die Kästen in der Abbildung dargestellt. Mittels der Linie zwischen den Bounded Contexts wird eine Verbindung und somit ein Austausch von Informationen dargestellt.

Ergänzend liefert DDD wichtige Konzepte, wie eine Form der Orchestrierung erfolgen kann. **Tabelle 1** gibt einen Überblick über die wichtigsten Muster für die Orchestrierung aus [Ver13]. Die jeweils genutzten Muster können ebenfalls in das Schaubild ergänzt werden, auf sie wurde lediglich im Zuge der Übersichtlichkeit verzichtet.

Resultierende Microservice-Architektur sowie deren Umsetzung

Die resultierende Microservice-Architektur ergibt sich durch den Verbund der Microservices, welche letztlich in ihrer Gesamtheit das Anwendungssystem widerspiegeln und damit die fachlichen Anforderungen umsetzen. Die fachliche Anforderung der Trainingsanalyse wird durch den gleichnamigen Bounded Context umgesetzt. Dieser kümmert sich um die Auswertung der aufgezeichneten Trainingsdaten und ermöglicht, dass die Analyseergebnisse in die Trainingspläne

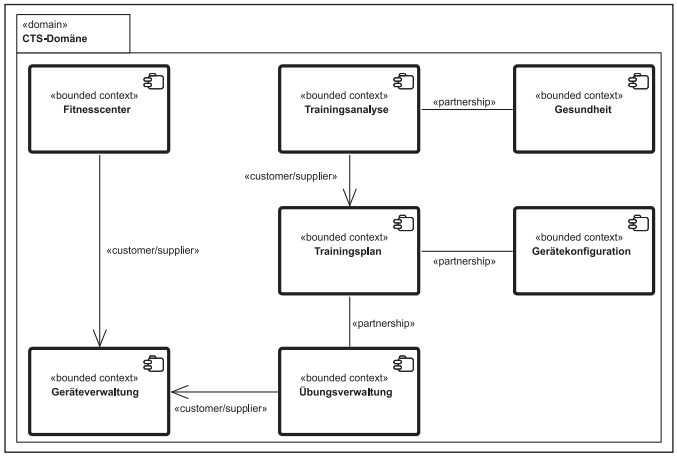

Abb. 4: Context Mapping zwischen den Bounded Contexts der CTS-Domäne

Muster	Beschreibung
Partnership	Die Teams kooperieren gemeinsam an den beteiligtem Bounded Contexts, um einen Misserfolg zu vermeiden
Shared Kernel	Es handelt sich um einen expliziter Bereich, welcher von vielen Teams geteilt wird
Customer-Supplier	Der Supplier (Lieferant) bietet die Funktionalität, die der Customer (Kunde) benötigt. Der Kunde hat dabei ein Mitspracherecht und einen Einfluss auf das Modell des Lieferanten
Conformist	Wie bei der Customer-Supplier-Beziehung, nur nutzt das abhängige Team das Modell des liefernden Teams ohne Mittbestimmungsrecht
Anticorruption Layer	Eine zusätzliche Schicht, welche das Modell von anderen Modellen trennt und die Domänenobjekte aufbereitet
Open Host Service	Eine Schnittstelle, welche Zugriff auf eine Menge von Diensten gewährt
Published Language	Der Dateiaustausch läuft über eine Schnittstelle und es werden die gleichen Datenformate verwendet
Separate Ways	Es besteht keine Kooperation zwischen den Teams und die eigene Umsetzung der benötigten Funktionalität

Tabelle 1: Übersicht über die Beziehungen zwischen Bounded Contexts

mit einfließen können. Gleichzeitig findet hier auch eine Bewertung des aktuellen Gesundheitszustands einer Person statt. Schließlich ist das System auch darauf ausgelegt, etwaige Trainierende mit körperlichen Beschwerden bestmöglich zu unterstützen und hier einen optimalen Heilungsprozess zu gewährleisten. Die aufbereiteten Daten erhält dann der zuständige Arzt oder Physiotherapeut, sodass dieser manuell den Trainingsplan den Bedürfnissen seines Patienten anpassen kann.

Die Verknüpfung von zwei Microservices erfolgt ausschließlich über die bereitgestellten Web-APIs. Der Bounded Context *Trainingsplan* verwendet die bereitgestellte Programmierschnittstelle des Bounded Context *Trainingsanalyse*. Die Orchestrierung zwischen den beiden Bounded Contexts ist durch eine Customer-Supplier-Beziehung realisiert, wobei *Trainingsplan* den Lieferanten darstellt und sich in der Rolle des Kunden befindet. Das Team, welches den Service des Trainingsplans umsetzt, hat ein gewisses Mitsprache- und Gestaltungsrecht bei der Erstellung der Web-APIs des Microservices der Trainingsanalyse. Die Programmierschnittstellen können beispielsweise einen ressourcenorientierten, hypermedia-basierten oder ereignisgetriebenen Architekturstil verfolgen. Die Wahl des entsprechenden Architekturstils ist immer von dem jeweiligen Anwendungsfall abhängig und sollte daher mit Bedacht gewählt werden. Wie bereits im Vorfeld erwähnt, neigen Unternehmen häufig zu ressourcenorientierten Web-APIs, was sich auch an der Anzahl veräußerter Dienstleistungen zeigt. Entscheidend für die Wiederverwendbarkeit ist die Trennung der Domänenlogik von der Anwendungslogik, da die Domänenlogik eine inhärente Wiederverwendbarkeit aufweist.

Die Umsetzung der Microservice-Architektur erfolgt heutzutage in Container-basierten Virtualisierungsumgebungen. Bekannte Technologievertreter sind hier *Docker* oder *rkt*. Die Verwaltung der laufenden Container übernehmen wiederum andere Lösungen, wie *Kubernetes* von Google [Bur17]. In diesem Zusammenhang ist auch häufig von Container-as-a-Service (CaaS)-Lösungen die Rede, welche einen ganzheitlichen Lebenszyklus zum Betreiben, Verwalten, Überwachen und Warten von container-basierten Lösungen anbieten. An dieser Stelle sei beispielsweise OpenShift [Pic17, Dum18] zu nennen, welche ebenfalls auf Kubernetes setzt. Da die Darstellung der Umsetzung zu weit führen würde, wird dies nicht weiter ausgeführt und stattdessen auf die entsprechenden Technologien und weiterführende Literatur [Bur17, Pic17, Dum18] verwiesen.

Wartung der Microservice-Architektur

Zur Bewahrung der Vorzüge einer Microservice-Architektur gilt es, nach der Entwicklung von microservice-basierten Anwendungssystemen auch die Pflege der entstandenen Bausteine zu berücksichtigen. Die Entscheidung, eine Microservice-Architektur als Architekturstil für die Anwendungslandschaft des eigenen Unternehmens zu verwenden, hat zur Folge, dass eine Vielzahl von verschiedenen Software- und Hardwarekomponenten verwaltet werden müssen.

Die Wartung der Microservices muss bereits auf der organisatorischen Ebene des Unternehmens stattfinden. Strukturen und Zuständigkeiten der einzelnen Komponenten der Architektur müssen festgelegt und vor allem berücksichtigt werden. Etabliert hat sich dabei die Zuordnung eines Entwicklungsteams auf genau einen Microservice. Dieses Entwicklungsteam hat die Hoheit, den Microservice fachlich und technisch voranzutreiben und fungiert gleichzeitig als Ansprechpartner für weitere Entwicklungsteams, die auf die Funktionalität des Microservice zugreifen möchten. Somit sind neben den Web-APIs auch die Kommunikationswege zwischen den Entwicklungsteams offengelegt.

Die Zuordnung eines Entwicklerteams zu genau einem Microservice wird durch das von Melvin Conway aufgestellte Gesetz *Conway's Law* untermauert [Con68]. Dieses besagt, dass die organisatorische Struktur des Unternehmens unmittelbar auf den Entwurf des Produktes Einfluss nimmt. Angewandt auf die Microservice-Architektur bedeutet dies, dass aus den Schnittstellen der Microservices die Kommunikationswege zwischen den Entwicklerteams abgeleitet werden können. Als positiver Nebeneffekt strukturiert sich das Unternehmen in kleine autonome Entwicklerteams. Die Größe des Entwicklungsteams spielt dabei auch eine entscheidende Rolle zum effektiven und effizienten Handeln.

Jeff Bezos, Chief Executive Officer (CEO) von Amazon, etablierte in seinem Unternehmen die *Two Pizzas Team*-Regel, welche besagt, dass ein autonomes Team von nur zwei Pizzen satt wird [Dye13]. Angewandt auf die Entwicklerteams der Microservices, stellt diese Regel eine gute Grundlage zur Bestimmung der Teamgrößen dar.

Je nach Komplexität der Microservice-Architektur variiert der Aufwand zur Wartung. Viele Anwendungen und viele Microservices sind ohne geeignete Unterstützung nicht zu bewältigen. Martin Fowler stellte in seinem Blog den Ansatz der *HumaneRegistry* vor [Fow08, Abe16], ein Service, der als Wiki-Portal für andere Services dient. Über die HumaneRegistry können somit die Microservices auffindbar gemacht werden. Dies ist ein essenzieller Faktor für den Erfolg einer Microservice-Architektur. Weiterhin kann die bereits vorgestellte Context Map genutzt werden. Bei der kontinuierlichen

Literatur & Links

[Abe16] S. Abeck, D. Forschner, P. Giessler, Service-Registry als zentrale Komponente in einer Microservice-Architektur, in: JavaSPEKTRUM, 06/2016

[Bra17] A. Brandolini, Introducing Event Storming, Leanpub, 2017

[Bur17] B. Burns, K. Hightower, J. Beda, Kubernetes: Up and Running, O'Reilly, 2017

[Con68] M. E. Conway, How do Committees Invent, Datamation 14.4, 1968, S. 28-31

[Dum18] G. Dumpleton, Deploying to OpenShift: A Guide for Busy Developer, O'Reilly, 2018

[Dye13] J. Dyer, H. Gregersen, The Secret to Unleashing Genius, Forbes September 2, 2013, S. 96-100

[EvaO3] E. Evans, Domain-Driven Desing: Tackling Complexity in the Heart of Software, Addison-Wesley, 2003

[FowO8] M. Fowler, HumaneRegistry, 2008, siehe: **https://martinfowler.com/bliki/HumaneRegistry.html**

[Fow10] M. Fowler, Richardson Maturity Model, 2010, siehe: **https://martinfowler.com/articles/richardsonMaturityModel.html**

[Mil15] S. Millett, Patterns, Principles and Practices of Domain-Driven Design, John Wiley & Sons, 2015

[New14] S. Newman, Building Microservices, O'Reilly, 2014

[Pic17] St. Picozzi, M. Hepburn, N. O'Connor, DevOps with OpenShift: Cloud Deployments Made Easy, O'Reilly, 2017

[Sma15] J. F. Smart, BDD in Action – Behavior-Driven Development for the Whole Software Lifecycle, Manning, 2015

[Ver13] V. Vernon, Implementing Domain-Driven Design, Addison-Wesley, 2013

[Wig17] A. Wiggins, The Twelve Factor App, 2017, siehe: **https://12factor.net/**

Pflege einer Context Map ist sie ein wertvolles Werkzeug zur Wahrung einer organisierten Microservice-Architektur.

Fazit

Der Entwurf von Microservices und letztlich der Microservice-Architektur kann auf unterschiedliche Arten erfolgen. Um letztlich die anvisierten Vorzüge eines modularen und verteilten Anwendungssystems zu erhalten, können die Konzepte aus DDD genutzt werden. So unterstützt DDD sowohl beim Entwurf eines einzelnen Microservice als auch bei der Strukturierung des gesamten Anwendungssystems. Bei der Gestaltung des Anwendungssystems wird die Domäne in den Mittelpunkt gerückt, wodurch eine hohe Wiederverwendbarkeit und Flexibilität des Anwendungssystems bei neuen Anforderungen erreicht werden kann.
Daneben kann durch die Modularisierung und Kapselung mittels Microservices eine hohe Wirtschaftlichkeit bei etwaigen Anpassungen erzielt werden, da diese durch das *Single Responsibility Principle (SRP)* nur an einer Stelle durchzuführen sind. Auch hier gilt es, letztlich etwaige Abwägungen zu tätigen, falls die resultierenden Kosten der organisatorischen Abhängigkeit zwischen Entwicklungsteams eine separate und lose gekoppelte Entwicklung überwiegen. ||

Die Autoren

Dr. Pascal Giessler
(pascal.giessler@kit.edu)
ist Head of Research and Development bei der syndikat7 GmbH und promovierte im Bereich des domänengetriebenen Entwurfs von Microservices und ressourcenorientierten Web-APIs am Karlsruher Institut für Technologie (KIT) innerhalb der Forschungsgruppe Cooperation & Management (C&M). Daneben ist er Dozent an der HECTOR School of Engineering and Management mit Schwerpunkt auf dem Entwurf und der Entwicklung von Microservices.

Benjamin Hippchen
(benjamin.hippchen@kit.edu)
ist Berater der Unternehmensberatung Scheer GmbH und befasst sich dort mit der Thematik des Process Mining. Neben seiner Beratertätigkeit ist er als wissenschaftlicher Mitarbeiter der Forschungsgruppe Cooperation & Management (C&M) am Karlsruher Institute für Technologie (KIT) angestellt. Im Rahmen seiner Promotion befasst er sich mit Microservice-Architekturen und dem domänengetriebenen Entwurf. Darüber hinaus ist er Teilnehmer des Software-Campus-Programms.

Michael Schneider
(michael.schneider@kit.edu)
ist wissenschaftlicher Mitarbeiter der Forschungsgruppe Cooperation & Management (C&M) am Karlsruher Institute für Technologie (KIT). Seine Forschungsschwerpunkte sind der domänengetriebene Entwurf, die verhaltensgetriebene Entwicklung und das Internet der Dinge.

Prof. Dr. Sebastian Abeck
(sebastian.abeck@kit.edu)
leitet an der KIT-Fakultät für Informatik die Forschungsgruppe Cooperation & Management (C&M). Seine Forschungsinteressen betreffen insbesondere die Microservice-Architekturen, Connected Cars und das Identitäts- und Zugriffsmanagement. Die von C&M entwickelten Konzepte werden intensiv mit Partnern aus der Forschung und der Industrie im Hinblick auf deren Einsetzbarkeit und Praxistauglichkeit erprobt.